LUND UNIVERSITY

Contributions to the Construction of Extensible Semantic Editors

Söderberg, Emma

2012

Link to publication

Citation for published version (APA):
Söderberg, E. (2012). *Contributions to the Construction of Extensible Semantic Editors*. [Doctoral Thesis (compilation), Department of Computer Science].

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Contributions to the Construction of Extensible Semantic Editors

Emma Söderberg

Doctoral Dissertation, 2012

Department of Computer Science
Lund University

ISBN 978-91-976939-8-1
ISSN 1404-1219
Dissertation 41, 2012
LU-CS-DISS:2012-4

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: emma.soderberg@cs.lth.se

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2012

© *2012 Emma Söderberg*

ABSTRACT

This dissertation addresses the need for easier construction and extension of language tools. Specifically, the construction and extension of so-called semantic editors is considered, that is, editors providing semantic services for code comprehension and manipulation. Editors like these are typically found in state-of-the-art development environments, where they have been developed by hand.

The list of programming languages available today is extensive and, with the lively creation of new programming languages and the evolution of old languages, it keeps growing. Many of these languages would benefit from proper tool support. Unfortunately, the development of a semantic editor can be a time-consuming and error-prone endeavor, and too large an effort for most language communities. Given the complex nature of programming, and the huge benefits of good tool support, this lack of tools is problematic.

In this dissertation, an attempt is made at narrowing the gap between generative solutions and how state-of-the-art editors are constructed today. A generative alternative for construction of textual semantic editors is explored with focus on how to specify extensible semantic editor services. Specifically, this dissertation shows how semantic services can be specified using a semantic formalism called reference attribute grammars (RAGs), and how these services can be made responsive enough for editing, and be provided also when the text in an editor is erroneous.

Results presented in this dissertation have been found useful, both in industry and in academia, suggesting that the explored approach may help to reduce the effort of editor construction.

CONTENTS

Preface **ix**

Acknowledgements **xiii**

Popular Scientific Summary in Swedish **xv**

I Introduction **1**
1 Problem Statements 2
2 Related Work 4
3 Overview of Contributions 7
4 Conclusions and Future Work 9
References 10

Included Papers 15

I Building Semantic Editors using JastAdd **17**
1 Introduction 17
2 Overview and Background 18
3 Example Editors 22
4 Generality and Limitations 26
5 Conclusions 27
References 27

II Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level **29**
1 Introduction 29
2 Control-flow Analysis 31
3 Dataflow Analysis 42

4 Dead Assignment Analysis 48
5 Language Extensions 51
6 Evaluation 52
7 Related Work 59
8 Conclusions 60
9 Acknowledgements 61
References 61

III Automated Selective Caching for Reference Attribute Grammars 65
1 Introduction 65
2 Reference Attribute Grammars 67
3 Attribute Instance Graphs 69
4 Computing a Cache Configuration 71
5 Evaluation 74
6 Related Work 83
7 Conclusions and Future Work 85
8 Acknowledgements 85
References 85

IV Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking 89
1 Introduction 90
2 Reference Attribute Grammars 91
3 Consistent Attribution 97
4 Dependency Tracking 98
5 Consistency Maintenance 100
6 Related Work 104
7 Conclusion and Future Work 105
References 106

V A Comparative Study of Incremental Attribute Grammar Solutions to Name Resolution 109
1 Introduction 109
2 Name Resolution 110
3 Comparison 113
4 Conclusions 115
References 115

VI Practical Scope Recovery using Bridge Parsing 117
1 Introduction 118
2 Background 119
3 Bridge Parsing 122
4 Bridge Parsing for Java 128
5 Evaluation 132

6 Conclusions 136
References 137

VII Natural and Flexible Error Recovery for Generated Modular Language Environments **141**
1 Introduction 141
2 Composite Languages and Generalized Parsing 144
3 Island Grammars 147
4 Permissive Grammars 149
5 Parsing Permissive Grammars with Backtracking 162
6 Layout-Sensitive Recovery of Scoping Structures 168
7 Layout-Sensitive Region Selection 171
8 Applying Error Recovery in an Interactive Environment 179
9 Implementation 184
10 Evaluation 186
11 Related Work 198
12 Conclusion 201
References 201

PREFACE

This dissertation is divided into two parts. The first part introduces the topic of this dissertation together with its contributions and conclusions, and the second part includes papers in support of these claims.

List of Included Papers

The following papers are included in this dissertation:

Paper I. Building Semantic Editors using JastAdd – Tool Demonstration Emma Söderberg and Görel Hedin. In proceedings of the *11th Workshop on Language Descriptions, Tools, and Applications (LDTA'11), ACM*, Saarbrücken, Germany, April 2011.

Paper II. Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level Emma Söderberg, Görel Hedin, Torbjörn Ekman, and Eva Magnusson. *Science of Computer Programming, 2012, Elsevier B.V.* In press.

Paper III. Automated Selective Caching for Reference Attribute Grammars Emma Söderberg and Görel Hedin. In proceedings of the *3rd International Conference on Software Language Engineering (SLE'10), Lecture Notes in Computer Science, 2011, Vol. 6563, pp. 2–21, Springer Berlin/Heidelberg*.

Paper IV. Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking Emma Söderberg and Görel Hedin. *Technical report, LU-CS-TR:2012-249, ISSN 1404-1200, Report 98, 2012*, Department of Computer Science, Lund University.

Paper V. A Comparative Study of Incremental Attribute Grammar Solutions to Name Resolution Emma Söderberg and Görel Hedin. In electronic proceedings of the *5th Conference on Software Language Engineering (SLE'12)*, Dresden, Germany, September 2012.

Paper VI. Practical Scope Recovery using Bridge Parsing Emma Nilsson-Nyman[1], Torbjörn Ekman, and Görel Hedin. In proceedings of the *1st International Conference on Software Language Engineering (SLE'08), Lecture Notes of Computer Science, 2009, Vol. 5452, pp. 95–113, Springer Berlin/Heidelberg*.

Paper VII. Natural and Flexible Error Recovery for Generated Modular Language Environments Lennart C.L. Kats, Maartje de Jonge, Emma Söderberg, and Eelco Visser. Accepted for publication in *ACM Transactions on Programming Languages and Systems, 2012*.

Contribution Statement

The author of this dissertation, Emma Söderberg, is the main contributor of Paper I-VI. For these papers, she was the main designer and author, and did all the implementation and evaluation work, with the exception of Paper VI where the evaluation was carried out jointly with Torbjörn Ekman.

The research amounting to Paper VII was carried out in collaboration with the SERG group at TU Delft. Emma Söderberg contributed to the integration of SGLR with bridge parsing, the evaluation of the approach, and the writing related to these parts.

List of Related Papers

The following papers are related to this dissertation, and have contributions by its author, but are not included:

- **Declarative Intraprocedural Flow Analysis of Java Source Code** Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. In proceedings of the *8th Workshop on Language Descriptions, Tools and Applications* (LDTA'08), Budapest, Hungary, April 2008. *Electronic Notes of Theoretical Computer Science, Vol. 238:5, pp. 155–171, 2009, Elsevier B.V.*.

[1] Emma Nilsson-Nyman is the maiden name of Emma Söderberg.

- **Providing Rapid Feedback in Generated Modular Language Environments: Adding Error Recovery to Scannerless Generalized-LR Parsing** Lennart C.L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, Eelco Visser. In proceedings of the *24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09), pp. 445–464, 2009, ACM*.

- **A Plan for Building Renaming Support for Modelica** Görel Hedin, Emma Nilsson-Nyman, and Johan Åkesson. In electronic proceedings of the *3rd Workshop on Refactoring Tools* (WRT'09), Orlando, Florida, USA, October 2009.

- **Natural and Flexible Error Recovery for Generated Parsers** Maartje de Jonge, Emma Nilsson-Nyman, Lennart C.L. Kats, and Eelco Visser. In proceedings of the *2nd International Conference on Software Language Engineering (SLE'09), Lecture Notes of Computer Science, 2010, Vol. 5969, pp. 204–223, Springer Berlin/Heidelberg*.

ACKNOWLEDGEMENTS

This work was funded by The Swedish Research Council, eLLIIT: The Linköping-Lund Initiative on IT and Mobile Communication, The Swedish Governmental Agency for Innovation Systems (VINNOVA), The Engineering and Physical Sciences Research Council of the UK (EPSRC), The Royal Physiographic Society in Lund, and the Google Anita Borg Memorial Scholarship.

I am very grateful to a lot of people for their support and advice during my work on this dissertation. I would like to thank my supervisor Görel Hedin, whose support, knowledge, and experience have been invaluable during this work. I would also like to thank my co-supervisor Boris Magnusson. I am grateful to Jörn Janneck for advice and enjoyable discussions, and to Torbjörn Ekman for early encouragement. I would also like to thank Roger Henriksson and Jonas Skeppstedt for advice.

All the work presented in this dissertation was done in collaboration with others, and I would like to extend my gratitude to my co-authors Torbjörn Ekman, Maartje de Jonge, Sven Gestegård Robertz, Görel Hedin, Lennart Kats, Boris Magnusson, Eva Magnusson, David Svensson Fors, Eelco Visser, and Johan Åkesson. An extra thanks to my co-authors at TU Delft (Lennart Kats, Maartje de Jonge and Eelco Visser) for fruitful discussions and splendid work.

During the composition of this dissertation I received valuable comments from several people and I would like to thank Flavius Gruian, Görel Hedin, Jörn Janneck, Roger Henriksson, and Christian Söderberg for taking the time to proof-read drafts.

While working on the content of this dissertation, I have had the pleasure to visit other research groups as well as companies. I would like to thank Oege de Moor and Torbjörn Ekman for inviting me to work at the Computing Laboratory at University of Oxford, and I would like to thank Eelco Visser for inviting me to work with his group at TU Delft. Thanks to Google UK for a splendid internship in

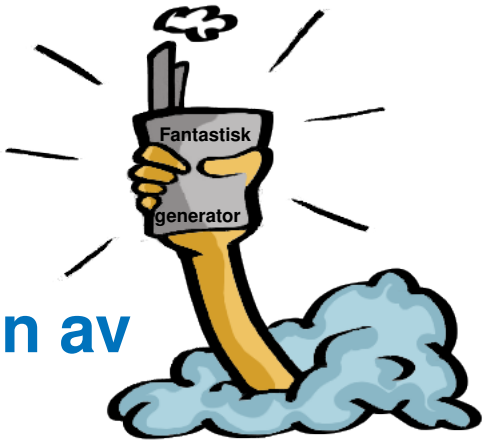
London, and thanks to Robert Führer for hosting my visit to IBM TJ Watson. I would also like to thank Modelon (Tove Bergdahl, Jesper Mattsson, Jon Sten, and Johan Åkesson) for joint work on the JModelica IDE, and John Lindskog, Erik Mossberg, Jesper Mattsson, and Philip Nilsson for using early versions of the editor architecture in their master thesis work.

A big thanks to all colleagues and students at the Department of Computer Science in Lund for providing a nice and relaxed place to work. I would like to thank all students and faculty members who have contributed to an enjoyable series of doktorandfikor, a special thanks to all presenters (Görel Hedin, Sten Henriksson, Thore Husfeldt, Jörn Janneck, Björn Regnell, Per Runeson), and I would like to thank all those who have contributed to an interesting series of JastAdd seminars (Tove Bergdahl, Niklas Fors, Görel Hedin, John Lindskog, Jesper Mattsson, Anders Nilsson, Jonas Rosenqvist, Jon Sten, Johan Åkesson, Jesper Öqvist). I am grateful to Anders Bruce, Peter Möller, Lars Nilsson, Lena Ohlsson, Thomas Richter, Anne-Marie Westerberg and the administrative staff for help with various practical matters. I would also like to thank Per Andersson, Klas Nilsson, Jonas Wisbrant, Linus Åkesson for fun discussions, and thank Mehmet Ali Arslan and Gustav Cedersjö for sharing some great music.

Finally, I would like to thank my family and friends for all their love and support, and last, but far from the least, thank you Christian for many happy moments and plenty to come.

Emma Söderberg
December 2012
Lund

POPULAR SCIENTIFIC SUMMARY IN SWEDISH



Effektiv konstruktion av utbyggbara semantiska editorer

Av Emma Söderberg

Instutionen för datavetenskap
Lunds universitet

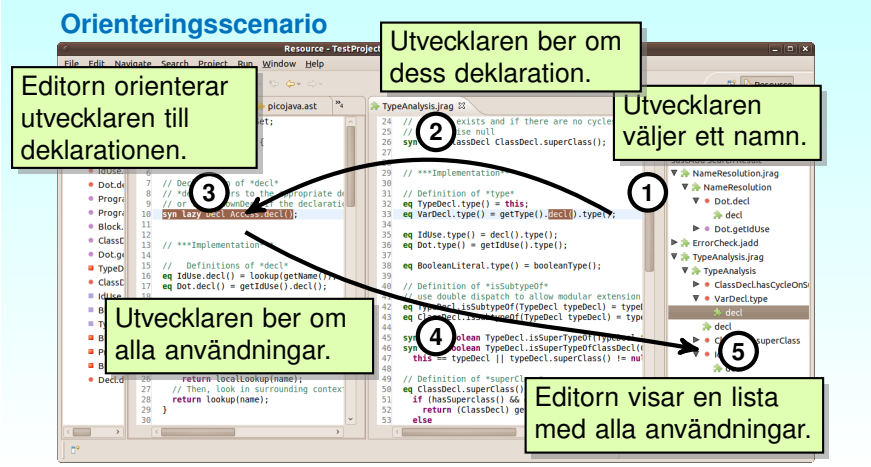
Programmeringsverktyg som semantiska editorer hjälper oss att utveckla och förstå mjukvara, vilket är viktigt i ett samhälle där vi måste kunna lita på att mjukvaran runt omkring oss fungerar. Tyvärr är utvecklingen av semantiska editorer ofta en tidskrävande och komplicerad uppgift, och majoriteten av språk saknar programmeringsverktyg. Hur gör vi det enklare att konstruera och underhålla semantiska editorer?

Programmeringsspråk utgör en central del av datavetenskapen, de avgör *vad* vi kan instruera en dator att göra och *hur* vi gör det. Efterhand som vi upptäcker nya bättre sätt att kommunicera med datorer utvidgar vi existerande språk eller skapar nya.

Dessutom, likt jargong, kan vi skapa språk ämnade för en specifik domän, som till exempel HTML, skapat för att konstruera websidor. Resultatet är en rik flora av språk och språkutvidgningar.

Samtidigt som programmeringsspråk hjälper oss att uttrycka lösningar på problem, hjälper programmeringsverktyg oss att utveckla lösningar, samt att förstå andras lösningar.

Tyvärr är utvecklingen av verktyg tiskrävande och komplex, och de flesta programmeringsspråk saknar verktyg.



Figur: Ett exempel på ett orienteringsscenario där editorn hjälper utvecklaren att förstå hur saker hänger ihop i ett program.

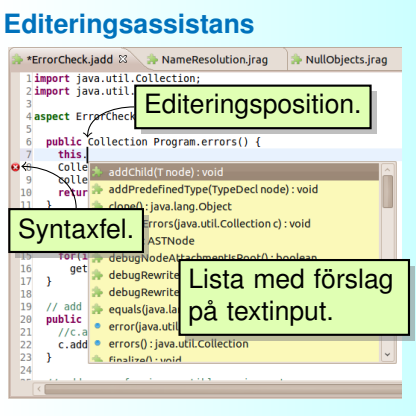
Om vi kan göra konstruktionen av verktyg enklare kan vi hjälpa fler mjukvaruutvecklare. Om vi dessutom kan göra det enkelt att utvidga dessa verktyg kan vi stödja språkutvidgningar.

Semantiska editorer

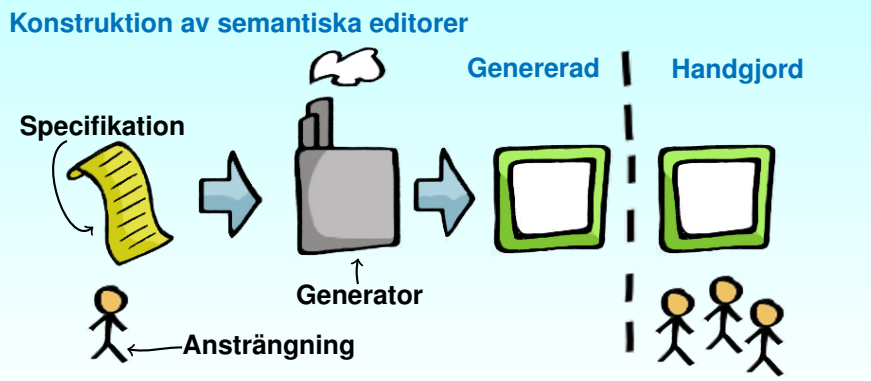
En semantisk editor är ett exempel på ett programmeringsverktyg som förstår sig på språksemantik. Denna förståelse gör det möjligt att erbjuda sofistikerade editortjänster. Exempelvis kan en semantisk editor hjälpa en utvecklare att orientera sig i ett program.

Ofta i programmering definieras ett namn på ett ställe i ett program för att sedan användas på många andra ställen i samma program. En editor kan visa var ett namn är deklarerat, samt var ett namn används. Något som kan vara svårt att reda ut för hand för språk med avancerad semantik.

Ett annat exempel på en semantisk editortjänst är en så kallad editeringsassistent som exempelvis kan hjälpa en utvecklare att leta upp vilka namn som är giltiga att använda på en viss plats i ett program.



Figur: Ett exempel på hur en editeringsassistent kan hjälpa en utvecklare.



Figur: En illustration av editorgenerering till vänster, och manuell konstruktion till höger. De små gubbarna symboliserar ansträngning.

Konstruktion av utbyggbara semantiska editorer

Ett vanligt angreppssätt för att göra mjukvaruutveckling effektivare är att generera kod, snarare än att skriva den själv, eller att återanvända kod.

För editorutveckling kan detta innebära att man försöker generera koden som beskriver hur editorn ska bete sig, eller att grafiska editorkomponenter återanvändas, eller båda.

I den här avhandlingen undersöks hur semantiska texteditorer kan genereras från formella språkbeskrivningar. Specifikt, undersöks hur semantiska editorer kan genereras från så kallade referensattributgrammatiker, en formalism för att beskriva språksemantik.

Avhandlingen redogör för hur semantiska editortjänster för orientering, alternativa programvyer, analys samt editeringsassistans kan genereras från en specifikation. Därtill visas hur sofistikerade editortjänster, beroende av kunskap om flödet i ett program, kan genereras och dessutom utvidgas för att hantera språkutvidgningar.

Utöver detta undersöks hur felaktig input kan hanteras i en texteditor. Detta kan vara utmanande eftersom program under editering ofta kan bli strukturellt trasiga. Det kan då vara svårt att förstå vad ett program betyder, men man vill ändå kunna erbjuda editortjänster som behöver förstå just detta.

För att reda ut detta behöver man laga programmet, och ju bättre man kan laga det, desto bättre tjänster kan man erbjuda. I avhandlingen beskrivs en teknik för att laga strukturellt trasiga program med hjälp av formatteringsinformation, som många gånger annars ignoreras vid textöversättning.

En annan viktig sak som avhandlingen undersöker är hur man effektivt kan beräkna den semantiska information som tjänsterna i en semantisk editor behöver genom att undvika att beräkna om information i onödan.

Sammanfattning

Det är utmanande att försöka göra det enklare att utveckla semantiska editorer, men bidragen i den här avhandlingen har tagit oss några viktiga steg på vägen.

INTRODUCTION

Today, integrated development environments (IDEs) such as Eclipse [Foua], IntelliJ IDEA [Jetb], and NetBeans [Cora] have grown popular with software developers and represent the state of the art. With knowledge of language semantics, these editors can provide developers with semantic services for code comprehension and manipulation [HW09]. For instance, with knowledge of the scope rules of a language, an editor can compute where a variable is declared in a program and where a variable is referenced. An editor with such information can provide services such as 'find declaration' and 'find references', that aid developers with code comprehension (Figure 1), and services such as 'name completion', that aid developers with code manipulation (Figure 2).

Semantic editors are useful, but can be both time-consuming to construct and difficult to maintain. For example, the Java editors offered by Eclipse, NetBeans and IntelliJ were all written more or less from scratch in Java. For Eclipse this work included the development of a Java compiler tailored for editing [Foua], while for NetBeans and IntelliJ it included interfacing to the javac compiler [Opeb], a compiler originally developed for batch compilation.

Once an editor has been created, the next step is to maintain it. Depending on the language, this work is more or less difficult. In the case with Java, the language is both complex and actively evolved by a lively community. During this language development, language extensions are considered and either postponed, discarded or included in the next language release. This release dictates what maintenance that needs to be done on editors supporting the language.

On rare occasions a language extension is provided with editor support already during the development process, to make it easier to evaluate the feature. For example, a Java language extension currently under consideration [Opea] (Project Lambda) has been provided with experimental editor support in upcoming releases of NetBeans [Corb] and IntelliJ [Jeta]. This kind of early editor support makes it easier to involve users in the language development process, and would be useful to have for more extensions.

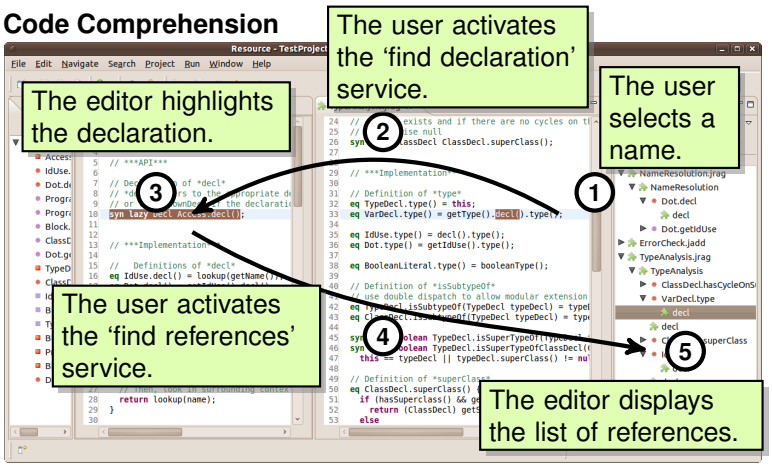


Figure 1: A Browsing Scenario An example of a browsing scenario where the editor helps the developer to understand the code via a 'find declaration' service and a 'find references' service.

1 Problem Statements

This dissertation addresses the need for easier construction and maintenance of semantic editors, by exploring generative approaches to editor construction. Generative techniques have grown popular for construction of language tools such as parsers, for which there are several generators available (e.g., ANTLR, JavaCC and Beaver), but have not yet gained the same popularity for editor construction.

To narrow the gap between generative approaches and how state-of-the-art editors are constructed today, this dissertation aims to demonstrate the potential of editor generation. Specifically, this dissertation explores how a semantic formalism called reference attribute grammars (RAGs) can be used to specify extensible semantic editor services.

Compilers and editors have many things in common, with regard to the kind of analyses they perform, and it is reasonable that a technique used for compiler construction may be used also for editor construction. RAGs have been successfully used both for specification of complex static semantics, for languages such as Java [EH07b] and Modelica [Åke+10], and for complex language extensions [EH07b; Öqv12; Hed+11]. These experiences suggest that RAGs could be useful for specification of extensible semantic editor services.

However, one concern with using RAGs for specification of editor services is performance. It has been shown that RAGs can provide reasonable performance for a generated compiler [EH07b], and it is reasonable that similar performance could be achieved for an editor. Still, this may not be good enough for editing

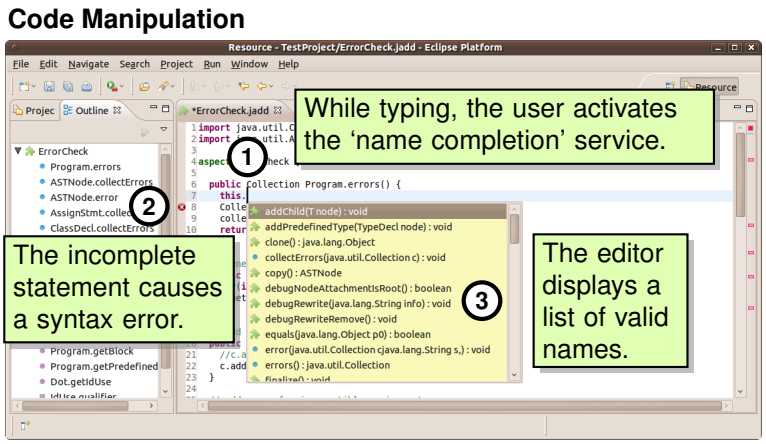


Figure 2: An Edit Scenario An example of an edit scenario where the user has entered an incomplete statement and the editor helps the user by suggesting valid names via a 'name completion' service.

where the user interacts with the tool in a different way than with a compiler. To not obstruct the work flow of the user, editor services need to be responsive enough for the user scenarios in which they are activated.

Responsiveness may mean different things for different services. User studies suggest that feedback should be given within $100 - 200$ ms for services activated by keystrokes [Mil68; DM01]. These limits may serve as a guideline for how fast a service such as 'name completion' needs to be. Notably, 'name completion' may be requested as often as copy/paste operations during editing [Mur+06], and feedback delays for such a service could severely hinder a users work flow.

Another concern for using RAGs is the robustness of editor services. A RAG-based approach is dependent on the availability of an abstract syntax tree (AST) in order to compute semantic information. In the event that a user has entered code which cannot be parsed, the editor may have trouble to provide editor services, as an AST to do computations on may be missing.

This dependency on the availability of an AST makes error recovery during parsing important. That is, it increases the importance of finding a parsing approach which can deal with the kinds of parsing problems that may occur during editing. RAGs in themselves do not provide a solution for parsing, and as this dissertation is exploring generative approaches, the natural complement to RAGs is to use a parser generator for this task.

2 Related Work

This section gives an overview of editor construction frameworks, parser generators and error recovery, and attribute grammars, the semantic formalism underlying RAGs.

2.1 Editor Construction Frameworks

To construct a semantic editor from scratch can be a task which is both error-prone and time-consuming, and to build a compiler, graphical components (views, pop-up boxes etc.), and editor services by hand is a lot of work. This effort may be reduced if some of these parts can be reused or generated.

During the last decade, the Eclipse platform [Foub] has greatly contributed to the construction of new editors. Eclipse is a plugin framework which, among other things, offers reusable editor components. These editor components may be extended, or reused, by new plugins added to the platform, and these new plugins may in turn offer new components to other plugins. In addition, the platform is a fully fledged integrated development environment with support for version control, debugging, etc. IntelliJ IDEA [Jetb] and NetBeans [Cora] are examples of similar platforms.

In recent years, a number of tool generating systems have emerged which extend the Eclipse Platform. For instance, the IDE Meta-tooling Platform (IMP) [Cha+09] provides a framework on top of Eclipse with semi-automatic generation of textual editors. Using wizards and generation of code skeletons, IMP reduces the amount of work the editor developer needs to do.

Another example is the xText system [EV06; Xte], which generates an editor from an annotated grammar. Notably, the behavior of editor services such as code completion and browsing are inferred from this grammar. However, unless the semantics of a language is very simple, the implementation of name resolution is left to the editor developer. This effort may be reduced as new languages like xSemantics [Xse], supporting specification of semantics, emerge for xText. Examples of systems with a similar approach include the MontiCore system [Grö+08] and the EMFText system [Hei+09].

Additionally, there are systems which both generate editors and support formal semantic specifications. One such example is the Spoofax language workbench [KV10], which uses formal syntax definitions together with strategic term rewriting [Vis01] to define editor services.

2.2 Parser Generators and Error Recovery

The first step in analyzing a text file with code is to parse the file, in order to construct a model suitable for analysis. This model is often a tree and normally referred to as the concrete syntax tree or the abstract syntax tree, depending on

how much syntax information is included in the tree. Implementing this translation from text to model by hand can be tricky and time-consuming, so over the years, many parser generators have been developed to make this development easier and more efficient.

Given a grammar of a language, a parser generator generates the source code for a parser capable of parsing the specified language. Typically, a parser generator supports a specific class of grammars. For instance, ANTLR [PQ95] and JavaCC [Jav] are both parser generators supporting so-called LL grammars [IS66]. In practice, this means that the generated parser is an LL parser which starts at the root of the syntax tree, and then builds it from the top down. An example of a slightly larger class of grammars is the LR grammars [Knu65], and in contrast, an LR parser starts at the leaves, and then builds the syntax tree from the bottom up. CUP [Hud+] and Beaver [Bea] are two examples of LR parser generators.

Generally, a grammar for a programming language is unambiguous, but combinations of grammars may be ambiguous. Some parser generators support ambiguous grammars by generating generalized LR (GLR) [Tom85] parsers, parsing all alternative interpretations of the given input. An interesting extension to GLR is scannerless GLR (SGLR) [Vis97b], supporting grammar composition. By incorporating the scanner into the parser, SGLR can distinguish between tokens in different contexts. That is, a token may be a reserved word in one context, like `enum` in Java, but be a variable in another context, for instance, in an embedding of a database query language like SQL.

The parser is a central part of any textual language tool, including semantic textual editors. However, parsing in an editor is different from parsing in a batch compiler, as code easily becomes erroneous during editing, but an editor should preferably still provide editor services. For these reasons, a parser needs powerful support for error recovery to be practical in editing.

Most parser generators have a strategy for handling of errors, where the simplest is to stop the parser when the first error is encountered, which is not so appropriate for editing. Typically, parser generators offer more support for error recovery. One common approach is to find so-called synchronizing tokens after a parse error, in order to continue parsing. In rare cases, more complicated error recovery algorithms are used which, in addition to synchronizing tokens, consider language scopes [Cha91].

2.3 Attribute Grammar Systems

Typically, the syntax of a language may be specified using a syntax formalism, while the semantics may be specified using a semantic formalism. Attribute grammars (AGs) introduced by Knuth [Knu68] is one such semantic formalism used to specify the static semantics of a language.

To compute semantic information, an AG adds context-sensitive information to an otherwise context-free grammar, by defining so-called *attributes* on grammar

productions. A grammar production represents a part of a language, for example, an `if` statement. The attributes added to a production are given values from functions evaluated in the context of the production. That is, based on the grammar, a syntax tree is built to represent the code being analyzed. In this syntax tree, nodes correspond to productions and each node has its own instances of the attributes defined for the corresponding production.

The functions computing attribute values may use the value of other attributes, resulting in a system where attributes are dependent on each other. These attribute systems can be evaluated in different ways. One strategy is to evaluate all attributes at once. Another strategy is to evaluate attributes as they are needed (on-demand) [Jou84]. In addition, to improve performance in editing, an AG can be evaluated incrementally, that is, attributes affected by a change are re-computed while values of non-affected attributes are reused.

Attribute Grammar Extensions

Originally, AGs [Knu68] included two kinds of attributes: *synthesized* and *inherited*. Synthesized attributes are used to propagate information upwards in the tree, i.e., from the leaves of the tree to the root, while inherited attributes are used to propagate information downwards in the tree, i.e., from the root of tree to the leaves. Since their introduction, several extensions to AGs have been proposed. For example, the following:

- **Circular**: Farrow introduced *circular attribute grammars* [Far86] supporting circular dependencies between attributes, not supported by Knuth's AGs.

- **Higher-order**: Vogt et al. [Vog+89] introduced *higher-order attribute grammars* (HAGs), allowing attributes to have attributed trees as values. HAGs provide a means to handle, for example, multi-pass compilation by step-wise refinement of a parse tree.

  A related technique called *forwarding*, introduced by van Wyk et al. [Wyk+02], allows for the creation of attributed subtrees, in a fashion similar to HAGs, where attribute calls are forwarded to the created subtree.

- **References**: Hedin introduced *reference attribute grammars* (RAGs) [Hed94; Hed00] allowing attributes to have references to other tree nodes as values. RAGs allow for modular and concise descriptions of, for instance, name analysis where use nodes can point directly to their declaration using references. Similar extensions have been presented by Boyland [Boy96; Boy05] and Poetzsch-Heffter [PH97].

- **Collections**: Boyland introduced *collection attributes* [Boy96] which can be used to compute sets of references to a declaration.

- **Rewrites**: Ekman et al. [EH04] introduced demand-driven rewrites for RAGs, which make use of attribute values to rewrite the syntax tree. Rewrites can, for example, be used to transform a `for each` statement in Java to a `for` statement.

The JastAdd System

JastAdd [Jas; EH07b] is one example of a system supporting an extended form of AGs. The JastAdd system supports RAGs with circular attributes, higher-order attributes, collection attributes and rewrites. In addition, the JastAdd system supports aspect-oriented and object-oriented specifications. Examples of systems developed with JastAdd include:

- **JastAddJ**: An extensible Java compiler [EH07a] which has been modularly extended from Java 1.4 to Java 5, and more recently to Java 7 [Öqv12].
- **JModelica**: A compiler [Åke+10] for the Modelica language [Moda], a language for modeling of physical systems. The compiler has been extended modularly to support the Optimica language [Åke08; Hed+11] for optimization of physical systems.
- **McSAF**: A static analysis framework [DH12] for the MATLAB ®[1] language, allowing for analysis of MATLAB programs and experimentation with language extensions to MATLAB.

Kiama [Slo+10] and Silver [Wyk+07] are two other examples of systems supporting AGs. Kiama is a Scala-based library supporting RAGs, while Silver is a standalone system supporting AGs with forwarding.

3 Overview of Contributions

Divided into three groups, the contributions of this dissertation are as follows:

3.1 Specification of Extensible Semantic Services

This dissertation shows how semantic editors with services such as 'find declaration', 'find references', 'name completion' and 'error feedback' can be specified modularly as compiler extensions (Paper I). Two editors are presented as examples: one for a small object-oriented language called PicoJava, and one for the JastAdd specification language, an extension of the Java language supporting RAGs. This latter editor shows how an editor extension may be added to a compiler which already has been extended in several steps (Figure 1 and Figure 2).

[1] MATLAB is a registered trademark of Mathworks, `mathworks.com/products/matlab`.

In this case, the underlying compiler is the JastAddJ compiler extended with support for inter-type declarations for aspect-oriented programming and support for RAGs.

In addition, this dissertation shows how 'flow analysis'-based editor services such as dead assignment detection can be specified modularly as compiler extensions, and extended to cater for language changes (Paper II). Three flow analysis modules are presented which specify control-flow, dataflow and dead-assignment analysis for Java, and the precision and performance of these modules is found to be on par with well-known analysis tools.

The editor construction framework, build using Eclipse and JastAdd, has been used in several master theses [Mat09; Mos09; Nil10], and is used in the JModelica IDE, providing an editor for the Modelica language [Modb]. The control-flow module is used as a central part of the JastAdd Refactoring Tools [Sch; Sch+08; Sch+09], performing refactoring of Java code.

3.2 Efficient Computation of RAGs

This dissertation introduces a profiler-based configuration approach for efficient computation of RAGs, and shows how the use of this method can significantly speed-up RAG computations (Paper III). By compiling a set of sample programs, where information of how attributes are used is collected, a cache configuration can be generated which gives better performance than caching of all attributes. The method is implemented as a part of the JastAdd system and has been evaluated on the JastAddJ compiler, for which speed-ups of 20% on average, compared to full caching, have been measured.

In addition, this dissertation introduces how to evaluate RAGs incrementally with the use of dynamic dependency tracking (Paper IV). This tracking accounts for dynamic dependencies caused by reference attributes, something not accounted for by earlier incremental evaluators for AGs. As a foundation for the approach, a notion of consistency for RAG attributions and an algorithm for maintaining consistency after edits is presented (Paper IV). To limit the effect of change, an optimization is introduced which compares attribute values in order to stop change propagation (Paper V).

This dissertation also shows that incremental RAG evaluators can provide better performance for editing than traditional incremental AG evaluators (Paper V). As an example, a comparison of incremental name resolution is presented, where the same name resolution problem is solved using an incremental RAG evaluator and an incremental AG evaluator. By counting the amount of work needed to recompute semantic information after changes, it is shown that the RAG approach significantly reduces the amount of work that needs to be done.

3.3 Textual Error Recovery for Editing

This dissertation presents a technique for light-weight textual scope recovery (Paper VI), which uses secondary notation such as layout information to recover broken scopes. With this information, the algorithm first identifies broken scopes and then recovers these scopes by identifying layout-based patterns, described in a so-called bridge grammar. The technique has been implemented for Java and has been shown to improve the error recovery quality of Java parsers generated with state-of-the-art parser generators.

In addition, this dissertation shows how this layout-sensitive recovery approach has contributed to the addition of error recovery support to SGLR (scanner-less generalized LR parsing) (Paper VII). The error recovery in SGLR automatically derives recovery rules from grammars, and takes layout information into consideration to efficiently provide natural recovery suggestions to users. With this addition of error recovery, SGLR performs on par with the hand-crafted parser of the Eclipse JDT, in terms of error recovery quality, while also providing the same recovery support to composed languages such as Java-SQL. The result of this work is implemented in the JSGLR [Jsg] parser generator and used in the Spoofax language workbench [KV10].

4 Conclusions and Future Work

An attempt at showing the potential of generative approaches to editor construction has been made in this dissertation. The focus has been on using RAGs for specification of extensible semantic editor services, and how such services can be added as extensions to an existing compiler, in addition to being extended in themselves. The examples presented in this dissertation suggest that RAGs can be used to specify editor services on par with the services offered by the state of the art, and potentially more.

In response to concerns regarding the performance of RAGs, a profiler-based method, which has been shown to significantly speed-up RAG computations, has been presented. To specifically address the performance needs of editing, a technique for incremental evaluation of RAGs has been presented, together with an evaluation showing how the technique improves performance compared to traditional AGs. These results suggest that RAG-based editor services can reach the performance level needed for editing.

A second concern with the approach was robustness and the capability of providing services despite erroneous textual input. In response to this concern, an error recovery approach has been presented which has been shown to improve error recovery quality for generated parsers. In addition, it has been shown how this layout-based approach together with SGLR provides error recovery quality on par with a hand-crafted state-of-the-art parser. These results suggest that RAG-based editor services can be provided also in the case of erroneous textual input.

The work presented in this dissertation opens several interesting directions for future work. With regard to specification of editor services, one such direction is to further raise the specification level. Recent work by Konat et al. [Kon+12] presents a language targeted at defining name analysis. This way of considering different semantic specification domains could likely be explored further, for instance, for editor construction.

With incremental evaluation in mind, there are challenges in generating evaluators for AG extensions. For instance, demand-driven tree rewrites [EH04] present interesting problems, which Bürger [Bür12] has started to explore. In addition, non-terminal attributes [Vog+89], used extensively in JModelica [Åke+10], also present challenges for incremental evaluation.

Another direction for incremental evaluation, is to reduce the amount of dependency information that is needed, as this information increases the size of the RAG-based models. The challenge is to reduce this information while still providing responsive editor services.

For layout-based error recovery, it would be interesting to experiment with languages where indentation is not secondary. Recent work by Erdweg et al. [Erd+12], explores the combination of layout-sensitive languages and generalized parsing. Similar experiments could be done using the layout-sensitive recovery approach presented in this dissertation.

In summary, editor generation is an area with a rich set of problems. This dissertation has presented solutions to several of them, and helped to narrow the gap between generative approaches and how state-of-the-art editors are constructed today.

References

[Åke08] Johan Åkesson. "Optimica – an extension of Modelica supporting dynamic optimization". In: *In 6th International Modelica Conference*. Modelica Association, 2008.

[Åke+10] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. "Implementation of a Modelica compiler using JastAdd attribute grammars". In: *Science of Computer Programming* 75.1-2 (2010), pp. 21–38.

[Bea] *Beaver*. `http://beaver.sourceforge.net`.

[Boy96] John Tang Boyland. "Descriptional Composition of Compiler Components". PhD thesis. University of California at Berkeley, 1996.

[Boy05] John Tang Boyland. "Remote attribute grammars". In: *Journal of the ACM* 52.4 (2005), pp. 627–687.

[Bür12] Christoff Bürger. *RACR: A* Scheme *Library for Reference Attribute Grammar Controlled Rewriting*. Tech. rep. TUD-FI12-09, ISSN 1430-211X. `https://code.google.com/p/racr/`. Technische Universität Dresden, 2012.

[Cha91] Philippe Charles. "A practical method for constructing efficient LALR(k) parsers with automatic error recovery". PhD thesis. New York, NY, USA: New York University, 1991.

[Cha+09] Philippe Charles et al. "Accelerating the creation of customized, language-Specific IDEs in Eclipse". In: *OOPSLA*. Ed. by Shail Arora and Gary T. Leavens. ACM, 2009, pp. 191–206.

[Cora] Oracle Corporation. *NetBeans IDE*. `http://netbeans.org`.

[Corb] Oracle Corporation. *NetBeans support for Project Lambda (JSR 335)*. `http://wiki.netbeans.org/Java_EditorJDK7`.

[DM01] James R. Dabrowski and Ethan V. Munson. "Is 100 Milliseconds Too Fast?" In: *CHI '01 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '01. Seattle, Washington: ACM, 2001, pp. 317–318.

[DH12] Jesse Doherty and Laurie J. Hendren. "McSAF: A Static Analysis Framework for MATLAB". In: *ECOOP*. Ed. by James Noble. Vol. 7313. Lecture Notes in Computer Science. Springer, 2012, pp. 132–155.

[EV06] Sven Efftinge and Markus Völter. "oAW xText: a framework for textual DSLs". In: *Eclipse Summit Europe, Eclipse Modeling Symposium*. Esslingen, Germany, 2006.

[EH04] Torbjörn Ekman and Görel Hedin. "Rewritable Reference Attributed Grammars". In: *ECOOP*. Ed. by Martin Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 144–169.

[EH07a] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler". In: *OOPSLA*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[EH07b] Torbjörn Ekman and Görel Hedin. "The JastAdd system - modular extensible compiler construction". In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[Erd+12] Sebastian Erdweg et al. "Layout-sensitive Generalized Parsing". In: *SLE*. Ed. by Krzysztof Czarnecki and Görel Hedin. Lecture Notes in Computer Science. Springer, 2012.

[Far86] Rodney Farrow. "Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars". In: *SIGPLAN Symposium on Compiler Construction*. Ed. by Richard L. Wexelblat. ACM, 1986, pp. 85–98.

[Foua] Eclipse Foundation. *The Eclipse Java Development Tools*. http://www.eclipse.org/jdt.

[Foub] Eclipse Foundation. *The Eclipse Platform*. http://www.eclipse.org.

[Grö+08] Hans Grönniger et al. "MontiCore: a framework for the development of textual domain specific languages". In: *ICSE*. 2008, pp. 925–926.

[Hed94] Görel Hedin. "An Overview of Door Attribute Grammars". In: *CC*. Ed. by Peter Fritzson. Vol. 786. Lecture Notes in Computer Science. Springer, 1994, pp. 31–51.

[Hed00] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[Hed+11] Görel Hedin, Johan Åkesson, and Torbjörn Ekman. "Extending Languages by Leveraging Compilers: From Modelica to Optimica". In: *IEEE Software* 28.3 (2011), pp. 68–74.

[Hei+09] Florian Heidenreich et al. "Derivation and Refinement of Textual Syntax for Models". In: *ECMDA-FA*. 2009, pp. 114–129.

[HW09] Daqing Hou and Yuejiao Wang. "An empirical analysis of the evolution of user-visible features in an integrated development environment". In: *CASCON*. Ed. by Patrick Martin, Anatol W. Kark, and Darlene A. Stewart. ACM, 2009, pp. 122–135.

[Hud+] Scott Hudson, Frank Flannery, and C. Scott Ananian. *CUP: LALR Parser Generator in Java*. http://www2.cs.tum.edu/projects/cup.

[IS66] Philip M. Lewis II and Richard Edwin Stearns. "Syntax Directed Transduction". In: *SWAT (FOCS)*. IEEE Computer Society, 1966, pp. 21–35.

[Jas] *JastAdd*. http://jastadd.org.

[Jav] *JavaCC*. http://java.net/projects/javacc.

[Jeta] JetBrains. *IntelliJ IDEA support for Project Lambda (JSR 335)*. http://confluence.jetbrains.net/display/IDEADEV/IDEA+12+EAP.

[Jetb] JetBrains. *IntelliJ IDEA*. http://www.jetbrains.com/idea.

[Jou84] Martin Jourdan. "An Optimal-time Recursive Evaluator for Attribute Grammars". In: *Symposium on Programming*. Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 167–178.

[Jsg] *JSGLR*. http://strategoxt.org/Stratego/JSGLR.

[KV10] Lennart C. L. Kats and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs". In: *OOPSLA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 444–463.

[Knu65] Donald E. Knuth. "On the Translation of Languages from Left to Right". In: *Information and Control* 8.6 (1965), pp. 607–639.

[Knu68] Donald E. Knuth. "Semantics of context-free languages". In: *Journal Theory of Computing Systems* 2.2 (1968), pp. 127–145.

[Kon+12] Gabriël D. P. Konat et al. "The spoofax name binding language". In: *SPLASH*. Ed. by Gary T. Leavens. ACM, 2012, pp. 79–80.

[Mat09] Jesper Mattsson. "The JModelica IDE: Developing an IDE Reusing a JastAdd Compiler". MA thesis. Lund, Sweden: Lund University, 2009.

[Mil68] Robert B. Miller. "Response time in man-computer conversational transactions". In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. AFIPS '68 (Fall, part I). San Francisco, California: ACM, 1968, pp. 267–277.

[Moda] ModelicaAssociation. *The Modelica Language*. `http://www.modelica.org`.

[Modb] Modelon. *The JModelica.org Project*. `http://www.jmodelica.org`.

[Mos09] Erik Mossberg. "Inspector – Tool for Interactive Language Development". MA thesis. Lund, Sweden: Lund University, 2009.

[Mur+06] Gail C. Murphy, Mik Kersten, and Leah Findlater. "How are Java software developers using the Elipse IDE?" In: *Software, IEEE* 23.4 (2006), pp. 76–83.

[Nil10] Philip Nilsson. "Semantic editing compiler extensions using JastAdd". MA thesis. Lund, Sweden: Lund University, 2010.

[Opea] Open JDK. *Project Lambda (JSR 335)*. `http://openjdk.java.net/projects/lambda`.

[Opeb] OpenJDK. *The javac compiler*. `http://openjdk.java.net/groups/compiler`.

[Öqv12] Jesper Öqvist. "Implementation of Java 7 Features in an Extensible Compiler". MA thesis. Lund, Sweden: Lund University, 2012.

[PQ95] Terence John Parr and Russell W. Quong. "ANTLR: A Predicated-*LL(k)* Parser Generator". In: *Softw., Pract. Exper.* 25.7 (1995), pp. 789–810.

[PH97] Arnd Poetzsch-Heffter. “Prototyping Realistic Programming Languages Based on Formal Specifications”. In: *Acta Informatica* 34.10 (1997), pp. 737–772.

[Sch] Max Schäfer. *JastAdd Refactoring Tool*. `http://code.google.com/p/jrrt`.

[Sch+08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. “Sound and extensible renaming for Java”. In: *OOPSLA*. Ed. by Gail E. Harris. ACM, 2008, pp. 277–294.

[Sch+09] Max Schäfer et al. “Stepping Stones over the Refactoring Rubicon”. In: *ECOOP*. Ed. by Sophia Drossopoulou. Vol. 5653. Lecture Notes in Computer Science. Springer, 2009, pp. 369–393.

[Slo+10] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. “A Pure Object-Oriented Embedding of Attribute Grammars”. In: *Electr. Notes Theor. Comput. Sci.* 253.7 (2010), pp. 205–219.

[Tom85] Masaru Tomita. “An efficient context-free parsing algorithm for natural languages and its applications”. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 1985.

[Vis97b] Eelco Visser. *Scannerless Generalized-LR Parsing*. Tech. rep. P9707. Programming Research Group, University of Amsterdam, 1997.

[Vis01] Eelco Visser. “Stratego: A Language for Program Transformation Based on Rewriting Strategies”. In: *RTA*. Ed. by Aart Middeldorp. Vol. 2051. Lecture Notes in Computer Science. Springer, 2001, pp. 357–362.

[Vog+89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *PLDI*. Ed. by Richard L. Wexelblat. ACM Press, 1989, pp. 131–145.

[Wyk+02] Eric Van Wyk et al. “Forwarding in Attribute Grammars for Modular Language Design”. In: *CC*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. London, UK: Springer-Verlag, 2002, pp. 128–142.

[Wyk+07] Eric Van Wyk et al. “Attribute Grammar-Based Language Extensions for Java”. In: *ECOOP*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 575–599.

[Xse] *XSemantics*. `http://xsemantics.sourceforge.net`.

[Xte] *xText*. `http://www.eclipse.org/Xtext`.

INCLUDED PAPERS

PAPER I

BUILDING SEMANTIC EDITORS USING JASTADD

Abstract

A semantic editor, providing services like completion and code browsing, can help users to quickly develop high-quality source code. However, a lot of languages still lack semantic editor support due to the difficulty and costs of development. Tool generation and reuse can greatly alleviate this development task. Specifically, tool generation from a formal specification, such as reference attribute grammars (RAGs), can increase development speed by reusing existing specifications. In this tool demonstration we demonstrate how semantic editors can be built with the aid of JastAdd, a meta-compilation tool based on RAGs. We demonstrate two editors built this way. One for a small object-oriented language, PicoJava, and one for the JastAdd specification language itself.

1 Introduction

Editors providing users with language-specific services during development can help users to rapidly produce high-quality code. Editors like these can be found in, for example, Eclipse[1], IntelliJ IDEA[2] or NetBeans[3]. Typical examples of ser-

[1] `http://www.eclipse.org`
[2] `http://www.jetbrains.com/idea`
[3] `http://netbeans.org`

Emma Söderberg and Görel Hedin.
In proceedings of the *11th Workshop on Language Descriptions, Tools and Applications (LDTA'11)*, ACM, Saarbrücken, Germany, April 2011.

vices found in these editors [HW09] are *name completion*, which helps users to quickly insert code appropriate for a certain context, and browsing services like *find declaration* or *find references*, which help users to quickly get an overview of the structure of a program. We call services like these, which take the semantics of a language into account, *semantic services*, and we call editors offering semantic services *semantic editors*.

Many language communities do not have the resources to develop editors like the above mentioned, which are all hand-coded and developed over several years. Several approaches with the goal of decreasing the costs of editor development are being explored in systems like xText [EV06], IMP [Cha+07], Spoofax [KV10] and MontiCore [Grö+08]. Most approaches involve some form of tool generation, formal specification of at least some parts, and in some cases reuse.

We are exploring the use of references attribute grammars (RAGs) [Hed00] for semantic editor generation. RAGs are an extension of attribute grammars (AGs) [Knu68] supporting references as values. These properties in RAGs make it simple to define structures needed in compiler construction, for example, declaration-use relations. Examples systems supporting RAGs include JastAdd [EH07b], Kiama [Slo+10] and Silver [Wyk+10]. The goal with using RAGs in semantic editor development is to easily construct semantic services by reusing the specification of an existing RAG-based compiler. If large parts of a compiler specification can be reused in an editor, development time can be decreased. Other systems have also used attribute grammars in editor development, an early example is the Synthesizer Generator [RT84]. A difference from this approach is that we use RAGs which have properties beneficial to many semantic services.

In this paper we present an architecture for editors built using JastAdd RAGs, and we show two examples of editors developed using RAGs. One for a small object-oriented language, PicoJava, and one for the JastAdd language itself.

The rest of this paper starts with an overview of the architecture and some background in Section 2. This is followed by a walk-through of two example editors in Section 3. Section 4 discusses the generality and limitations of the approach and Section 5 sums up the paper with some concluding remarks.

2 Overview and Background

The idea with our approach is to reuse an existing RAG-based compiler specification. Ideally, to develop an editor we would like to *just* specify the behavior of our semantic services as extensions to an existing compiler and then get a running editor. With this in mind, we have developed a RAG-based editor framework where we provide generic editor behavior, leaving the specific language behavior to the editor implementor.

To build this framework we have used the JastAdd and Eclipse systems. JastAdd is a system supporting RAGs, as described in the introduction, and Eclipse is an

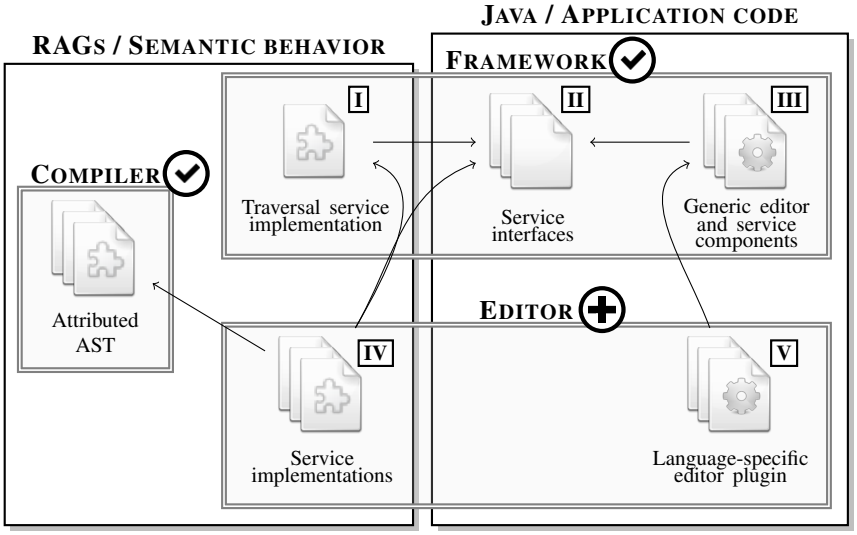


Figure 1: Overview of the semantic editing framework The right side shows Java modules and the left side shows RAGs modules. Java modules working as Eclipse plugins are marked with a cogwheel. Modules are grouped into parts representing the compiler, the editing framework and the editor. Arrows show dependencies between RAGs and Java modules. Parts marked as checked are reused and parts marked with a plus are added.

extensible plugin-based application framework providing basic editor components. The JastAdd system is used to generate attributed program representations in Java, or *abstract syntax trees* (AST), used as the semantic model in an editor, and the Eclipse system is used as the runtime platform of an editor. The Eclipse system supports Java code, while the JastAdd system generates Java code. The code, or specifications, in the two systems are connected via Java interfaces. Figure 1 gives an overview of how the generic framework connects to compiler and editor modules. The figure separates modules into JastAdd modules and Java modules. In addition, modules are grouped into compiler, framework and editor parts. To develop an editor, a developer would have to add the editor part. In more detail, the different modules in the figure contain the following:

COMPILER: The compiler contains an AST with reusable attributes, for example, references from identifier uses to their declarations, and types of expressions.

FRAMEWORK: The framework provides a generic editor with the following modules (see Figure 2 for module sizes):

I: A generic implementation of an AST traversal interface.

II: Service interfaces that describe which information that needs to be provided for a certain language in order for a service to be supported. Currently supported interfaces include outline, browsing, type hierarchy view, name completion and error feedback.

III: The generic editor contains service components that interact with the AST via the service interfaces.

The framework can easily be extended by an editor with additional service interfaces (II) and corresponding service component implementations (III).

EDITOR: The editor specification contains language-specific behavior.

IV: The service implementations provide language-specific behavior by defining attributes that implement the service interfaces, reusing attributes in the compiler.

V: The language-specific editor plugin provides concrete classes for the generic editor, as well as non-semantic services like syntax highlighting. This code is of simple boiler-plate character.

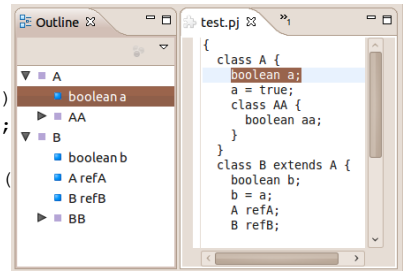
As example of a semantic service implementation, we will consider the *outline service* for the PicoJava language. An outline visualizes the contents of a file as a tree in a view next to the editor. Figure 3 shows the service interface for the outline service, its implementation using RAGs, and a screenshot of the outline view. The service interface includes methods needed to construct an outline tree (lines $2-4$), along with a method selecting which node types to display (5).

	I	II	III	Total
FRAMEWORK	4 LOC	137 LOC	1.7 kLOC	1.8 kLOC

Figure 2: Summary of modules in the framework

Outline interface (II):

```
1  interface IOutlineNode {
2      public List outlineChildren();
3      public String outlineLabel();
4      public Image outlineImage();
5      public boolean showInOutline(
6  }
```



Outline implementation (IV):

```
1  aspect Outline {
2      ASTNode implements IOutlineNode;
3      syn List ASTNode.outlineChildren()  = getChildren();
4      syn String ASTNode.outlineLabel()   = getClass().getName();
5      syn Image ASTNode.outlineImage()    = null;
6      syn boolean ASTNode.showInOutline() = true;
7      eq ClassDecl.outlineLabel() = getName();
8      eq VarDecl.outlineLabel()   = type().getName() + " " + getName();
9  }
```

Figure 3: The outline service The upper left code shows the service interface, the lower left code shows its implementation for PicoJava and the upper right picture shows the PicoJava editor with an outline view.

The language-specific behavior in this particular example should be to display all nodes in the outline, but use different labels for some of the nodes. To do this we start by letting the superclass of all AST nodes (`ASTNode`) implement the outline service interface (2). Next, we define an attribute for each method for the node type `ASTNode` ($3-6$). On line 3 we compute outline children and on line 4 we let the labels be the name of the node types, reusing traversal methods in the compiler (`getChildren`, `getClass`, `getName`). We choose no default icon on line 5 and choose to display all nodes on line 6. On lines 7 and 8 we specialize the behavior for class and variable declarations. For classes we let the label be the name of the class (`getName`). For variable declarations, we let the label be the name of the variable along with the name of its type. Here, we reuse an attribute in the compiler (`type`) computing the type of a variable.

		EDITOR		
Language	**COMPILER**	Services (**IV**)	Editor (**V**)	**Total**
JastAdd	29.2 kLOC	1.1 kLOC	3.2 kLOC	4.3 kLOC
PicoJava	210 LOC	420 LOC	180 LOC	600 LOC
Size Relation	x139.0	x2.6	x17.8	x7.2

Figure 4: Summary of compiler and editor modules The columns show the sizes of compiler and editor modules. The bottom row show the relation in size between the module in the same column (JastAdd/PicoJava).

3 Example Editors

To demonstrate how we use the generic framework presented in the previous section we will show how we specify semantic services for two editors. One for a small object-oriented language, PicoJava, and one for the JastAdd language. We selected these two languages as examples due to their size. The PicoJava is very small with only 15 parse productions, while the JastAdd language is much larger with 253 parse productions. The JastAdd language includes full Java, inter-type declarations found in AspectJ [Kic+01], and RAG-specific constructs like attributes and equations. Figure 4 shows the sizes of compiler and editor modules for the two example languages. It is clear from the comparison in the figure that the JastAdd compiler specification is many times larger than the PicoJava compiler specification. The difference between the total editor sizes are, in relation to the compiler specification difference, quite small. The difference between the editor modules is largest for the editor modules (V). This is because the JastAdd editor adds non-semantic services like syntax highlighting, while the PicoJava editor does not.

3.1 The Browsing Service

To give a detailed example of how a more advanced service is added to one of our example editors, we will show how we add browsing service to the JastAdd editor that supports *find declaration* and *find references*. For the JastAdd editor we want the find declaration service to jump to the declaration of a use, and we want the find references service to list all references to a declaration. Figure 5 shows the browsing interface and its implementation, and Figure 6 shows a screenshot of how browsing works in the editor. The first two methods in the interface (line $2-3$) connects a browsing node to its declaration and its references. The last two interface methods describe how to visualize browsing nodes when they appear as search results of the find reference service. In the implementation we select two types as browsing nodes, `TypeDecl` and `TypeAccess`, on line 2 and 8. These nodes represent type declarations and uses.

Browsing interface:

```
1  interface IBrowsingNode {
2      public IBrowsingNode browsingDecl();
3      public List browsingRefs();
4      public String browsingLabel();
5      public Image browsingImage();
6  }
```

Browsing implementation:

```
1  aspect Browsing {
2      TypeDecl implements IBrowsingNode;
3      syn IBrowsingNode TypeDecl.browsingDecl() = this;
4      syn List TypeDecl.browsingRefs() = references();
5      syn String TypeDecl.browsingLabel() = getID();
6      syn Image TypeDecl.browsingImage() = ..
7
8      TypeAccess implements IBrowsingNode;
9      syn IBrowsingNode TypeAccess.browsingDecl() = type();
10     syn Collection<IBrowsingNode> TypeAccess.browsingRefs() =
11         type().browsingRefs();
12     syn String TypeAccess.browsingLabel() = getID();
13     syn Image TypeAccess.browsingImage() = ..
14
15     coll HashSet TypeDecl.references() [new HashSet()] with add;
16     TypeAccess contributes this to TypeDecl.references() for type();
17 }
```

Figure 5: The browsing services The upper code section shows the browsing interface, and the lower code listing shows an abbrevated implementation for the JastAdd language.

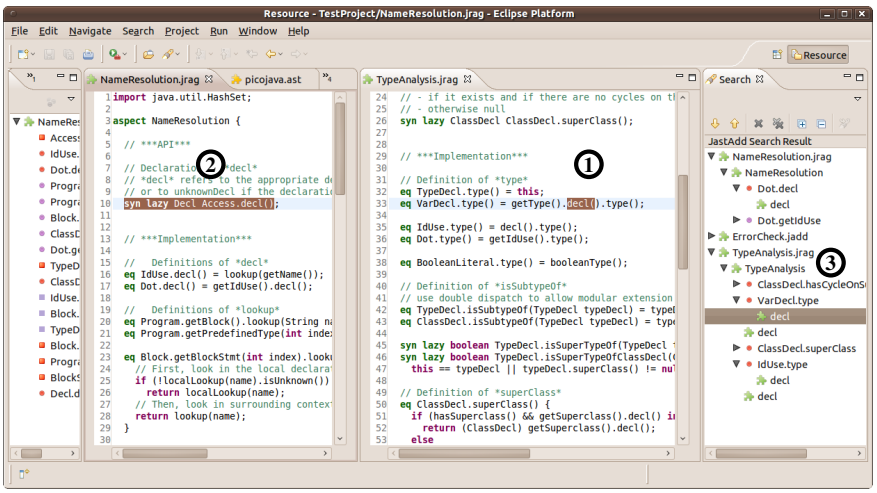


Figure 6: Browsing The find declaration service is activated for the marked method call at (1), and as a result the declaration is marked in the left editor at (2). The find reference service is activated for the declaration at (2), which results in the list of references being displayed in the search page at (3). The marked reference at (3) corresponds to the original method access at (1).

For the type declaration node, we define `this` to be the result of a find declaration call (3) and the result of an attribute `references` to be the result of a find declaration call. This `references` attribute is not previously defined in the compiler and is therefore defined in the implementation module. It is defined on line $15-16$ as a so called *collection attribute* (`coll`) [Mag+09]. Collection attributes collect a set of references from a set of contributors, defined with the `contribute` keyword. During the evaluation of this attribute, which is done on demand, `TypeAccess` nodes will add themselves to the `reference` collection of their type declaration (`TypeDecl`). The connection to the `TypeDecl` node is given by an attribute `type`, defined and used during type analysis in the Java compiler. For the `TypeAccess`, we let the declaration be the result of the `type` attribute (9) and we let the references be the references of the declaration ($10-11$). For both node types, we use their identifier (`getID`) as label in the search result of a find references call.

3.2 The Completion Service

Another example of a service supported by the generic framework is the completion service. In the JastAdd editor we want this service to provide valid name

```
interface ICompletionNode {
    public Collection<ICompletionProposal>
        completionProposals(int offset);
}

aspect Completion {
    Access implements ICompletionNode;
    // ...
}

// abstract grammar extension
ErrorStmt : Stmt ::= ErrorAccess;
ErrorAccess : Access:

// concrete grammar extension
Stmt statement = error SEMICOLON
{: return new ErrorStmt(new ErrorAccess
```

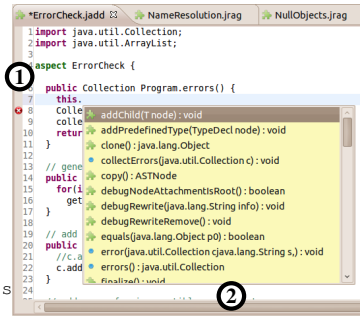


Figure 7: Completion Completion is activated at (1), a popup of alternatives are displayed. The completion causes a syntactic error marked at (1). The list of completions show valid completions for `this`, including the encolsing method `errors` at (2).

completion suggestions sensitive to the context in which we activate the service. Figure 7 shows the completion interface and implementation along with a screenshot of the service activated in the JastAdd editor. The ICompletionProposal type in the return type of the method in the service interface is used by Eclipse to describe completion proposals (2). In the language-specific behavior we need to choose a node type to offer the service. For this service, we choose the `Access` node which represent uses (2).

Completion and syntax errors

A completion service will typically be triggered during editing where the code being edited is in an syntactically erroneous state. This calls for robust parsing. Robust parsing can be accomplished in different ways. In LALR parsing so called *error productions* can be inserted into the concrete grammar allowing the parser to produce error nodes at some positions with syntax errors. For example, an error statement node can be inserted at positions where a statement is expected but no parse match can be found.

The JastAdd systems allows us to both modularly extend the abstract grammar and the concrete grammar. We can, for example, extend the abstract grammar with a new statement error node as a subclass to the statement node, and we can extend the concrete grammar with a new parse production, constructing error statement nodes for cases where no other statement node match. In addition, if we let this error node include an error access node, inheriting from the access class, we can

Service	**Size**:JastAdd	**Size**:PicoJava	**Size relation**:JastAdd/PicoJava
Outline	96	37	x2.6
Browsing	101	24	x4.2
Completion	128	74	x1.7
Type Hierarchy	80	21	x3.8
Error Feedback	86	104	x0.8

Figure 8: Summary of service modules The sizes of the service modules in the PicoJava and JastAdd editors given in LOC, and the size relation of service modules between the two languages with regard to size.

collect valid completion suggestions from these error nodes. Figure 7 shows completion via an error node at a syntax error, along with the node interface, aspect implementation, abstract grammar extension and concrete grammar extension.

3.3 Summary of Services

The sizes of the service modules in the two example editors for PicoJava and JastAdd are summarized in Figure 8. One size that sticks out is the size of the error feedback module for PicoJava which is larger than the corresponding module for JastAdd. The larger size is due to the need to add error handling code which was not present in the PicoJava compiler specification. For cases like this one, where there is a gap between the compiler and editor specifications, the service module will increase in size.

Considering the size of the JastAdd service modules compared to their corresponding PicoJava service module, we see that there is not a huge difference. On average the JastAdd service module is about twice as large as the same PicoJava service module. In relation to the difference in size between the compiler specifications, shown in Figure 4, where the JastAdd compiler is more than 100 times larger than the PicoJava compiler, this difference is very small.

4 Generality and Limitations

The editor framework makes some assumptions about the language and editors that it will support. For example, for browsing it assumes that there are nodes corresponding to declarations and that there are nodes corresponding to uses. For some languages this might not be the case, but for most languages with need for this kind of editor we assume that this is a common design. Further, the editor implementation requires an existing RAG-based compiler implementation.

The editor framework can handle arbitrary semantic services due to the generality of attribute grammars. Currently, the framework supports a few common semantic services, but the framework can be extended to support others. There

is ongoing work on adding refactoring support. The framework focuses on semantic services and has no specific support for purely lexical services like syntax-highlighting, debugging services, versioning services etc. These services have to be implemented in the conventional way.

The architecture of the framework is general and should work for other RAG-based systems. The framework does not yet support incremental attribute updating, which means that services may take time to be recomputed during editing. Incremental evaluation of RAGs is non-trival due to dynamic dependencies, preventing a static scheduling of evaluation dependencies. A solution to incremental evaluation of RAGs is ongoing work.

5 Conclusions

In this tool demonstration paper we have briefly presented an editor framework along with two example editors for JastAdd and PicoJava. We have shown how we use the editor framework to concisely add semantic services to these example editors. The size of the service modules for these editors suggest that the effort of adding a service is reasonably small. Particularly, when the language is large but the service modules remain small. We see a number of possible ways to continue the work presented in this paper. The editor framework could, for example, provide more default service behavior and additional common semantic services, like refactorings. Also, the framework which currently has a focus on text editors and read-only tree views could be extended to support editable tree or graph views. In supporting editable tree views, there is a need to support structural updating which relates to incremental updating. Incremental evaluation of RAGs is an open problem which we are working on. As a part of this work we want to evaluate the performance of RAG-based editors.

References

[Cha+07] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton, Jr. “IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse”. In: *ASE*. Ed. by R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer. ACM, 2007, pp. 485–488.

[EV06] Sven Efftinge and Markus Völter. “oAW xText: a framework for textual DSLs”. In: *Eclipse Summit Europe, Eclipse Modeling Symposium*. Esslingen, Germany, 2006.

[EH07b] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[Grö+08] Hans Grönniger et al. “MontiCore: a framework for the development of textual domain specific languages”. In: *ICSE*. 2008, pp. 925–926.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[HW09] Daqing Hou and Yuejiao Wang. “An empirical analysis of the evolution of user-visible features in an integrated development environment”. In: *CASCON*. Ed. by Patrick Martin, Anatol W. Kark, and Darlene A. Stewart. ACM, 2009, pp. 122–135.

[KV10] Lennart C. L. Kats and Eelco Visser. “The spoofax language workbench: rules for declarative specification of languages and IDEs”. In: *OOPSLA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 444–463.

[Kic+01] Gregor Kiczales et al. “An Overview of AspectJ”. In: *ECOOP*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 327–353.

[Knu68] Donald E. Knuth. “Semantics of context-free languages”. In: *Journal Theory of Computing Systems* 2.2 (1968), pp. 127–145.

[Mag+09] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. “Demand-driven evaluation of collection attributes”. In: *Automated Software Engineering* 16.2 (2009), pp. 291–322.

[RT84] Thomas W. Reps and Tim Teitelbaum. “The Synthesizer Generator”. In: *Software Development Environments (SDE)*. Ed. by William E. Riddle and Peter B. Henderson. ACM, 1984, pp. 42–48.

[Slo+10] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. “A Pure Object-Oriented Embedding of Attribute Grammars”. In: *Electr. Notes Theor. Comput. Sci.* 253.7 (2010), pp. 205–219.

[Wyk+10] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming* 75.1-2 (2010), pp. 39–54.

PAPER II

EXTENSIBLE INTRAPROCEDURAL FLOW ANALYSIS AT THE ABSTRACT SYNTAX TREE LEVEL

Abstract

We have developed a new approach for implementing precise intraprocedural control-flow and dataflow analysis at the abstract syntax tree level. Our approach is declarative, making use of reference attribute grammars augmented with circular attributes and collection attributes. This results in concise executable specifications of the analyses, allowing extensions both to the language and with further source code analyses. To evaluate the new approach, we have implemented control flow, dataflow and dead assignment analysis for Java, by extending the JastAdd Extensible Java Compiler. We have compared our results to several well-known analysis frameworks and tools, using a set of Java programs as benchmarks. These results show that our approach performs well concerning both efficiency and preciseness.

1 Introduction

Control-flow and dataflow analysis are key elements in many static analyses, and useful for a variety of purposes, e.g., code optimization, refactoring, enforcing

Emma Söderberg, Görel Hedin, Torbjörn Ekman, and Eva Magnusson.
Science of Computer Programming, 2012, Elsevier B.V. In press.

coding conventions, bug detection, and metrics. Often, such analyses are carried out on a normalized intermediate code representation, rather than on the abstract syntax tree (AST). This simplifies the computations by not having to deal with the full source language. However, doing these analyses directly at the AST level can be beneficial, since the high-level abstractions are not compiled away during the translation to intermediate code. This is particularly important for tools that are integrated in interactive development environments, such as refactoring tools and tools supporting bug detection and coding convention violations.

In this paper, we present a new approach for computing intra-procedural control-flow and dataflow at the AST level. Our approach is declarative, making use of attribute grammars. Advantages include compact specification and modular support for language extensions, while giving sufficient performance for practical use.

To make the approach work, we rely on a number of extensions to Knuth's original attribute grammars [Knu68]: *Reference attributes* [Hed00] allow the control-flow edges to be represented as references between nodes in the AST. *Higher-order attributes* [Vog+89] are used for reifying entry and exit nodes in the control-flow graph as objects in the AST. *Circular attributes* [Far86; MH07] are used for writing down mutually recursive equations for dataflow as attributes, automatically solved through fixed-point iteration. Finally, *collection attributes* [Boy05; Mag+09], enable the simple specification of reverse relations, for example, computing the set of predecessors, given the set of successors. These mechanisms are all supported in the JastAdd system [EH07b], which we have used to implement our approach.

As a case study, we have implemented control-flow graphs and dataflow analysis for Java by extending JastAddJ (the JastAdd Extensible Java Compiler) [EH07a]. The control flow graph is precise: it is implemented at the expression level and covers non-trivial control flow including Java exception handling, taking exception types into account, and short-circuited boolean expressions. For dataflow, we have implemented both liveness analysis and reaching definition analysis. As an example of a tool-oriented analysis, we have implemented a detector of dead assignments to local variables.

The implementation is modular and extensible. Similar to the internal modularization of JastAddJ [EH07a], each module can be viewed as an object-oriented framework, with a client API representing the result of the analysis, and an extension API for the attributes that need to be defined by a language extension module. In many cases, new language features can reuse the existing analyses as they are, but for language constructs affecting control-flow, rules need to be added. We exemplify this by considering the effect on the analyses when extending Java 1.4 to Java 5.

These are the main contributions of this paper:

- We present a new approach to implementing precise control-flow graphs at the AST level, using reference attribute grammars. An attribute framework for control-flow graphs is presented that allows the modular addition of language constructs, classified into non-directing, internal flow, and abruptly

completing constructs. We furthermore provide attribute grammar solutions for specifying precise control flow of exceptions and short-circuiting of boolean expressions.

- We present how the control-flow framework can be modularly extended with liveness analysis and reaching definition analysis. These dataflow analyses are specified using circular attributes, resulting in declarative implementations very similar to textbook definitions.
- We have implemented control flow graphs and dataflow analysis using our approach for full Java 1.4 and with a modular extension to support Java 5. The implementation is available at the JastAdd site [Jas].
- We report performance and preciseness results of our approach by comparing it to three well known analysis frameworks and tools for Java: Soot [VR+99], PMD [Cop05], and FindBugs [Aye+08]. This is done by comparing the results from a dead assignment analysis (implemented on top of the dataflow analyses) on a set of benchmark Java programs from the DaCapo suite [Bla+06], the largest being 130 000 lines of code. Our results show that our approach present precise results on par with Soot, and provides better performance than the selected set of tools for almost all selected benchmarks.

The rest of this paper is structured as follows. The implementation of control-flow analysis is described in Section 2, and the dataflow analyses in Section 3. An application doing dead assignment analysis is given in Section 4, and Section 5 discusses how to extend the analysis when the source language is extended. Section 6 provides a performance evaluation of our method. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 Control-flow Analysis

In control-flow analysis, the goal is to build a control-flow graph (CFG) where nodes represent blocks of executable code, and successor edges link the blocks in their possible order of execution. The nodes typically correspond to basic blocks, i.e., linear sequences of program instructions with one entry and one exit point [All70]. Each node n has a set of immediate successors, $succ(n)$, and a set of immediate predecessors, $pred(n)$, both of which can be empty.

2.1 Control-flow API

In JastAdd, a program is represented as an AST, with nodes that are objects with attributes. To represent the CFG, we superimpose it on the AST, treating statement and expression nodes as nodes in the CFG. We represent the $succ$ and $pred$ sets as

attributes on an interface **`CFGNode`** implemented by expressions and statements. To represent the entry and exit points of a method, we add synthetic empty statements to the method declaration.

```
public Set<CFGNode> CFGNode.succ();
public Set<CFGNode> CFGNode.pred();

public CFGNode MethodDecl.entry();
public CFGNode MethodDecl.exit();
```

Figure 1: The generated Java API for the control flow graph of a method.

JastAdd builds on Java, and generates an ordinary Java API for the AST and its attributes. Figure 1 shows the generated Java API for the CFG of a method. JastAdd specs can use this API to specify additional analyses, for example dataflow. The API can also be used by ordinary Java code, for example, an integrated development environment implemented in Java.

2.2 Language Structure

Figure 2 shows an example Java method and parts of its corresponding AST. We will use this as an example to illustrate how the control-flow graph is superimposed on the AST. To keep the example concise, we have omitted parameters and local declarations in the code.

```
void m() {
  if(c > 2)
    x = c;
  while(c < 10) {
    x += p();
    c++;
  }
}
```

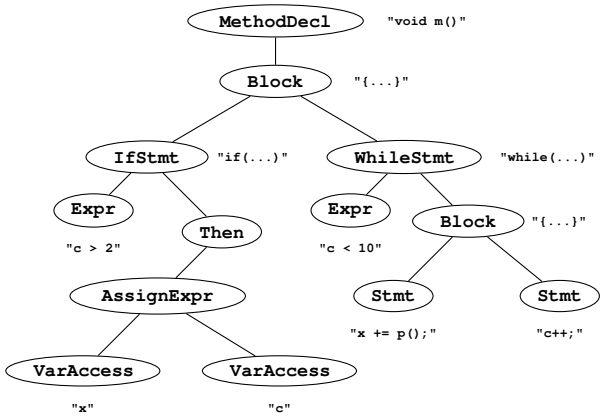


Figure 2: Sample Java method and its abstract syntax tree.

A simplified part of the abstract grammar for Java is shown in Figure 3. It is written in an object-oriented form with abstract classes **`Stmt`** and **`Expr`**, and subclasses for the individual statements and expressions such as **`WhileStmt`** and **`VarAccess`**.

```
MethodDecl ::= ParamDecl* Block;
ParamDecl  ::= <Type:String> <Name:String>;

abstract Stmt;
Block      : Stmt ::= Stmt*;
IfStmt     : Stmt ::= Expr Then:Stmt [Else:Stmt];
WhileStmt  : Stmt ::= Expr Stmt;
ExprStmt   : Stmt ::= Expr;
VarDecl    : Stmt ::= <Type:String> <Name:String> [Init:Expr];
ReturnStmt : Stmt ::= [Expr];
EmptyStmt  : Stmt;

abstract Expr;
AssignExpr : Expr ::= LValue:Expr RValue:Expr;
VarAccess  : Expr ::= <Name:String>;
MethodCall : Expr ::= <Name:String> Arg:Expr*;
```

Figure 3: Simplified parts of the Java abstract grammar in Figure 2.

The grammar uses a typical syntax with the Kleene star for list children, angle brackets for tokens, and square brackets for optional children. Children are either named after their types, such as a **`Block`** child of a **`MethodDecl`**, or with given names preceding the type name. For example, the left and right children of an **`AssignExpr`** are named **`LValue`** and **`RValue`**.

Certain constructs in Java can act as both expressions and statements, for example assignments and method calls. They are represented as expressions in the grammar, for example **`AssignExpr`**, and the class **`ExprStmt`** serves the purpose of adapting such expressions to serve as statements. The full grammar for Java is available at the JastAdd web site [Jas].

2.3 The control-flow graph

Figure 4 shows how the AST has been attributed with successor edges and synthetic nodes, to form the CFG for the example method. The statement nodes constitute the nodes of the CFG, and reference attributes represent the successor edges. Two synthetic nodes are added to represent the entry and exit of the graph.

Some nodes can be viewed as explicitly transferring control, whereas others merely let the control flow through them. For example, the **`AssignExpr`** in Figure 4

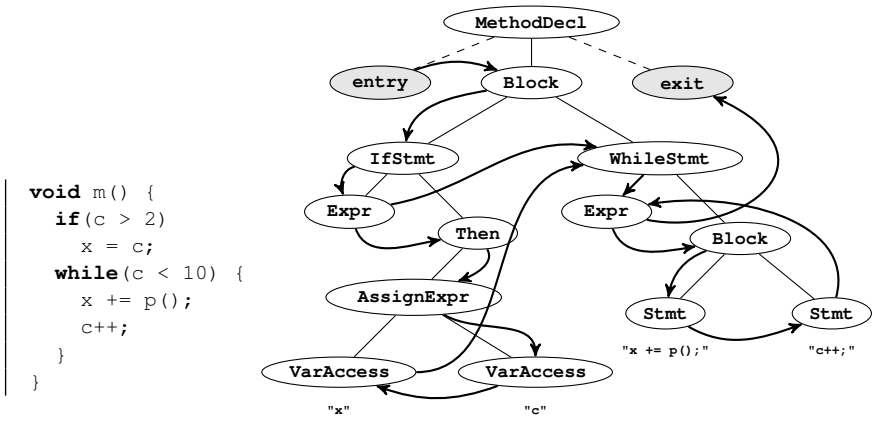


Figure 4: Example method and its CFG, excluding control flow internal to omitted children. Successors are shown as directed edges. Synthetic nodes are grey and the dashed lines show parent-child relations to these nodes.

transfers the control first to its right-hand side (the read of `c`) and then to its left-hand side (the assignment to `x`). After that, it transfers control to some location decided by its context (to the **`WhileStmt`** in this case). For a **`VarAccess`**, the control simply flows through, transferring to a location decided by its context.

Based on this observation, we distinguish between the following three categories of nodes.

Non-directing nodes which merely transfer control to the next node, as decided by their context. A **`VarAccess`** is an example of a node in this category.

Internal flow nodes which may transfer control to and between their children. Examples of nodes in this category are **`Block`**, **`WhileStmt`**, and **`AssignExpr`**.

Abruptly completing node which may transfer control to a specific location outside itself, in effect ending the execution of one or more enclosing nodes. Examples of such nodes in Java include *breaks*, *throws*, *returns* and method calls [Gos+96].

In the following subsections, we will discuss how the different parts of the CFG are specified, and how these different categories of nodes are handled.

2.4 The successors framework

Figure 5 shows a small attribution framework for the successor edges. It specifies the behavior for non-directing nodes, and can be specialized to handle internal flow and abruptly completing nodes. The framework introduces four attributes: **`succ`**,

following, **followingTrue** and **followingFalse**. The **succ** attribute is a set of references to nodes, and represents the successor edges in the CFG. The **following** attribute of a node n, is its set of successors as seen from its enclosing node, i.e., without any knowledge of the internal flow or possible abruptly completing nodes inside n.

The attributes **followingTrue** and **followingFalse** are used for handling control flow of short-circuited boolean expressions. For instance, in "`e1 && e2`", the evaluation of `e2` should be skipped if `e1` is *false*. If this boolean expression is enclosed in some other boolean expression or conditional construct, the place to skip to may be a different one from the ordinary following set. The attributes capture the appropriate place to skip to.

In the framework, **succ** is defined to be equal to **following**, thus capturing the behavior of non-directing nodes. Subclasses of **Expr** and **Stmt** can override this definition to cater for internal flow or abrupt completion.

```
// The successor edges in the CFG
syn Set<CFGNode> CFGNode.succ();

// Nodes that follow a node, as seen from its context
inh Set<CFGNode> CFGNode.following();

// By default, they are the same.
eq CFGNode.succ() = CFGNode.following();

// The following node for conditional branches.  By default, these
are empty
inh CFGNode.followingTrue();
inh CFGNode.followingFalse();
```

Figure 5: The attribution framework for successors.

The attribute **succ** is *synthesized*, whereas **following**, **followingTrue** and **followingFalse** are *inherited*[1]. The difference is that synthesized attributes must be defined in the node in which they are declared, whereas inherited attributes must be defined in an ancestor node. So, **succ** is defined by an equation in **CFGNode**, and can have overriding equations in subclasses of **Expr** and **Stmt**, similar to ordinary virtual methods. The attribute **following** of a node n, must instead be defined by one of the ancestor nodes of n. So to use this framework, equations must be provided that define the value of **following** for all possible nodes. The same applies to **followingTrue** and **followingFalse**.

[1] Note that this use of the term *inherited* stems from Knuth [Knu68] and is unrelated to and different from the object-oriented use of the term.

Node kind	**Examples**	**`succ`**	**`following*`**
Non-directing	variable	–	–
Internal-flow	block if assignment	direct flow to an internal node	possibly redefine for internal nodes
Abruptly completing	break return throw	direct flow to a special location	–

Figure 6: How different kinds of nodes extend the successors framework to achieve the control-flow graph.

The table in Figure 6 shows how the CFG is achieved by extending the successors framework: For non-directing nodes, no additional equations are needed. For internal-flow nodes, the equation for **`succ`** needs to be overridden, and equations may need to be added for constituents' **`following`**, **`followingTrue`** and **`followingFalse`** attributes. For abruptly completing nodes, **`succ`** is overridden.

As an example of an internal-flow node, consider the **`Block`** whose CFG specification is shown in Figure 7. To capture the internal flow, **`Block`** overrides the definition of its own **`succ`** attribute, transferring control to its first internal statement, if there is one. Since a block has a list of statement children, it must also define the value of **`following`** for each of these children. This is done by the equation **`Block.getStmt(int i).following = ...`** which applies to the **`i`**:th statement child of a block. For the last child, **`following`** is simply the same as for the block itself. For other children, **`following`** contains a reference to the next child in the block. The function **`singleton`** used in this definition returns a set containing a single given reference.

```
eq Block.succ() =
    (getNumStmt() = 0) // no children
    ? following()
    : singleton(getStmt(0));

eq Block.getStmt(int i).following() =
    (i = getNumStmt()-1) // last child
    ? following()
    : singleton(getStmt(i+1));
```

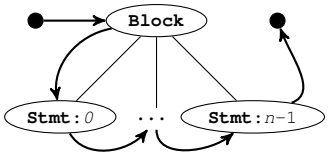


Figure 7: Specializing the successors framework for **`Block`**.

Another example of an internal-flow node is the **`IfStmt`**, whose CFG specifi-

cation is shown in Figure 8. The equation overriding **succ** states that control will be transferred to the **Expr** part (the condition). To allow boolean expressions in the condition to short-circuit to the correct branch, equations are given defining the **followingTrue** and **followingFalse** attributes. For normal (non-short-circuited) control flow, transfer is possible to both branches as defined by the equation for the **following** attribute.

Note that it is not necessary to define the **following** attribute for the **Then** and **Else** parts, since they should have the same value as **following** for the **IfStmt** itself, so the same equation in some ancestor applies to these parts.

```
eq IfStmt.succ() = singleton(getExpr());
eq IfStmt.getExpr().followingTrue() = singleton(getThen());
eq IfStmt.getExpr().followingFalse() = hasElse() ?
    singleton(getElse()) : following();
eq IfStmt.getExpr().following() =
    getExpr().followingTrue().union(getExpr().followingFalse());
```

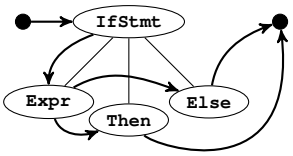


Figure 8: Specializing the successors framework for **IfStmt**.

Before we give examples of abruptly completing statements, we will introduce the framework for entry and exit nodes.

2.5 The entry and exit framework

To make sure there will always be well-defined entry and exit nodes, even for empty methods, we add two synthetic empty statements to each method. Nodes can be added declaratively to an AST by means of *higher-order attributes*, also known as *non-terminal attributes* (NTAs) [Vog+89]. An NTA is like a non-terminal in that it is a node in the AST. However, instead of being constructed as part of the initial AST, typically built by a parser, it is defined by an equation, just like an attribute. So in this sense, it is both an attribute and an AST node, hence the term higher-order. The right-hand side of an equation for an NTA must denote a *fresh* object, i.e. an object not already part of the AST, typically computed by a *new* expression.

Figure 9 shows the attribution framework defining the entry and exit nodes. Since the method declaration is the parent of both the entry and exit nodes, as well

```
syn nta Stmt MethodDecl.entry() = new EmptyStmt();
syn nta Stmt MethodDecl.exit() = new EmptyStmt();

eq MethodDecl.entry().following() = singleton(getBlock());
eq MethodDecl.getBlock().following() = singleton(exit());
eq MethodDecl.exit().following() = empty();

inh Stmt Stmt.exit();
eq MethodDecl.getBlock().exit() = exit();
eq MethodDecl.entry().exit() = exit();
eq MethodDecl.exit().exit() = exit();
```

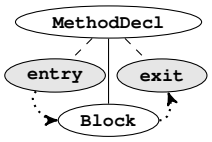


Figure 9: Attribution framework for entry and exit nodes. Dotted directed edges indicate elements in the **`following`** sets.

as of the main block, it furthermore needs to define their **`following`** attributes. Naturally, the entry is followed by the main block, which is followed by the exit node, which in turn has no following statements, as specified in the equations. The function **`empty`**, used when defining **`following`** for the exit node, simply returns the empty set.

The framework additionally defines an inherited attribute **`exit`** which gives all nodes access to the **`exit`** node. This is useful for abruptly completing nodes which need to transfer control directly to the **`exit`** node.

As a simple example of an abruptly completing node, consider the **`return`** statement. Figure 10 shows how it directs the control flow directly to the **`exit`** node by overriding the **`succ`** attribute. This definition is simplified, however, and does not take Java exception handling into account. A full treatment of these issues is given in the next section.

2.6 Handling Java Exceptions

The Java statements **`break`**, **`throw`**, **`continue`** and **`return`** are abruptly completing nodes, transferring control to a specific location outside of themselves.

The successor of an abrupt node is called the *target* node. For example, the target of a **`return`** statement is normally the **`exit`** node, as was shown in Figure 10.

```
eq ReturnStmt.succ() = exit();
```

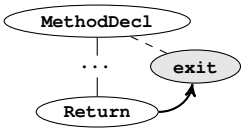


Figure 10: Using the entry and exit framework to abruptly transfer control from return statements to the end of the method. (Simplified definition that ignores Java exceptions.)

However, if the abrupt node is inside the `try` block of a Java exception handler with a `finally` block, the `finally` block will intercept control before transferring control to the normal target(s). Figure 11 shows an example.

```
try {
  return;
} finally {
  n();
}
```

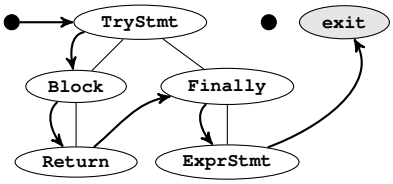


Figure 11: The control flow from a `return`, in the presence of a `finally` block.

In a similar way, other abrupt nodes also have a normal target to which control is transferred if there are no enclosing `try` statements with `finally` blocks. For `throw` it is a matching `catch`, or the `exit` node. For `break` the normal target is the statement following a matching enclosing loop or labeled statement. For `continue` the normal target is the first part of a matching enclosing loop. Figure 12 shows example normal control-flow (without `finally` blocks).

We will now show how control flow of abrupt nodes is handled in the presence of `finally` blocks. As an example, we will take a closer look at the `break` statement. The other abrupt nodes are handled in an analogous way. We introduce an inherited attribute `breakTarget`, returning a singleton set with the matching target, or the empty set if no target is found (corresponding to a compile-time error). For the `break` statement, this attribute will be the true successor, i.e., either the normal target (e.g., a while loop), or a `finally` block.

The attribute `breakTarget` is also defined for the `try` statement, by which the `finally` block can find its successor, i.e., usually the normal target. This solution works also for nested `try` statements with `finally` blocks, in which case control is transferred from the `break` statement, through all the `finally` blocks of enclosing `try` statements, and finally to the normal target.

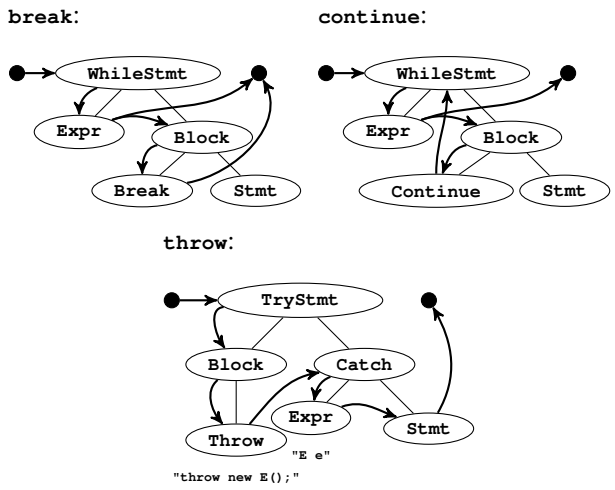


Figure 12: Control flow for some abrupt nodes.

The `breakTarget` attribute is parameterized by the `BreakStmt` to allow the target for the correct `BreakStmt` to be found. This attribution solution, using parameterized inherited attributes, is similar to the JastAdd implementation of Java name analysis, as presented in [EH05].

The successor of a `BreakStmt` is now simply defined as the `breakTarget` of itself. Figure 13 shows the specification. There are several equations defining `breakTarget`, and if there is more than one in a chain of ancestors, the closest equation applies. Therefore, if a `BreakStmt` is enclosed by a `TryStmt`, and then by a `BranchTargetStmt` (e.g., a while loop), the equation in the `TryStmt` will hold. If the `BreakStmt` is not enclosed by any of these kinds of statements, the equation defined in `BodyDecl` will hold, defining the target to be the empty set. To illustrate how this works, consider Figure 14, showing the values of `breakTarget` for an example program.

To handle the remaining abrupt statements, `continue`, `return`, and `throw`, we define one target attribute for each of them and use them in a similar fashion. With this approach we end up with potentially several abrupt nodes transferring control to the `finally` block. The potential successors of the `finally` block is thus the set of normal targets for all these intercepted abrupt nodes. For this reason, we introduce an attribute `interceptedAbruptNodes` which contains references to these nodes. Given this attribute, the `TryStmt` can define the `following` attribute for its `finally` block, as shown in Figure 15. Here, the attribute `targetAt` uses the double dispatch pattern [Ing86] to let each kind of abrupt node decide how to

```
eq BreakStmt.succ() = breakTarget(this);

inh Set BreakStmt.breakTarget(BreakStmt stmt);
inh Set TryStmt.breakTarget(BreakStmt stmt);

// Equations for breakTarget
eq BodyDecl.getChild().breakTarget(BreakStmt stmt) = empty();
eq BranchTargetStmt.getChild().breakTarget(BreakStmt stmt) =
    targetOf(stmt)
    ? following()
    : breakTarget(stmt);
eq TryStmt.getBlock().breakTarget(BreakStmt stmt) =
    hasFinally()
    ? singleton(getFinally())
    : breakTarget(stmt);
```

Figure 13: Specializing the successor framework for **`BreakStmt`**. The **`targetOf`** attribute is defined in the compiler frontend.

compute its target[2].

Handling unchecked exceptions

In addition to explicitly thrown exceptions, using the **`throw`** statement, exceptions can be thrown implicitly by the runtime system at runtime errors such as null pointer dereferencing, division by zero, out of memory, etc. Unless these errors are caught, they are propagated back to the calling method, making also method calls a source of such implicit exceptions. So in this sense, more or less every expression and statement can have abrupt completion. Instead of adding explicit successor edges for all these possible control paths, we define an inherited attribute **`uncheckedExceptionTarget`** for **`Expr`** and **`Stmt`** nodes, and in that way make all nodes aware of these potential successors. By default, this attribute is a set containing the **`exit`** node. But if there are **`catch`** clauses that match **`RuntimeException`** or **`Error`**, these clauses are also added.

This approach is inspired by the factored control-flow graph explained in [Cho+99] where unchecked exception branches are summarized at the end of basic blocks to limit the number of branches.

[2] The equation for `following` uses an assignment and a for loop which might be surprising since our approach is declarative. However, because we use Java method body syntax to define attribute values, it is natural to use imperative code here. This is perfectly in agreement with the declarative approach as long as that code has no net side effects, i.e., only local variables are modified.

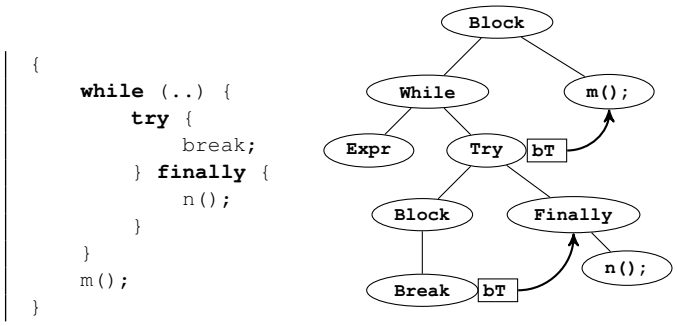


Figure 14: Values of the `breakTarget` attribute (**bT**).

2.7 Predecessors

To complete the implementation of the control-flow API, we now define the set of predecessors. This is simply the inverse of the successors relation, so if there is a successor edge from a to b, there will be a predecessor edge from b to a. Such inverse relations are easily defined using *collection* attributes [Boy05; Mag+09]. The attributes we have seen so far have been defined using an equation located in an AST node. A collection, in contrast, is an attribute whose value is defined by the combination of a number of *contributions*, distributed over the AST. This way, we can define the predecessor sets by letting each node contribute itself to the predecessor sets of its successors. Figure 16 shows the JastAdd specification.

Rule (1) is the declaration of the collection attribute `pred` for `CFGNode`. The rule states the type of the attribute (`Set`), the initial value (the empty set), and the operation used to add contributions (`add`). For correct evaluation, it is assumed that the operation is commutative, i.e., that the order of adding the contributions is irrelevant, which is indeed the case for the `add` method for the Java class `Set`.

Rules (2) and (3) declare that each `Stmt` and `Expr` node contributes itself (`this`) to the `pred` attribute of each of its successors. A more detailed presentation of collection attributes and their evaluation in JastAdd is available in [Mag+09].

3 Dataflow Analysis

We want to analyze dataflow on the control-flow graph defined in the previous section. Two typical examples of dataflow analyses are liveness analysis and reaching definition analysis. We describe our implementation of these analyses using JastAdd in the following two subsections.

```
eq TryStmt.getFinally().following() {
    Set flw =
      (getFinally().canCompleteNormally())
      ? following()
      : empty();
    for (Stmt abrupt : interceptedAbruptStmts) {
      flw = flw.union(abrupt.targetAt(this));
    }
    return flw;
}

syn Set Stmt.targetAt(TryStmt t) = empty();
eq BreakStmt.targetAt(TryStmt t) = t.breakTarget(this);
eq ContinueStmt.targetAt(TryStmt t) = t.continueTarget(this);
...
```

Figure 15: Specializing the successor framework for **`TryStmt`**.

```
coll Set CFGNode.pred() [empty()] with add;                  // (1)
Stmt contributes this to CFGNode.pred() for each succ();     // (2)
Expr contributes this to CFGNode.pred() for each succ();     // (3)
```

Figure 16: Using a collection attribute to define the predecessors.

3.1 Liveness Analysis

A variable is *live* at a certain point in the program, if its assigned value will be used by successors in the control-flow graph. If a variable is assigned a new value before an old value has been used, the old assignment to the variable is unnecessary, also called *dead*.

We express liveness in the same fashion as Appel in [App02] using four sets – in, out, def and use. The def set of a node n contains the variables assigned a value in n, and the use set contains the variables whose values are used in n. From these two sets we calculate the in and out sets, i.e., variables live into a node and variables live out of a node, using the following equations:

Definition 1 *Let n be a node and $succ[n]$ the value of the* **`succ`** *attribute for the node n:*

$$\begin{aligned} in[n] &= use[n] \cup (out[n] \setminus def[n]) \\ out[n] &= \bigcup_{s \in succ[n]} in[s] \end{aligned}$$

We note that the equations for the in and the out sets are recursive and mutually dependent, i.e. they have a circular dependency to each other. Equations like these are usually solved by iteration until a fixpoint is reached, which is guaranteed if all intermediate values can be organized in a finite height lattice and all operations are monotonic on that lattice. We will explain how circular equations like these can be implemented as circular attributes in JastAdd [MH07].

The use and def sets

The main challenge in computing the use set for each node, is to support all kinds of statements and expressions in the source language. A complex language such as Java has more than 20 statements and 50 expressions. Fortunately, it is quite easy to support all these constructs in JastAddJ (the JastAdd Extensible Java Compiler), since each expression that accesses a local variable encapsulates a **`VarAccess`** node performing the actual binding. Moreover, each **`VarAccess`** node has two **`boolean`** attributes, **`isDest`** and **`isSource`**, determining whether the access acts as a definition (*l-value*) or use (*r-value*). Some nodes actually act as both. For example, a **`VarAccess`** that is the child of the post increment operator '**`++`**', will both read from and write to the variable. JastAddJ also defines an attribute **`decl`** for **`VarAccess`** nodes, referring to the appropriate declaration node. Figure 17 summarizes the JastAdd API used.

```
public boolean VarAccess.isDest();
public boolean VarAccess.isSource();
public Decl VarAccess.decl();
```

Figure 17: JastAddJ API used by liveness analysis

In the liveness analysis, we represent use and def as sets of references to declaration nodes in the AST. We implement them using synthesized attributes, and let **`VarAccess`** nodes add themselves to the appropriate collection, depending on their role as an *r-value* and/or *l-value*. The variable, parameter and field declarations are also viewed as assignments, so they contribute themselves to their own def set. Figure 18 shows the implementation of these attributes.

These two attributes effectively compute the use and def sets for all intraprocedural control-flow nodes in Java. If we add a new language construct that modifies a local variable we need only make sure it encapsulates a **`VarAccess`** and provide equations for the inherited attributes **`isDest`** and **`isSource`**, which are needed elsewhere in the frontend anyway, and the use set and def set attributes are still valid.

```
// def
syn Set<Decl> CFGNode.def();
eq Stmt.def() = empty();
eq Expr.def() = empty();
eq VarAccess.def() = isDest() ? singleton(decl()) : empty();
eq VarDecl.def() = singleton(this);
eq ParamDecl.def() = singleton(this);

// use
syn Set<Decl> CFGNode.use() = empty();
eq VarAccess.use() = isSource() ? singleton(decl()) : empty();
```

Figure 18: Implementation of def and use for liveness analysis

The in and out sets for liveness

The equations for the in set and out set in Definition 1 are mutually dependent. As mentioned earlier, such equations can be solved by iteration as long as the values form a finite height lattice and all functions are monotonic. This is clearly the case for our equations since the power set of the set of local variables, ordered by inclusion, forms a finite lattice, with the empty set as bottom, on which union is monotonic. A fixpoint will thus be reached if we start with the bottom value and iteratively apply the equations as assignments until no values change.

JastAdd has explicit support for fixpoint iteration through circular attributes, as described in [MH07]. If we declare an attribute as circular and provide a bottom value, then the attribute evaluator will perform the fixpoint computation automatically. This allows us to implement the in and out sets using circular attributes, resulting in a specification very close to the textbook definition, as shown in Figure 19.

In our actual implementation, we use an even more concise specification of the out set by defining it as a collection attribute, reversing the direction of the computation by making use of the predecessors instead of the successors. See Figure 20.

An alternative to using circular attributes would be to manually implement the fixpoint computation imperatively. Such a solution requires manual book keeping to keep track of change, which significantly increases the size of the implementation and the essence of the algorithm gets tangled with book keeping code. Also, it is necessary to either statically approximate the sets of attributes involved in the cycle to iterate over, or to manually keep track of such dependencies dynamically. This is all taken care of automatically by the attribute evaluation engine in JastAdd when using circular attributes.

```
// in
syn Set<Decl> CFGNode.live_in() circular [empty()] =
    use().union(live_out().compl(def()));

// out
syn Set<Decl> CFGNode.live_out() circular [empty()] {
    Set<Decl> set = empty();
    for(Stmt s : succ()) {
        set = set.union(s.live_in());
    }
    return set;
}
```

Figure 19: Implementation of liveness in and out sets, using circular attributes.

```
coll Set<Decl> CFGNode.live_out() circular [empty()] with add;
Stmt contributes live_in() to CFGNode.live_out() for each pred();
Expr contributes live_in() to CFGNode.live_out() for each pred();
```

Figure 20: Alternative implementation of the out set, using a circular collection.

3.2 Reaching Definition Analysis

In computing *reaching definitions*, we are interested in sets of *definitions* (assignments), rather than in sets of variable declarations. Because definitions may occur in several different syntactic constructs, not just in assignment statements, we define an interface `Definition` to abstract over the relevant AST classes, namely `VarAccess`, `VarDecl`, and `ParamDecl`. Not all variable accesses are definitions, but the `isDest` attribute can be used to decide this.

A definition of a variable is said to *reach* a use of a variable if there is a path in the control-flow graph from the definition to the use. A variable use may be reached by more than one variable definition in which case the actual value of the variable can not be decided statically. For cases where there is only one reaching definition the use might be replaceable with a constant, a property typically used in, for example, constant propagation.

We define five sets – $defs$, gen, $kill$, in and out, in the same fashion as Appel [App02]. The $defs$ set of a variable declaration v contains all definitions of that variable. The gen set of a node n contains the definitions in n, i.e., corresponding to the new variable values generated by that node. The $kill$ set of a node n is the set of definitions killed by definitions made in n. Consider a definition d of a certain variable v. The $kill$ set for a definition d is the $defs$ for v, minus the definition d

itself, see Definition 2. The $kill$ set for a statement is simply the union of the $kill$ sets of its gen set.

The in set of a node n is the set of definitions that reach the beginning of n, and out is the set that reaches the end of n. Given the $kill$ and gen sets, in and out are defined as shown in Definition 3. Note that the equations for in and out are recursive and mutually dependent, hence requiring a fixpoint iteration for evaluation.

Definition 2 *Let d be a definition of a variable v:*

$$d : v \leftarrow \ldots : kill[d] = defs[v] \setminus \{d\}$$

Definition 3 *Let n be a node and $pred[n]$ the value of the* **`pred`** *attribute for the node n:*

$$\begin{aligned} in[n] &= \bigcup_{p \in pred[n]} out[p] \\ out[n] &= gen[n] \cup (in[n] \setminus kill[n]) \end{aligned}$$

The *defs* set

To implement the $defs$ set, we use a collection attribute on **`Variable`**, which is an interface implemented by **`VarDecl`** and **`ParamDecl`**. We then let the definitions contribute themselves to their declaration. Contributing **`VarAccess`** nodes check that they are actually acting as definitions using the attribute **`isDest`**. The implementation is shown in Figure 21.

```
coll Set<Definition> Variable.defs() [empty()] with add;
VarAccess contributes this
    when isDest() to Variable.defs() for decl();
VarDecl contributes this to Variable.defs() for this;
ParDecl contributes this to Variable.defs() for this;
```

Figure 21: Implementation of $defs$ using attributes.

The *gen* and *kill* sets

The gen set of a node contains all the definitions inside the node. We use a synthesized attribute to implement this set and let variable declarations, parameter declarations and **`VarAccess`** nodes, that serve as definitions, contribute themselves to their own gen. The $kill$ set is implemented using the same strategy, see Figure 22.

```
// gen
syn Set<Definition> CFGNode.gen();
eq Stmt.gen() = empty();
eq Expr.gen() = empty();
eq VarAccess.gen() = isDest() ? singleton(this) : empty();
eq VarDecl.gen() = singleton(this);
eq ParamDecl.gen() = singleton(this);

// kill
syn Set<Definition> CFGNode.kill();
eq Stmt.kill() = empty();
eq Expr.kill() = empty();
eq VarAccess.kill() = isDest() ? defs().compl(this) : empty();
eq VariableDeclaration.reaching_kill() = defs().compl(this);
eq ParameterDeclaration.reaching_kill() = defs().compl(this);
```

Figure 22: Implementation of gen and $kill$.

The in and out sets for reaching definitions

In Definition 3 the sets in and out are defined as two mutually dependent equations using the $kill$ and gen sets. Again we use circular attributes, obtaining an implementation very similar to the textbook definition of these sets. See Figure 23.

```
// out
syn Set<Definition> CFGNode.reach_out() circular [empty()];
eq CFGNode.reach_out() = gen().union(reach_in().compl(kill()));

// in
coll Set<Definition> CFGNode.reach_in() circular [empty()] with add;
Stmt contributes reach_out() to CFGNode.reach_in() for each succ();
Expr contributes reach_out() to CFGNode.reach_in() for each succ();
ParamDecl contributes reach_out() to CFGNode.reach_in() for each succ();
```

Figure 23: Implementation of the in and out sets for reaching definitions.

4 Dead Assignment Analysis

To evaluate the efficiency and scalability of our approach, we have implemented a simple intraprocedural analysis for Java which detects dead assignments. In more detail, we locate assignments whose values are not used later in a body declaration,

i.e., in a method, constructor, instance initializer, static initializer, or field declaration. We only include assignments to local non-constant variables and parameters in the analysis:

```
syn lazy boolean CFGNode.includeInDeadAssignAnalysis() = false;
eq VarAccess.includeInDeadAssignAnalysis() =
    isDest() && isLocalStore();
eq VarDecl.includeInDeadAssignAnalysis() =
    hasInit() && isLocalVariable() && !isConstant();
```

We try out two versions on this selection: one based on liveness analysis, and one combining liveness analysis with analysis of reaching definitions.

4.1 Collecting Dead Assignments

To collect all dead assignments of a compilation unit, we add a collection (`coll`) attribute `deadAssignments` to the `CompilationUnit` class. This class represents a file with one or more classes which might contain one or more body declarations (methods, constructors etc.):

```
coll Set<Stmt> CompilationUnit.deadAssignments() [empty()] with add;
```

The `CompilationUnit` class is connected to the grammar in Figure 3 as follows (here, only including methods):

```
CompilationUnit ::= ClassDecl*;
ClassDecl       ::= MethodDecl*;
MethodDecl      ::= ...
```

Dead assignments contribute themselves to the collection of their enclosing `CompilationUnit` using a `contributes` clause. The reference to the `CompilationUnit` node is propagated to descending statement nodes using an inherited attribute `enclosingCompilationUnit`:

```
VarAccess contributes this
    when includeInDeadAssignAnalysis() && isDeadAssign()
    to CompilationUnit.deadAssignments()
    for enclosingCompilationUnit();
VarDecl contributes this
    when includeInDeadAssignAnalysis() && isDeadAssign()
    to CompilationUnit.deadAssignments()
    for enclosingCompilationUnit();
```

Each of these nodes, `VarAccess` and `VarDecl`, contribute to the collection, if they are included in the selection of the analysis, and their `isDeadAssign` attribute is true. We define this attribute to be false by default for all control flow nodes:

```
syn boolean CFGNode.isDeadAssign() = false;
```

4.2 Analyzing using Liveness

Definition 4 *If a variable is defined, but not live immediately after the node, the assignment is considered dead in the sense that the assignment is unnecessary. That is, an assignment a is dead when:*

$$kill[a] \neq \emptyset \wedge kill[a] \cap out[a] = \emptyset$$

Using liveness analysis, we can define an assignment to be dead when a defined variable is not live after the assignment, as defined in Definition 4. With this in mind, we can define a very useful attribute **`isDead`** which we can use to define equations for **`isDeadAssign`** as follows:

```
syn lazy boolean CFGNode.isDead();
eq CFGNode.isDead() = !def().compl(liveness_out()).isEmpty();

eq VarAccess.isDeadAssign() = isDead();
eq VarDecl.isDeadAssign() = isDead();
```

4.3 Analyzing using Liveness and Reaching Definition

We can combined liveness analysis with reaching definition analysis, by adding a condition to the equations of the **`isDeadAssign`** attribute, as follows:

```
eq VarAccess.isDeadAssign() = isDead() || allReachedUsesAreDead();
eq VarDecl.isDeadAssign() = isDead() || allReachedUsesAreDead();
```

The consequence of combining these two analyses, is that we can find additional dead assignments on the form:

```
a = 0; // Also dead because b is dead (the reached use)
b = a; // b is dead
```

Here, the assignment to `a` is dead because the assignment to `b` is dead, which is the only reached use of `a`. To get this behavior, we need to define the attribute **`allReachedUsesAreDead`**, which investigates whether all reached uses are dead:

```
syn boolean ReachingDef.allReachedUsesAreDead() circular [false];
eq Stmt.allReachedUsesAreDead() {
    for (ReachedUse use : reachedUses())
        if (!use.inDeadAssign())
            return false;
    return true;
}
```

The **reachedUses** attribute is defined on an interface **ReachingDef**, implemented by nodes defining values, and it returns a set of reached uses, implementing an interface **ReachedUse**. Nodes implementing the **ReachedUse** interface has an additional attribute **inDeadAssign** returning true if the use is in the right-hand side of an assignment that is dead:

```
inh boolean ReachedUse.inDeadAssign();
eq VarDecl.getInit().inDeadAssign() = isDead();
eq AssignExpr.getSource().inDeadAssign() =
    getDest().isLocalStore() && getDest().isDead();
eq Program.getChild().inDeadAssign() = false; // default value
```

It might be the case that an assignment that is dead has, for instance, a method call on its right-hand side, but we do not want to consider variables given to the method as dead. To avoid cases like these, we can add an equation to, for example, a method call as follows:

```
eq MethodAccess.getArg(int i).inDeadAssign() = false;
```

5 Language Extensions

The previous examples have illustrated how the control-flow specification for individual statements can be written modularly. Similarly, the control-flow implementation for Java 1.4 can be extended modularly to support Java 1.5. The only new language constructs that affect the CFG are the new enhanced **for** statement and **enum** constant, which is a new kind of body declaration. As an example we will considering the enhanced **for** statement in more detail, which has the following abstract syntax:

```
EnhancedFor : BranchTargetStmt ::= VarDecl Expr Stmt;
```

This statement iterates over the elements in the iterable object denoted by **Expr**. In each iteration, a new element is assigned to **VarDecl**, and the **Stmt** is executed. To capture this flow, we let the **EnhancedFor** itself represent the initialization of the iterator. We provide equations defining the **succ** attribute for **EnhancedFor** and the **following** attributes of its constituents. Figure 24 shows the specification.

Note that since the analyses of liveness, reaching definitions, and dead assignments are defined in terms of the control-flow graph, they will work automatically also for these new constructs.

```
eq EnhancedForStmt.succ() = singleton(getExpr());
eq EnhancedForStmt.getExpr().followingTrue() =
    singleton(getVarDecl());
eq EnhancedForStmt.getExpr().followingFalse() = following();
eq EnhancedForStmt.getExpr().following() =
    getExpr().followingTrue().union(
        getExpr().followingFalse());
eq EnhancedForStmt.getVarDecl().following()=
    singleton(getStmt());
eq EnhancedForStmt.getStmt().following()=
    singleton(getExpr());
```

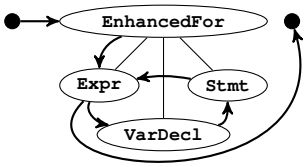


Figure 24: Control flow for `EnhancedFor`

6 Evaluation

To evaluate our approach, we have run the dead assignment analysis on a set of Java benchmark applications, and compared the results and performance to other analysis tools. We have also measured the size of our specification modules in order to evaluate development effort, and compared them to another tool.

6.1 Setup

Selection of Benchmarks

For evaluating our analyses, we have selected four Java applications of varying size from the DaCapo benchmark suite [Bla+06]: **ANTLR**, **Bloat**, **Chart** and **Apache FOP**. ANTLR is a parser and translator generator, Bloat is a byte-code level optimization and analysis tool, Chart is a charting utility tool and Apache FOP is a print formatting tool. Figure 25 gives an overview of the selected benchmarks with regard to size (lines of code), number of flows (methods, instance initializers etc.), and average size of these flows (number of nodes in the control-flow graph). ANTLR and Bloat are of similar size, but we include both because they differ substantially in their average flow size.

Name	Version	Lines of Code	Candidates	# Flows	Avg. Flow Size
ANTLR	2.7.7	37 730	3 826	3 332	47.0
Bloat	1.0	38 581	5 740	5 095	136.0
Chart	1.0	9 968	1 818	1 469	39.0
FOP	0.95	130 300	18 203	19 632	110.0

Figure 25: Java benchmarks. *Candidates* are the number of local variable declarations and assignments in an application. The last two columns show the number of intraprocedural flows (methods etc.) in an application and the average flow size, i.e. the average number of nodes in a flow.

The figure also shows the number of possible dead assignments, or *candidates*, in each application. For a node to be a candidate it needs to be either a variable declaration with an initializing assignment, or an assign expression. For reason of comparison, we exclude constants of primitive types (integer, double etc.) and strings from the set of candidates. Constants like these may be removed by default by some analysis tools, excluding them from the dead assignment analysis that we want to compare to.

Selection of Analysis Tools

We compare our JastAdd-based analysis results to those of **Soot** (2.4.0), **FindBugs** (1.3.9) and **PMD** (4.3.5). Soot is a very well known Java optimization framework, working at the byte code level [VR+99]. It is interesting for comparison as it can be expected to have very high precision and correctness. FindBugs [Aye+08] and PMD [Cop05] are two well known tools for detection of bugs and anomalies in Java source code. They are interesting for us to compare to since they exemplify the developer-oriented tools we have in mind for our AST-based analysis. FindBugs performs the analysis on byte code, whereas PMD analyzes the source code directly.

All these tools support a number of different analyses, but for our comparison we are only interested in dead assignment analysis. In order to get these results from each tool we have used the following configurations:

Soot The Soot framework is made up by a set of phases, each connected to a certain kind of analysis. For example, there is a phase called `jb` which translates input to a three-address code called *jimple*, and there are phases for whole program analysis, for example, `cg`, `wjtp`, `wjop`. We are interested in the intra-procedural analyses found in a phase called `jop`. So we disable all other phases, except for the `jb` phase. Inside a phase there are several packs, one for each analysis. For the `jop` phase, we are only interested in the `jop.dae` pack, performing dead assignment elimination, and hence we disable all other packs in the `jop` phase.

We want to easily find which assignments that Soot wants to eliminate. With this in mind, we have added a flag `-only-tag` to the `jop.dae` pack, which causes the analysis to tag an assignment rather than removing it. This way, we can print out the jimple code and find which assignments are detected as dead.

Since Soot operates on a jimple representation it might find a lot of dead assignments to temporaries in its own representation which do not correspond to assignments in the source. To partly deal with this issue we only consider assignments on the line of a source assignment, i.e., on the line of a candidate. However, given that one source assignment may be represented on several lines in Soot, it is still possible that Soot will remove an assignment but not the actual source assignment. For cases like these, i.e., where Soot does not remove all jimple lines of a candidate, we do a manual check.

FindBugs FindBugs performs a number of identifications of so called *bug patterns*, i.e., patterns in the code possibly corresponding to a bug. One such bug pattern identifies dead local stores (DLS), that is, dead assignments. We have configured FindBugs to only include the DLS pattern in its analysis. The results are given on a file and source line basis which makes it easy for us to map the result to candidates in a benchmark. To get as good precision as possible and to find all pattern matches for DLS we run FindBugs with the `-effort:max` and `-low` flags.

PMD PMD supports the definition of rules using Java or XPath, but also provides a default set of rules. One such rule set looks for so called dataflow anomalies of three kinds, and two of these locate dead assignments – DU and DD. DD by identifying when a variable is assigned twice in a row with out a use in between, and DU by identifying if an assigned value is not used in the scope it is defined. The third finds undefined variables (UR) which is not interesting for our comparison. Results are obtained on a file and line basis which makes it easy for us to map the results to candidates in a benchmark.

Comparison of Result

In order to compare the results of different tools we need a unified way to identify which assignments that are found to be dead. To accomplish this we pretty-print the source code of each benchmark and let each assignment start on a new line. This way we can identify a candidate by file name and source line.

In the case where the analysis is not performed on source code, we may need to maintain a mapping to source. For Soot we maintain a mapping between each source line and its corresponding jimple lines, to know if an assignment has been found completely dead or partially dead. In the case with FindBugs, which analyses bytecode, the result includes information of source lines and no extra mapping is required.

Performance Measurement

All performance measurements have been performed on a Lenovo Thinkpad X61 running Ubuntu 10.10 (Maverick Meerkat). For comparison between tools, we use the average time of 10 runs from a terminal, measuring execution time with the Unix command `time`. For JastAdd, we also provide performance measurements using the multi-iteration approach with a pre-heated VM, as presented in [Bla+08].

6.2 Correctness and Precision

Dead Assignments Found (#)

Tool: A,B	only A	both	only B
JA, Soot	8	308 (22)	3
JA, PMD	57	259	658
JA, FB	278	38	0
Soot, PMD	57	254 (21)	663
Soot, FB	276	35 (21)	3
PMD, FB	885	32	6

(a) **Results for ANTLR**

Tool: A,B	only A	both	only B
JA, Soot	8	78 (26)	3
JA, PMD	32	54	466
JA, FB	58	28	0
Soot, PMD	31	50 (6)	470
Soot, FB	59	22 (10)	6
PMD, FB	502	18	10

(b) **Results for Bloat**

Tool: A,B	only A	both	only B
JA, Soot	8	22 (4)	0
JA, PMD	0	30	104
JA, FB	19	11	0
Soot, PMD	0	22 (4)	112
Soot, FB	16	6 (3)	5
PMD, FB	123	11	0

(c) **Results for Chart**

Tool: A,B	only A	both	only B
JA, Soot	13	226 (31)	6
JA, PMD	22	217	1705
JA, FB	193	46	0
Soot, PMD	10	222 (27)	1700
Soot, FB	191	41 (21)	5
PMD, FB	1884	38	8

(d) **Results for Apache FOP**

Figure 26: Results The numbers show the number of dead candidate assignments found by pairs of tools: Soot, JA$_{live}$ (JA), FindBugs (FB) and PMD. For each tool pair, the number of assignments only found in one of the tools and the number of assignments found in both are shown. For the assignments found by both tools, where one tool is Soot, the number of cases where Soot only removed some jimple lines are shown within parentheses.

Figure 26 shows the number of dead candidate assignments found by each tool. The results are grouped into four subfigures, one for each Java benchmark.

For JastAdd we only include the results for JA$_{live}$, i.e., the dead assignment analysis only using liveness. The results for JA$_{live+reaching}$ are slightly more precise, but at a substantial additional cost in execution time. JA$_{live+reaching}$ only identified an additional three cases, one in ANTLR and two in Chart, all on the form:

```
s = s + a; // dead in JA_live
s = b; // also dead in JA_live+reaching
```

None of the other tools found any of these cases.

JastAdd and Soot both find very similar numbers of dead assignments among the selected candidates. JastAdd finds a few dead assignments that Soot does not find, and we have manually verified that they are indeed dead. Soot also finds a few dead assignments that JastAdd does not find. We have looked at each of these manually. One of these cases correspond to a true dead assignment at the source level:

```
int a = 0;
while (expr) {
    a++; // dead in Soot but not in JastAdd
}
```

Here, `a` is kept alive in the JastAdd analysis, while not in Soot.

In the other cases where Soot identifies dead candidate assignments, and JastAdd not, it is actually not the source level assignment that is detected, but assignments to temporary variables introduced in the jimple code. These do thus not correspond to dead assignments at the source level.

There are some cases where both Soot and JastAdd have identified a dead assignment, but Soot has only removed some of the corresponding jimple lines. This is because the right-hand side is a construct that might have side effects, typically a method call, and the call is therefore still present. The behavior for these cases is equivalent for JastAdd and Soot, but we needed to look at the jimple code manually to determine this.

In addition to the dead assignments found on candidates shown in the figure, Soot also finds an additional number of dead assignments not matching candidates (ANTLR=256, Bloat=902, Chart=106 and FOP=5212). We have not been able to manually check all these assignments, but after looking at many of them, we have only found cases that are either due to constant propagation (which we do not do), or to temporary variables introduced in the jimple code.

PMD reports very many dataflow anomalies of type DD and DU. After inspecting several of those that are neither reported by Soot nor JastAdd, we have only found false positives. It seems that arrays appear to be treated as ordinary variables, and that the control-flow is not fine enough, ignoring, for example, short-circuiting of boolean expressions. Like Soot, PMD reports dead assignments for non-candidates (ANTLR=18, Bloat=99, Chart=22, FOP=625). These non-candidates may, for example, be fields. It should be pointed out that the DD and DU reports are described by PMD to be anomalies that are *potentially* dead assignments. PMD does not claim that they are dead.

FindBugs finds comparatively few dead assignments, and reports no dead assignments for non-candidates. All the dead assignments found by FindBugs are

Bench.	JA$_{live}$	JA$_{live+reach}$	Soot	FindBugs	PMD
ANTLR	11.8 ± 0.3	22.3 ± 0.2	26.0 ± 5.2	105.6 ± 18.1	17.9 ± 2.4
Bloat	15.0 ± 0.4	46.8 ± 11.8	37.0 ± 8.9	115.5 ± 14.8	61.9 ± 10.1
Chart	7.4 ± 0.2	17.2 ± 4.4	20.2 ± 5.2	53.0 ± 12.0	7.6 ± 0.1
FOP	59.4 ± 11.6	278.9 ± 27.3	256.3 ± 2.6	250.3 ± 38.3	38.9 ± 9.3

Figure 27: Average total execution time (in seconds)

found also by JastAdd.

6.3 Performance

Benchmark	Plain	JA$_{live}$	JA$_{live+reach}$
ANTLR	1.7 ± 0.1	2.9 ± 0.06	9.1 ± 0.04
Bloat	2.3 ± 0.1	3.6 ± 0.06	17.5 ± 0.08
Chart	1.0 ± 0.07	1.3 ± 0.1	9.3 ± 0.2
FOP	10.0 ± 0.05	16.2 ± 0.07	182.4 ± 16.5

Figure 28: In-memory performance for JastAdd. In seconds, using a pre-heated VM. *Plain* is the static-semantic analysis only.

Figure 27 shows average total execution times in seconds for all tools, measured using `time`. All average times are given with a confidence interval of 95%.

JA$_{live}$ is faster than Soot on all four applications, and it is the fastest tool for three of the applications, with the exception of FOP where PMD is faster. For PMD, the performance for Bloat sticks out, which may be due to the large average flows in Bloat. FindBugs generally gets the worst performance, except for FOP where both JA$_{live+reach}$ and Soot are worse. Both JastAdd and PMD perform analysis on source which is likely to result in smaller control-flow graphs with less nodes. This might also explain the difference in performance between Soot/FindBugs and JastAdd/PMD.

One motivation for doing this type of analysis on source rather than on byte code is the applicability in interactive settings, for example, in editors. In an editing scenario a model of the edited program will be kept in memory. This model, which is typically an AST, will be updated in response to user actions, like code modifications. The time needed for re-computation of information will affect the response time experienced by the user, and a translation to byte code would potentially slow down performance. Figure 28 shows JastAdd performance measures for an in-memory AST with a pre-heated VM. We show both our analyses, JA$_{live}$ and JA$_{live+reach}$, as well as a plain analysis only doing semantic analysis. These

numbers show the performance for a full analysis of the whole benchmark application. Preferably, in an editing scenario each edit action should not trigger a full analysis of the application being edited, but employ some incremental evaluation mechanism for limiting unnecessary re-computations.

6.4 Effort

Modules			Number of Rules				
Name	**Version**	**LOC**	`syn`	`inh`	`eq`	`coll`	`contr.`
Java Frontend	1.4	10 352	471	168	1 453	0	0
	1.5	4 909	166	48	588	0	0
Control Flow	1.4	444	17	26	185	2	5
	1.5	20	0	0	9	0	0
Liveness	1.4	29	4	1	10	1	3
Reaching	1.4	96	8	1	30	3	7
Helpers	1.4	33	11	1	13	1	1
Dead assignment	1.4	25	3	1	5	2	5

Figure 29: Size of modules using lines of code (LOC) and number of JastAdd rules separated into different columns for – `syn`, `inh`, `eq`, `coll`, `contributes`. The modules for the alternative variants of liveness (JA_{live} and $JA_{live+reaching}$) have the same size and are only included once.

In order to estimate effort of implementation, we look at the actual size of the implementations. By making use of higher-level abstractions in the form of attributes, our wish is to decrease the development effort needed for the analyses.

Figure 29 shows an overview of the different modules for the JastAdd approach, including the frontend of JastAddJ. Each module is separated into two rows when there is a modular extension from Java version 1.4 to Java version 1.5. For cases where such an extension is unnecessary due to reused behavior, only numbers for version 1.4 are given. Besides size, we also show the number of JastAdd rules divided into different columns depending on rule type. For completeness, the size of a Helpers module, needed by the Control Flow, Liveness, Reaching Definition and Dead Assignment modules, is also included.

The total number of lines for the JastAdd analyses is 647. In comparison, the corresponding Soot implementation is 1308 lines of code, i.e., more than twice as large. This includes 186 for the dead assignment analysis, 481 for dataflow analysis, and 641 for control-flow including the handling of exceptions. We have not found it meaningful to compare with the implementation sizes of PMD and FindBugs, since the results they report are so different.

7 Related Work

Silver is a recent attribute grammar system with many similarities to JastAdd, but which does not support circular attributes. It has also been applied for declarative flow analysis [Wyk+07], but using a different approach than ours. In Silver, the specification language itself is extended to support the specification of control-flow and dataflow analysis. The actual dataflow analysis is not carried out by the attribute grammar system, but by an external model checking tool. This approach is motivated by the difficulty of declaratively specifying dataflow analysis on the same program representation as, for example, type analysis. No performance figures for this approach are reported. In contrast, we have shown how both control flow and dataflow can be specified in a concise way directly using the general attribute grammar features of JastAdd, in particular relying on the combination of reference attributes, circular attributes and collection attributes.

Farrow introduced circular attributes, and used liveness as a motivating example [Far86]. He builds on traditional attribute grammars without reference attributes, and does therefore not build any explicit control-flow graph. The dataflow analysis is instead defined directly in terms of the underlying syntax, with rules for each kind of statement.

Another declarative approach to dataflow analysis (both inter- and intra) is to use techniques based on logic programming and deductive databases, running queries on a database of facts extracted from the program code [Rep94]. Deductive database languages like Datalog have been used for interprocedural flow analyses of Java [WL04; BS09]. In this approach, the source program needs to be preprocessed, for example to resolve names, in order to extract the relevant facts. In contrast, the attribute grammar approach can be used seamlessly for all analysis after parsing. However, it should be pointed out that our current implementation concerns *intraprocedural* flow analysis only. Implementation of interprocedural flow analyses using reference attribute grammars is still future work.

Soot, [VR+99], is a framework for optimizing, analyzing, and annotating Java bytecode. The framework provides a set of inter- and intraprocedural program optimizations with a much wider scope than the analyses presented in this paper. Soot is based on several kinds of intermediate code representations, including typed three-address code, and provides seamless translations between the different representations. Java source code is first translated into one of these representations in which some high-level structure is lost. The control-flow and data-flow frameworks in Soot are indeed quite powerful with reasonably small APIs. A major difference, as compared to our approach, is that the Soot approach is not declarative and therefore relies on manual scheduling when combining analyses, or adding new analyses as new specializations of the framework.

Schäfer et al. have used a variant of our analyses modules in the implementation of experimental refactoring tools for Java. They report performance on par with industrial strength refactoring tools [Sch+08].

8 Conclusions

Control-flow and intraprocedural dataflow analysis is important for source-level tools like bug detectors and refactoring tools. Doing such analysis at the source level, rather than at the level of intermediate code, is desirable from a tool integration point of view. The downside of working at the source level is that it requires all language constructs to be taken into account, and that the analyses need to be extended when new language features are added.

In this paper, we have presented a new approach to source level control-flow and dataflow analysis, based on reference attribute grammars that are augmented with circular attributes and collection attributes. We argue that this provides an excellent foundation for implementing these analyses, leading to concise specifications that are close to text book definitions, and that are easy to extend modularly when the language evolves.

We have demonstrated that the approach works well for practical applications by implementing control flow, dataflow, and dead assignment analysis for Java, and comparing with Soot (a well known Java optimization framework), and with PMD and FindBugs (both well known tools for bug and code anomaly detection).

Our evaluation shows that JastAdd analyses are concise and easy to extend modularly. The JastAdd specification for Java 1.4 is only 627 lines for Java 1.4, and only a 20-line module is needed to extend the control-flow analysis to Java 5, and the other analyses can be reused as they are. In comparison, the corresponding Soot implementation is 1308 lines.

To evaluate correctness and precision, we compared the results of dead assignment analysis on a number of Java benchmark programs, the largest being over 130 000 lines of code. Due to its focus on optimization, Soot can be expected to have both high correctness and preciseness, and manual inspection of deviating results between the tools confirmed this. We found that the results from JastAdd and Soot were almost identical, with both finding a very small number of dead assignments that the other did not find. Both PMD and FindBugs found substantially fewer dead assignments, and PMD had many false positives.

Concerning performance, our JastAdd-based solution is between four and nine times faster than FindBugs, and at the same time more precise. The performance comparison between JastAdd and the other two tools, Soot and PMD, is less clear cut. While JastAdd is the fastest on most benchmarks, Soot and PMD find dead assignments also outside the candidate set we have tested. While we believe that most of these reports are due to constant propagation and internal optimizations of jimple code for Soot, and false positives for PMD, this would need to be manually verified.

We implemented two variants of dataflow analysis: liveness only, and liveness combined with reaching definitions. The difference between these variants was extremely small: adding the reaching definitions analysis accounted for merely three additional dead assignments detected in the four benchmark programs together,

and which none of the other tools detected. The performance cost was quite large, however, resulting in an analysis that was between two to five times slower.

There are several interesting ways to continue this work. One is to investigate more advanced interactive tool support that need precise intraprocedural dataflow analysis. For example, more advanced bug and code anomaly detectors. Another direction is to extend the work to interprocedural analyses, in particular to object-oriented call graph construction and interprocedural points-to analysis. We already have promising work in the direction of call graphs and simple whole program devirtualization analysis [Mag+09]. Because evaluation of reference attribute grammars is demand-driven, they should lend themselves to interprocedural analyses.

A third direction is to apply these results to analysis on intermediate code, and to develop declarative frameworks building SSA form and declarative implementation of related analyses.

9 Acknowledgements

We are very grateful to Max Schäfer for refactorings and improvements to the Control Flow Graph module.

References

[All70] Frances E. Allen. “Control flow analysis”. In: *Proceedings of a symposium on Compiler optimization*. Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19.

[App02] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[Aye+08] Nathaniel Ayewah et al. “Using Static Analysis to Find Bugs”. In: *IEEE Software* 25.5 (2008), pp. 22–29.

[Bla+06] Stephen M. Blackburn et al. “The DaCapo benchmarks: Java benchmarking development and analysis”. In: *OOPSLA*. Ed. by Peri L. Tarr and William R. Cook. Portland, Oregon, USA: ACM, 2006, pp. 169–190.

[Bla+08] Stephen M. Blackburn et al. “Wake up and smell the coffee: evaluation methodology for the 21st century”. In: *Communications of the ACM* 51.8 (2008), pp. 83–89.

[Boy05] John Tang Boyland. “Remote attribute grammars”. In: *Journal of the ACM* 52.4 (2005), pp. 627–687.

[BS09] Martin Bravenboer and Yannis Smaragdakis. “Strictly declarative specification of sophisticated points-to analyses”. In: *OOPSLA*. Ed. by Shail Arora and Gary T. Leavens. ACM, 2009, pp. 243–262.

[Cho+99] Jong-Deok Choi et al. “Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs”. In: *PASTE*. Ed. by William G. Griswold and Susan Horwitz. ACM, 1999, pp. 21–31.

[Cop05] Tom Copeland. *PMD applied*. Centennial Books, 2005.

[EH05] Torbjörn Ekman and Görel Hedin. “Modular Name Analysis for Java Using JastAdd”. In: *GTTSE*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science. Springer, 2005, pp. 422–436.

[EH07a] Torbjörn Ekman and Görel Hedin. “The JastAdd extensible Java compiler”. In: *OOPSLA*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[EH07b] Torbjörn Ekman and Görel Hedin. “The JastAdd system - modular extensible compiler construction”. In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[Far86] Rodney Farrow. “Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars”. In: *SIGPLAN Symposium on Compiler Construction*. Ed. by Richard L. Wexelblat. ACM, 1986, pp. 85–98.

[Gos+96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[Ing86] Daniel H. H. Ingalls. “A Simple Technique for Handling Multiple Polymorphism.” In: *OOPSLA*. Ed. by Norman K. Meyrowitz. ACM, 1986, pp. 347–349.

[Jas] *JastAdd*. `http://jastadd.org`.

[Knu68] Donald E. Knuth. “Semantics of context-free languages”. In: *Journal Theory of Computing Systems* 2.2 (1968), pp. 127–145.

[MH07] Eva Magnusson and Görel Hedin. “Circular Reference Attributed Grammars - their Evaluation and Applications”. In: *Science of Computer Programming* 68.1 (2007), pp. 21–37.

[Mag+09] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. “Demand-driven evaluation of collection attributes”. In: *Automated Software Engineering* 16.2 (2009), pp. 291–322.

[Rep94] Thomas W. Reps. “Solving Demand Versions of Interprocedural Analysis Problems”. In: *CC*. Ed. by Peter Fritzson. Vol. 786. Lecture Notes in Computer Science. Springer, 1994, pp. 389–403.

[Sch+08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. “Sound and extensible renaming for Java”. In: *OOPSLA*. Ed. by Gail E. Harris. ACM, 2008, pp. 277–294.

[VR+99] Raja Vallée-Rai et al. “Soot - a Java bytecode optimization framework”. In: *CASCON*. Ed. by Stephen A. MacKay and J. Howard Johnson. IBM, 1999, p. 13.

[Vog+89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. “Higher-Order Attribute Grammars”. In: *PLDI*. Ed. by Richard L. Wexelblat. ACM Press, 1989, pp. 131–145.

[WL04] John Whaley and Monica S. Lam. “Cloning-based context-sensitive pointer alias analysis using binary decision diagrams”. In: *PLDI*. Ed. by William Pugh and Craig Chambers. ACM, 2004, pp. 131–144.

[Wyk+07] Eric Van Wyk et al. “Attribute Grammar-Based Language Extensions for Java”. In: *ECOOP*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 575–599.

AUTOMATED SELECTIVE CACHING FOR REFERENCE ATTRIBUTE GRAMMARS

PAPER III

Abstract

Reference attribute grammars (RAGs) can be used to express semantics as super-imposed graphs on top of abstract syntax trees (ASTs). A RAG-based AST can be used as the in-memory model providing semantic information for software language tools such as compilers, refactoring tools, and meta-modeling tools. RAG performance is based on dynamic attribute evaluation with caching. Caching all attributes gives optimal performance in the sense that each attribute is evaluated at most once. However, performance can be further improved by a selective caching strategy, avoiding caching overhead where it does not pay off. In this paper we present a profiling-based technique for automatically finding a good cache configuration. The technique has been evaluated on a generated Java compiler, compiling programs from the Jacks test suite and the DaCapo benchmark suite.

1 Introduction

Reference attribute grammars (RAGs) [Hed00] provide a means for describing semantics as super-imposed graphs on top of an abstract syntax tree (AST) using *reference attributes*. Reference attributes are defined by functions and may have

Emma Söderberg and Görel Hedin.
Lecture Notes in Computer Science, 2011, Vol. 6563, pp. 2–21, Springer Berlin/Heidelberg.

values referring to distant nodes in the AST. RAGs have been shown useful for the generation of many different software language tools, including Java compilers [Wyk+07; EH07a], Java extensions [Hua+08; Ibr+09; Pto], domain-specific language tools [Jou+08; Åke+10], refactoring tools [Sch+08], and meta-modeling tools [Bür+10]. Furthermore, they are being used in an increasing number of meta-compilation systems [EH07b; Wyk+10; Slo+10; Kat+10a].

RAG evaluation is based on a dynamic algorithm where attributes are evaluated on demand, and their values are cached (memoized) for obtaining optimal performance [Jou84]. Caching all attributes gives optimal performance in the sense that each attribute is evaluated at most once. However, caching has a cost in both compilation time and memory consumption, and caching does not pay off in practice for all attributes. Performance can therefore be improved by *selective caching*, caching only a subset of all attributes, using a *cache configuration*. But determining a good cache configuration is not easy to do manually. It requires a good understanding of how the underlying attribute evaluator works, and a lot of experience is needed to understand how different input programs can affect the caching inside the generated language tool. Ideally, the language engineer should not need to worry about this, but let the system compute the configuration automatically.

In this paper we present a profiling-based approach for automatically computing a cache configuration. The approach has been evaluated experimentally on a generated compiler for Java [EH07a], implemented using JastAdd [EH07b], a meta-compilation system based on RAGs. We have profiled this compiler using programs from Jacks (a compiler test suite for Java) [Jac] and DaCapo (a benchmark suite for Java) [Bla+06]. Our evaluation shows that it is possible to obtain an average compilation speed-up of 20% while only using the profiling results from one application with a fairly low attribute coverage of 67%. The contributions of this paper include the following:

- A profiling-based approach for automatic selective caching of RAGs.
- An implementation of the approach integrated with the JastAdd meta-compilation system.
- An evaluation of the approach, comparing it both to full caching (caching all attributes) and to an expert cache configuration (produced manually).

The rest of this paper is structured as follows. Section 2 gives background on reference attribute grammars and their evaluation, explaining the JastAdd caching scheme in particular. Section 3 introduces the concept of an AIG, an attribute instance graph with call information, used as the basis for the caching analysis. Section 4 introduces our technique for computing a cache configuration. Section 5 presents an experimental evaluation of the approach. Section 6 discusses related work, and Section 7 gives a conclusion along with future work.

2 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) [Hed00], extend Knuth-style attribute grammars [Knu68] by allowing attributes to be references to nodes in the abstract syntax tree (AST). This is a powerful notion because it allows the nodes in an AST to be connected into the graphs needed for compilation. For example, reference attributes can be used to build a type graph connecting subclasses to superclasses [EH05], or a control-flow graph between statements in a method [NN+09a]. Similar extensions to attribute grammars include Poetzsch-Heffter's occurrence algebras [PH97] and Boyland's remote attribute grammars [Boy05].

In attribute grammars, attributes are defined by equations in such a way that for any attribute instance in any possible AST, there is exactly one equation defining its value. The equations can be viewed as side-effect-free functions which make use of constants and of other attribute values.

In RAGs, it is allowed for an equation to define an attribute by following a reference attribute and accessing its attributes. For example, suppose node n_1 has attributes a and b, where b is a reference to a node n_2, and that n_2 has an attribute c. Then a can be defined by an equation as follows:

$$a = b.c$$

For Knuth-style attribute grammars, dependencies are restricted to attributes in parents or children. But the use of references gives rise to non-local dependencies, i.e., dependencies that are independent of the AST hierarchy: a will be dependent on b and c, where the dependency on b is local, but the dependency on c is non-local: the node n_2 referred to by b could be anywhere in the AST. The resulting attribute dependency graph cannot be computed without actually evaluating the reference attributes, and it is therefore difficult to statically precompute evaluation orders based on the grammar alone. Instead, evaluation of RAGs is done using a simple but general dynamic evaluation approach, originally developed for Knuth-style attribute grammars, see [Jou84]. In this approach, attribute access is replaced by a recursive call which evaluates the equation defining the attribute. To speed up the evaluation, the evaluation results can be cached (memoized) in order to avoid evaluating the equation for a given attribute instance more than once. Caching all attributes results in optimal evaluation in that each attribute instance is evaluated at most once. Because this evaluation scheme does not require any pre-computed analysis of the attribute dependencies, it works also in the presence of reference attributes.

Caching is necessary to get practical compiler performance for other than the tiniest input programs. But caching also implies an overhead. Compared to caching all attributes, *selective caching* may improve performance, both concerning time and memory.

Non-cached version:

```
class Node {
  A a() {
    return b().c();
  }
}
```

Cached version:

```
class Node {
  boolean a_cached = false;
  A a_value;
  A a() {
    if (! a_cached) {
      a_value = b().c();
      a_cached = true;
    }
    return a_value;
  }
}
```

Figure 1: Caching scheme for non-parameterized attributes

2.1 The JastAdd Caching Scheme

In JastAdd, the dynamic evaluation scheme is implemented in Java, making use of an object-oriented class hierarchy to represent the abstract grammar. Attributes are implemented by method declarations, equations by method implementations, and attribute accesses by method calls. Caching is decided per attribute declaration, and cached attribute values are stored in the AST nodes using two Java fields: one field is a flag keeping track of if the value has been cached yet, and another field holds the value. Figure 1 shows the implementation of the equation $a = b.c$, both in a non-cached and a cached version. It is assumed that a is of type A. The example shows the implementation of a so called *synthesized attribute*, i.e., an attribute defined by an equation in the node itself. The implementation of a so called *inherited attribute*, defined by an equation in an ancestor node, is slightly more involved, but uses the same technique for caching. The implementation in Figure 1 is also simplified as compared to the actual implementation in JastAdd which takes into account, for example, circularity checking. These differences are, however, irrelevant to the caching problem.

This caching scheme gives a low overhead for attribute accesses: a simple test on a flag. On the other hand, the caching pays off only after at least one attribute instance has been accessed at least twice. Depending on the cost of the value computation, more accesses than that might be needed for the scheme to pay off.

JastAdd allows attributes to have parameters. A *parameterized attribute* has an unbounded number of values, one for each possible combination of parameter values. To cache accessed values, the flag and value fields are replaced by a map where the actual parameter combination is looked up, and the cached values are stored. This is a substantially more costly caching scheme, both for accessing attributes and for updating the cache, and more accesses per parameter combination will be needed to make it pay off.

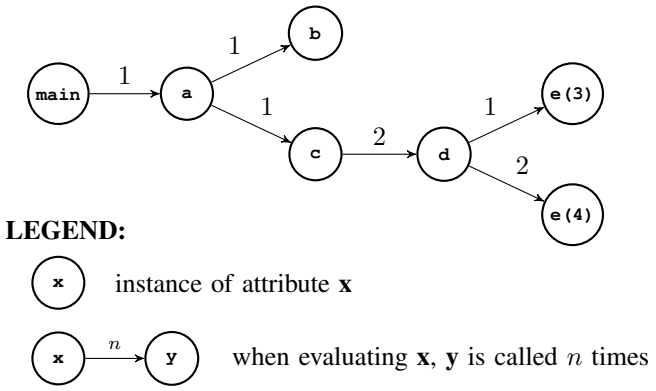


Figure 2: Example AIG

3 Attribute Instance Graphs

In order to decide which attributes that may pay off to cache, we build a graph that captures the attribute dependencies in an AST. This graph can be built by instrumenting the compiler to record all attribute accesses during a compilation. By analyzing such graphs for representative input programs, we would like to identify a number of attributes that are likely to improve the compilation performance if left uncached. We define the *attribute instance graph* (AIG) to be a directed graph with one vertex per attribute instance in the AST. The AIG has an edge (a_1, a_2) if, during the evaluation of a_1, there is a direct call to a_2, i.e., indirect calls via other attributes do not give rise to edges. Each edge is labeled with a *call count* that represents the number of calls. This count will usually be 1, but in an equation like $c = d + d$, the count on the edge (c, d) will be 2, since d is called twice to compute c.

The main program of the compiler is modeled by an artificial vertex `main`, with edges to all the attribute instances it calls. This may be many or few calls, depending on how the main program is written.

To handle parameterized attributes, we represent each accessed combination of parameter values for an attribute instance by a vertex. For example, the evaluation of the equation $d = e(3) + e(4) + e(4)$ will give rise to two vertices for e, one for $e(3)$ and one for $e(4)$. The edges are, as before, labeled by the call counts, so the edge $(d, e(3))$ is labeled by 1, and the edge $(d, e(4))$ by 2, since it is called twice. Figure 2 shows an example AIG for the following equations:

$$\begin{aligned} a &= b.c \\ c &= d + d \\ d &= e(3) + e(4) + e(4) \end{aligned}$$

```
abstract Expr;
Use : Expr ::= ...;
Literal : Expr ::= ...;
AddExpr : Expr ::=
    e1:Expr e2:Expr;

Decl ::= Type ... ;

abstract Type;
Integer : Type;
Unknown : Type;

...
```

```
syn Type Expr.type();
syn Type Decl.type() = ...;

eq Literal.type() =
  stdTypes().integer();
eq Use.type() = decl().type();
eq AddExpr.type() =
  (left.type().sameAs(right.type()) ?
  left.type() : stdTypes.unknown();

syn Decl Use.decl() = lookup(...);
inh Decl Use.lookup(String name);
inh Type Expr.stdTypes();

syn boolean Type.sameAs(Type t) = ...;
...
```

Figure 3: Example JastAdd attribute grammar

and where it is assumed that a is called once from the main program.

3.1 An Example Grammar

Figure 3 shows parts of a typical JastAdd grammar for name and type analysis. The abstract grammar rules correspond to a class hierarchy. For example, `Use` (representing a use of an identifier) is a subclass of `Expr`. The first attribution rule:

```
syn Type Expr.type();
```

declares a synthesized attribute of type `Type`, declared in `Expr` and of the name `type`. All nodes of class `Expr` and its subclasses will have an instance of this attribute. Different equations are given for it in the different subclasses of `Expr`. For example, the equation

```
eq Use.type() = decl().type();
```

says that for a `Use` node, the value of `type` is defined to be `decl().type()`. The attribute `decl()` is another attribute in the `Use` node, referring to the appropriate declaration node, possibly far away from the `Use` node in the AST. The `decl()` attribute is in turn defined using a parameterized attribute `lookup`, also in the `Use` node. The `lookup` attribute is an inherited attribute, and the equation for it is in an ancestor node of the `Use` node (not shown in the grammar). For more information on name and type analysis in RAGs, see [EH05].

Figure 4 shows parts of an attributed AST for the grammar in Figure 3. The example program contains two declarations: `"int a"` and `"int b"`, and two add

expressions: `"a + b"` and `"a + 5"`. For the `decl` attributes of `Use` nodes, the reference values are shown as arrows pointing to the appropriate `Decl` node. Similarly, the `type` attributes of `Decl` nodes have arrows pointing to the appropriate `Type` node. The nodes have been labeled `A`, `B`, and so on, for future reference.

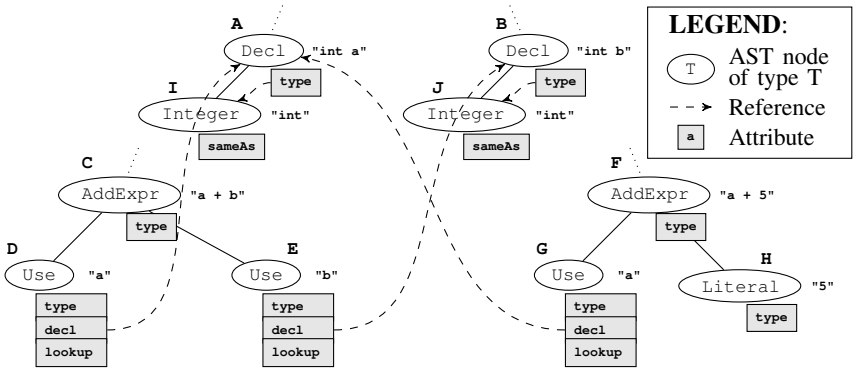


Figure 4: An example attributed AST

Figure 5 shows parts of the AIG for this example. In the AIG we have grouped together all instances of a particular attribute declaration, and labeled each attribute instance with the node to which it belongs. For instance, since the node `D` has the three attributes (`type`, `decl`, and `lookup`), there are three vertices labeled `D` in the AIG. For parameterized attribute instances, there is one vertex per actual parameter combination, and their values are shown under the vertex. For instance, the `sameAs` attribute for `I` is called with two different parameters: `J` and `K`, giving rise to two vertices. (`K` is a node representing integer literal types and is not shown in Figure 4.) All call counts in the AIG are 1 and have therefore been omitted.

4 Computing a Cache Configuration

Our goal is to automatically compute a good cache configuration for a RAG specification. A cache configuration is simply the set of attributes configured to be cached. Among the different attribute kinds, there are some that will always be cached, due to properties of the kind. For example, circular attributes [Far86], which may depend on themselves, and non-terminal attributes (NTAs) [Vog+89], which may have ASTs as values. There is no cache decision to make for these attributes, i.e., they are *unconfigurable*. We let PRE denote the set of unconfigurable attributes. Since the attributes in the PRE set are always cached, we exclude them from remaining definitions in this paper. We let ALL denote the remaining set of *configurable* attributes. This ALL set can further be divided into two disjoint sets

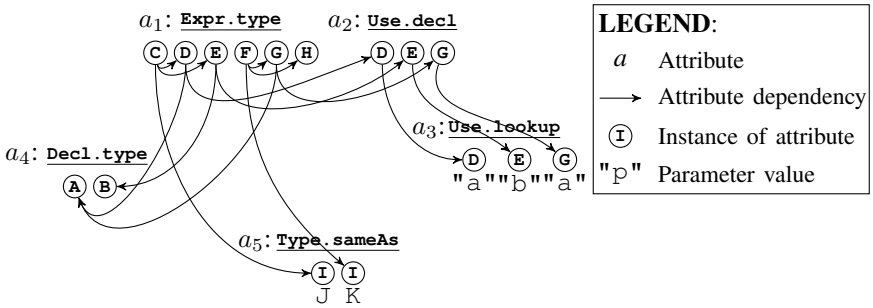


Figure 5: Parts of the AIG for the example

PARAM and NONPARAM, for parameterized and non-parameterized attributes respectively. For the rest of this paper we will refer to configurable attributes when we write attributes.

As a basis for our computation, we do profiling runs of the compiler on a set of test programs, producing the AIG for each program. These runs are done with all attributes cached, allowing us to use reasonably large test programs, and making it easy to compute the AIG which reflects the theoretically optimal evaluation with each attribute instance evaluated at most once. We will refer to these test programs as the *profiling input* denoted by the set P. Further, a certain profiling input ($p \in$ P) will, depending on its structure, require that a certain number of attributes are evaluated. We call this set of attributes the USED_p set. However, it cannot be assumed that a single profiling input uses all attributes. We define the set of unused attributes for a profiling input p as follows:

$$\text{UNUSED}_p = \text{ALL} \setminus \text{USED}_p \tag{1}$$

4.1 The ONE set

The calls label on the edges in the AIG reflects the number of attribute calls in a fully cached configuration. To find out if a certain attribute is worth uncaching, we define $\mathrm{extra_evals}(a_i)$, i.e., the number of extra evaluations of the attribute instance a_i that will be done if the attribute a is not cached:

$$\mathrm{extra_evals}(a_i) = \begin{cases} \mathrm{calls}(a_i) - 1, & \text{if } a \in \text{NONPARAM} \\ \sum_{c \in \mathrm{params}(a_i)} (\mathrm{calls}(c) - 1), & \text{if } a \in \text{PARAM} \end{cases} \tag{2}$$

where $\mathrm{params}(a_i)$ is the set of vertices in the AIG representing different parameter combinations for the parameterized attribute instance a_i. The number of extra evaluations is a measure of what is lost by not caching an attribute. The total number of extra evaluations for an attribute a is simply the sum of the extra evaluations

of all its instances:

$$\text{extra_evals}(a) = \sum_{a_i \in \text{I}_{\text{called}}(a)} \text{extra_evals}(a_i); \tag{3}$$

where $\text{I}_{\text{called}}(a)$ is the set of attribute instances of a that are called at least once. Of particular interest is the set of attributes for which all instances are called at most once. These should be good candidates to leave uncached since they do not incur any extra evaluations for a certain profiling input (p). We call this set the ONE_p set, and for a profiling input p it is constructed as follows:

$$\text{ONE}_p = \{a \in \text{USED}_p | \text{extra_evals}(a) = 0\} \tag{4}$$

The $\text{USED}_p \setminus \text{ONE}_p$ set contains the remaining attributes in the AIG, i.e., the attributes which may gain from being cached, depending on the cost of their evaluation.

4.2 Selecting a good profiling input

To obtain a good cache configuration, it is desirable to use profiling input that is realistic in its attribute usage, and that has a high *attribute coverage*, i.e., as large a USED_p set as possible. We define the attribute coverage (in percent) for a profiling input, $p \in \text{P}$, as follows:

$$\text{coverage}(p) = (|\text{USED}_p|/|\text{ALL}|) * 100 \tag{5}$$

Furthermore, for tools used in an interactive setting with continuous compilation of potentially erroneous input, it is important to also take incorrect programs into account. To help fulfill these demands, different profiling inputs can be combined. In particular, a compilation test suite may give high attribute coverage and test both correct and erroneous programs. But test suites might contain many small programs that do not use the attributes in a realistic way. In particular, attributes which most likely should be in the $\text{USED}_p - \text{ONE}_p$ set for an average application may end up in the ONE_p sets of the test suite programs because these are small. By combining the test suite with a large real program, better results may be obtained. Still, even with many applications and a full test suite, it may be hard to get full coverage. For example, there may be semantic checks connected to uncommon language constructs and, hence, attributes rarely used.

4.3 Choosing a cache configuration

In constructing a good cache configuration we want to consider the USED_p, UNUSED_p, ONE_p and ALL sets. From these sets we can experiment with two interesting configurations:

$$\text{ALLONE}_p = \text{ALL} \setminus \text{ONE}_p \tag{6}$$

$$\text{USEDONE}_p = \text{USED}_p \setminus \text{ONE}_p \tag{7}$$

Presumably, the first configuration, which includes the UNUSED_p set, will provide robustness for cases where the profiling input is insufficient, i.e., the USED_p set is too small. In contrast, the second configuration may provide better performance in that it uses less memory for cases where the profiling input is sufficient.

4.4 Combining cache configurations

In order to combine the results of several profiling inputs, for example, A, B and C in P, we need to consider each of the resulting sets USED_p and ONE_p. One attribute might be used in the compilation of program A but not in the compilation of program B. If an attribute is used in both B and C, it might belong to ONE_B, but not to ONE_C, and so on. We want to know which attributes that end up in a total ONE_P set for all profiling inputs ($p \in \text{P}$), i.e., the attributes that are used by at least one profiling input, but that, if they are used by a particular profiling input, they are in its ONE_p set. More precisely:

$$\text{ONE}_\text{P} = \bigcup_{p \in \text{P}} \text{USED}_p \setminus \bigcup_{p \in \text{P}} (\text{USED}_p \setminus \text{ONE}_p) \tag{8}$$

These attributes should be good candidates to be left uncached. By including or excluding the UNUSED_P set, we can now construct the following combined cache configuration, for a profiling input set P, in analogy to Definition 6 and Definition 7:

$$\text{ALLONE}_\text{P} = \text{ALL} \setminus \text{ONE}_\text{P} \tag{9}$$

$$\text{USEDONE}_\text{P} = \text{USED}_\text{P} \setminus \text{ONE}_\text{P} \tag{10}$$

5 Evaluation

To evaluate our approach we have applied it to the frontend of the Java compiler JastAddJ [EH07a]. This compiler is specified with RAGs using the JastAdd system. We have profiled the compilation with one or several Java programs as profiling input, and used the resulting AIGs to compute different cache configurations. We have divided our evaluation into the following experiments:

Experiment A: The effects of no caching

Experiment B: The effects of profiling using a compiler test suite

Experiment C: The effects of profiling using a benchmark application

Experiment D: The effects of combining B and C

Throughout our experiments we use the results of caching all attributes and the results of using a manual configuration, composed by an an expert, for comparison.

5.1 Experimental setup

All measurements were run on a high-performing computer with two Intel Xeon Quad Core @ 3.2 GHz processors, a bus speed of 1.6 GHz and 32 GB of primary memory. The operating system used was Mac OS X 10.6.2 and the Java version was Java 1.6.0._15.

The JastAddJ compiler The frontend of the JastAddJ compiler (for Java version 1.4 and 1.5) has an ALL set containing 740 attributes and a PRE set containing 47 unconfigurable attributes (14 are circular and 33 are non-terminal attributes). The compiler comes with a configuration MANUAL, with 281 attributes manually selected for caching by the compiler implementor, an expert on RAGs, making an effort to obtain as good compilation speed as possible. The compiler performs within a factor of three as compared to the standard javac compiler, which is good considering that it is generated from a specification. MANUAL is clearly an expert configuration, and it cannot be expected that a better one can be obtained manually.

Measuring of performance The JastAddJ compiler is implemented in Java (generated from the RAG specification), so measuring its compilation speed comes down to measuring the speed of a Java program. This is notoriously difficult, due to dynamic class loading, just-in-time compilation and optimization, and automatic memory management [Bla+06]. To eliminate as many of these factors as possible, we use the multi-iteration approach suggested in [Bla+08]. We start by warming up the compiler with a number of non-measured compilations (5), thereby allowing class loading and optimization of all relevant compiler code to take place, in order to reach a steady state. Then we turn off the just-in-time compilation and run a couple of extra unmeasured compilations (2) to drain any JIT work queues. After that we run several (20) measured compilation runs for which we compute 95% confidence intervals. In addition to this, we start each measured run with a forced garbage collection (GC) in order to obtain as similar conditions as possible for each run. Memory usage is measured by checking of available memory in the Java heap after each forced GC call and after each compilation. The memory measurements are also given with a 95% confidence interval. We present a summary of these results in Figure 7, Figure 8, Figure 9 and Figure 10. All results have a confidence interval of less than $\pm 0.03\%$. These intervals have not been included in the figures since they would be barely visible with the resolution we need to use. A complete list of results are available on the web [Söd10].

Profiling and test input As a basis for profiling input, we use the Jacks test suite [Jac], the DaCapo benchmark suite [Bla+06; Dac] and a small hello world program. We use 4170 tests from the Jacks suite, checking frontend semantics, and the following applications from the DaCapo suite (lines of code (LOC)):

ANTLR: an LL(k) parser generator (ca 35 000 LOC).

Bloat: a program for optimization and analysis of Java bytecode (ca 41 000 LOC).

Chart: a program for plotting of graphs and rendering of PDF files (ca 12 000 LOC).

FOP: parses XSL-FO files and generates PDF files (ca 136 000 LOC).

HsqlDb: a database application (ca 138 000 LOC).

Jython: a Python interpreter (ca 76 000 LOC).

Lucene: a program for indexing and searching of large text corpuses (ca 87 000 LOC).

PMD: a Java bytecode analyzer for a range of source code problems (ca 55 000 LOC).

Xalan: a program for transformation of XML documents into HTML (ca 172 000 LOC).

In our experiments, we use different combinations of these applications and tests as profiling input. We will denote these profiling input sets as follows:

J: The Jacks test suite profiling input set

D: The DaCapo benchmarks profiling input set

"APP": The benchmark APP of the DaCapo benchmarks. For example, ANTLR means the ANTLR benchmark.

HELLO: The hello world program

We combine these profiling input sets in various ways, for example, the profiling input set J + ANTLR means we combine the Jacks suite with the benchmark ANTLR. Finally, as test input for performance testing we use the benchmarks from the DaCapo suite and the hello world program.

Cache configurations We want to compare the results of using the cache configurations defined in Section 4. In addition, the JastAddJ specification comes with a manual cache configuration (MANUAL) which we want to compare to. We also have the option to cache all attributes (ALL), or to cache no attributes (NONE):

MANUAL: This expert configuration is interesting to compare to, as it would be nice if we could obtain similar results with our automated methods.

ALL: The ALL configuration is interesting as it is easily obtainable and robust with respect to performance: there is no risk that a particular attribute will be evaluated very many times for a particular input program, and thereby degrade performance.

NONE: The least possible configuration is interesting as it provides a lower bound on the memory needed during evaluation. However, this configuration will in general be useless in practice, leading to compilation times that increase exponentially with program size.

From each profiling input set P, we compute USED_P, and ONE_P, and construct the configurations USEDONE_P and ALLONE_P (according to Definition 9 and 10):

USEDONE_P: This is an interesting cache configuration as it should give good performance by avoiding caching of unused attributes and attributes used only once by P. The obvious risk with this configuration is that other programs might use attributes unused by P, causing performance degradation. There is also a risk that the attributes in the ONE_P set may belong to another program's $\text{USED} \setminus \text{ONE}$ set, also causing a performance degradation. However, if attributes in ONE_P are only used once in a typical application, they are likely to be used only once in most applications.

ALLONE_P: This configuration is more robust than the USEDONE_P configuration in that also unused attributes are cached, which prevents severe performance degradation for those attributes.

Attribute coverage Figure 6 gives an overview of the $\text{USED}_\text{P} \setminus \text{ONE}_\text{P}$, ONE_P and UNUSED_P sets for the profiling inputs from the DaCapo suite. The figure also includes the combined sets for DaCapo (D) and Jacks (J). Not surprisingly, Hello World has the lowest attribute coverage. Still, it covers as much as 29%. The high coverage is due to analysis of standard library classes needed to compile the program. The combined results for the DaCapo suite and two of its applications have better or the same coverage as the Jacks suite, i.e., the USED_J set of Jacks *does not* enclose the USED_D set of DaCapo neither does it have an empty UNUSED_J. These observations are interesting since they might indicate that additional tests could be added to Jacks. We can also note that the attribute coverage is not directly proportional to the size of an application, as shown by PMD and Lucene which both are smaller than Xalan and FOP in regard to LOC. This may not be surprising since the actual attribute coverage is related to the diversity of language constructs in an application rather than to the application size.

5.2 Experiment A: The effects of no caching

To compare the behavior of *no caching* with various other configurations, we profiled a simple Hello World program (HELLO) and then tested performance by compiling the same program using the configurations ALL, NONE, MANUAL, $\text{USEDONE}_\text{HELLO}$ and $\text{ALLONE}_\text{HELLO}$. The results are shown in Figure 7. It is clear from these results that the minimal NONE configuration is not a good configuration, not even on this small test program. Even though it provides excellent

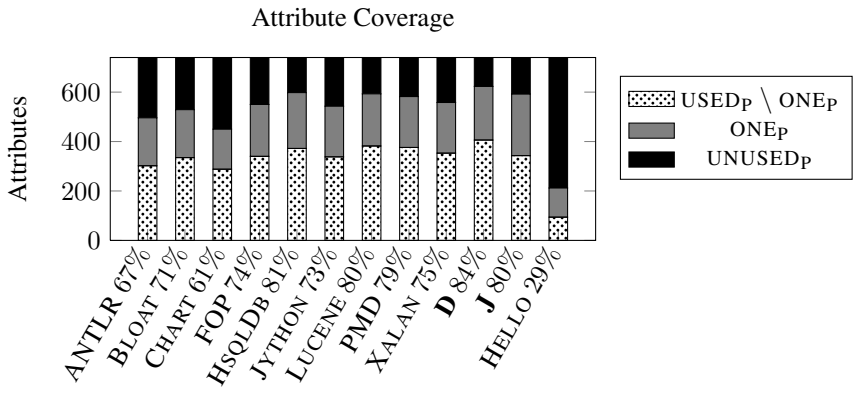


Figure 6: Attribute coverage for the benchmarks in the DaCapo suite, the combined coverage for all the DaCapo applications (D), the combined coverage for the programs in the Jacks suite (J) and for a hello world program. The attribute coverages are given next to the names of the application/combination.

memory usage, the compilation time is more than twice as slow as any of the other configurations. For a larger application the NONE configuration would be useless.

5.3 Experiment B: The effects of profiling using a compiler test suite

To show the effects of using a compiler test suite we profiled the compilation of the Jacks suite and obtained the two configurations USEDONE$_J$ and ALLONE$_J$. We then measured performance when compiling the DaCapo benchmarks using these configurations. The results are shown in Figure 8 and are given as percent in relation to the compilation time and memory usage of the ALL configuration[1]. The results for the MANUAL configuration are included for comparison.

Clearly, the MANUAL configuration performs better with regard to both compilation time and memory usage, with an average compilation time / memory usage of 75% / 47% in relation to the ALL configuration. The average results for USEDONE$_J$ is 83% / 67%. The ALLONE$_J$ configuration has the same average compilation time 83% / 72%, but higher average memory usage. It is interesting to note that USEDONE$_J$ is robust enough to handle the compilation of all the DaCapo benchmarks. So it seems that the Jacks test suite has a sufficiently large coverage, i.e., we can use the USEDONE$_J$ configuration rather than the ALLONE$_J$

[1]The absolute average results for ALL are the following: Antlr (1.462s/0.270Gb), Bloat (1.995s/0.339Gb), Chart (0.928s/0.177Gb), FOP (8.328s/1.362Gb), HsqlDb (6.054s/1.160Gb), Jython (3.257s/0.611Gb), Lucene (4.893s/0.930Gb), PMD (3.921s/0.691Gb), Xalan (6.606s/1.141Gb)

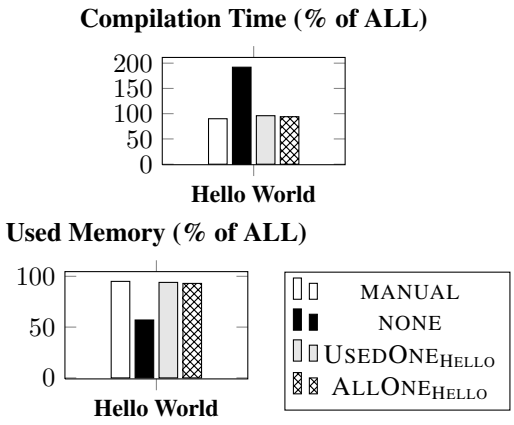


Figure 7: Results from Experiment A: Compilation of Hello World using static configurations along with configurations obtained using Hello World (HELLO) as profiling input. The average compilation time / memory usage when compilating with the ALL configuration were 50.0 ms / 14.7 kb. The corresponding values for the NONE configuration were 95.9 ms / 8.4 kb.

configuration. In doing so, we can use less memory with the same performance and robustness.

5.4 Experiment C: The effects of profiling using a benchmark program

To show the effects of using benchmarks we profiled using each of the DaCapo benchmarks obtaining the USEDONE and ALLONE configurations for each benchmark. We also combined the profiling results for all the benchmarks to create the combined configurations USEDONE_D and ALLONE_D. We then measured performance when compiling the DaCapo benchmarks using these configurations. A selected set of the results are shown in Figure 9, including the combined results and the best USEDONE and ALLONE configurations from the individual benchmarks. All results are given as percent in relation to the compilation time and memory usage of the ALL configuration. The results for the MANUAL set are also included for comparison. Note that not all results are shown in Figure 9. Two of the excluded configurations $\text{USEDONE}_\text{ANTLR}$ and $\text{USEDONE}_\text{CHART}$ performed worse than full caching (ALL). These results validate the concern that the USEDONE_p configuration would have robustness problems for insufficient profiling input. In this case, neither ANTLR nor Chart were sufficient as profiling inputs on their own. We can

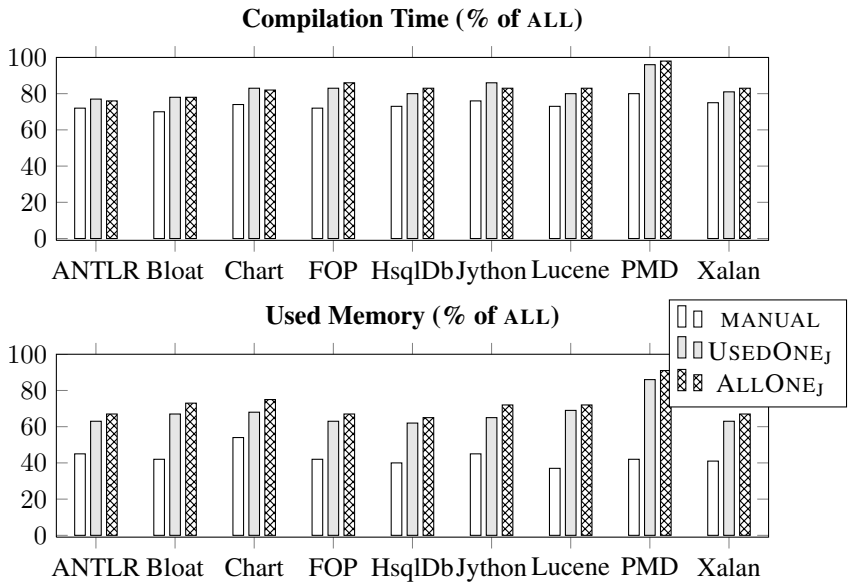


Figure 8: Results from Experiment B: Compilation of DaCapo benchmarks using configurations from the Jacks suite. All results are given in relation to the compilation time and memory usage of the ALL configuration and results for MANUAL are included for comparison.

note that these two applications have the least coverage among the applications from the DaCapo suite (67% for ANTLR and 61% for Chart).

The USEDONE configurations

The USEDONE configurations for ANTLR and Chart perform worse than the ALL configuration for several of the applications in the DaCapo benchmarks: FOP, Lucene and PMD. The remaining USEDONE configurations can be sorted with regard to percent of compilation time, calculated as the geometric mean of the DaCapo benchmark program compilation times (each in relation to the ALL configuration), as follows:

80%: $\text{USEDONE}_{\text{BLOAT}}$ (mem. 62%)

82%: $\text{USEDONE}_{\text{FOP}}$ (mem. 63%), $\text{USEDONE}_{\text{XALAN}}$ (mem.66%)

83%: $\text{USEDONE}_{\text{HSQLDB}}$ (mem. 68%), $\text{USEDONE}_{\text{JYTHON}}$ (mem. 65%), $\text{USEDONE}_{\text{LUCENE}}$ (mem. 68%), $\text{USEDONE}_{\text{PMD}}$ (mem. 66%)

84%: $\text{USEDONE}_{\text{D}}$ (mem. 70%)

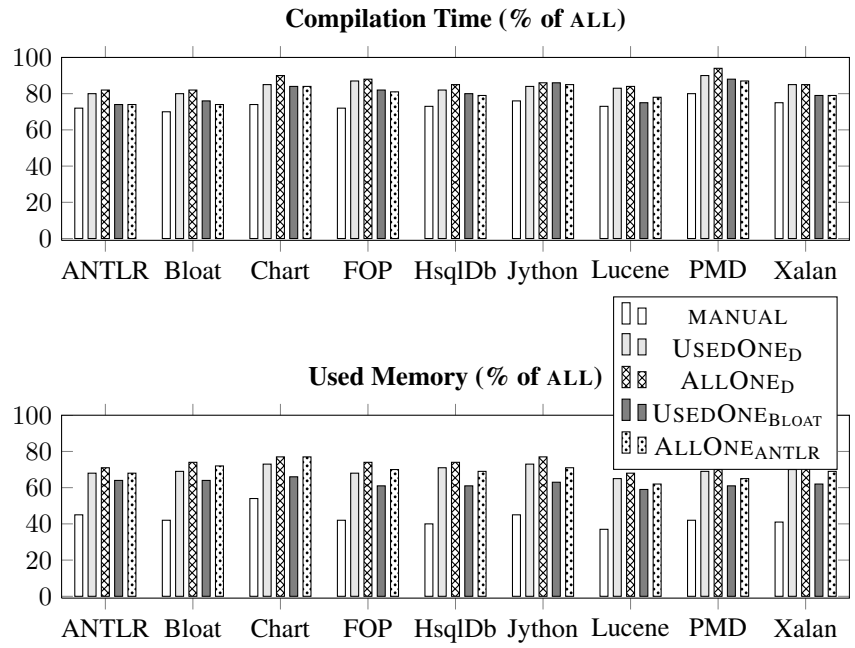


Figure 9: Results from Experiment C: Compilation of DaCapo benchmarks using configurations from the DaCapo benchmarks. All results are shown as compilation time and memory usage as percent of the results for the ALL configuration and results for MANUAL are included for comparison.

These results indicate that a certain coverage is needed in order to obtain a robust USEDONE configuration. It is also interesting to note that the combined USEDONE$_D$ configuration for DaCapo performs the worst (except for the non-robust configurations). One possible explanation to this performance might be that some attributes ending up in the USEDONE$_D$ set might be used rarely or not at all in several compilations. Still, these attributes are cached which leads to more memory usage.

The ALLONE configurations

The ALLONE configurations generally perform worse than the USEDONE configurations. This result might be due to the fact that these configurations include unused attributes for robustness. However, this strategy for robustness pays off in that all ALLONE configurations become robust, i.e., they compile all the DaCapo benchmarks faster than the ALL configuration. The ALLONE configurations can

be sorted as follows, with regard to percent of compilation time:

80%: ALLONE$_{\text{ANTLR}}$ (mem. 69%)

82%: ALLONE$_{\text{BLOAT}}$ (mem. 69%), ALLONE$_{\text{FOP}}$ (mem. 69%)

83%: ALLONE$_{\text{CHART}}$ (mem. 71%)

84%: ALLONE$_{\text{HSQLDB}}$ (mem. 73%), ALLONE$_{\text{JYTHON}}$ (mem. 70%), ALLONE$_{\text{XALAN}}$ (mem. 70%)

85%: ALLONE$_{\text{LUCENE}}$ (mem. 73%), ALLONE$_{\text{PMD}}$ (mem. 71%)

86%: ALLONE$_{\text{D}}$ (mem. 73%)

These results indicate that a profiled application does not necessarily need to be large, or have the best coverage, for the resulting configuration to provide good performance. The best individual ALLONE configuration is obtained from profiling ANTLR which is remarkable since ANTLR has the next lowest attribute coverage, while the combined ALLONE$_{\text{D}}$ configuration for DaCapo performs the worst on average. This result might be due to the fact that the combined configuration caches attributes that might be in the ONE set of an individual application. This fact is also true for several of the individual configurations but apparently the complete combination takes the edge off the configuration.

5.5 Experiment D: The effects of combining B and C

To show the effects of profiling using a compiler test suite (B) together with profiling a benchmark program (C) we combine the cache configurations from experiment B and C. We then measured performance when compiling the DaCapo benchmarks using these configurations. A selected set of the results are shown in Figure 10, including the fully combined results and the best USEDONE and ALLONE configurations, obtained from combining configurations from ANTLR and Jacks. We can sort the USEDONE configurations, with regard to their percent of compilation time, as follows:

81%: USEDONE$_{\text{J+ANTLR}}$ (mem. 64%), USEDONE$_{\text{J+BLOAT}}$ (mem. 67%), USEDONE$_{\text{J+CHART}}$ (mem. 65%)

82%: USEDONE$_{\text{J+FOP}}$ (mem. 68%), USEDONE$_{\text{J+XALAN}}$ (mem. 69%)

83%: USEDONE$_{\text{J+JYTHON}}$ (mem. 69%)

84%: USEDONE$_{\text{J+HSQLDB}}$ (mem. 70%), USEDONE$_{\text{J+LUCENE}}$ (mem. 70%), USEDONE$_{\text{J+PMD}}$ (mem. 70%)

85%: USEDONE$_{\text{J+D}}$ (mem. 71%)

We can sort the ALLONE configurations in the same fashion:

82%: ALLONE$_{J+ANTLR}$ (mem. 66%), ALLONE$_{J+BLOAT}$ (mem. 68%)

84%: ALLONE$_{J+CHART}$ (mem. 66%)

85%: ALLONE$_{J+D}$ (mem. 72%)

88%: ALLONE$_{J+FOP}$ (mem. 69%), ALLONE$_{J+JYTHON}$ (mem. 69%), ALLONE$_{J+XALAN}$ (mem. 70%)

89%: ALLONE$_{J+LUCENE}$ (mem. 71%), ALLONE$_{J+PMD}$ (mem. 71%)

We note that the influence of the benchmarks improve the average performance of the Jacks configurations (83%) with one or two percent. It is interesting to note that the benchmarks providing the best performance on average for Jacks, independent of configuration, are those with small coverage and few lines of code. These results indicate that it is worth combining a compiler test suite with a normal program, but that the program should not be too large or complicated. This way, we will end up with a configuration that caches attributes that end up in the USEDONE set of any small intricate program, as well as in the USEDONE set of larger programs, but without caching attributes that seem to be less commonly used many times. Further, it should be noted that the memory usage results for the combined ALLONE$_{J+D}$ and USEDONE$_{J+D}$ present unexpected results when compiling Jython. Presumably, the first configuration should use more memory than the second configuration, but the results show the reverse. The difference is slight but still statistically significant. At this point we have no explanation for this unexpected result.

6 Related Work

There has been a substantial amount of research on optimizing the performance of attribute evaluators and to avoid storing all attribute instances in the AST. Much of this effort is directed towards optimizing static visit-oriented evaluators, where attribute evaluation sequences are computed statically from the dependencies in an attribute grammar. For RAGs, such static analysis is, in general, not possible due to the reference attributes. As an example, Saarinen introduces the notion of *temporary attributes* that are not needed outside a single visit, and shows how these can be stored on a stack rather than in the AST [Saa78]. The attributes we have classified as ONE correspond to such temporary attributes: they are accessed only once, and can be seen as stored in the stack of recursive attribute calls. Other static analyses of attribute grammars are aimed at detecting *attribute lifetimes*, i.e., the time between the computation of an attribute instance until its last use. Attributes whose instances have non-overlapping lifetimes can share a global variable, see,

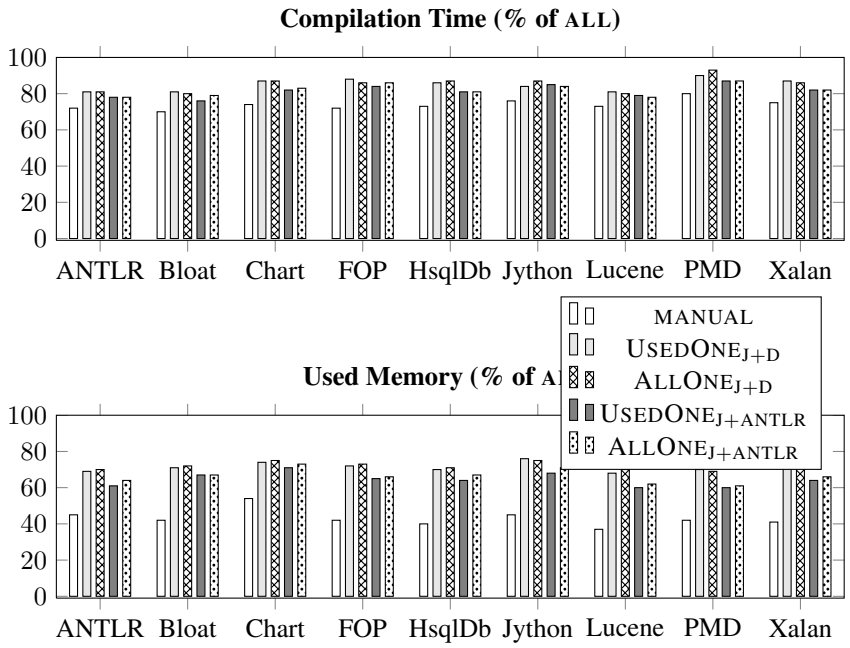


Figure 10: Results from Experiment D: Compilation of DaCapo benchmarks using combined configurations from the Jacks and DaCapo suites. All results are shown as compilation time and memory usage as percent of the results for the ALL configuration and results for MANUAL are included for comparison.

e.g., [Kas87]. Again, such analysis cannot be directly transfered to RAGs due to the use of reference attributes.

Memoization is a technique for storing function results for future use, and is used, for example, in dynamic programming [Bel57]. Our use of cached attributes is a kind of memoization. Acar et al. present a framework for selective memoization in a function-oriented language [Aca+03]. However, their approach is in a different direction than ours, intended to help the programmer to use memoized functions more easily and with more control, rather than to find out which functions to cache. There are also other differences between memoization in function-oriented programming, and in our object-oriented evaluator. In function-oriented programming, the functions will often have many and complex arguments that can be difficult or costly to compare, introducing substantial overhead for memoization. In contrast, our implementation is object-oriented, reducing most attribute calls to parameterless functions which are cheap to cache. And for parameterized attributes, the arguments are often references which are cheap to compare.

7 Conclusions and Future Work

We have presented a profiling technique for automatically finding a good caching configuration for compilers generated from RAG specifications. Since the attribute dependencies in RAGs cannot be computed statically, but depend on the evaluation of reference attributes, we have based the technique on profiling of test programs. We have introduced the notion of an attribute dependency graph with call counts, extracted from an actual compilation. Experimental evaluation on a generated Java compiler shows that by profiling on only a single program with an attribute coverage of only 67%, we reach a mean compilation speed-up of 20% and an average decrease in memory usage of 38%, as compared to caching all configurable attributes. This is close to the average compilation speed-up obtained for a manually composed expert configuration (25%). The corresponding average decrease in memory usage for the manual configuration (53%) is still significantly better. Our evaluation shows that we get similar performance improvements for both tested cache configuration approaches. Given these results, we would recommend the ALLONE configuration due to its higher robustness.

We find these results very encouraging and intend to continue this work with more experimental evaluations. In particular, we would like to study the effects of caching, or not caching, parameterized attributes, and to apply the technique to compilers for other languages. Further, we would like to study the effects of analyzing the content of the attribute equations. Most likely there are attributes which only return a constant or similar and, hence, should not benefit from caching independent of the number of calls. Finally, it would be interesting to further study the differences between the cache configurations from the profiler and the manual configuration.

8 Acknowledgements

We are grateful to Torbjörn Ekman for fruitful discussions and for implementing support in JastAdd for profiling and separate cache configurations. Thanks also to anonymous reviewers for comments on an earlier version of this paper.

References

[Aca+03] Umut A. Acar, Guy E. Blelloch, and Robert Harper. “Selective memoization”. In: *POPL*. Ed. by Alex Aiken and Greg Morrisett. ACM, 2003, pp. 14–25.

[Åke+10] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. “Implementation of a Modelica compiler using JastAdd attribute grammars”. In: *Science of Computer Programming* 75.1-2 (2010), pp. 21–38.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Bla+06] Stephen M. Blackburn et al. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *OOPSLA*. Ed. by Peri L. Tarr and William R. Cook. Portland, Oregon, USA: ACM, 2006, pp. 169–190.

[Bla+08] Stephen M. Blackburn et al. "Wake up and smell the coffee: evaluation methodology for the 21st century". In: *Communications of the ACM* 51.8 (2008), pp. 83–89.

[Boy05] John Tang Boyland. "Remote attribute grammars". In: *Journal of the ACM* 52.4 (2005), pp. 627–687.

[Bür+10] Christoff Bürger, Sven Karol, and Christian Wende. "Applying Attribute Grammars for Metamodel Semantics". In: *Proceedings of the International Workshop on Formalization of Modeling Languages*. ACM Digital Library, 2010.

[EH05] Torbjörn Ekman and Görel Hedin. "Modular Name Analysis for Java Using JastAdd". In: *GTTSE*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science. Springer, 2005, pp. 422–436.

[EH07a] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler". In: *OOPSLA*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[EH07b] Torbjörn Ekman and Görel Hedin. "The JastAdd system - modular extensible compiler construction". In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[Far86] Rodney Farrow. "Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars". In: *SIGPLAN Symposium on Compiler Construction*. Ed. by Richard L. Wexelblat. ACM, 1986, pp. 85–98.

[Hed00] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[Hua+08] Shan Shan Huang et al. "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary". In: *ECOOP*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008.

[Ibr+09] Ali Ibrahim et al. "Remote Batch Invocation for Compositional Object Services". In: *ECOOP*. Ed. by Sophia Drossopoulou. Vol. 5653. LNCS. Springer, 2009, pp. 595–617.

[Jac] *Jacks*. `http://sources.redhat.com/mauve`.

[Jou84] Martin Jourdan. "An Optimal-time Recursive Evaluator for Attribute Grammars". In: *Symposium on Programming*. Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 167–178.

[Jou+08] Wilfried Jouve et al. "A SIP-Based Programming Framework for Advanced Telephony Applications". In: *IPTComm*. Ed. by Henning Schulzrinne, Radu State, and Saverio Niccolini. Vol. 5310. Lecture Notes in Computer Science. Springer, 2008, pp. 1–20.

[Kas87] Uwe Kastens. "Lifetime Analysis for Attributes". In: *Acta Informatica* 24.6 (1987), pp. 633–651.

[Kat+10a] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. "Domain-Specific Languages for Composable Editor Plugins". In: *Electr. Notes Theor. Comput. Sci.* 253.7 (2010), pp. 149–163.

[Knu68] Donald E. Knuth. "Semantics of context-free languages". In: *Journal Theory of Computing Systems* 2.2 (1968), pp. 127–145.

[NN+09a] Emma Nilsson-Nyman et al. "Declarative Intraprocedural Flow Analysis of Java Source Code". In: *Electr. Notes Theor. Comput. Sci.* 238.5 (2009), pp. 155–171.

[PH97] Arnd Poetzsch-Heffter. "Prototyping Realistic Programming Languages Based on Formal Specifications". In: *Acta Informatica* 34.10 (1997), pp. 737–772.

[Pto] *Ptolemy*. `http://www.cs.iastate.edu/~ptolemy`.

[Saa78] Mikko Saarinen. "On Constructing Efficient Evaluators for Attribute Grammars". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. Vol. 62. Lecture Notes in Computer Science. Springer, 1978, pp. 382–397.

[Sch+08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. "Sound and extensible renaming for Java". In: *OOPSLA*. Ed. by Gail E. Harris. ACM, 2008, pp. 277–294.

[Slo+10] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. "A Pure Object-Oriented Embedding of Attribute Grammars". In: *Electr. Notes Theor. Comput. Sci.* 253.7 (2010), pp. 205–219.

[Söd10] Emma Söderberg. *Evaluation Link: Automated Selective Caching for Reference Attribute Grammars*. `http://svn.cs.lth.se/svn/jastadd-research/public/evaluation/sle-10-caching`. 2010.

[Dac] *The DaCapo Benchmarks*. `http://dacapobench.org`.

[Vog+89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. "Higher-Order Attribute Grammars". In: *PLDI*. Ed. by Richard L. Wexelblat. ACM Press, 1989, pp. 131–145.

[Wyk+07] Eric Van Wyk et al. “Attribute Grammar-Based Language Extensions for Java”. In: *ECOOP*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 575–599.

[Wyk+10] Eric Van Wyk et al. “Silver: An extensible attribute grammar system”. In: *Science of Computer Programming* 75.1-2 (2010), pp. 39–54.

INCREMENTAL EVALUATION OF REFERENCE ATTRIBUTE GRAMMARS USING DYNAMIC DEPENDENCY TRACKING

PAPER IV

Abstract

Reference attribute grammars (RAGs) have proven practical for generating production-quality compilers from declarative descriptions, as demonstrated by the JastAdd system. Recent results indicate their applicability also to generating semantic services in interactive editors. For use in editors, it is necessary to update the attribution after edit operations. Earlier algorithms based on statically scheduled incremental attribute evaluation are, however, not applicable to RAGs, as they do not account for the dynamic dependencies that reference attributes give rise to. In this report, we introduce a notion of consistency for RAG attributions, along with an algorithm for maintaining consistency after edit operations, based on dynamic dependency tracking. That is, we introduce a means to do incremental evaluation of RAGs using dynamic dependency tracking.

Emma Söderberg and Görel Hedin.
Technical report, LU-CS-TR:2012-250,
ISSN 1404-1200, Report 99, 2012.
Lund University

1 Introduction

Today's industry-standard language-based editors, like the Eclipse JDT or IntelliJ IDEA for Java, have become indispensable tools for efficient software development. More languages would benefit from this kind of editor support, but to develop an editor from scratch is a major endeavor. Especially, since such interactive tools need support for efficient updating of their internal representation.

To instead generate such editors from declarative descriptions is an area of research that has been extensively investigated. Much focus has been put on editor descriptions building on the formalism of attribute grammars (AGs) [Knu68], and the development of incremental evaluation algorithms for such grammars [Rep82]. This research has resulted in several generator systems [RT84; Jou+90; KS98].

However, a major deficiency of the pure AG approach is its limitation of only supporting attribute dependencies running along the structure of the syntax tree. A restriction to such dependencies, leads to shuffling of large aggregated values along the syntax tree. Also, these restrictions make it difficult to express graph-like properties, like use-def chains or object-oriented inheritance. Many efforts have been made to remove these obstacles through extensions to AGs [Boy05; PH97].

One such recent extension, *reference attribute grammars* [Hed00] (RAGs), allows attributes to be references to other abstract syntax tree (AST) nodes. In effect, this allows graphs to be super-imposed on top of the AST, making it easy to, for instance, express use-def chains. RAGs have further been extended with *parameterized attributes*, which remove the need to shuffle large aggregated values up and down the syntax tree. In addition, RAGs have been extended with so called ReRAGs or *rewrites*, that is, demand-driven transformations depending on attribute values. ReRAGs can, for instance, be used for syntax normalization, or context-based specialization of nodes in the syntax tree. Together, these extensions, tackle the earlier mentioned practicality issues of pure AGs, making it possible to more easily express complex semantics. This expressive property has been clearly demonstrated by the JastAdd system [EH07b], where a full Java compiler has been generated from a RAG specification, performing within a factor of three from handwritten compilers [EH07a].

The graph properties of RAGs make them highly attractive for the generation of interactive services in language-based editors, such as refactorings, name completion, and cross-references [Sch+08; SH11]. However, there is so far no general algorithm for updating of RAG attributions after edits. Earlier developed incremental algorithms for AGs are based on static analysis of attribute dependencies, where dependencies follow the tree structure of the AST. RAGs, in contrast, are more general and may have attribute dependencies that follow a graph structure emanating from reference attributes, making these static algorithms inapplicable. Instead, RAGs are evaluated dynamically on-demand using a recursive algorithm, originally formulated for AGs [Jou84], during which attribute values may

be cached to prevent unnecessary re-computations.

In this report, we explore different approaches for maintaining consistency of RAG attributions after edit operations in an interactive setting. We consider the spectrum of possible approaches: from the crude batch solution, which restores consistency by rebuilding the AST with a complete re-parsing of the source code, to the fine-grained incremental solution, that seeks to retain as many valid attribute values as possible. Previous incremental algorithms statically compute an evaluation order based on static dependencies. These algorithms work under the assumption that all attributes should be evaluated, and that all dependencies are known before evaluation, or a at least a good approximation thereof.

In our setting, neither is true: the exact set of evaluated attributes depends on the syntax tree, and the set of dependencies depends on the values of references attributes, and is therefore not known before evaluation. To find the exact dependencies we are left to using a dynamic algorithm where we construct a dependency graph during evaluation. Once we have this dependency graph, we can react to change and restore consistency after edits. In addition to the batch and the incremental approach, we consider a so called full flush approach, where we restore consistency by removing all computed attribute values, but avoid the re-parsing needed in the batch approach. In this full flush approach, and in the fine-grained incremental approach, we incorporate support for reversal of rewrites.

To our knowledge, this is the first work on consistency maintenance of RAGs. The main contributions of this report are the following:

- Basic notions of consistency and attribute dependencies for RAGs.
- A dynamic dependency tracking approach for dependencies in RAGs.
- An incremental algorithm for consistency maintenance for RAGs.

The rest of this report is structured as follows: We start with a brief introduction to RAGs in Section 2 and a description of the concept of a consistent RAG attribution in Section 3. This is followed by an explanation of how the dynamic dependency tracking works in Section 4, and how it is used to maintain a consistent RAG attribution in Section 5. Related work is covered in Section 6 and, finally, conclusions and a summary of future work ends the report in Section 7.

2 Reference Attribute Grammars

This section describes RAGs and the problems in applying a statically scheduled attribute evaluation.

2.1 Traditional Attribute Grammars

Attribute grammars (AGs) [Knu68] provide context-sensitive information by associating *attributes* to nodes of an abstract syntax tree (AST) defined by a context-

free grammar. The attribute values are defined using so called *semantic functions* of other attributes. Traditionally, there are two kinds of attributes: **inh**erited and **syn**thesized, propagating information downwards and upwards in the AST. The following example shows a synthesized attribute propagating the sum of an addition upwards in the AST:

```
// Grammar: Add ::= Left Right
syn int Add.sum = Left.val + Right.val;
```

In the example, the comment shows a simplified grammar with an AST node `Add` with two children – `Left` and `Right`. The attribute `sum` is declared as an **int**eger defined as the sum of its children's `val` attributes (definitions of `val` are not included in the example).

The definition of an attribute value is called an *equation*, whose left hand side is the defined attribute, and whose right hand side is an application of a semantic function to other attributes. In this case, the semantic function is "+", and it is applied to the attributes `Left.val` and `Right.val`. To propagate information in the reverse direction, downwards, we can use inherited attributes. The following example shows an inherited attribute which is used to compute the nesting depth:

```
// Grammar: Program ::= Block
// Grammar: Block ::= Assign | Block
inh int Assign.nesting;
inh int Block.nesting;
eq Block.Block.nesting = nesting + 1;
eq Block.Assign.nesting = nesting + 1;
eq Program.Block.nesting = 0;
```

Again, the comment provides a simplified grammar, this time with a `Block` node which may have either an assignment (`Assign`) or another block as a child. Each of these possible children are defined to have an inherited attribute `nesting` returning an integer. In contrast to synthesized attributes, the equation for an inherited attribute is not given in the node in which the attribute is declared, instead it is provided by an ancestor in the AST. With this in mind, we provide equations for the attribute in `Block` – one for the `Assign` child and one for the `Block` child. In these equations, we make use of the `nesting` attribute in `Block` itself to increase the value for its children. Finally, the root of the AST (`Program`) provides an equation for the `nesting` attribute of its `Block` child.

2.2 Reference Attributes and Parameters

In traditional AGs, attribute types are value types, for example, integers and booleans. RAGs extend AGs with *reference attributes*, that is, by allowing attribute values to be direct references to distant nodes in the AST. Reference attributes allow for easy specification of structures that do not follow the AST tree structure, like call

```
// Grammar: Program ::= Block
// Grammar: Block ::= Decl (Use | Block)
// Grammar: Decl ::= <Type:String> <Name:String>
// Grammar: Use ::= <Name:String>
syn String Use.type = decl.Type;
syn Decl Use.decl = lookup(Name);
inh Decl Use.lookup(String name);
inh Decl Block.lookup(String name);
syn boolean Decl.declares(String name) =
    Name == name;
eq Block.Use.lookup(String name) =
    Decl.declares(name) ? Decl : lookup(name);
eq Block.Block.lookup(String name) =
    Decl.declares(name) ? Decl : lookup(name);
eq Program.Block.lookup(String name) =
    ''Unknown Decl''
```

Figure 1: Lookup example, illustrating parameterized attributes.

graphs and inheritance relations useful in compiler construction. A reference attribute can be used to access information in a distant node, as in the following example:

```
// Grammar: Decl ::= <Type:String> <Name:String>
// Grammar: Use ::= <Name:String>
syn Decl Use.decl = ...
syn String Use.type = decl.Type;
```

Here, we have two AST nodes, `Decl` and `Use`, representing declarations and uses of names in a language. The `Decl` node has two terminals of type `String` while the `Use` node has one. Two synthesized attributes are defined: the first providing a reference to the declaration of a use, and the second providing the type of a use as a string. In the latter equation, `Type` is a terminal of the `Decl` object referred to by the `decl` attribute.

To complete the example and provide an equation for the `decl` attribute, we need a means to look up the declaration corresponding to the name of the use. Traditionally, AGs use inherited aggregate-valued attributes, usually named `environment`, to propagate information about all visible names to each use. With RAGs, we can instead provide a distributed symbol table [EH05] using *parameterized attributes*, another central extension in RAGs. Using parameterized attributes, we define inherited attributes, typically called `lookup`, that take a name as a parameter and return a reference to the appropriate declaration node. Parameterized attributes allow nodes to be queried for information, rather than having to construct large aggregate attribute values with all potentially interesting informa-

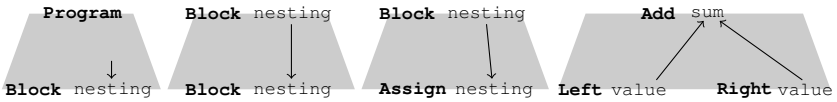


Figure 2: Example of production-based static dependency graphs. An arrow from b to a indicates that a depends on b.

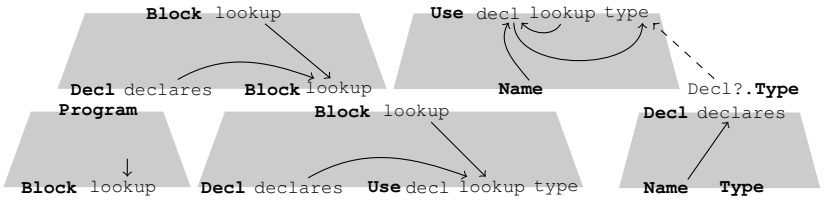


Figure 3: Production-based static dependency graphs for the lookup example listed in Figure 1. Dashed arrows show dependencies, which cannot be captured by these graphs, due to reference attributes.

tion.

An example is shown in Figure 1. The top comments in the figure show a simplified grammar with `Decl` and `Use` nodes as presented earlier, a `Block` node with a `Decl` node followed by a `Block` or `Use` node, and a root `Program`. The previous `decl` attribute is here defined by an equation calling an inherited attribute `lookup`, taking the name of the use node as parameter. Three equations are provided for the `lookup` attribute: two in `Block` and one final in `Program`, returning a representation of an "unknown declaration". The equations in `Block` check whether the `Decl` child declares `name`, in which case the `Decl` node is returned, or calls `lookup` of `Block` itself.

2.3 Attribute Evaluation and Dependencies

If the value of an attribute b is used when evaluating the right hand side of the equation for another attribute a, we say that a *depends on* b. For traditional AGs, all attribute dependencies are *static*, in the sense that they can be deduced from the AG alone, without taking the attribute values of a particular AST into account. Most incremental algorithms for AGs, e.g., the well-known optimal algorithm by Reps [Rep82], make use of this fact.

These algorithms do not apply to RAGs since the possibility to access attributes of distant nodes, via reference attributes, makes the dependencies dependent on the values of individual reference attributes. I.e., some of the dependencies are

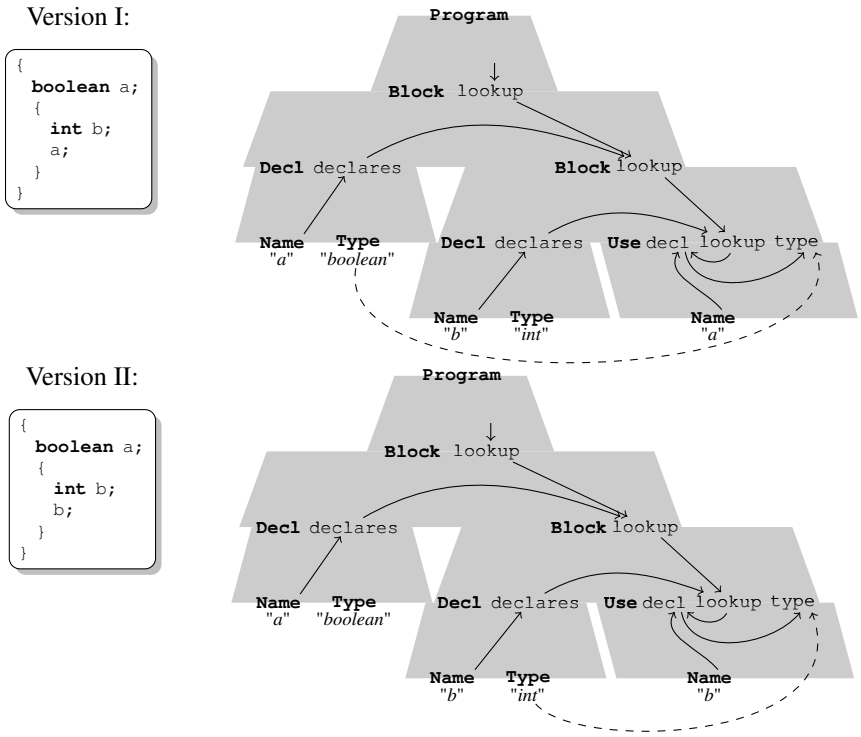


Figure 4: Two example showing how the dynamic dependency graph is obtained for a derivation tree by pasting together instances of the production-based static dependency graphs. The dashed arrows show dynamic dependencies not captured by these graphs.

dynamic, in that they can be decided only after actually evaluating some of the attributes.

Considering the examples in Section 2.1, the static evaluation order is quite clear: to compute `Add.sum` we must first compute `Left.val` and `Right.val`, and to compute `Assign.nesting` we must first compute the `nesting` of its parent block. These static dependencies can be illustrated using production-based static dependency graphs, also used in [Der+86], as illustrated in Figure 2.

Using the same notation, we can capture the static dependencies of the lookup example presented in Section 2.2, as shown in Figure 3. However, here we have trouble capturing the dependency of the `type` attribute, as indicated by the dashed arrow. We know that `type` depends on the `Type` terminal of a `Decl` node but exactly which node depends on the AST. Figure 4 illustrates these dynamic dependencies with two examples of possible ASTs. The two examples are identical except for the `Use` node: in version I there is a use of the name `a` and in version II there is a use of the name `b`.

2.4 Demand-driven Transformations

In addition to the previous mentioned extensions, reference attributes and parameterized attributes, RAGs support demand-driven transformations called *rewrites* [EH04]. Rewrites are defined as conditional transformations on node types, and are triggered and evaluated on first access to a node of that type. During traversal of an AST, on each access to a child, potential rewrites will be evaluated on that child before it is returned. At the point where a child is returned, it is considered to be *final*. Initially, only the root node is considered final, but this final "region" of the root node will spread downwards in the AST as new nodes are accessed and evaluated. In practice, this means that rewrites are evaluated top-down, from parent to child, starting at the root of the AST.

There is no limit to the number of rewrites in an AST. In theory, all nodes except the root node may have rewrites, but in practice rewrites are mainly used for smaller transformations. For example, desugaring of syntax or specialization of access nodes based on context. The extent to which rewrites are evaluated depend on which AST nodes are that accessed, and in the set of accessed nodes, the actual set of rewritten nodes depend on which rewrite conditions that have become true. A rewrite condition may contain attribute values, and these values may depend on the syntax tree. That is, a rewrite may happen in one syntax tree but not in another.

In order to incrementally update an AST constructed using rewrites, we need to know the dependencies of rewrite conditions and we need a means to reverse rewrites if their conditions turn to false after an update. Finding the dependencies of rewrite conditions, boils down to the finding of dependencies between reference attributes, and the reversal, or *flushing*, of rewrites, requires knowledge of which value to reverse back to. Regardless of approach, the solution to these problems

needs to be integrated with the tracking of attribute dependencies and flushing of attribute values.

3 Consistent Attribution

In this section we describe what is meant by a RAG attribution, and what it means for it to be consistent.

3.1 Attribution

The value of an attribute instance is found by evaluating the right-hand side of its defining equation, and recursively evaluating any attribute instances used in this equation. For efficiency, the value can be *cached*, i.e., stored at the first access, so that subsequent accesses can return the value directly, rather than have to recompute it [Jou84]. In theory, all attribute values should be cached, to minimize the number of computations. However, in practice, there are performance gains in selecting only a subset of attributes to be cached [SH10].

We will refer to attributes that store their value as *cacheable* and attributes that do not as *uncacheable*. A cacheable attribute instance is either in the state *cached*, meaning it has a currently stored value, or *decached*, meaning it does not. Initially, all cacheable attributes are decached. Evaluation of a decached attribute computes its value, stores it, and takes the attribute to the cached state. A cached attribute can also be *flushed*, removing the value and taking the attribute back to the decached state.

To be able to reason about edits of the AST, we will regard the child and parent links as *intrinsic* reference attributes, and terminals as intrinsic value attributes. Intrinsic attributes have a stored value that is given a priori, when the AST is constructed. They are similar to cached attributes in that they have a stored value, but different in that they generally have no defining equation, and are not flushed. The collective state of all intrinsic and cacheable attributes is called an *attribution*.

Adding rewrites to a RAG system is then like adding equations to certain intrinsic attributes, in this case child links. Rewrites use the values that are given a priori as *base values* for their evaluation: the rewrite condition will be evaluated based on this value and, if the condition is true, the final value of the rewrite will be constructed using this value. Attribution-wise when using rewrites, the intrinsic child attribute can be considered as a cached attribute with a more complex attributed value: Flushing the rewritten child attribute will bring it back to its base value.

3.2 Consistency

When accessing an attribute we expect that we will get the same value as we would if we evaluated the right-hand side of its defining equation. If this is the case for

all attributes, we say that the attribution is *consistent*. For the different kinds of attributes, we define consistency as follows:

intrinsic attributes are by definition consistent, rewrites are here not considered to be pure intrinsic attributes, but cached intrinsic attributes.

uncacheable attributes are by definition consistent

decached attributes are by definition consistent

cached attributes are consistent if all cached attributes they (transitively) depend on are consistent, and if their stored value is equal to the value computed by evaluating the right-hand side of their defining equation.

rewrites are considered to be cached intrinsic attributes. In their decached state, they are consistent if they have their base value, and in their cached state, their consistency follows from the definition for cached attributes.

It follows that an initial AST is consistent since it has no cached attributes. Evaluation of cacheable attributes and caching of their values will keep the attribution consistent, since the expressions in the right-hand side of equations are side-effect free. These conditions also hold for rewrites, which behave like cached attributes.

However, after editing an AST, i.e., changing the value of an intrinsic attribute, cached attributes may have become inconsistent. With knowledge of dependencies, potentially inconsistent attributes can be found and consistency can be restored, either by decaching attributes or by re-caching attributes, i.e., by re-evaluating attributes. In this paper, we focus on decaching of attributes, i.e., flushing.

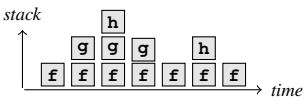
4 Dependency Tracking

This section describes how dynamic dependencies are found by tracking during evaluation of attributes and rewrites.

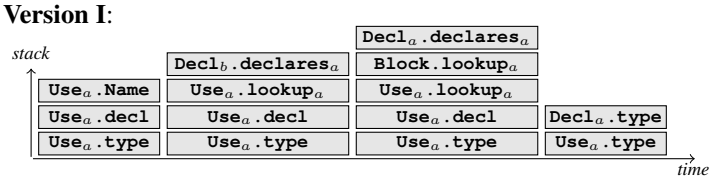
4.1 Stack-based Dependency Tracking

Reference attributes are evaluated recursively using an evaluation stack [Jou84]. To find dynamic dependencies, dependency tracking is done during evaluation, recording how attribute instances depend on each other. The example below shows the stack during the evaluation of an attribute `f`.

```
int f = g + h;
int g = h;
int h = 4;
```



Version I:



Version II:

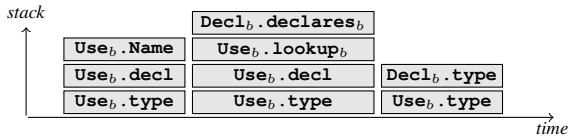
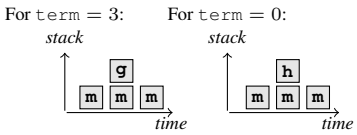


Figure 5: The stacks correspond to the evaluation stack at points during the evaluation where the stack is about to decrease. The examples being evaluated are taken from Figure 4.

The evaluation stack is in effect a call stack, where each call reflects that the callee is dependent on the caller, giving the following dependencies: `f`←`g`←`h`, and `f`←`h`. In this example, the dependencies are static and could have been deduced from the attribute definitions alone. However, in the case of dynamic dependencies, the call stack is needed to capture the exact dependencies. Consider the following example involving a terminal `term` (i.e., an intrinsic attribute) and a conditional equation right-hand side:

```
int m =
  term > 2 ? g : h;
int g = 5;
int h = 4;
```



Here, the stack depends on the value of `term`, resulting in different dependencies: for `term` = 3, we get `m`←`g`, and for `term` = 0 we get `m`←`h`. Here, a static approach would lead to an approximation of the exact dependencies: `m`←`{g,h}`.

The need for dynamic dependencies is even more apparent in examples using reference attributes. Figure 5 shows the call stacks for the examples in Figure 4. Here, we show the stacks only at points during evaluation when the stack is about to decrease. Version I of the program gives rise to the following dependencies: Use$_a$.type ← Use$_a$.decl ← {Use$_a$.Name, Use$_a$.lookup$_a$}, Use$_a$.lookup$_a$ ← {Decl$_b$.declares$_a$, Block.lookup$_a$}, Block.lookup$_a$ ← Decl$_a$.declares$_a$, and Version II give rise to the following dependencies: Use$_b$.type ← Use$_b$.decl ← {Use$_b$.Name, Use$_b$.lookup$_b$}, Use$_b$.lookup$_b$ ← Decl$_b$.declares$_b$. We can note that Version II induces fewer dependencies, due to the closeness of the declaration to the use.

4.2 Tracking of Attribute Dependencies

Each attribute is implemented as a method containing evaluation code. For cacheable and intrinsic attributes this code accesses the stored state (computing and storing the value in case the cacheable attribute was previously decached). For uncacheable attributes, the evaluation code simply computes the value according to the equation right-hand side.

We represent each intrinsic and cacheable attribute instance a by a *dependency handler object* that keeps a set of dependents, i.e., a set of references to handler objects for the attribute instances that depend on a. Initially, before any attribute evaluation starts, all dependents sets are empty. For cacheable parameterized attributes, the cached value is stored for each used combination of parameter values, and a handler is created for each new such combination used.

To track dependencies during evaluation, we instrument the evaluation code of these attributes to maintain a global stack of handlers, adding and removing the handler for the evaluated attribute instance to this stack as the evaluation code is entered and exited. Furthermore, at each evaluation code entry, the previous top of stack is added to the dependents set of the new top of stack.

4.3 Tracking of AST Structure and Rewrites

Accesses to child and parent links also give rise to dependencies. Even more so, when a child has rewrites, since then there is a rewrite condition potentially depending on attribute values. Dependencies for these rewrite conditions need to be tracked like for any cached attribute. In fact, this tracking needs to be done for all intrinsic attributes that may be changed. Rewritten children may be changed due to an update of a dependency, and non-rewritten children may be changed due to an AST edit. In this sense, we may consider parent and child links as reference attributes, with all child links of a node as one parameterized child attribute.

The dependency graphs in Figure 4 are thus actually incomplete. In particular, dependencies to child links are missing, since equations may return references to children. For example, the equations for `lookup` in `Block` in Figure 1 may return a reference to the `Decl` child. Possibly less apparent, is that each time an attribute of a child is used in an equation, there is actually also a dependency on the child link. Also, each inherited attribute actually depends on the parent link up to the node holding its defining equation.

5 Consistency Maintenance

During development, a developer makes changes to a program. These changes will result in a sequence of AST edits handled by the editor. After each edit, the previously consistent attribution of the AST, corresponding to the program being edited, may have become inconsistent. The goal of consistency maintenance is

to bring the AST into a consistent state after each edit. In practice, this means keeping track of dependencies and notifying affected cached attributes of change when needed, so that these attribute values may be flushed.

In our setting, we assume the following: 1) that the AST is initially syntactically correct with a consistent attribution before any edits have taken place, 2) that the AST is syntactically correct after an edit, and 3) that any new nodes or subtrees added to the AST are consistent before the addition, typically with all cacheable attributes in the decached state.

5.1 AST Edits

There are several possible AST edits, for example, a child link may be replaced, added, removed or inserted, or a terminal may be replaced. Edits to child links may be considered to be the most complex edits given that its an edit to a list structure, where succeeding children may be affected. In contrast, a parent link or an intrinsic terminal value, like the name of a variable, can only be replaced.

As an example, consider the removal of a child k in a list of n children (from 0 to $n-1$), dependants to the removed child must be notified, but also dependants to child $k+1$ to $n-1$, since these children are being moved as a consequence. In comparison, the replacement of a child is a simpler edit, since then only the dependencies of the child being replaced needs to be notified.

In general, an edit can be described as changing the values of a set I of intrinsic attributes, i.e., parent and child links, and terminals, followed by a notification of dependencies which are (transitively) dependent on the set I. Clearly, these are the set of cached attributes that can become inconsistent due to the edit.

Notably, edits to rewritten children are not very different than edits to children without rewrites. For example, if a rewritten child is replaced, then the new value is used as the base value of the rewrite, and the rewrite is considered to be decached.

5.2 Flushing

If a cached attribute is notified of a change, with the current approach, it should be flushed. A flush means marking the attribute as decached and returning the value of the attribute to its base value given at AST construction.

Base values for rewrites To flush a rewrite, the base value needs to be stored. The trivial approach for storing base values is to make an *base copy* of the value of a rewrite before it is evaluated. This approach is, however, quite memory demanding, especially when rewrites are nested, or if rewrites occur for larger subtrees, and not useful in practice. A slightly trimmed alternative, is for rewrites to share base copies when possible. That is, nested rewrites, or *inner rewrites*, share their base copy with enclosing rewrites, or *outer rewrites*. In practice, this means that during evaluation outer rewrites make copies while inner rewrites do not.

Flushing of rewrites Given that we use the slightly trimmed copying of base values, we get a situation where we have flushing of inner and outer rewrites. The flushing of an outer rewrite, then includes the following steps: 1) setting the rewrite to decached, 2) setting all inner rewrites to decached, 3) restoring the value of the base value, and 4) notifying dependencies of the rewrite. In contrast, the flushing of an inner rewrite involves the following steps: 1) setting the rewrite to decached, and 2) locating the enclosing outer rewrite and notifying it of change.

5.3 Algorithm for Consistency Maintenance

A general technique for maintaining RAG consistency after edits is to flush all attributes that transitively depend on the edited set of intrinsic attributes I. We represent I by the set of corresponding handler objects, handling dependencies and acting as nodes in the dependency graph. The algorithm for restoring consistency can then be expressed as follows:

```
RESTORE-CONSISTENCY(I)
1  for each intrinsic handler h ∈ I
2       do TRANSITIVE-FLUSH(h)

FLUSH(h)
1  ▷ Flush the cacheable
2  attribute handled by h

TRANSITIVE-FLUSH(h)
1  deps ← dependents(h)
2  dependents(h) ← ∅
3  for each cacheable handler h ∈ deps
4       do FLUSH(h)
5          TRANSITIVE-FLUSH(h)
```

Notably, the $dependents$ set of h is cleared before dependents are transitively flushed. This clean up prevents the algorithm from going into endless recursion if there are circular dependencies between attributes. Circular attributes [Far86; MH07] are excluded from the examples in this paper, but are nonetheless supported by this approach.

5.4 Aborting Transitive Flush

The simple algorithm above does not take the attribute values into account: if an attribute happens to have the same value after the change, all its dependent attributes will be flushed. In principle, it would be possible to abort the transitive flush for attributes that are known to have the same value after the change. However, this would require that a new value is computed before the flush is done. To compute this new value, we cannot, however, use the cached values it depends on since they might be inconsistent.

In principle, new values can be computed without using cached values, i.e., by evaluating corresponding equations rather than using cached values. However, in general, this can become extremely expensive, since evaluating attributes without using caching may lead to exponentially growing evaluation times. Therefore, in

general, such abortion is not likely to be profitable. In specific cases, however, it can still pay off.

In particular, if an attribute does not depend (transitively) on any cacheable attributes, it can be evaluated in the same amount of time before or after the flush. We call such an attribute *cache-independent*. To take such cache-independent attributes into account, the algorithm for TRANSITIVE-FLUSH would be altered as follows:

```
TRANSITIVE-FLUSH(h)
 1  deps ← dependents(h)
 2  dependents(h) ← ∅
 3  for each cacheable handler h ∈ deps
 4       do if CACHE-INDEPENDENT(h)
 5            then
 6                 value_new ← EVALUATE-ATTRIBUTE-OF(h)
 7                 if value_new! = CACHED-VALUE-OF(h)
 8                   then
 9                        SET-VALUE-OF(h, value_new)
10                        FLUSH(h)
11                        TRANSITIVE-FLUSH(h)
12
13            else
14                 FLUSH(h)
15                 TRANSITIVE-FLUSH(h)
```

The identification of attributes that are cache-independent could either be done statically, by annotating the attributes as such (and checking this property), or dynamically, by keeping track of which attributes are cache-independent during the dependency tracking.

In practice, abortion of transitive flush will be particularly important for parameterized attributes that check terminal values, like the attribute `Decl.declares` in Figure 1. Suppose the `Name` terminal of a `Decl` is edited. Calls to `Decl.declares` will have the same value for all parameters that are different from the old and new `Name` terminal. In practice, there may be many such calls due to the block structure in a program.

5.5 Implementation

The algorithm described in Section 5.3 has been implemented and tested in the JastAdd system [Jas]. The implementation supports incremental consistency maintenance for all AST edits allowed by the system, that is, removal, addition and insertion of children. The JastAddJ extensible Java compiler [EH07a] has been used as a test platform for the implementation, and all attributes and transformations occurring in the JastAddJ compiler are supported. This includes synthesized,

inherited, and parameterized attributes, higher-order attributes [Vog+89], circular attributes [Far86], and rewrites [EH04]. The abortion of transitive flush is currently under implementation.

6 Related Work

There is an extensive amount of previous work on the incremental evaluation of attribute grammars with the goal of supporting interactive language-based editors.

For classical AGs, Demers, Reps, and Teitelbaum presented a two-pass algorithm, which first nullifies dependent attributes and then reevaluates them [Dem+81]. The dependency analysis is done based on the static dependencies in the AG. Reps improved this approach by presenting an optimal *change propagation* algorithm [Rep82], where old and new attribute values are compared, and avoiding to propagate the change to dependents if the values are equal. This algorithm was proven optimal in the sense that it does work proportional to the number of affected attributes, i.e., the attributes that actually do get new values.

A problem with classical AGs is that, even if the algorithm is optimal, the number of affected attributes becomes very large: to handle complex computations, large amounts of information, typically symbol tables, are bundled together into single aggregate-valued attributes that are copied throughout the AST. A small change to one declaration thereby causes all the copies to become affected, even if very few attributes actually make use of the changed declaration. A number of different solutions to these problems were proposed, focusing on special support for these aggregate-valued attributes, e.g., [HT86].

Other work focused on extending the classical AGs themselves, to make the complex computations more straightforward to express. This includes work by Poetzsch-Heffter [PH97], Boyland's Remote AGs [Boy98], and our RAGs [Hed00]. All these formalisms make use of some kind of mechanism for remote access of attributes, thereby inducing dependencies that are difficult to deal with by static analysis of the AG.

In Boyland's Remote AGs, AST nodes can have local objects with fields, and attributes can be references to such objects. This allows graph structures to be built, and equations can read the fields of an object remotely. In Remote AGs, an AST node can also have collection fields, which are aggregate-valued attributes like sets, and where the definition can be spread out on multiple sites in the AST, each contributing to the collection, e.g., adding a particular element.

Boyland developed a static algorithm for the evaluation of Remote AGs, where control attributes are automatically added to take care of scheduling of remote attributes and collections [Boy98]. He also developed an incremental algorithm for remote AGs that combines static scheduling for "ordinary" attributes with dynamic scheduling for remote attributes and collections [Boy02]. Preliminary experiments with this algorithm on a small procedural toy language and synthetic benchmark

programs, showed substantial speedups for edits of declarations as compared to reevaluating all attributes.

Boyland uses collection attributes for solving name analysis problems, representing local symbol tables as collections of declaration objects. Although JastAdd does support collection attributes (whose incremental evaluation is not treated in this paper), name analysis in RAGs is typically solved using parameterized attributes, as in the examples in this paper.

All the algorithms mentioned above are based on data-driven attribute evaluation, i.e., all attributes are evaluated, regardless of if they are actually used or not. After an edit, all affected attributes are updated. In RAGs, the attribute evaluation is instead demand-driven [Jou84], evaluating only attributes whose values are actually needed. After an edit, we decache all attributes that might be inconsistent. This might well be a larger set than the actually affected set. However, because aggregate values are avoided, we do not get the inflated affected sets that classical AGs suffer from.

7 Conclusion and Future Work

We have presented a basic fine-grained algorithm for incremental evaluation of RAGs. The algorithm restores consistency after edits to the abstract syntax tree, and is based on dynamic dependency tracking to flush all possibly affected attributes. We have also discussed how the flush propagation can be aborted by comparing old and new attribute values, and how this can be done without extra cost for cache-independent attributes.

As future work, we will evaluate the algorithm experimentally, and investigate several optimizations. The basic algorithm has extensive overhead due to the fine-grained dependency information that is maintained. We expect that more coarse-grained approaches will perform better in practice, and we are investigating approaches based on partitioning the AST and the attribute set. Furthermore, some practical RAGs such as the JastAddJ extensible Java compiler [EH07a], make use of large attribute values for collecting local declarations into a map data structure. This is done in order to avoid that multiple queries for declarations repeatedly search the AST. Unfortunately, this use of large values causes the same kind of dependency imprecision that ordinary AGs suffer from. To obtain both the higher performance of the maps and fine-grained dependencies, we are investigating the introduction of a new kind of *bound* parameterized attribute, where all results (for all possible parameter values) can be computed at the first call to the attribute.

There is also a potential for improving the evaluation of rewrites. In our current implementation, attributes depending on rewritable parts of the AST are not cached until those parts are rewritten [EH04]. By using the incremental evaluation it might be possible to cache such attributes already during rewrite. Other possible improvements include support for incrementally updating rather than recomputing

affected higher-order attributes [Vog+89], and support for incremental evaluation of collection attributes [Mag+09].

References

[Boy05] John Tang Boyland. "Remote attribute grammars". In: *Journal of the ACM* 52.4 (2005), pp. 627–687.

[Boy98] John Boyland. "Analyzing Direct Non-local Dependencies in Attribute Grammars". In: *CC*. Ed. by Kai Koskimies. Vol. 1383. Lecture Notes in Computer Science. Springer, 1998, pp. 31–49.

[Boy02] John Boyland. "Incremental Evaluators for Remote Attribute Grammars". In: *Electr. Notes Theor. Comput. Sci.* 65.3 (2002), pp. 9–29.

[Dem+81] Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. "Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors". In: *POPL*. Ed. by John White, Richard J. Lipton, and Patricia C. Goldberg. ACM Press, 1981, pp. 105–116.

[Der+86] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *A Survey on Attribute Grammars, Part I: Main Results on Attribute Grammars*. Tech. rep. Rapport de Recherche 485. Rocquencourt, France: INRIA, 1986.

[EH04] Torbjörn Ekman and Görel Hedin. "Rewritable Reference Attributed Grammars". In: *ECOOP*. Ed. by Martin Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 144–169.

[EH05] Torbjörn Ekman and Görel Hedin. "Modular Name Analysis for Java Using JastAdd". In: *GTTSE*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science. Springer, 2005, pp. 422–436.

[EH07a] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler". In: *OOPSLA*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[EH07b] Torbjörn Ekman and Görel Hedin. "The JastAdd system - modular extensible compiler construction". In: *Science of Computer Programming* 69.1-3 (2007), pp. 14–26.

[Far86] Rodney Farrow. "Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars". In: *SIGPLAN Symposium on Compiler Construction*. Ed. by Richard L. Wexelblat. ACM, 1986, pp. 85–98.

[Hed00] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[HT86] Roger Hoover and Tim Teitelbaum. "Efficient incremental evaluation of aggregate values in attribute grammars". In: *SIGPLAN Symposium on Compiler Construction*. Ed. by Richard L. Wexelblat. ACM, 1986, pp. 39–50.

[Jas] *JastAdd*. `http://jastadd.org`.

[Jou84] Martin Jourdan. "An Optimal-time Recursive Evaluator for Attribute Grammars". In: *Symposium on Programming*. Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 167–178.

[Jou+90] Martin Jourdan et al. "Design, implementation and evaluation of the FNC-2 attribute grammar system". In: *SIGPLAN Notices* 25.6 (1990), pp. 209–222.

[Knu68] Donald E. Knuth. "Semantics of context-free languages". In: *Journal Theory of Computing Systems* 2.2 (1968), pp. 127–145.

[KS98] Matthijs F. Kuiper and João Saraiva. "Lrc - A Generator for Incremental Language-Oriented Tools". In: *CC*. Ed. by Kai Koskimies. Vol. 1383. Lecture Notes in Computer Science. Springer, 1998, pp. 298–301.

[MH07] Eva Magnusson and Görel Hedin. "Circular Reference Attributed Grammars - their Evaluation and Applications". In: *Science of Computer Programming* 68.1 (2007), pp. 21–37.

[Mag+09] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. "Demand-driven evaluation of collection attributes". In: *Automated Software Engineering* 16.2 (2009), pp. 291–322.

[PH97] Arnd Poetzsch-Heffter. "Prototyping Realistic Programming Languages Based on Formal Specifications". In: *Acta Informatica* 34.10 (1997), pp. 737–772.

[Rep82] Thomas W. Reps. "Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors". In: *POPL*. Ed. by Richard A. DeMillo. ACM Press, 1982, pp. 169–176.

[RT84] Thomas W. Reps and Tim Teitelbaum. "The Synthesizer Generator". In: *Software Development Environments (SDE)*. Ed. by William E. Riddle and Peter B. Henderson. ACM, 1984, pp. 42–48.

[Sch+08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. "Sound and extensible renaming for Java". In: *OOPSLA*. Ed. by Gail E. Harris. ACM, 2008, pp. 277–294.

[SH10] Emma Söderberg and Görel Hedin. "Automated Selective Caching for Reference Attribute Grammars". In: *SLE*. Ed. by Brian Malloy, Steffen Staab, and Mark van den Brand. Vol. 6563. LNCS. Springer, 2010, pp. 2–21.

[SH11] Emma Söderberg and Görel Hedin. "Building semantic editors using JastAdd: tool demonstration". In: *LDTA*. Ed. by Claus Brabrand and Eric Van Wyk. ACM, 2011, p. 11.

[Vog+89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. "Higher-Order Attribute Grammars". In: *PLDI*. Ed. by Richard L. Wexelblat. ACM Press, 1989, pp. 131–145.

A Comparative Study of Incremental Attribute Grammar Solutions to Name Resolution

Abstract

Attribute grammars are useful in integrated editing environments for obtaining automatic incremental compilation features. However, traditional attribute grammars use aggregated values during name resolution, resulting in large sets of affected attribute instances after edits. In this paper, we show how reference attribute grammars (RAGs) can significantly reduce the number of affected attributes. We also introduce a notion of cache independent attributes used to limit propagation after edits. Our results indicate that RAGs are a highly viable alternative for use in editing environments.

1 Introduction

Traditional attribute grammars (AGs) [Knu68; Boc76; Dem+81] use aggregated values to propagate symbol tables with valid names used during semantic analysis for name resolution. In incremental evaluation of AGs, attribute values are updated

Emma Söderberg and Görel Hedin.
In electronic proceedings of the *5th International Conference on Software Language Engineering (SLE'12)*, Dresden, Germany, September 2012.

after edits to the syntax tree [Dem+81], and a static attribute evaluation order allows for an optimal propagation of change [Rep82]. Using re-evaluation, attribute values can be tested for change and propagation can be limited for unaffected attributes. However, this optimization does not work well for aggregated values, because aggregated values are composed of small parts, and even if only a small part depends on a change the entire value becomes affected, see e.g., [HT86].

Reference attribute grammars (RAGs) [Hed00] extend AGs with *reference attributes*, i.e., attributes that may refer to distant nodes in the abstract syntax tree, and *parameterized attributes*, i.e., attributes may have parameters. These two extensions allow for a different approach to name resolution, where parameterized attributes are used for name lookup, and reference attributes connect uses to declarations [EH05]. RAGs can be incrementally evaluated by construction of a dynamic dependency graph [SH12]. We show in this paper that RAGs do not have the same problem as AGs with aggregated values since the support of parameterized attributes allow for more fine grained attribute values. We also show how detection of *cache independent* attributes, i.e. attributes only depending on tokens, can help to limit notification after edits.

We start this paper with a description of the name resolution problem and how this typically is solved using an AG-based approach and how it is solved using a RAG-based approach in Section 2. We compare the presented approaches in Section 3 where we show how the RAG-based approach significantly reduces the number of affected attributes, and how detection of cache independent attributes clearly reduces the number of notified attributes for RAGs. Finally, we conclude the paper in Section 4.

2 Name Resolution

As a basis for the comparison in this paper, we use a simple language with nesting, declarations and uses taken from Bochmann [Boc76]. For the benefit of our comparison, the grammar has been translated to an object-oriented model. We include an assignment statement to hold uses, and include types and type literals. The abstract grammar is shown in Figure 1. We use this grammar to do *name resolution* using AGs and RAGs. The name resolution task can be defined as follows: define an attribute `type` for each `Use` node with a value corresponding to the type of its declaration.

Traditional AGs [Knu68], decorate AST nodes with attributes defined by equations (also called semantic functions) over other attributes. There are two kinds of attributes, *synthesized* and *inherited*, used for propagating information upwards and downwards in the AST. Name resolution is typically implemented by pairing names and types from declarations and synthesizing this information upwards in the AST to the nearest block. The pairs are aggregated into a symbol table map

```
Program ::= Block;
Block   ::= StmtList;
abstract StmtList;
CombStmtList : StmtList ::=
    StmtList Stmt;
SingleStmtList : StmtList ::= Stmt;
abstract Stmt;
BlockStmt  : Stmt ::= Block;
DeclStmt   : Stmt ::= Decl;
AssignStmt : Stmt ::=
    Left:Use Right:Use;
```

```
Decl ::= Type <ID:String>;
Type ::= <ID:String>;
abstract Use;
IdUse : Use ::= <ID:String>;
IntLiteral : Use ::=
    <VAL:String>;
abstract BoolLiteral : Use;
TrueLiteral  : BoolLiteral;
FalseLiteral : BoolLiteral;
```

Figure 1: Abstract grammar for the Block language Each line corresponds to an AST node type. A type `X` may be abstract (in an object-oriented sense), and may inherit from another type `Y` (indicated with '`X : Y`'). Children of a node are listed on the right-hand side of '`::=`', with tokens given within angle brackets.

which is propagated downwards, using inherited attributes, to reach nodes where names are used.

The left part of Figure 2 shows the attributed AST for a small program, following Bochmann's example AG. Here, `dec` is the name-type pair for a single declaration, and `used` is the symbol table map propagated down to nodes that might use names. The `used` attribute of a particular node contains the name-type pairs for all declarations visible at that node. A `Use` node can use this attribute to look up its type. The attributes `org` and `upt` are maps that collect name-type pairs in order to help construct the `used` attributes. This solution builds heavily on aggregating information in maps and propagating these large aggregate values around in the AST.

RAGs extend traditional AGs with *reference* and *parameterized* attributes [Hed00]. A reference attribute has a value that refers to a distant AST node. Attributes of that distant node can be accessed via the reference attribute. A parameterized attribute is an attribute that takes arguments, i.e., it can be called as a function. Together, these extensions allow for a different approach to name resolution where aggregated values are avoided [EH05].

The right part of Figure 2 shows the RAG-decorated AST for the example program, exemplifying typical name resolution: Each `Use` node is decorated with a reference attribute `decl` that points to the appropriate declaration node. A `Use` node can use this `decl` attribute to access its type. To define the `decl` attributes, parameterized attributes `lookup(String)` return a reference to the declaration for a given name. These lookup attributes serve a similar role as the symbol table attributes in traditional AGs, but instead of aggregating all information about visible declarations into a map, they delegate queries to other parameterized attributes in order to find the appropriate declaration. Typically, a query can be delegated to a

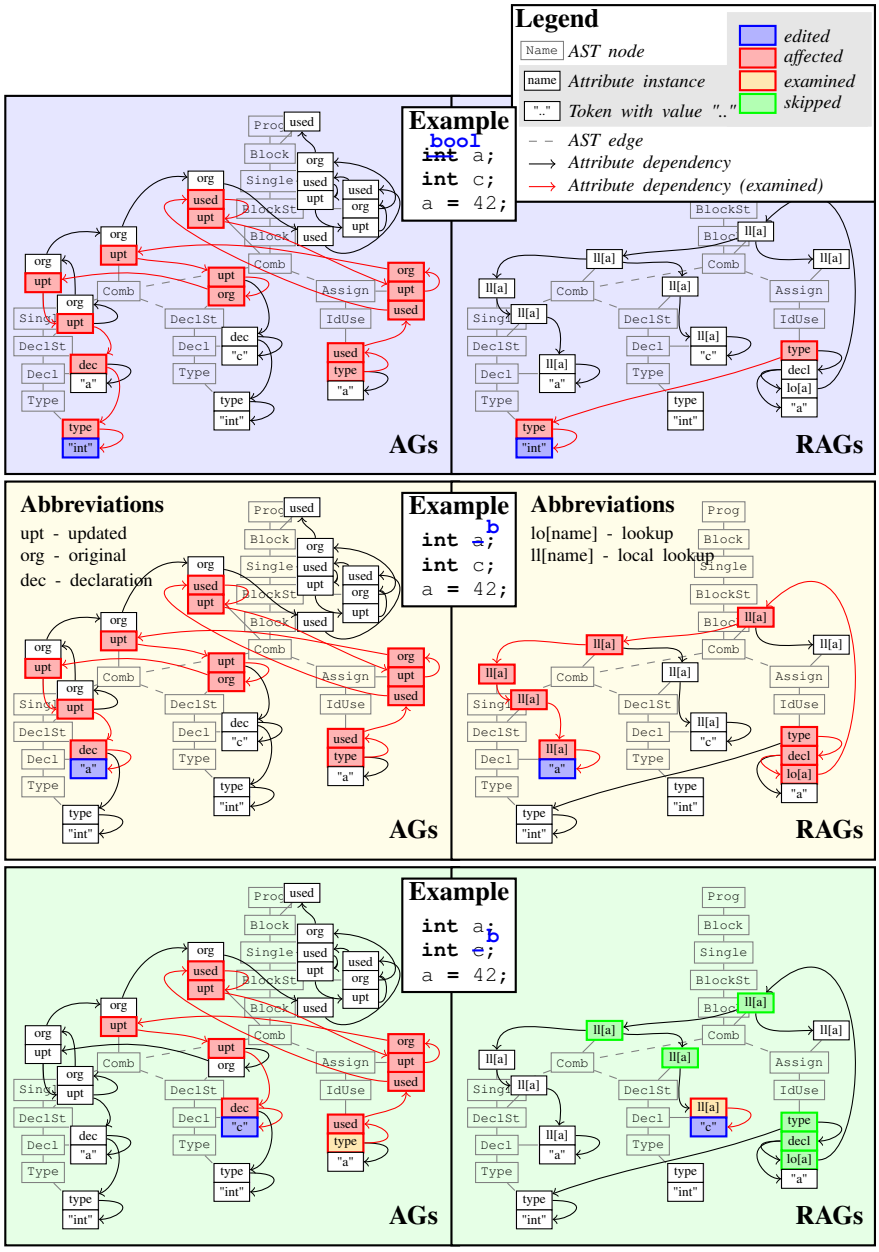


Figure 2: Three token edit scenarios An attribute instance that depends on an edit is either *affected* (re-evaluates to a different value), *examined* (notified, but with an unaffected value), or *skipped* (not notified, but guaranteed to have an unaffected value). Without the optimization for cache-independent attributes, all skipped attributes would need to be notified as well. (Picture best viewed in color)

`localLookup(String)` attribute that performs a local search for a requested name, e.g., in a `Block`. In RAGs, only attributes that are accessed are computed and stored (cached). Furthermore, for parameterized attributes, individual call results are cached. A traditional AG typically computes and stores all attributes.

Given the two presented approaches, we want to incrementally update values after an edit to the syntax tree. Typically, this is done by keeping track of dependencies, either statically, based on the attribute grammar, or dynamically, based on the actual syntax tree. When an edit is performed, dependent attributes are notified so that their values may be updated. To prevent that unaffected attributes are notified, attribute re-evaluation and notification can be interleaved so that notification to dependent attributes is only done if a re-evaluated attribute is affected, i.e., if its value has changed. This approach is called *change propagation* and was introduced for AGs in [Dem+81].

However, even though change propagation limits notification it can cause unnecessary re-computations for attributes notified more than once due to multiple dependencies on the same edit. As a solution, Reps introduced an optimal algorithm which uses static information about attribute evaluation order to find an optimal notification order [Rep82]. However, in the presence of reference attributes, there is no useful static evaluation order, making this algorithm inapplicable to RAGs.

For RAGs, we instead track dependencies dynamically, and notify all attributes dependent on an edit. To get some of the benefits of the change propagation algorithm, we use a simple scheme where we dynamically detect so called *cache independent* attributes, that is, attributes that do not depend on other stored/cached attributes, but only on tokens. Cache independent attributes are re-evaluated during the notification phase, and if their value is unchanged, their dependent attributes are not notified. Figure 2 shows the notifications after three edit scenarios on a small example program where this approach is used. We can note that both the number of dependent and affected attributes are much lower for the RAG solution than for the AG solution. Furthermore, the use of cache-independent attributes for RAGs is very effective, resulting in very few notifications of unaffected attributes.

3 Comparison

To compare the AG and RAG approaches, we have implemented two name resolvers for the Block language, one for each approach, using the JastAdd system [Jas]. Each analyzer performs name resolution and supports incremental evaluation after edits of tokens. The full implementation used in this comparison is available on-line [Söd12].

Input programs We have constructed a Block program that can be generated to different sizes. The program is constructed as a pattern extendible at `REPEATED`

HERE, corresponding to a repetition of lines 4 to 9 in the program. The program size is measured in the number of repetitions used, so the size of the program below is 1. The table below shows the program size, the number of AST nodes (|AST|), and the number of nodes in the dynamic dependency graph (|DDG|) [SH12] for the RAG and AG approaches.

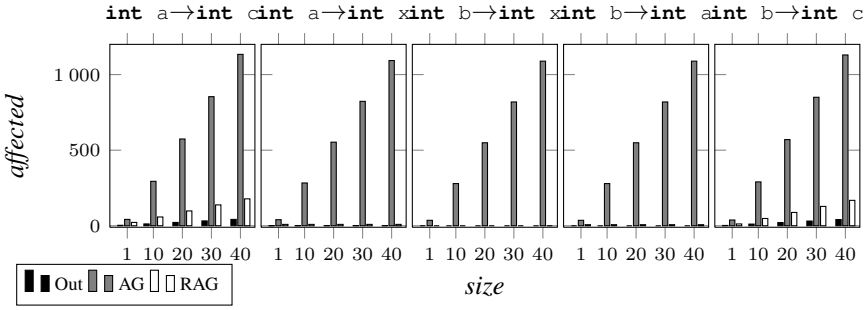
```
1 int a;//change A,B
2 int b;//change C,D,E
3 a = c;
4 {
5   int a;
6   a = c;
7   d = e;
8   REPEATED HERE
9 }
```

Program size	|AST|	$\|DDG_{RAG}\|$	$\|DDG_{AG}\|$
1	29	142	118
10	164	852	721
20	314	1643	1391
30	463	2433	2061
40	614	3223	2731

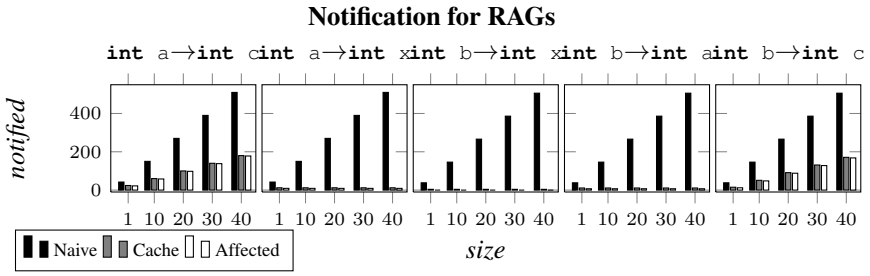
Edits When tokens are edited, a declaration may switch between being used and *not* used, and a use may switch between being declared and *not* declared. The figure below captures such edit scenarios. All edits to `int a` means an edit to the declaration on line 1, and all edits to `int b` means a change to the declaration on line 2. Note that these declarations have the same line number regardless of program size.

Affected attributes For our comparison, we consider affected attributes for each approach in relation to the number of affected *output* attributes, i.e., attributes corresponding to features directly visible to the user. Output attributes in this case are the `type` attributes for `Use` nodes. The figure below shows the number of affected output attributes (Out) along with the total number of affected attributes for the two approaches (AG, RAG). As shown in the figure, the RAG approach significantly reduces the number of affected attributes compared to the AG approach. The results for RAGs are strongly correlated with Out, whereas for AGs the results are correlated with program size. The distance between Out and RAG is due to the helper attributes used to compute the values of the output attributes.

Affected attributes after edits for AGs and RAGs



Notifications In addition, we compare the difference between using the naive change notification approach – notifying all dependent attributes, and using the detection of cache independence approach. We focus on RAGs for this comparison since a large difference between the number of affected and dependent attributes is key to making this optimization useful. The figure below shows the number of dependent attributes (Naive), the number of notified attributes using the cache independence approach (Cache), and the number of affected attributes for RAGs (Affected). As shown in the figure, the Naive approach will notify many unaffected attributes, whereas the Cache approach is very close to the affected set (the optimum).



4 Conclusions

We have shown how the use of RAGs significantly reduces the number of affected attributes, and that the number of affected attributes is more correlated with the number of attributes viewed by the user, than with program size. Furthermore, we have exemplified how dynamic detection of cache-independent attributes can reduce the set of notified attributes almost down to the optimal affected set. These results indicate that RAGs are a highly viable alternative for use in integrated editing environments. Future work includes evaluation of larger languages and examples, and exploration of coarser-grained dependency analysis to reduce memory footprint.

References

[Boc76] Gregor von Bochmann. “Semantic Evaluation from Left to Right”. In: *Communications of the ACM* 19.2 (1976), pp. 55–62.

[Dem+81] Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. “Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors”. In: *POPL*. Ed. by John White, Richard J. Lipton, and Patricia C. Goldberg. ACM Press, 1981, pp. 105–116.

[EH05] Torbjörn Ekman and Görel Hedin. "Modular Name Analysis for Java Using JastAdd". In: *GTTSE*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science. Springer, 2005, pp. 422–436.

[Hed00] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[HT86] Roger Hoover and Tim Teitelbaum. "Efficient incremental evaluation of aggregate values in attribute grammars". In: *SIGPLAN Symposium on Compiler Construction*. Ed. by Richard L. Wexelblat. ACM, 1986, pp. 39–50.

[Jas] *JastAdd*. `http://jastadd.org`.

[Knu68] Donald E. Knuth. "Semantics of context-free languages". In: *Journal Theory of Computing Systems* 2.2 (1968), pp. 127–145.

[Rep82] Thomas W. Reps. "Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors". In: *POPL*. Ed. by Richard A. DeMillo. ACM Press, 1982, pp. 169–176.

[Söd12] Emma Söderberg. *Evaluation Link: A Comparative Study of Incremental Attribute Grammar Solutions to Name Resolution*. `http://svn.cs.lth.se/svn/jastadd-research/public/evaluation/sle-12-inc`. 2012.

[SH12] Emma Söderberg and Görel Hedin. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*. Tech. rep. 98. LU-CS-TR:2012-249, ISSN 1404-1200. Lund University, 2012.

PRACTICAL SCOPE RECOVERY USING BRIDGE PARSING

Abstract

Interactive development environments (IDEs) increase programmer productivity, but unfortunately also the burden on language implementors since sophisticated tool support is expected even for small domain-specific languages. Our goal is to alleviate that burden, by generating IDEs from high-level language specifications using the JastAdd meta-compiler system. This puts increased tension on scope recovery in parsers, since at least a partial AST is required by the system to perform static analysis, such as name completion and context sensitive search. In this paper we present a novel recovery algorithm called bridge parsing, which provides a light-weight recovery mechanism that complements existing parsing recovery techniques. An initial phase recovers nesting structure in source files making them easier to process by existing parsers. This enables batch parser generators with existing grammars to be used in an interactive setting with minor or no modifications. We have implemented bridge parsing in a generic extensible IDE for JastAdd based compilers. It is independent of parsing technology, which we validate by showing how it improves recovery in a set of typical interactive editing scenarios for three parser generators: ANTLR (LL(variable lookahead) parsers), LPG (LALR(k) parsers), and Beaver (LALR(1) parsers). ANTLR and LPG both contain sophisticated support for error recovery, while Beaver requires manual er-

Emma Nilsson-Nyman, Torbjörn Ekman, and Görel Hedin.
Lecture Notes of Computer Science, 2009, Vol. 5452, pp. 95–113, Springer Berlin/Heidelberg.

ror productions. Bridge parsing complements these techniques and yields better recovery for all these tools with only minimal changes to existing grammars.

1 Introduction

Interactive development environments (IDE) have become the tool of choice in large-scale software development. This drastically increases the burden on language developers since sophisticated tool support is expected even for small domain-specific languages. The work presented in this paper is part of a larger effort to generate IDEs from high-level language specifications based on attribute grammars in the JastAdd meta-compiler tools. The AST is used as the predominant data structure and all static semantic analyses are implemented as attribute grammars on top of that tree. This approach has been used successfully to implement a production-quality Java compiler [EH07a], extensible refactoring tools [Sch+08], source level control-flow and data-flow analyses [NN+09a], and various Java extensions [Avg+08; Hua+08].

One key insight from our earlier work is that we can use the AST as the only model of the program and then superimpose additional graph structure using attributes on top of that tree, e.g., name bindings, inheritance hierarchies and call graphs. The IDE can then reuse and extend this model to provide services such as name completion, cross-referencing, code outline, and semantic search facilities. However, this allows us to use the same extension mechanisms that have proven successful in building extensible compilers to build extensible IDEs. This puts extreme tension on the error recovery facilities in parsers since incomplete programs are the norm rather than the exception during interactive development, and a recovered AST is necessary for instant feedback to the user. An unfortunate consequence of this challenge is that, despite the wealth of research in automatic parser generators from high-level grammars and sophisticated error recovery mechanisms, most IDEs still rely on hand crafted parsers to provide such services to the user.

In this paper we present an algorithm for scope recovery, preserving as much of the AST structure as possible, that is neither tied to a particular parsing technology nor to a specific parser generator. A light-weight pre-processor takes an incomplete program and recovers scope nesting before the adjusted source file is fed to a traditional parser. Existing error recovery mechanisms are then sufficient to build a partial AST suitable for static semantic analysis. This approach even makes it feasible to use a batch parser rather than an incremental parser since the speed of parsing a complete source unit is usually satisfactory even for interactive editing on today's hardware.

The approach has proven successful when combined with several parser generators in our IDE generator for JastAdd based compilers in the Eclipse IDE framework. We have integrated bridge parsing in an IDE framework for compilers where

all services use the AST as the only model to extract program information from. An IDE based on Eclipse is generated from an attribute grammar for static semantic analysis and our largest example is an environment for Java which includes support for name completion, content outline, cross-referencing, and various semantic search facilities.

We demonstrate that the approach is independent of parsing technology and parser generator by combining bridge parsing with three different parser generators: ANTLR which generates an LL(variable lookahead) parser, LPG which generates an LALR(k) parser, and Beaver which generates an LALR(1) parser. LPG and ANTLR are particularly interesting because of their sophisticated error recovery mechanisms. We show that on a set of typical interactive editing scenarios, bridge parsing improves recovery for both these tools and the few cases with degraded performance can be mitigated by minor enhancements in the IDE. The contributions of this paper are:

- A general algorithm for recovering scope information suitable for arbitrary parsing technologies.
- An implementation in an IDE generator for JastAdd based compilers in Eclipse.
- A careful comparison of its effect on error recovery in state of the art parser generators.

The rest of the paper is structured as follows. Section 2 explains the requirements on error recovery and outlines previous work and room for improvement. Bridge parsing is introduced in Section 3 and an example of language sensitive recovery for Java is presented in Section 4 and evaluated in Section 5. We finally discuss future work and conclude in Section 6.

2 Background

We first outline scenarios requiring error recovery in the setting described above, and then survey existing recovery mechanisms and explain why they are not sufficient or have room for improvement. This serves as a background and related work before we introduce bridge parsing in Section 3. A more thorough survey of error recovery techniques is available in [DP95].

2.1 Error recovery scenarios

The interactive setting this approach is used in puts extreme pressure on error recovery during parsing. Incomplete programs with errors are the norm rather than the exception, and we rely on an AST to be built to perform most kinds of IDE services to the user. It is therefore of paramount importance that common editing

scenarios produce an AST rather than a parse failure to allow for services such as name completion while editing. There is also a tension between sophisticated error recovery and the goal to lower the burden on language developers building IDEs. Ideally, she should be able to reuse an existing parser, used in the compiler, in the IDE with minimal changes while providing satisfactory recovery. We define the following desirable properties of error recovery in this context:

- Support for arbitrary parser generators and parsing formalisms.
- Only moderate additions to add recovery to a specific language grammar.
- Effective in that recovery is excellent for common editing scenarios.

The motivation behind the goal of using arbitrary parser generators is that JastAdd defines its own abstract grammar and as long as the parser can build an AST that adheres to that grammar it can be used with JastAdd. We can thus benefit from the wealth of available parsing techniques by selecting the most appropriate for the language at hand. Notice that many proposed parser formalisms are orthogonal to the problem of handling incomplete programs, e.g., GLR-parsing[Tom85], Earley-parsing[Ear70], and Parsing Expression Grammars [For04], all deal with ambiguities in grammars rather than errors in programs.

The overall goal of the project is to lower the burden on language developers who want to provide IDE support for their languages. The extra effort to handle incomplete programs should therefore be moderate compared to the overall effort of lexical and syntactic analysis.

The effectiveness criterion is a very pragmatic one. We want the automatic recovery to be as close as possible to what a user would manually do to correct the problem. Here we borrow the taxonomy from Pennello and DeRemer [PD78] and consider a correction *excellent* if it repairs the text as a human reader would have, otherwise as *good* if the result is a reasonable program and no spurious errors are introduced, and otherwise as *poor* if spurious errors are introduced.

Consider the simple incomplete program below. A class `C` with a method `m()` and a field `x` is currently being edited. There are two closing curly braces missing. An excellent recovery, and the desired result of an automatic recovery, would be to insert a curly brace before and after the field `x`. A simpler recovery, with poor result, would be to insert two curly braces at the end of the program.

```
class C {
  void m() {
    int y;
  int x;
```

Notice that we need to consider indentation to get an excellent result. Changing the indentation of the field `x` to the same as for the variable `y` should, for instance, change its interpretation to a variable and result in a recovery where both

closing braces are inserted at the end of the file. Meaning that the simpler recovery alternative above would be sufficient.

2.2 Simple recovery

The simplest form of recovery is to let the parser automatically replace, remove, or insert single tokens to correct an erroneous program. This information can easily be extracted from the grammar and is supported by many parser generators. However, for incomplete programs this is insufficient since series of consecutive tokens are usually missing in the source file. It should be noted that this kind of recovery serves as a nice complement to other recovery strategies by correcting certain spelling errors.

2.3 Phrase recovery

A more advanced form of recovery is phrase level recovery where text that precedes, follows, or surrounds an error token is replaced by a suitable nonterminal symbol [Gra+79]. Beacon symbols, e.g., delimiters, and keywords, are used to re-synch the currently parsed production and erroneous tokens are removed from the input stream and replaced by a particular error token. These symbols are language specific and the parsing grammar therefore needs to be extended with error productions unless automatically derived by the parser generator. This form of recovery works very well in practice when beacon symbols are available in the input stream and implemented in many parsing tools. Incomplete programs usually lack some beacons required to re-synch the parser and often result in a parsing failure where the recovery can not proceed.

2.4 Scope recovery

Hierarchical structure is usually represented by nested scopes in programming languages, e.g., blocks and parentheses. It is a common error to forget to close such scopes and during interactive editing many scopes will be incomplete. Burke and Fisher introduced scope recovery to alleviate such problems [BF87]. Their technique requires the language developer to explicitly provide symbols that are closing scopes. Charles improves on that analysis by automatically deriving such symbols from a grammar by analyzing recursively defined rules [Cha91]. Scope recovery can drastically improve the performance of phrase recovery since symbols to open and close scopes are often used as beacons to re-synch the parser. Scope recovery usually discards indentation information which unfortunately limits the possible result of the analysis to good rather than excellent for the previously outlined example.

2.5 Incremental parsing

An interactive setting makes it possible to take history into account when detecting and recovering from errors. This opens up for assigning blame to the actual edit that introduced an error, rather than to the location in the code where the error was detected. A source unit is not parsed from scratch upon a change but instead only the changes are analyzed. Wagner and Graham present an integrated approach of incremental parsing and a self versioning document model in [WG98; Wag98]. It is worth noting that this requires a deep integration of the environment and the generated parser, which makes it less suitable for our setting due to the goal of supporting multiple parser generators.

2.6 Island parsing

Island parsing is not an error recovery mechanism per se but rather a general technique to create robust parsers that are tolerant to syntactic errors, incomplete source code, and various language dialects [Moo01]. It is based on the observation that for source model extraction one is often only interested in a limited subset of all language features. A grammar consists of detailed productions describing language constructs of interests, that are called islands, and liberal productions matching the remaining constructs, that are called water. This allows erroneous programs to be parsed as long as the errors are contained in water. The parts missing in an incomplete program are usually a combination of both water and islands which makes this approach less suitable for extracting hierarchical structure on its own.

3 Bridge Parsing

We now present *bridge parsing* as a technique to recover scope information from an incomplete or erroneous source file. It combines island parsing with layout sensitive scope recovery. In [Moo01] Moonen defines island grammars [Moo02; DK99] as follows:

*An **island grammar** is a grammar that consists of two parts: (i) detailed productions that describe the language constructs that we are particularly interested in (so called **islands**), and (ii) liberal productions that catch the remainder of the input (so called **water**).*

In bridge parsing, tokens which open or close scopes are defined as islands while the rest of the source text is defined as water or reefs. Reefs are described further in Section 3.1. This light-weight parsing strategy is used to detect open scopes and to close them. The end product of the bridge parser is a recovered source representation suitable for any parser that supports phrase level recovery.

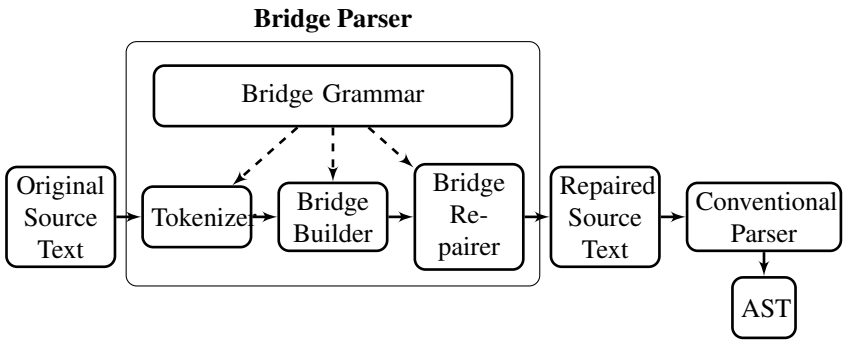


Figure 1: A bridge parser consists of three parts – a tokenizer returning a token list, a bridge builder taking a token list and returning a bridge model and a bridge repairer which takes a bridge model and generates a repaired source text.

A bridge parser is built from three parts, as illustrated in Figure 1. The first part, the tokenizer, takes a source text and produces a list of tokens based on definitions in the *bridge grammar*. The second part, the bridge builder, constructs a bridge model from the token list and the last part, the bridge repairer, analyses and repairs the bridge model.

3.1 Tokenizer

The goal of the tokenizer is to produce a list of tokens from a source text. Token definitions are provided by a bridge grammar which extends the island grammar formalism with bridges and reefs:

*A **bridge grammar** extends an island grammar with the notions of **bridges** and **reefs**: (i) reefs are attributed tokens which add information to nearby islands, and (ii) bridges connect matching islands. All islands in a bridge grammar are seen as potential bridge abutments.*

We are primarily interested in tokens that define hierarchical structure through nested scopes. Tokens that open or close a scope are therefore represented as islands in the grammar, e.g., braces and parentheses in Java, while most other syntactic constructs are considered water. However, additional tokens in the stream may be interesting to help match two islands that open and close a particular scope. Typical examples include indentation and delimiters, and we call such tokens *reefs*. Reefs can be annotated with attributes which enables comparison between reefs of the same type. Indentation reefs may for instance have different indentation levels. We call such tokens attributed tokens:

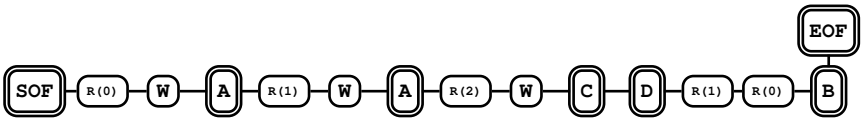


Figure 2: The token list is a simple double-linked list of *islands* (A, B, C, D), *reefs* (R) and *water* (W). Two additional nodes representing *start of file* (SOF) and *end of file* (EOF) are added first and last in the list. The numbers within parentheses show the attribute values of the reefs.

Attributed tokens *are tokens which have attributes, which potentially makes them different from other attributed tokens of the same type. This difference makes it possible to compare attributed tokens to each other.*

The first step when defining a bridge grammar, is to specify islands and reefs. The following example, which we will use as a running example, shows the first part of a bridge grammar relevant to the tokenizer:

Listing VI.1: Tokenizer Definitions

```
1 islands SOF, EOF, A=.., B=.., C=.., D=..
2 reefs R(attr)=..
```

The bridge grammar defines four island types and one reef type. The SOF and EOF islands, represent *start of file* and *end of file*. The A and B islands could, for instance, be open and close brace and C and D open and close parenthesis. The reef could represent indentation where the value of $attr$ is the level of indentation. In our implementation each island corresponds to a class that accepts and consumes the substring matching its lexical representation. The information given so far in the example grammar is sufficient to create a lexer which will produce a list of tokens, e.g., the example in Figure 2.

3.2 Bridge Builder

The bridge builder takes a token list and produces a bridge model defined as follows:

A ***bridge model*** *is a token list where matched islands are linked with bridges in alignment with the nesting structure in a source text. Bridges can be enclosing other bridges or be enclosed by other bridges, or both, but they never cross each other in an ill-formed manner. Unmatched islands point out broken parts of the nesting structure.*

In the bridge model, islands opening and closing a scope should be linked with a bridge. For the bridge builder to know between which islands to build bridges we need to add definitions to the bridge grammar.

Listing VI.2: Bridge Builder Definitions

```
1 bridge from SOF to EOF { ... }
2 bridge from [a:R A] to [b:R B] when a.attr = b.attr { ... }
3 bridge from [a:R C] to [b:R D] when a.attr = b.attr { ... }
```

The first bridge, the *file bridge*, between the `SOF` island and `EOF` island does not need any additional matching constraints i.e., constraints that define when two islands of given types match. For other bridges additional constraints besides type information i.e., an island of type `A` and type `B` match, are usually needed. In the example above the islands of type `A` and `B` match if both islands have reefs of type `R` to the left which are equal. The constraints for the third bridge are similar.

The order of the bridge abutments in the definition should correspond to the order in the source text e.g., the bridge start of type `C` is expected to occur before the bridge end of type `D`.

With the information given so far in our example bridge grammar we can construct a bridge model. The BRIDGE-BUILDER algorithm will construct as many bridges as possible. If at some point no matching island can be found the algorithm will jump over the unmatched island and eventually construct a bridge enclosing the island. The algorithm is finished when the file bridge has been constructed.

The BRIDGE-BUILDER algorithm, listed in Figure 4, will iterate through the token list until it manages to build the file bridge. For each unmatched bridge start it will try to match it to the next island. If there is a match a bridge will be built and otherwise the algorithm will continue with the next unmatched bridge start. Each time a bridge is encountered it will be crossed. In this way the number of encountered unmatched islands will decrease each iteration.

The algorithm will try to match an unmatched island to the next unmatched island within the current tolerance. The tolerance defines how many unmatched islands that are allowed below a bridge. This value is only changed between iterations and depends on the success of the last iteration. The tolerance should be as low as possible. We start out with a tolerance of zero, meaning no unmatched islands under a bridge. This value will increase if we fail to build bridges during an iteration. If we have a successful iteration we reset the tolerance to zero. If we apply this algorithm to the running example we end up with four iterations, illustrated in Figure 3.

3.3 Bridge Repairer

The goal of the bridge repairer is to analyze the bridge model and repair it if necessary. The BRIDGE-REPAIRER algorithm needs to locate remaining unmatched

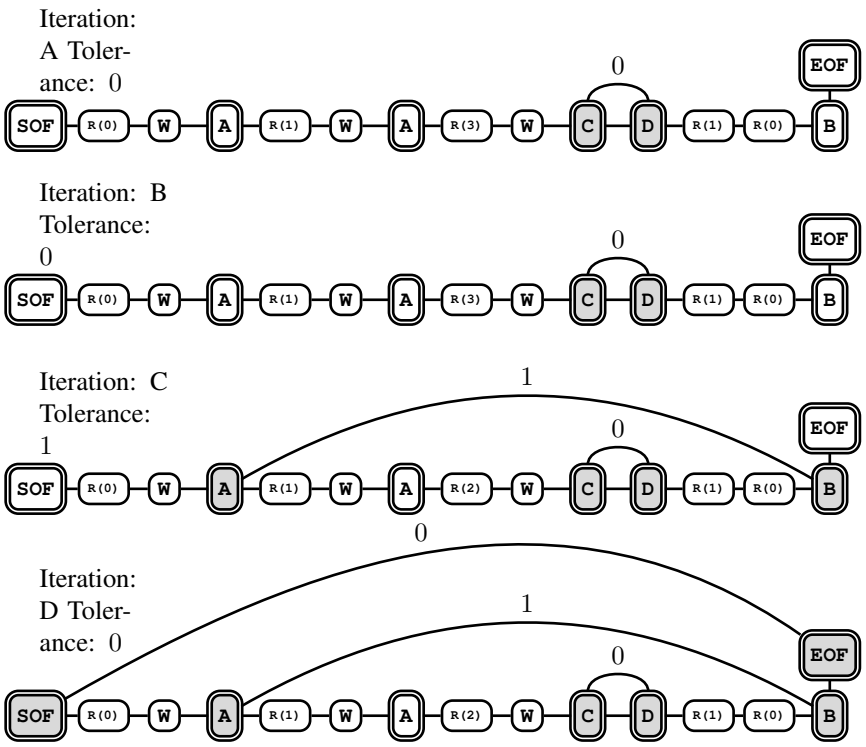


Figure 3: The resulting bridge model after running the BRIDGE-BUILDER algorithm. No bridges were built during iteration B which results in an increased tolerance in iteration C. During iteration C a bridge is built which means the tolerance is reset to zero in iteration D.

```
BUILD-BRIDGES(sof)
 1  tol ← 0
 2  while ¬HAS-BRIDGE(sof)
 3        do start ← sof
 4           change ← FALSE
 5           while start ≠ NIL
 6                 do end ← NEXT-UNMATCHED-ISLAND(start, tol)
 7                    if BRIDGE-MATCH(start, end)
 8                      then BUILD-BRIDGE(start, end)
 9                           change ← TRUE
10                           start ← NEXT-UNMATCHED-START-ISLAND(end)
11                      else start ← NEXT-UNMATCHED-START-ISLAND(start)
12           if ¬ change
13             then tol ← tol +1
14             else if tol > 0
15                      then tol ← 0
```

Figure 4: The BUILD-BRIDGES algorithm constructs a bridge model from a token list. The NEXT-UNMATCHED-ISLAND returns the next unmatched island to the right of the given island. The tolerance defines the number of unmatched islands to jump over before the procedure returns.
NEXT-UNMATCHED-START-ISLAND is similar but only looks for bridge starts.

islands and to add *artificial islands* which will match them. Each island defined in the bridge grammar can potentially be missing. When an island is missing, the algorithm will search for an appropriate *construction site* where an artificial island can be inserted. The search for a construction site starts from the bridge start if the bridge end is missing, and vice versa, and ends when a match is found or an enclosing bridge is encountered. With this in mind we add additional definitions to our example bridge grammar:

Listing VI.3: Bridge Repair Definitions

```
1 bridge from [a:R A] to [b:R B] when a.attr = b.attr {
2     missing[A] find [c:R] where c.attr = b.attr insert after
3     missing[B] find [c:R] where c.attr = a.attr insert after
4 }
5 bridge from [a:R C] to [b:R D] when a.attr = b.attr {
6     missing[C] find [c:R] where c.attr = b.attr insert after
7     missing[D] find [c:R] where c.attr = a.attr insert after
8 }
```

If an island of type `A` is missing, a construction site will be found in the interval starting at the unmatched island of type `B` and ending at the start of the enclosing bridge. The first reef of type `R` which is equal to the reef to the left of the unmatched island of type `B` points out a construction site. The final information we need is how to insert the artificial island. In this case the artificial island should be inserted after the reef. The definitions for the remaining islands are similar.

The BRIDGE-REPAIRER algorithm, listed in Figure 5, recursively repairs unmatched islands under a bridge, starting with the file bridge. When an unmatched island is encountered the MEND algorithm, listed in Figure 6, will locate a construction site and insert an artificial island.

In the running example, an island of type `A` is missing an island of type `B`. The island of type `A` has a reef of type `R` on its left hand side with value `1` which means the algorithm will search for for a another reef the same type with the same value. The search is stopped when either a reef is found or a closing island of an enclosing scope is encountered. In this case there is a reef of the right type and value, before the enclosing island of type `B`, which points out a construction site.

The result of the BRIDGE-REPAIRER algorithm is shown in Figure 7. An artificial island of type `B` has been inserted to form a bridge with the previously unmatched island of type `A`.

4 Bridge Parsing for Java

To construct a bridge parser for Java we need to create a bridge grammar which can provide information to each of the three parts of the bridge parser, illustrated

```
BRIDGE-REPAIRER(bridge)
 1  start ← START(bridge)
 2  end ← END(bridge)
 3  island ← NEXT-ISLAND(start)
 4  while island ≠ end
 5        do if ¬HAS-BRIDGE(island)
 6              then if START-OF-BRIDGE(island)
 7                      then MEND-RIGHT(island, end)
 8                      else MEND-LEFT(start, island)
 9           bridge ← BRIDGE(island)
10           BRIDGE-REPAIRER(bridge)
11           island ← NEXT-ISLAND(END(bridge))
```

Figure 5: The BRIDGE-REPAIRER algorithm constructs artificial islands to match unmatched islands. The MEND-RIGHT algorithm is described further in Figure 6. The MEND-LEFT algorithm is symmetric to the MEND-RIGHT algorithm.

```
MEND-RIGHT(broken, end)
1  node ← NEXT(broken)
2  while node ≠ end
3        do if HAS-BRIDGE(node)
4              then node ← NEXT(BRIDGE-END(node)
5              else if POSSIBLE-CONSTRUCTION-SITE(broken, node)
6                      then CONSTRUCT-ISLAND-AND-BRIDGE(broken, node)
7                           return
8              else node ← NEXT(node)
9  CONSTRUCT-ISLAND-AND-BRIDGE(broken, PREVIOUS(end))
```

Figure 6: The MEND-RIGHT algorithm constructs an artificial bridge end in the interval starting at the unmatched bridge start (*broken*) and ending at the end of the enclosing bridge (*end*).

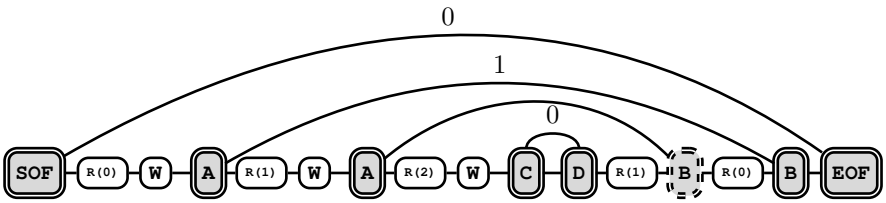


Figure 7: The changes in the example bridge model after the BRIDGE-REPAIRER algorithm is done. The artificial island is marked with a dashed double edge.

in Figure 1. We will define this bridge grammar for Java in three steps which gradually will include more complex language-specific information. The performance impact of these different levels of language sensitivity is evaluated in Section 5.

For Java, and other languages with similar language constructs, we need to consider how to deal with comments and strings. These constructs might enclose text which would match as reefs or islands but which should be ignored. In our implementation we handle this separately in the lexer implementation.

4.1 Scopes

The first level of language sensitivity is to only consider tokens directly defining scopes. We therefore include braces and parentheses as islands since they have a direct impact on the nesting structure of Java code. Indentation is also included to enhance matching of islands in incomplete code. The complete bridge grammar looks like this:

Listing VI.4: Bridge Repairer Definitions

```
1  islands SOF, EOF, LBRACE, RBRACE, LPAREN, RPAREN
2  reefs INDENT(pos)
3
4  bridge from SOF to EOF
5
6  bridge from [a:INDENT LBRACE] to [b:INDENT RBRACE]
7      when a.pos = b.pos {
8      missing [RBRACE]
9          find [c:INDENT] where (c.pos <= a.pos) insert after
10     missing [LBRACE]
11         find [c:INDENT] where (c.pos <= a.pos) insert after
12 }
13
14 bridge from [a:INDENT LPAREN] to [b:INDENT RPAREN]
15     when a.pos = b.pos {
16     missing [RPAREN]
17         find [c:ISLAND] insert before
```

```
18          find [c:INDENT] where (c.pos <= a.pos) insert after
19      missing [LPAREN]
20          find [c:ISLAND] insert before
21          find [c:INDENT] where (b.pos <= c.pos) insert before
22  }
```

The `pos` attribute of the `INDENT` reef corresponds to the indentation level. Comparing two reefs of this type corresponds to comparing their `pos` attribute.

For the islands corresponding to right and left parentheses there are two **`find`** conditions. For cases like these the first occurrence that fulfills all its conditions decide which action to take.

4.2 Delimiters

To improve matching of parentheses we add additional reefs for delimiters. The following code snippet illustrates the benefit of defining commas as reefs during recovery:

```
void m(int a) {
  n(o(, a);   // Recover to "n(o(),a)" and not to "n(o(,a))"
}
```

We have a call to `o()` as the first argument in the call to `n()` in the method `m()`. The comma tells us that we are editing the first element in the list of arguments and that the call to `o()` should be closed right before the comma rather than after reading the `a` parameter. If we modify the code snippet and remove the other end parenthesis instead we end up with a different scenario:

```
void m(int a) {
  n(o(), a;   // Recover to "n(o(),a);" and not to "n(o(),a;)"
}
```

The analysis should ideally place a closing parenthesis somewhere after the comma but before the semicolon. The reason is that the comma separates elements within the parentheses and the call to `o()` is within the call to `n()`, while the semicolon separates elements in the block. To deal with these scenarios we define additional reefs to match delimiters and add additional **`find`** declarations to the missing parenthesis blocks.

Listing VI.5: Bridge Repairer Definitions

```
1  reefs .., COMMA, DOT, SEMICOLON
2
3  missing [RPAREN]
4      ..
5      find [c:COMMA] where (previous(c) != WATER) insert before
```

```
6       find [c:DOT] where (previous(c) != WATER) insert before
7       find [c:SEMICOLON] insert before
8
9   missing [LPAREN]
10      ..
11      find [c:COMMA] where (next(c) != WATER) insert after
12      find [c:DOT] where (next(c) != WATER) insert after
13      find [c:SEMICOLON] insert after
```

These definitions should be seen as extensions to the bridge grammar presented in the previous section. In the above **find** declaration for RPAREN we have added actions for when we encounter a COMMA, DOT or SEMICOLON while searching for a construction site.

4.3 Keywords

To further improve matching we can add keywords as reefs. This can be useful since keywords separate statements from each other. The following code snippet shows an example:

```
boolean m(boolean a) {
  if a == true)      // Recover to "if (a == true)"
    return false;    // and not to "(if a == true)"
  return true;
}
```

Ideally, the analysis should put the missing left parenthesis after the `if` keyword If the keyword has been defined as a reef this is possible, otherwise not since then keywords will be considered to be water. To deal with scenarios such as these we add additional keywords and **find** definitions to the bridge grammar:

Listing VI.6: Bridge Repairer Definitions

```
1   reefs KEYWORD (if, for, while ..)
2
3   missing [RPAREN]
4       find [c:KEYWORD] insert before
5
6   missing [LPAREN]
7       find [c:KEYWORD] insert after
```

5 Evaluation

We have chosen to evaluate bridge parsing on common editing scenarios for Java using the specifications described in Section 4. Our bridge parsing implementation

has been combined with a set of Java parsers, generated using state of the art parser generators based on LALR(1) and LL(variable lookahead) grammars. The parsers were generated with the following parser generators and settings:

- **Antlr** Generated using Antlr (v.3.0).
- **AntlrBT** Generated using a forth-coming version of Antlr (v.3.1 beta) with backtracking turned on. This version of Antlr introduces new error recovery not yet available in the latest stable version.
- **Beaver** Generated with Beaver (v.0.9.6.1).
- **BeaverEP** Generated with Beaver (v.0.9.6.1), with error productions manually added to the grammar.
- **LPG** Generated using LPG (v.2.0.12). While this is the newest version of LPG it does not yet provide a complete automatic scope recovery as suggested by Charles[Cha91]. To allow comparison with Charles approach the test cases where manually edited to correspond to the suggested recovery from the generated parser.

In lack of an existing appropriate benchmark suite we have created a test suite for incomplete and erroneous Java programs to use during evaluation. The test suite consists of several series of tests, each series with one *correct program* and one or more *broken programs*. The correct program in each series corresponds to the intention of the programmer, while the broken programs illustrate variations of the correct program where one or more tokens are missing. Broken programs in this setting illustrate how programs may evolve during an editing session. For each test series and parser we build a tree representing the correct program and try to do the same for each broken program. For the Antlr, AntlrBT and LPG we build parse trees, while for the Beaver and BeaverEP we build ASTs.

As a metric of how close a tree constructed from one of the broken programs is to the tree constructed for the correct program we use tree alignment distance, as described in [Jia+95]. To calculate tree alignment distance, a cost function is required which provides a cost for insertion, deletion and renaming of nodes. We use a simple cost function where all these operations have the cost one. As a complementary classification of success we use the categorization of recovery as excellent, good, or poor by Pennello and DeRemer [PD78].

5.1 Benchmark examples

The test suite consists of 10 correct test cases which have been modified in various ways to illustrate possible editing scenarios. The test suite provides a total of 41 tests. Full documentation of the test suite can be found at [Söd09]. We have focused on three editing scenarios:

Incomplete code Normally, programs are written in a top-down, left-right fashion. In this scenario we put test cases with incomplete code, to illustrate how code evolves while being edited in an IDE. An example where a user is adding a method `m()` to a class `C` which already contains two fields `x` and `z` may look like this:

```
class C {
    int x;
    void m() {
        int x;
        if (true) {
            int y;

    int z;
}
```

Missing start This scenario highlights situations which might occur when the normal course of writing a program top-down, left-right is interrupted. A typical example is that the user goes back to fix a bug or to change an implementation in an existing program. Consider the example below where the programmer has started changing a while loop which causes a missing opening brace:

```
class C {
    void m() {
        // while (true) {
            int a;
        }
    }
}
```

Tricky indentation Since bridge parsing relies on indentation it is reasonable to assume that tricky indentation is its Achilles heel. We therefore included a set of test cases with unintuative indentation to evaluate its performance on such programs. An example of nested blocks where indentation is not increasing is shown below:

```
class C {
    void m() {
}
 }
```

Another scenario leading to a similar situation is when a programmer pastes a chunk of code with different indentation in the middle of her program, as illustrated by this example:

```
class C {
    void n() { .. }
void m() {
}
}
```

5.2 Results

The results, after running through the test suite with our parser suite, are shown in Table 8, 9 and 10. The first table shows the results for tests with incomplete code, the second table shows results for tests with missing starts and the last table shows results for tests with tricky indentation. Each table has a column for each parser generator containing a set of tree editing distances: without bridge parsing, with bridge parsing using the scopes version, with bridge parsing using the delimiters version. After each tree editing distance set the result for the test case and parser generator is summarized with a letter indicating excellent (**E**), improved (**I**), status quo (**S**) or worse (**W**). A tree alignment distance of 0 indicates a full recovery and an excellent result while a missing value indicates total failure which is when no AST or parse tree could be constructed.

The leftmost column for each parser in the tables shows that the test suite presents many challenges to all parsers which manifest themselves in less than excellent recoveries, except for the test cases in Table 10 where only indentation is changed to trick the bridge parser. Without bridge parsing, LPG is much better than the other parser generators, most likely due to the built-in support for scope recovery.

The second column for each parser shows that all parser generators benefit vastly from bridge parsing in most cases, of 41 cases 19 improve from good or poor recovery to excellent. LPG still has the edge over the other generators with superior error recovery. We notice that using indentation can indeed improve scope recovery since the LPG results are improved in many cases.

The third column in each column set shows that there is almost no change when we use the delimiters version of the bridge parser instead of the scopes version. Generally, nothing changes or there are small improvement.

There are some problems with bridge parsing, as shown in Table 10, when there are inconsistencies in the layout. This problem could be alleviated by IDE support to automatically correct indentation during editing and pasting. Because of the current problems with some layout scenarios the bridge parser is only run when the parser fails to construct an AST and there is no other way to acquire an AST.

We have run tests with keyword sensitive bridge parsing as well, but saw no improvement using our test suite. There are certainly cases where this could yield an improvement but we could not easily come up with a convincing realistic editing scenario to include in the test suite.

Test	Antlr	AntlrBT	Beaver	BeaverEP	LPG
A1	-, 0, 0 **E**	65, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	6, 0, 0 **E**
A2	-, 0, 0 **E**	65, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	4, 0, 0 **E**
A3	75, 0, 0 **E**	1, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	0, 0, 0 **E**
A4	-, 0, 0 **E**	65, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	2, 0, 0 **E**
B1	75, 0, 0 **E**	28, 0, 0 **E**	73, 0, 0 **E**	30, 0, 0 **E**	0, 0, 0 **E**
B2	-, 0, 0 **E**	11, 0, 0 **E**	73, 0, 0 **E**	73, 0, 0 **E**	0, 0, 0 **E**
B3	29, 2, 0 **E**	29, 1, 0 **E**	33, -, 0 **E**	33, -, 0 **E**	8, 2, 1 **E**
B4	1, 0, 0 **E**	1, 0, 0 **E**	-, 0, 0 **E**	-, 0, 0 **E**	0, 1, 1 **E**
B5	75, 0, 0 **E**	1, 0, 0 **E**	73, 0, 0 **E**	73, 0, 0 **E**	8, 0, 0 **E**
C1	-, 7, 7 **I**	249, 7, 7 **I**	207, 5, 5 **I**	207, 5, 5 **I**	9, 5, 5 **I**
C2	249, 0, 0 **E**	29, 0, 0 **E**	207, 0, 0 **E**	123, 0, 0 **E**	0, 0, 0 **E**
C3	-, 0, 0 **E**	33, 0, 0 **E**	207, 0, 0 **E**	207, 0, 0 **E**	19, 0, 0 **E**
D1	168, 0, 0 **E**	114, 0, 0 **E**	124, 0, 0 **E**	81, 0, 0 **E**	12, 0, 0 **E**
D2	168, 0, 0 **E**	37, 0, 0 **E**	124, 0, 0 **E**	124, 0, 0 **E**	16, 0, 0 **E**
D3	168, 0, 0 **E**	65, 0, 0 **E**	124, 0, 0 **E**	105, 0, 0 **E**	2, 0, 0 **E**
D4	168, 0, 0 **E**	15, 0, 0 **E**	124, 0, 0 **E**	124, 0, 0 **E**	10, 0, 0 **E**
E1	31, -, - **W**	28, 18, 17 **I**	109, 109, 109 **S**	47, 47, 109 **W**	18, 12, 10 **I**
E2	-, -, - **S**	38, 18, 17 **I**	-, 109, 109 **I**	-, 109, 37 **I**	24, 10, 10 **I**
E3	125, 0, 0 **E**	16, 0, 0 **E**	109, 0, 0 **E**	109, 0, 0 **E**	11, 0, 0 **E**
F1	151, 0, 0 **E**	67, 0, 0 **E**	106, 0, 0 **E**	54, 0, 0 **E**	25, 0, 0 **E**
F2	151, 0, 0 **E**	44, 0, 0 **E**	106, 0, 0 **E**	54, 0, 0 **E**	9, 0, 0 **E**
F3	151, 0, 0 **E**	48, 0, 0 **E**	106, 0, 0 **E**	-, 0, 0 **E**	9, 0, 0 **E**
G1	1, 0, 0 **E**	1, 0, 0 **E**	-, 0, 0 **E**	-, 0, 0 **E**	0, 0, 0 **E**
G2	-, 1, 1 **I**	13, 1, 1 **I**	154, 0, 0 **E**	114, 0, 0 **E**	11, 0, 0 **E**
G3	-, -, - **S**	36, 34, 34 **I**	154, 154, 154 **S**	154, 154, 154 **S**	2, 2, 2 **S**
H1	116, 2, 0 **E**	116, 1, 0 **E**	96, -, 0 **E**	96, -, 0 **E**	13, 2, 1 **I**
H2	-, 2, 0 **E**	97, 1, 0 **E**	73, 11, 0 **E**	73, 11, 0 **E**	16, 2, 0 **E**
H3	-, -, - **S**	57, 4, 4 **I**	-, 117, 117 **I**	-, 50, 50 **I**	4, 1, 1 **I**
H4	-, -, - **S**	8, 7, 7 **I**	117, 15, 15 **I**	117, 15, 15 **I**	13, 5, 5 **I**
I1	1, 1, 1 **S**	2, 1, 1 **I**	19, 19, 19 **S**	19, 19, 19 **S**	0, 0, 0 **S**
I2	2, 1, 1 **I**	2, 1, 1 **I**	21, 21, 21 **S**	21, 21, 21 **S**	15, 15, 15 **S**
I5	0, 5, 5 **W**	0, 3, 3 **W**	15, 105, 105 **W**	15, 15, 15 **W**	0, 1, 1 **W**
I4	0, 0, 0 **E**	0, 1, 1, **W**	15, -, -, **W**	15, -, -, **W**	0, 3, 3, **W**
I6	1, 1, 1 **S**	2, 1, 1 **W**	23, 23, 23 **S**	23, 23, 23 **S**	0, 0, 0 **S**

Figure 8: Results for test cases with incomplete code

6 Conclusions

We have presented bridge parsing as a technique to recover from syntactic errors in incomplete programs with the aim to produce an AST suitable for static seman-

Test	Antlr	AntlrBT	Beaver	BeaverEP	LPG
A5	9, 0, 0 **E**	9, 0, 0 **E**	63, 0, 0 **E**	63, 0, 0 **E**	5, 0, 0 **E**
C4	248, 4, 2 **I**	193, 193, 2 **I**	207, 207, 34 **I**	153, 153, 34 **I**	4, 4, 1 **I**

Figure 9: Results for test cases with missing starts

Test	Antlr	AntlrBT	Beaver	BeaverEP	LPG
A6	0, 3, 3 **W**	0, 3, 3 **W**	0, 2, 2 **W**	0, 2, 2 **W**	0, 1, 1 **W**
B6	0, 10, 10 **W**	0, 64, 64 **W**	0, 73, 73 **W**	0, 23, 23 **W**	0, 4, 4 **W**
I3	0, -, 0 **E**	0, 14, 0 **E**	0, -, 105 **W**	0, -, 23 **W**	0, 4, 4 **W**
J1	0, 17, 17 **W**	0, 59, 59 **W**	0, 49, 49 **W**	0, 49, 49 **W**	0, 15, 15 **W**
J2	0, 0, 0 **E**	0, 0, 0 **E**	0, 13, 13 **W**	0, 13, 13 **W**	0, 3, 3 **W**

Figure 10: Results for test cases with tricky indentation

tic analysis. This enables tool developers to use existing parser generators when implementing IDEs rather than writing parsers by hand. The approach has proven successful when combined with several parser generators in our IDE generator for JastAdd based compilers in Eclipse.

The approach is highly general and can be used in combination with many different parsing technologies. We have validated this claim by showing how it improves error recovery for three different parser generators in common interactive editing scenarios.

One of the main goals of this work is to lower the burden on language developers who want to provide IDE support for their language. It is pleasant to notice that the language models for bridge parsing are very light-weight, yet yield good recovery on complex languages as exemplified by Java in this paper. We believe that it would be easy to adjust the bridge parser presented in this paper to support other languages as well.

As future work we would like to investigate the possibility of integrating history based information into the bridge model, work on improving the handling of incorrect layout and investigate how to derive bridge grammars from existing baseline grammars [KL03]. An other area we would like to look into is to improve the test suite by observing editing patterns passively from existing code and actively during development.

References

[Avg+08] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. "Modularity First: A Case for Mixing AOP and Attribute Grammars". In: *AOSD*. ACM Press, 2008.

[BF87] Michael G. Burke and Gerald A. Fischer. "A practical method for LR and LL syntactic error diagnosis and recovery". In: *ACM Trans. Program. Lang. Syst.* 9.2 (1987), pp. 164–197.

[Cha91] Philippe Charles. "A practical method for constructing efficient LALR(k) parsers with automatic error recovery". PhD thesis. New York, NY, USA: New York University, 1991.

[DP95] Pierpaolo Degano and Corrado Priami. "Comparison of syntactic error handling in LR parsers". In: *Journal of Software: Practices and Experience* 25.6 (1995), pp. 657–679.

[DK99] Arie van Deursen and Tobias Kuipers. "Building Documentation Generators". In: *IEEE International Conference on Software Maintenance*. 1999, pp. 40–49.

[Ear70] Jay Earley. "An efficient context-free parsing algorithm". In: *Communications of the ACM* 13.2 (1970), pp. 94–102.

[EH07a] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler". In: *OOPSLA*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18.

[For04] Bryan Ford. "Parsing expression grammars: a recognition-based syntactic foundation". In: *SIGPLAN Notices* 39.1 (2004), pp. 111–122.

[Gra+79] Susan L. Graham, Charles B. Haley, and William N. Joy. "Practical LR error recovery". In: *SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler construction*. Denver, Colorado, United States: ACM, 1979, pp. 168–175.

[Hua+08] Shan Shan Huang et al. "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary". In: *ECOOP*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008.

[Jia+95] Tao Jianga, Lusheng Wang, and Kaizhong Zhang. "Alignment of trees - an alternative to tree edit". In: *Theoretical Computer Science*. Vol. 143. Elsevier Science B.V., 1995, pp. 137–148.

[KL03] Steven Klusener and Ralf Lämmel. "Deriving tolerant grammars from a base-line grammar". In: *ICSM*. IEEE Computer Society, 2003, p. 179.

[Moo01] Leon Moonen. "Generating Robust Parsers using Island Grammars". In: *Proceedings. Eighth Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2001, pp. 13–22.

[Moo02] Leon Moonen. “Lightweight Impact Analysis using Island Grammars”. In: *Proceedings of the 10th IEEE International Workshop of Program Comprehension*. IEEE Computer Society, 2002, pp. 219–228.

[NN+09a] Emma Nilsson-Nyman et al. “Declarative Intraprocedural Flow Analysis of Java Source Code”. In: *Electr. Notes Theor. Comput. Sci.* 238.5 (2009), pp. 155–171.

[PD78] Thomas J. Pennello and Frank DeRemer. “A Forward Move Algorithm for LR Error Recovery”. In: *POPL*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM Press, 1978, pp. 241–254.

[Sch+08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. “Sound and extensible renaming for Java”. In: *OOPSLA*. Ed. by Gail E. Harris. ACM, 2008, pp. 277–294.

[Söd09] Emma Söderberg. *Evaluation Link: Practical Scope Recovery using Bridge Parsing*. `http://svn.cs.lth.se/svn/jastadd-research/public/evaluation/sle-08-bp`. 2009.

[Tom85] Masaru Tomita. “An efficient context-free parsing algorithm for natural languages and its applications”. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 1985.

[Wag98] Tim A. Wagner. “Practical Algorithms for Incremental Software Development Environments”. PhD thesis. CA, USA: University of California at Berkeley, 1998.

[WG98] Tim A. Wagner and Susan L. Graham. “Efficient and flexible incremental parsing”. In: *ACM Trans. Program. Lang. Syst.* 20.5 (1998), pp. 980–1013.

NATURAL AND FLEXIBLE ERROR RECOVERY FOR GENERATED MODULAR LANGUAGE ENVIRONMENTS

Abstract

Integrated development environments (IDEs) increase programmer productivity, providing rapid, interactive feedback based on the syntax and semantics of a language. 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, this paper introduces a novel error recovery mechanism. Our approach is language-independent, and relies on automatic derivation of recovery rules from grammars. By taking layout information into consideration it can efficiently suggest natural recovery suggestions.

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, in-

Lennart C.L. Kats, Maartje de Jonge, Emma Söderberg, and Eelco Visser.
Accepted for publication in *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

teractive 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. Modern IDEs use a parser to obtain the syntactic structure of a program with every change that is made to it, ensuring rapid syntactic and semantic feedback as a program is edited. As programs are often in a syntactically invalid state as they are edited, parse error recovery is needed to diagnose and report parse errors, and to construct a valid abstract syntax tree (AST) for syntactically invalid programs. Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

The development and maintenance costs of complete parsers with recovery support are often prohibitive when general-purpose programming languages are used for their construction. Parser generators address this problem by automatically generating a working 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.

In this paper 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.

Generalized parsers A limitation of most parser generators 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 [Kat+10b].

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) [Vis97b], even scanner-level composition problems such as reserved keywords are avoided.

Error recovery To provide rapid syntactic and semantic feedback, modern IDEs interactively parse programs as they are edited. A 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. Users still want editor feedback for the incomplete programs they are editing, even if this feedback is incomplete or only partially correct. For services that apply modifications to the source code such as refactorings, errors and warnings can be provided to warn the user about

[1] Generalized LR [Tom88] parses deterministic grammars in linear time and gracefully copes with non-determinism and ambiguity with a cubic worst-case complexity.

the incomplete state of the program. These days, the expected behavior of IDEs is to provide editor services, even for syntactically invalid programs.

Parse error recovery techniques can diagnose and report parse errors, and can construct a valid AST for programs that contain syntax errors [DP95]. The recovered AST forms a speculative interpretation of the program being edited. Since all language specific services crucially depend on the constructed AST, the quality of this AST is decisive for the quality of the feedback provided by these services. Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

Challenges Three important criteria for the effectiveness and applicability of parser generators for use in IDEs are *1)* the grammar classes they support, *2)* the performance guarantees they provide for those grammar classes, and *3)* the quality of the syntax error recovery support they provide. Parse error recovery for generalized parsers such as SGLR has been a long-standing open issue. In this paper we implement an error recovery technique for generalized parsers, thereby showing that all three criteria can be fulfilled.

The scannerless, generalized nature of SGLR parsers poses challenges for the diagnosis and recovery of 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, increasing the difficulty of diagnosis and recovery. Second, 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. Together, these two challenges make it harder to apply traditional error recovery approaches, as scannerless and generalized parsing increases the search space for recovery solutions and makes it harder to diagnose syntax errors and identify the offending substring.

Approach overview In this paper we address the above challenges by introducing additional "recovery" production rules to grammars that make it possible to parse syntax-incorrect inputs with added or missing substrings. These rules are based on the principles of island grammars (Section 3). We show how these rules can be specified and automatically derived (Section 4), and how with small adaptations to the parsing algorithm, the added recovery rules can be activated only when syntax errors are encountered (Section 5). By using the layout of input files, we improve the quality of the recoveries for scoping structures (Section 6), and ensure

efficient parsing of erroneous files by constraining the search space for recovery rule applications (Section 7).

Contributions This paper integrates and updates our work on error recovery for scannerless, generalized parsing [Kat+09b; Jon+09] and draws on our work on bridge parsing [NN+09b]. We implemented our approach based on the modular syntax definition formalism SDF [Hee+89; Vis97c] and JSGLR[2], a Java-based implementation of the SGLR parsing algorithm. The present paper introduces general techniques for the implementation of an IDE based on a scannerless, generalized parser, and evaluates the recovery approach using automatic syntax error seeding to generate representative test sets for multiple languages.

2 Composite Languages and Generalized Parsing

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 [Duc+06] or aspects [Kic+97] 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 [Bra+10]. Such an embedding both increases the expressiveness of the host language and facilitates static checking of queries. Figure 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 [Vis02]. Transformation languages such as Stratego [Bra+08] and ASF+SDF [Bra+02a] allow fragments of a language that undergoes transformation to be embedded in the specification of rewrite rules. Figure 2

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

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

Figure 1: An extension of Java with SQL queries.

```
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: Program transformation using embedded object language syntax.

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.

2.1 Parsing Composite Languages

The key to effective realization of composite languages is a modular, reusable language description, 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. 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 [SC89; SC95], these issues can be elegantly addressed [Bra+06].

The *Scannerless Generalized-LR* (SGLR) parsing algorithm [Vis97b] realizes scannerless parsing by incorporating the generalized-LR parsing algorithm [Tom88].

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). 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.[3]

Language composition often results in grammars that contain ambiguities. Generalized parsing allows declarative disambiguation of ambiguous interpretations, implemented as a filter on the parse tree, or rather the *parse forest*. 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.

Non-determinism in grammars can negatively affect parser performance. With traditional backtracking parsers, this would lead to exponential execution time. Packrat parsers use a form of backtracking with memoization to parse in linear time [For02]; but, as with other backtracking parsers, they greedily match the first possible alternative instead of exploring all branches in an ambiguous grammar [Sch06]. In contrast, GLR parsers explore all branches in parallel and run in cubic time in the worst case. Furthermore, they have the attractive property that they parse the subclass of deterministic LR 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 [Vis97a].

2.2 Defining Composite Languages

The syntax definition formalism SDF [Hee+89; Vis97c] 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 [Bra+02b]. Implicit disambiguation mechanisms such as 'longest match' are avoided. Other approaches, including PEGs [For02], language inheritance in MontiCore [Kra+08], and the composite grammars of ANTLR [PF11], implicitly disambiguate grammars by forcing an ordering on the

[3] Note that [SVW09] 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 3: Syntax of Java with embedded SQL queries, adapted from [Bra+10]. The 'cons' annotation defines the name of the constructed ATerm.

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++ [WY07], PHP [Bra+10], and Java [BV04; Kat+08]. The example grammar shown in Figure 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.

3 Island Grammars

Island grammars [DK99; Moo01; Moo02] combine grammar production rules for the precise analysis of parts of a program and selected language constructs with general rules for skipping over the remainder of an input. 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 [Moo01]. In this paper we use island grammars as inspiration for error recovery using additional production rules.

Using an island grammar, a parser can skip over any uninteresting bits of a file ("water"), including syntactic errors or constructs found only in specific language dialects. A small set of declarative context-free production rules specifies only the interesting bits (the "islands") that are parsed "properly". Island grammars were originally developed using SDF [DK99; Moo01]. The integration of lexical and context-free productions of SDF allows island grammars to be written in a single,

```
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 4: An island grammar for extracting calls from a legacy application; adapted from [Moo01].

declarative specification that includes both lexical syntax for the definition of water and context-free productions for the islands. A parser using an island grammar behaves similar to one that implements a noise-skipping algorithm [LT93]. It can skip over any form of noise in the input file. However, using an island grammar, this logic is entirely encapsulated in the grammar definition itself.

Figure 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 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 (i.e. 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 [Bra+02b]. First, 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. Second, 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.

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

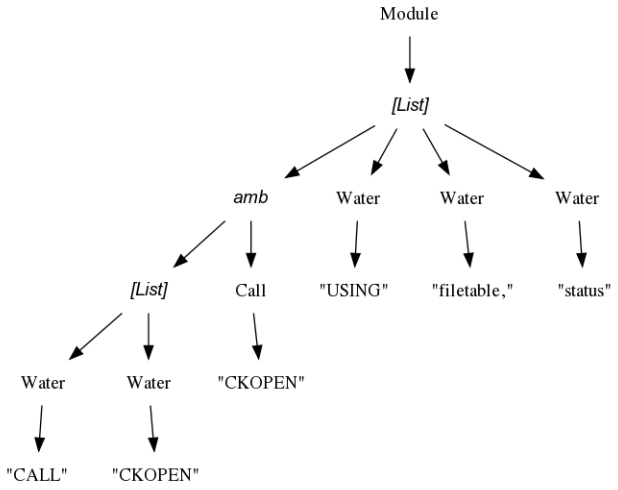


Figure 5: The unfiltered abstract syntax tree for a COBOL statement, constructed using the ExtractCalls grammar.

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

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 paper, we focus on abstract syntax trees (derived from the parse trees) where only the `{cons(`*name*`)}` constructor labels appear in the tree. Figure 5 shows the complete, ambiguous AST for our example input program. Note in particular the *amb* node, 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.

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 defi-

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

Figure 6: Part of the standard Java grammar in SDF; adapted from [Bra+06].

nitions, 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 4.1 to 4.3. Then, in Section 4.4, we explain how different forms of recovery can be combined. Finally, in Section 4.5 we discuss automatic derivation of recovery rules from the grammar, while Section 4.6 explains how the set of generated recovery rules can be customized by the language developer.

Without loss of generality, we focus many of our examples on the familiar Java language. Figure 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}` annotation states that the interpretation of an `IfElse` term with a nested `If` subterm, must be avoided in favour 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 [BV04; Bra+06].

4.1 Chunk-Based Water 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 chunks.

Figure 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. 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.

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 6. In Figure 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 recovery 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 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 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

```
module Java-15-Permissive-Avoid
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 7: Chunk-based recovery rules for Java using `avoid`.

```
module Java-15-Permissive-ChunkBased
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 8: Chunk-based recovery rules using `recover`.

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 5.

Throughout this section we use only the standard, unaltered SDF specification language, adding only the `{recover}` annotation.

Limitations of Chunk-Based Rules We can extend the grammar of Figure 8 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. However, 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 number of occurrences of the `{recover}` annotation in the ambiguous branches.

The technique of selectively adding water recovery rules to a grammar allows any existing grammar to be adapted. It avoids having to rewrite grammars from the ground up to be more "permissive" in their inputs. Grammars adapted in this fashion produce parse trees even for inputs with syntax errors that cannot be parsed by the original grammar. The `WATER` constructors in the ASTs indicate the location of errors, which can then be straightforwardly reported back to the user.

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.

4.2 General Water Recovery Rules

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 3, any form of layout, which may include comments, is implicitly interleaved in the patterns of concrete syntax productions. The parser skips over these parts in a similar fashion to the noise skipping of island grammars. However, 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 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 gram-

mar 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 [Vis97c], the underlying core language of SDF, as follows[4]:

```
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 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 a new variation of the water recovery rules we have shown so far. Consider Figure 9, which combines elements of the comment definition of Figure 6 and the chunk-based recovery rules from Figure 8. It introduces optional water into the grammar, 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 water into small chunks, each associated with its own significant `{recover}` annotation, we distinguish between `WATERWORD` and `WATERSEP` tokens. The production for the `WATERWORD` token allows to skip over identifier strings, while the production for the `WATERSEP` 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 water recovery rule. Based on the number of `{recover}` annotations, the following would be the preferred interpretation:

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

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

```
module Java-15-Permissive-Water
imports Java-15
exports
lexical syntax
  [A-Za-z0-9\_]+            -> WATERWORD {recover}
  ~[A-Za-z0-9\_\ \t\12\r\n] -> WATERSEP  {recover}
  WATERWORD                 -> WATER
  WATERSEP                  -> WATER
  WATER                     -> LAYOUT    {cons("WATER")}
lexical restrictions
  WATERWORD -/- [A-Za-z0-9\_]
```

Figure 9: Water recovery rules.

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" nodes into the AST. For reporting errors, the original parse tree, which *does* contain "water" nodes, can be inspected instead.

The adapted grammar of Figure 9 no longer depends on hand-picking particular symbols at different granularities to introduce water recovery rules. Therefore, it is effectively language-independent, and can be automatically constructed using only the `LAYOUT` definition of the grammar.

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:

```
"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.

```
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 10: Insertion recovery rules for literal symbols.

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 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 definition of a literal non-terminal, effectively making it optional by indicating that they may match the empty string. 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 1 and Figure 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". 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 11 specifies recover rules for terminating the productions of the `StringLiteral` and `Comment` symbols, first seen in Figure 6. Both rules have a `{recover}` annotation on their starting literal. Alternatively, the annotation could be placed on the complete production:

[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" -> StringLiteral {cons("INSERTEND")}
 "\""                          -> INSERTSTARTQ  {recover}
 INSERTSTARTC CommentPart* EOF -> Comment      {cons("INSERTEND")}
 "/*"                          -> INSERTSTARTC  {recover}
```

Figure 11: Insertion recovery rules for lexical symbols.

```
lexical syntax
   "\"" StringPart* "\n" -> StringLiteral
      {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 nodes that represent a missing code construct, for example a `NULL()` node. Since placeholder nodes 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 Section 8.

4.4 Combining Different Recovery Rules

The water recovery rules of Section 4.2 and the insertion rules of Section 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-Water
  Java-15-Permissive-LiteralInsertions
  Java-15-Permissive-LexicalInsertions
```

Together, the two strategies maintain a fine balance between discarding and inserting substrings. Since the water recovery rules incur additional cost for each water substring, insertion of literals will generally be preferred over discarding multiple substrings. This ensures that most of the original (or intended) user input is preserved.

4.5 Automatic Derivation of Permissive Grammars

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 generic, automated 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 water 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 [DP95]. 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).

```
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 12: A selection of context-free productions that appear in the Java grammar.

- 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 four 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 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 [Bra+06]) 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 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 10 which are a subset of the complete set of closing literal insertion rules for Java. From the productions of Figure 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 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 10. 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 water recovery rules as outlined in Figure 9. The complete set of derived rules is available from [Kat+11].

4.6 Customization of Permissive Grammars

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* [DP95]. 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")}
```

[6]Note that for narrative reasons, we did not include an insertion rule for this keyword in Figure 10.

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 [Bra+02b]. 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.

5 Parsing Permissive Grammars with Backtracking

When all recovery rules are taken into account, permissive grammars provide many different interpretations of the same code fragment. As an example, Figure 13 shows many possible interpretations of the string `i=f(x)+1;`. The alternative interpretations are obtained by applying recovery 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 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 recovery productions negatively affect time and space requirements. With a generalized parser, all interpretations are explored in

```
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 13: Interpretations of `i=f(x)+1;` with insertion recovery rules (underlined) and water recovery rules.

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.

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 (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 recovery productions during normal parsing, and employ backtracking to apply the recovery rules only once an error is detected. Backtracking

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

Figure 14: The superfluous closing bracket is detected at the `while` keyword.

parsers exhibit exponential behavior in the worst case [Joh+04]. For pathological cases with repetitive backtracking, the parser is aborted, and a secondary, non-correcting, recovery technique is applied.

5.2 Selecting Choice Points for Backtracking

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 [DP95]. Consider for example the erroneous code fragment in Figure 14. 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 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 [Tom88]. 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.

5.3 Applying Recovery Rules

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, 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 (recover rule applications) that are made.

Figure 15 illustrates how the search heuristic is applied to recover the Java fragment of Figure 14. 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 and for a restricted number of recovery rule applications. For each following iteration the size of the area and the number of recovery rule applications are increased.

Figure 15a 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 15b) 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 recovery production are considered; alternative interpretations formed by a second recovery 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 15c, 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 changes are considered, resuming the interpretations that were previously cut off.

In Figure 15d, the search space is again expanded with the preceding line. This time, a valid recovery is found: the application of a water recovery rule that discards the closing bracket leads to a valid interpretation of the erroneous code

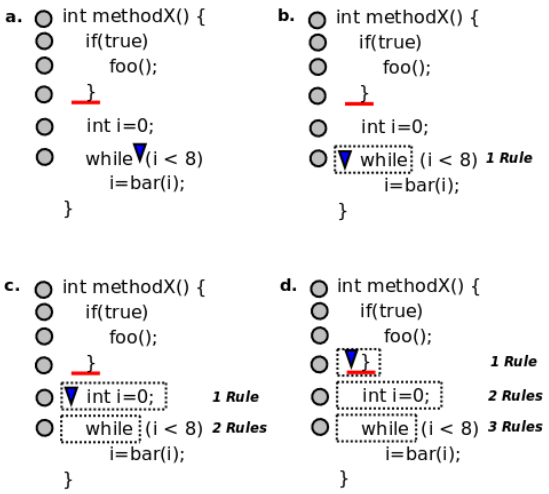


Figure 15: Applying error recovery rules with backtracking. The initial point of failure and the start of the recovery search space is indicated by a triangle. The entire search space is indicated using dashed lines, where the numbers to the side indicate the number of recovery rules that can be applied at that line.

fragment. Once the original line where the error was detected can be successfully parsed, normal parsing continues.

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. And 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 16 shows the recovery algorithm in pseudo code. The `Recover` function controls the iterative search process described in Section 5.3. The function starts with some initial configuration (line 2–3), initializing the `candidates` variable, 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):

```
RECOVER(choicePoints, failureOffset)
 1  ▷ Constructs a recovery stack structure (GSS) for the parse input
 2     after the failure location
 3  candidates ← {}
 4  choicePoint ← Last inserted choicepoint
 5  do
 6          (stacks, candidates) ← PARSECANDIDATES(candidates,
 7                  choicePoint, failureOffset)
 8          choicePoint ← Preceding choicepoint (or choicePoint if none)
 9  until | stacks | > 0
10  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 that were created
13          in previous loop
14     choicePoint - The start configuration for the parser
15     failureOffset - Location were the parser originally failed
16  ▷ Output:
17     stacks - recovered stacks at the accept location
18     newCandidates - new unexplored recover branches for
19          the parsed fragment
20
21  stacks ← choicePoint.stacks
22  offset ← choicePoint.offset
23  newCandidates ← {}
24  do
25          stacks ← stacks ∪ { c | c ∈ candidates ∧ c.offset = offset}
26          (stacks, recoverStacks) ← PARSECHARACTER(stacks,
27                  offset, true)
28          newCandidates ← newCandidatess ∪ recoverStacks
29          offset = offset +1
30  until offset = (failureOffset + ACCEPT_INTERVAL)
31  return (stacks, newCandidates)

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

Figure 16: A backtracking algorithm to apply recovery rules.

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 cut off recover interpretations of a prefix of the program. The `ParseCandidates` function reparses the fragment that starts at the choice point location and ends at the accept location (line 19–26). 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 24–25, line 31).

The main idea, implemented in line 23-25 and the `ParseCharacter` function (line 28–32), is to postpone the exploration of branches that require multiple recover productions, thereby implementing the expanding search space heuristic described in Section 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 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 water 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.

6 Layout-Sensitive Recovery of Scoping Structures

In this section, we describe a recovery technique specific for errors in scoping structures. Scoping structures are usually recursive structures specified in a nested fashion [Cha91]. Omitting brackets of scopes, or other character sequences marking scopes, is a common error made by programmers. These errors can be addressed by common parse error recovery techniques that insert missing brackets.

However, as scopes can be nested, there are often many possible positions where a missing bracket can be inserted. The challenge is to select the most appropriate position.

As an example, consider the Java fragment in Figure 17. This fragment could be recovered by inserting a closing bracket at the end of the line with the second opening bracket, or at any line after this line. However, the use of indentation suggests the best choice may be just before the `int x;` declaration.

```
class C {
  void m() {
    int y;
  int x;
}
```

Figure 17: Missing `}`.

One approach to handle this problem is to take secondary notation like indentation into account during error recovery. Bridge parsing, introduced by [NN+09b][7], uses this particular approach. This scope recovery approach can be combined with the permissive grammar approach presented in the previous section.

6.1 Bridge Parsing

Bridge parsing provides a technique specifically targeted at improved recovery of scope errors using secondary notation such as indentation. The technique as such is independent of any specific parsing formalism. It may be used as a standalone processor of erroneous files where recovery otherwise fails: given an erroneous file, or section of a file, the bridge parser analyses the content and provides suggestions on where to insert missing brackets. Based on a set of rules that describe the typical relation between scopes and layout for Java, a bridge parser can correctly recover cases such as the example above.

Internally, a bridge parser contains three parts: a *tokenizer*, a *model builder*, and a *repairer*. The tokenizer provides a list of interesting tokens from an input text. Tokens starting and ending scopes are referred to as *islands*; tokens interesting for construction of scopes, or recovery of scopes, are referred to as *reefs*; and remaining tokens are considered to be *water*. The terms island and water are used in the same fashion as in island grammars [DK99; Moo01; Moo02]. Reefs, added for bridge parsing, are tokenwise like islands, but have a different role in the model constructed from the token list. Figure 18 shows an example of a token list for the program fragment in Figure 17. Each part of the fragment is mapped to either an island, reef, or water. For the benefit of the model builder algorithm, the token list is padded with some additional tokens at the start and end.

After tokenization, the model builder constructs scopes based on information in the token list. For instance, each reef in the token list in Figure 18 has a number indicating indentation level.[8] This indentation information is key to construction of scopes, represented as bridges, connecting two islands, in the model. The model

[7] Emma Nilsson-Nyman, the first author of the cited bridge parser paper, has changed her name to Emma Söderberg and is one of the authors of this paper.

[8] Note that in our implementation, we determine the indentation level by counting the number of spaces, treating tabs as a fixed number of spaces. The relation between tabs and spaces could also be determined from the editor settings.

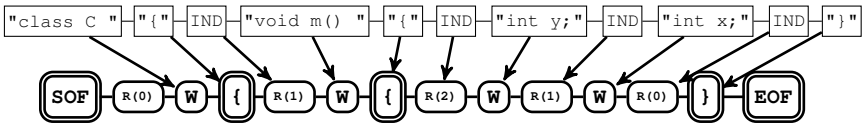


Figure 18: A tokenization of the example program in Figure 17 where text is mapped to islands (double edges), water (`W`), and reefs (`R(`n`)`). The number n in a reef `R(`n`)` represents the indentation level of the reef.

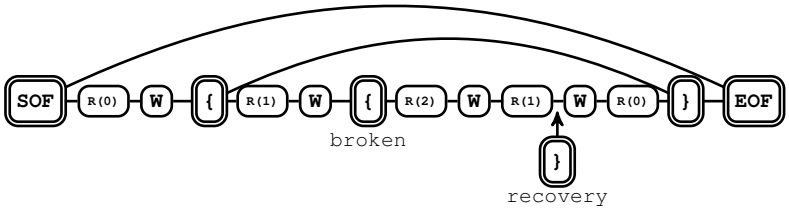


Figure 19: A bridge parser model with bridges (arches) between matching islands (double edged nodes). Islands missing a bridge correspond to broken scopes (`broken`). The bridge repairer will try to recover such scopes by insertion of new matching islands (`recovery`).

builder decides which two islands to connect using an algorithm that considers patterns of tokens surrounding islands, and rules for when patterns match. For instance, in Figure 19 bridges have been added to the token list in Figure 18. The added bridges connect the start and end of the fragment, and two of the islands, while one island remains unmatched. In this example, islands are matched based on the indentation of the first reef to their left. For the two matched islands their corresponding reef shares the same indentation, while their is no such match for the island without a bridge. Islands like this one, without a bridge, are considered broken and representatives of broken scopes.

After construction of bridges, the repairer takes over. The purpose of the repairer is to recover broken scopes based on as set of patterns and rules. The purpose of the patterns is to identify appropriate so called *construction sites* for a recovery. Once such a construction site has been found, the rules are used to decide how to insert a matching so called *artificial island* and create a bridge. For instance, in Figure 19 a construction site is found based on a pattern identifying indentation shifts, and an artificial islands is inserted to match the broken island and recover the scope error. Insertion of islands, like in this example, correspond to the recovery suggestions a bridge parser provides after it is done.

A more complete description of the algorithm, incrementally constructing multiple bridges, is given by [NN+09b].

6.2 Combining Permissive Grammars and Bridge Parsing

As a recovery technique, bridge parsing forms a supplementary approach that can be used together with permissive grammars introduced in Section 4. Permissive grammars and bridge parsing share their inspiration from island grammars [DK99; Moo01; Moo02], with the difference that a bridge parser employs a scanner.

The use of a scanner in bridge parsing may appear contrary to the scannerless nature of SGLR. One could imagine that a scannerless version of a bridge parser would be better suited for an integration to SGLR. That is, based on an accurate (scannerless) lexical analysis, additional reefs could be identified using the keywords of a language. However, previous results showed that doing so only marginally improves recovery quality [Jon+09]. Also, practical experience has shown that a bridge parser is most time and memory-efficient when independent from a specific grammar, focusing just on the scoping structures of the language. For this reason and for simplicity, the bridge parsers used in this paper only include scope tokens and layout reefs.

This combined approach has limitations with regard to embedded languages, where a token may have different syntactic meanings: `{` might be a scope delimiter in one language and an operator in another. Still, the layout-sensitive bridge model gives an approximation of the scoping structure in those cases, which can improve recovery results when used in combination with recovery rules. As a layout-sensitive technique, bridge parsing served as an inspiration to the layout-sensitive regions discussed in the next section.

7 Layout-Sensitive Region Selection

In this section we describe a layout-sensitive region recovery algorithm that improves recovery efficiency and helps cope with pathological cases not easily addressed with only permissive grammars, backtracking, and bridge parsing. 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 distance between the error location and the detection location is exceptionally large, or when the recovery requires many combined recovery rule applications. The latter can occur when multiple errors are tightly clustered, or when no suitable recovery rule is at hand for a particular error. In general, a valid parse can be found after expanding the search space, but at a 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. Section 4.3 discusses an example in which an entire SQL fragment would be parsed as (severely incorrect) Java code.

To address these concerns, this section introduces an approach to identify the *region* in which the actual error is situated. By constraining the recovery sugges-

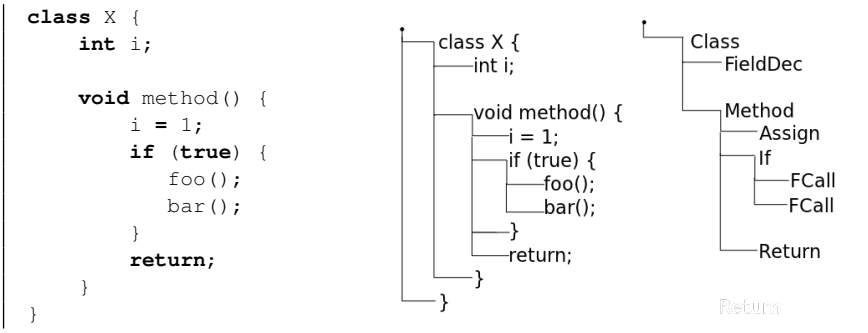


Figure 20: Indentation closely resembles the hierarchical structure of a program.

tions 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.

7.1 Nested Structures as Regions

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 list elements are suitable regions for regional error recovery.

The bridge parsing technique discussed in Section 6 exploits layout characteristics to detect the intended nesting structure of a program. In this section, we present a region selection technique that, inspired by bridge parsing, uses indentation to detect erroneous structures. Indentation typically follows the logical nesting structure of a program, as illustrated in Figure 20. 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.

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

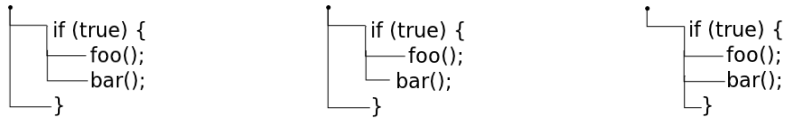


Figure 21: TODO

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.

7.2 Regions based on Indentation

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 21 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 nodes are more indented than their parent, however, the siblings in these fragments do not all have the same indent value.

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 22 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

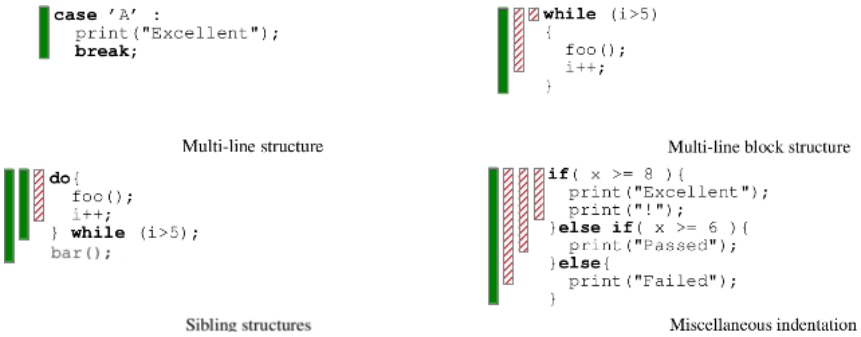


Figure 22: Multi-line Java constructs with various indentation patterns. The solid bars indicate layout regions that correspond to code regions, the hatched bars indicate layout regions that are in fact unwished artifacts.

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 22 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, can not be distinguished from real regions. Notice that most of these ambiguities could 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. However, we implemented the algorithm in a language independent way.

7.3 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

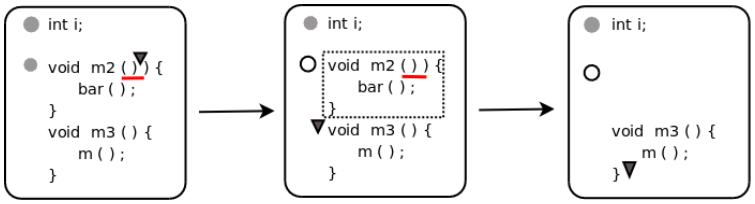
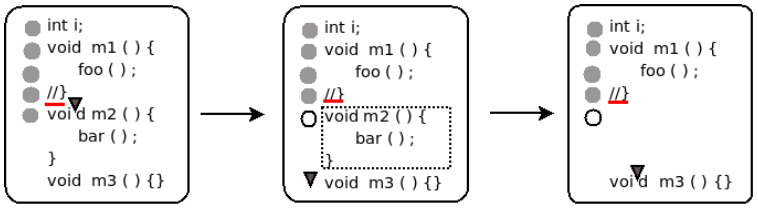
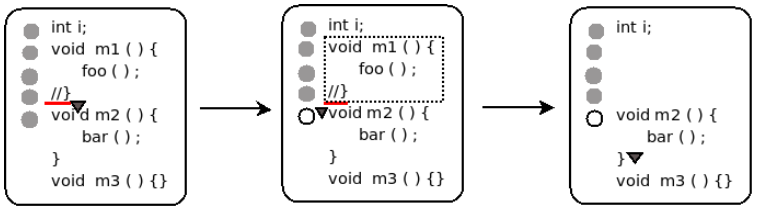


Figure 23: A candidate region is validated and successfully discarded.



(a) A candidate region is rejected.



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

Figure 24: Iterative search for a valid region.

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 7.4 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. We show example scenarios in Figure 23 and Figure 24. Figure 23 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 VII.24(a) 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 Fig-

ure VII.24(b) an alternative candidate region is selected. This region is validated as erroneous.

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.

7.4 Algorithm

Figure 25 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 nested `for` loops (line 6,7) implement the iterative search process described in Section 7.3. The iteration starts with the smallest region (line 6, `sibCount=0`) that can be constructed for the failure line (line 7, `bwSibIndex=0`). In the first iteration (line 7), regions are selected at increasing distance from the failure location. The second iteration (line 6) increases the size of the selected regions. The iteration stops in case a selected region is validated as erroneous (lines 11-13). If no erroneous region is found, the search process continues by recursively visiting the parent of the failure line (line 16). 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.

Figure 26 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 (`bwSibIndex`) and the region size (`sibCount`). The right most bar represents the parent region visited in the recursion step.

```
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  ▷ MAX_SIBLINES_COUNT: Max number of sibling lines in
 5        the candidate regions
 6  ▷ MAX_BW_INDEX: Max number of sibling lines that are
 7        backtracked
 8  for sibCount in 0 to MAX_SIBLINES_COUNT
 9        for bwSibIndex in 0 to MAX_BW_INDEX
10              startLine ← GETPRECEDINGSIBLINE(failureLine,
11                    bwSibIndex)
12              sibLine ← GETFOLLOWINGSIBLINE(startLine,
13                    sibCount)
14              endLine ← GETLASTDESCENDANTLINE(sibLine)
15              if startLine, endLine exist and TRYSKIPREGION(
16                    startLine, endLine)
17                then return (startLine, endLine) ▷ erroneous region
18              end
19        end
20  end
21  return SELECTERRONEOUSREGION(GETPARENTLINE(failureLine))

TRYSKIPREGION(startline, endline)
17  ▷ Output: true iff discarding the region startline ... endline
18  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 25: Algorithm to select a discardable region that contains the syntax error.

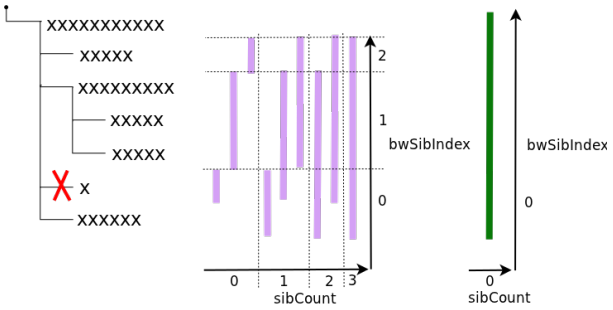


Figure 26: Candidate regions subsequently tested for the indented code fragment at the left. Candidate regions are selected by backtracking ($bwSibIndex$) and by extending the number of sibling lines that are contained in the region ($sibCount$). Finally, the parent line is visited in the recursion step.

7.5 Practical Considerations

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

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

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

Multi-line Comments and Strings The selection procedure can generally select erroneous regions that are not located at 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. 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.

```
/* Comments ...
int foo(){
   ...
}
...
EOF
```

8 Applying Error Recovery in an Interactive Environment

A key goal of error recovery is its application in the construction of IDEs. Modern IDEs rely heavily 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. Experience with modern IDEs shows that for most services it is not a problem to operate on inaccurate or incomplete information as a consequence of syntax errors; for some services such as refactorings, errors and warnings can be presented to the user. In this section, we describe the role of error recovery in different editor services and show language-parametric techniques for using error recovery with these services.

8.1 Efficient Construction of Languages and Editor Services

While IDEs for languages have been constructed and used for several decades, only recently did they become significantly more sophisticated and indispensable for productivity of software developers. In early 2001, IntelliJ IDEA [Sau+06]

revolutionized the IDE landscape [Fowb], 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.

As IDEs become both more commonplace and more sophisticated, it becomes increasingly important to lower the threshold of creating new languages and developing IDEs for these languages. In order to make this possible, *language workbenches* have been developed that combine the construction of languages and editor services. Language workbenches improve the productivity of language engineers by providing specialized languages, frameworks, and tools [Fowa]. Examples of language workbenches for textual languages include EMFText [Hei+09], MontiCore [Kra+08; Grö+08], Spoofax [KV10], TCS [Jou+06], and Xtext [EV06].

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 generated parser runs in the background with each key press or after a small delay passes, and 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 every time the source code changes, they have access to more accurate, more up-to-date information, but they also crucially depend on the parser's performance and its support for error recovery.

8.2 Guarantees on Recovery Correctness

Using permissive grammars, bridge parsing and regional recovery, the parser can construct ASTs for syntactically incorrect inputs. These trees can be constructed using generated or handwritten recovery rules, and may have gaps for regions that could not be parsed. 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.

While error recovery is ultimately a speculative interpretation of an incorrect input, our approach does guarantee well-formedness of ASTs. That is, it will only produce ASTs with tree nodes that conform to the abstract structure imposed by production rules of the original (non-permissive) grammar. This property is maintained for all our recovery techniques. With respect to permissive grammars (Section 4 and 5), water recovery rules (Section 4.2) and literal insertion recovery rules (Section 4.3 and 4.5) do not contribute AST nodes, while insertion recovery rules for lexical productions (Section 4.3, 4.5) only contribute lexical tree nodes that correspond to the recovered lexicals. Bridge parsing (Section 6) and region recovery (Section 7) do not compromise the well-formedness property of the parse

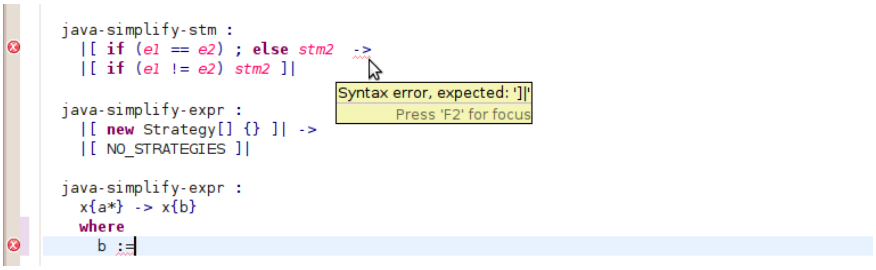


Figure 27: An editor for Stratego with embedded quotations of Java code.

result since both techniques only modify the input string respectively by adding a literal and by skipping over a text fragment.

The property of well-formedness of trees significantly simplifies the implementation and specification of editor services, as they do not require any special logic to handle badly parsed constructs with missing nodes or special constructors. This approach also ensures separation of concerns: error recovery is purely performed by the parser, while editor services do not have to treat syntactically incorrect programs differently. This separation of concerns means that all editor services could be implemented without any logic specific for error recovery. Still, there are a number of editor services that inherently require some interaction with the recovery strategy, which we discuss next.

8.3 Syntactic Error Reporting

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. We use generic error messages that depend on the class of recovery (Section 4.5). For water recovery rules and for region recoveries, we use "[string] not expected," for insertion rules 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.

Figure 27 shows a screenshot of an editor for Stratego with embedded Java. The shown code fragment contains two syntax errors. Due to error recovery, the editor can still provide syntax highlighting and other editor services, while it marks all the syntax errors inline with red squiggles.

8.4 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. Using the grammar, a scanner can recognize tokens in a stream, which can be used to assign colors instead.

More recent implementations of syntax highlighting do a full context-free syntax analysis, or even use the semantics of a language for syntax highlighting. For example, they may assign Java field accesses a different color than local variable accesses.

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. In situations like this one, that may not be possible, resulting in no highlighting for the `new` keyword.

Fallback syntax highlighting Syntax highlighting is equally or possibly more important for syntactically incorrect programs than for syntactically correct programs, as it indicates how the editor interprets the program as a programmer is editing it. A *fallback syntax highlighting* mechanism is needed to address this issue.

A natural way of implementing fallback syntax highlighting is by using a lexical analysis for those cases where the full context-free parser is unable to distinguish the different words to be highlighted. This analysis can be performed by a rudimentary tokenizer that can recognize separate words such that they can be distinguished for colorization. Simple coloring rules can then be applied to any tokens that do not belong to recovered tree nodes, e.g. highlighting all the reserved keywords and string literals. Consequently, programmers get highly responsive syntax highlighting as they are typing, even if the program is not (yet) syntactically correct. 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.

8.5 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: 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 context completion is applied to an incomplete expression, the syntactic context of that expression must be recovered. This is especially challenging for language constructs with many elements, such as the "for" 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 4 this is not always the case. Water 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 only insert missing terminal symbols.

We introduce specific recovery rules for content completion that specify what abstract representation to use for incomplete syntactic constructs. These rules use the `{ast(`p`)}` annotation of SDF to specify a pattern p as the abstract syntax to construct. Figure 28 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 statement if the `Stm` non-terminal is omitted, using a placeholder pattern `NULL()` in place of the abstract representation of the omission. The third rule handles the case where both non-terminals are omitted.

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. Since these rules preserve the wellformedness property, they are also applicable for normal error recovery. For example, in the second rule of Figure 28, the pattern `Block([])` can be used instead of the `NULL()` placeholder (Figure 29). 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 the empty list `[]` as the basic default for list productions.

```
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 28: Java `ForEach` production and its derived completion rules.

```
context-free syntax

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

Figure 29: Java `ForEach` completion rule with placeholder pattern that matches the signature of the original production.

Runtime support Completion recovery rules are designed to support the special scenario of recovering the expression where content completion is requested. The cursor location provides a hint about the location of the (possible) error. Instead of backtracking after an error is found, we apply completion recovery rules if they apply to 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 water and insertion recovery rules for the content completion scenario.

9 Implementation

We implemented our approach in Spoofax [KV10], which is a language development environment that combines the construction of languages and editor services. Using SDF and JSGLR[9], Spoofax has the distinguishing feature that it supports language compositions and embeddings. In this section we give an overview of the general system and we discuss the adaptations we made for error recovery.

Figure 30 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 is used by the *JSGLR* parser, which constructs a parse tree for a (possible erroneous) input

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

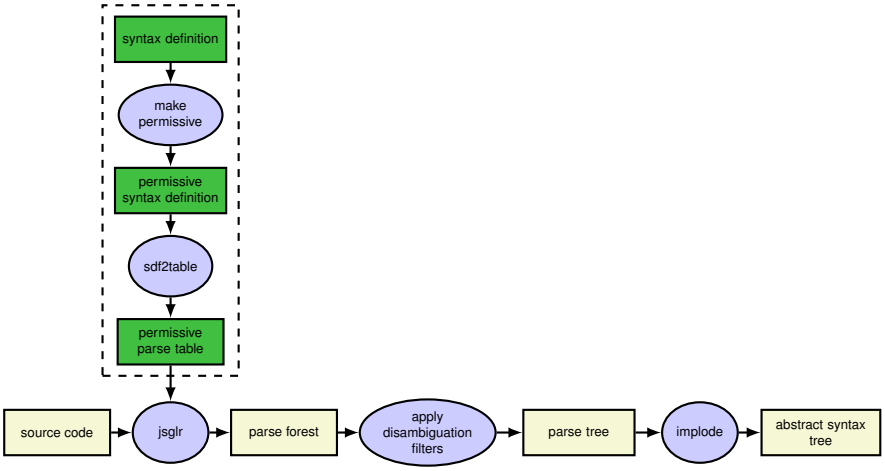


Figure 30: Overview tool chain. Make-permissive generates a permissive version of the original grammar, for which a parse table is constructed by sdf2tbl. The (permissive) parse table is used by JSGLR to construct a parse tree for a (possible erroneous) input file, which is then imploded into an AST.

file. The parse tree is first disambiguated by applying post-parse filters, and then imploded into an AST.

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 that applies the heuristic rules described in Section 4.5 and Section 8.5 that guide the generation of recovery rules. The tool is implemented in Aster [Kat+09a], 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. Furthermore, we implemented two additional recovery techniques, namely, bridge parsing and region selection. 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 productions labeled with the recover annotation during normal parsing
- ignore productions labeled with the completion annotation, unless the production applies to a character sequence that overlaps with the cursor loca-

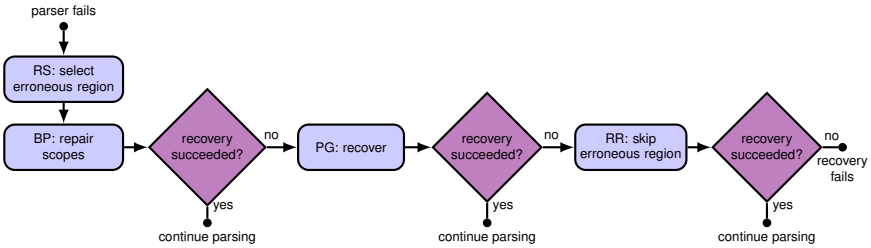


Figure 31: Overview integrated recovery approach implemented in JSGLR.

tion, and the completion service is triggered by the user.

- runtime disambiguation filter that selects the branch with the lowest number of recover/completion productions, preferring insertions over water productions.
- implementations for the different recovery techniques described in Sections 5, 6, and 7.
- some code to integrate the different recovery techniques, as described below.

Integrating recovery techniques We combine the different techniques described in this paper in a multi-stage recovery approach (Figure 31). Region selection (RS) is applied first to detect the erroneous region. In case region selection fails to select the erroneous region, the whole file is selected instead. In the second stage, the erroneous region is inspected by one of the correcting techniques, bridge parsing (BP) or permissive parsing (PG). Since bridge parsing provides the most natural recoveries from a user perspective, it is applied first. The bridge parser returns a set of recovery suggestions based on bracket insertions, which are applied during a re-parse of the erroneous region. In case the bridge parser suggestions do not lead to a successful recovery, the permissive grammars approach described in Sections 4 and 5 is used, where backtracking is restricted to the erroneous region. In case both correcting techniques fail, the erroneous region is skipped (region recovery, RR) as a fallback recovery strategy.

10 Evaluation

We evaluate our approach with respect to the following properties:

- **Quality of recovery**: How well does the environment recover from input errors?

- **Performance and scalability**: What is the performance of the recovery technique? Is there a large difference in parsing time between erroneous and correct inputs? Does the approach scale up to large files?
- **Editor feedback**: How well do editor services perform based on the recovered ASTs?

In the remainder of this section we describe our experimental setup, experimentally select an effective combination of techniques and recovery rules, and show the quality and performance results of the selection.

10.1 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.

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. There are many factors involved for selecting the test inputs, such as the type of grammar, the type of error, distribution of errors over the file, and the layout characteristics of the test files. With these factors in mind, we have taken the approach of generating 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 [JV12]. We distinguish the following categories for seeded errors:

- *Incomplete constructs*, language constructs that miss one or more symbols at the suffix, e.g. an incomplete `for` loop `for (x = 1; x`.
- *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.
- *Combined errors*, two or more errors from the above mentioned categories, randomly distributed over the source file.

Test Oracle

To measure the quality and performance of a recovery, we compare the results obtained for the recovered file against the results for the base file or expected file. In some cases, the base file does not realistically reflect the expected result, as information is lost in the generated erroneous file. For these cases we construct an expected result, a priori. For example, for a "for" loop with an *Incomplete construct* error – such as `for (x = 1; x` – the original body of the construct is lost. For this "for" loop, 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 [PD78]. 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 Pennello and DeRemer criteria represent the state of the art evaluation method for syntactic error recovery applied in, amongst others, [PD78; PK80; DP95; Cor+02].

Since human criteria form an evaluation method that is arguably subjective, as a second method, we also do an automated comparison of the abstract syntax. For this, we print the AST of the recovered file to text using the ATerm format [Bra+00], formatted so that nested structures appear on separate lines. We then count the number of lines that differ in the recovered AST compared to the AST of the expected file (the "diff"). The advantage of this approach is that it is objective, and assigns a larger penalty to recoveries for which a larger area of the text does not correspond to the expected file, where structures are nested improperly, or when multiple deviations appear on what would be a single line of pretty-printed code. Furthermore, using this approach the comparison can be automated, which makes it feasible to apply 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" (1–10 lines of abstract syntax) roughly corresponds to the *good* qualification, and a "large diff" ($>$ 10 lines) 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 10.2.

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 correct base file or expected file from the parse time of the incorrect variation of this 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 paper, an average, collected after three runs, is used. 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.[10]
- *The Dryad compiler*: An open compiler for the Java platform [Kat+08] written using Stratego-Java.
- *The StringBorg project*: A tool and grammar suite that defines different embedded languages [Bra+10], providing Java-SQL code.
- *JSGLR*: A Java implementation of the SGLR parser algorithm.[11]

We selected five representative base files from each project, and generated test files using the error seeding technique. We applied a sanity check to ensure that generated test cases are indeed syntactically incorrect and that there are no duplicates. In total, we generated 334 Stratego-Java test cases, 190 WebDSL test cases, 195 Java-SQL test cases, and 329 Java test cases. In addition, we generated a second test set consisting of 314 Stratego-Java test cases in the *Incomplete construct* and *Erroneous context* categories specifically to evaluate the content completion editor service. Finally, for testing of scalability, we manually constructed a test set consisting of 28 erroneous Stratego-Java files of increasing size in the interval of $1,000$–$15,000$ LOC.

10.2 Experiments

There are a large number of configurations to consider in evaluating the presented approach: combinations of languages, recovery rule sets, and recovery techniques. In order to limit the size of the presented results, we first concentrate on one language and experiment with different configuration of recovery rule sets and recovery techniques. For these initial experiments we use the Stratego-Java language –

[10] `http://www.yellowgrass.org/`.
[11] `http://strategoxt.org/Stratego/JSGLR/`.

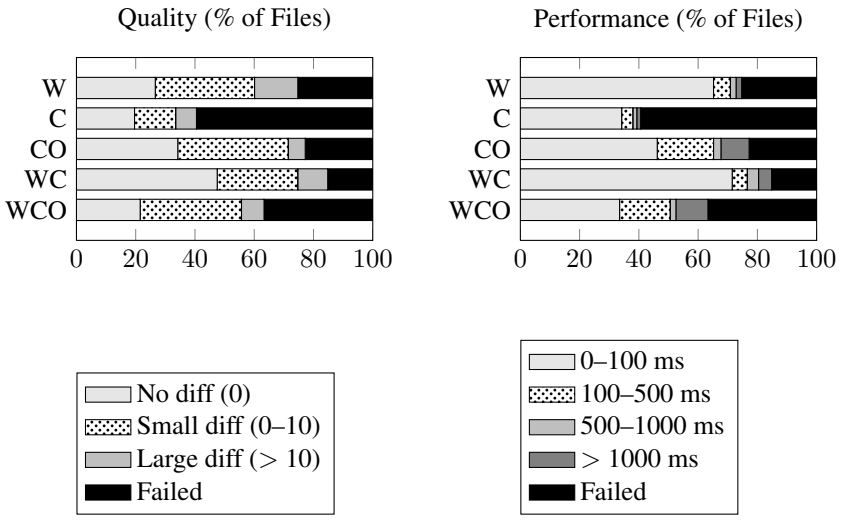


Figure 32: Quality and performance (recovery times) using a permissive grammar with different recovery rule sets for Stratego-Java. **W** - Water, **C** - Insertion of closing brackets, **O** - Insertion of opening brackets.

a fairly complex language embedding. After selecting an effective configuration, we perform additional experiments with other languages.

Selecting a Recovery Rule Set

In this experiment we focus on selecting the most effective recovery rule set for a permissive grammar with respect to quality and performance. The permissive grammar technique is used in combination with region selection, described in Section 7. That is, the recovery rules are applied on a selected erroneous region, but the fallback region recovery technique is disabled since it obscures failed recoveries obtained for the evaluated rule sets. In this experiment, we set a time limit of 5 seconds to cut off recoveries that take an (almost) infinite time to complete.

For the permissive grammars approach of Section 4, there are three recovery rule sets that we evaluate in isolation and in combination – *Water* (W), *insertion of Closing brackets* (C), and *insertion of Open brackets* (O). Results from the experiment are shown in Figure 32. The figure includes results for W, C, CO, WC and WCO for a Stratego-Java grammar. The remaining combinations, O and WO, were 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 water rules (W) both contribute to the quality of a recovery. Combined together (WC) they further improve recovery results. The insertion of opening brackets

(O) does improve the recovery quality for insertion-only grammars, which follows from comparing C to CO. However, when all rules are combined (WCO), the recovery quality decreases in comparison with the WC 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 (WCO) allows many creative recoveries that often do not correspond to the human intended recoveries. We conclude that WC seems to be the best trade off between Quality and Performance.

In this experiment we only set a limit on the number of lines (75) that were inspected during backtracking, and a time limit of 5 seconds to cut off recoveries that take an (almost) infinite time to complete. The performance diagram shows that this leads to objectionable parse times in certain cases, $4.4\% > 1.0$ seconds and $15.2\% > 5.0$ seconds (failures) for WC. For these cases, a practical implementation would opt for an inferior recovery result obtained by applying a fallback strategy (region skipping in our approach). We apply this strategy in the remainder of this section, setting a time limit of 1000 milliseconds on the time spent applying recovery rules.

Selecting Recovery Techniques

In this experiment, we focus on selecting the best parser configuration combining the recovery techniques presented in this paper: the permissive grammars and backtracking approach of Section 4 and 5 (PG), bridge parsing of Section 6 (BP), and the region selection technique of Section 7 (RS), which can be applied as a fall back recovery technique (RR) by skipping the selected region. We use the WC recovery rule set of Section 10.2. and the Stratego-Java test set. We first applied the techniques in isolation: first regional recovery by skipping regions (RR), and then parsing with permissive grammars (PG). Bridge parsing is not evaluated separately, since it has a limited application scope and only works as a supplementary method. 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), and finally the combination of all three techniques together (RS-BP-PG-RR). Throughout this experiment, we set a time limit of 1 second for applying recovery rules (PG). The results from the experiment are shown in Figure 33.

Figure 33 (Performance) shows the performance results for the different combinations of techniques. The results show that region recovery (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). Furthermore, applying the bridge parsing technique (BP) does not negatively affect performance according to Figure 33 (RS-PG-BP-RR versus RS-PG-RR). Since all techniques

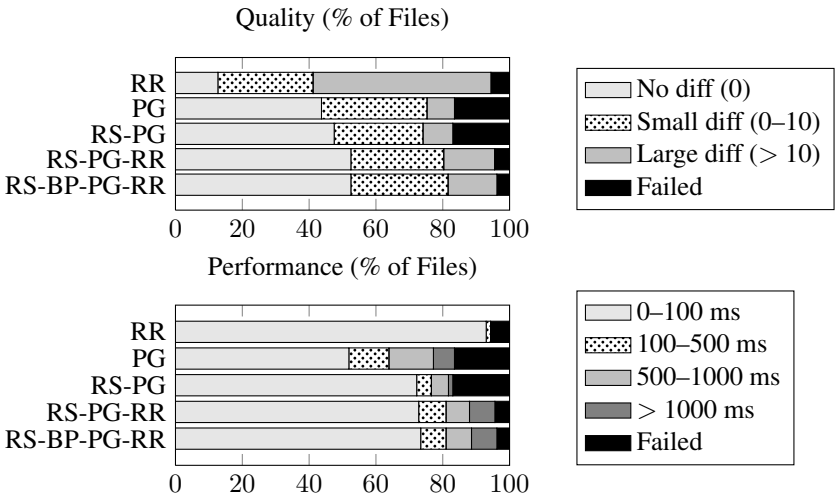


Figure 33: Quality and performance (recovery times) using combinations of techniques for Stratego-Java. **RR** - Region selection and recovery, **PG** - Permissive grammars, **RS** - Region selection, **BP** - Bridge parsing.

give reasonable performance, we focus on quality to find the best combination of techniques.

Considering the Quality part of Figure 33 and the results of PG, we see that it has the largest number of failed recoveries (17%), but regardless of this fact it still leads to reasonable recoveries (< 10 diff lines) in the majority of cases (75%). Restricting PG to a selected erroneous region (RS-PG) leads to more excellent recoveries (48% versus 44%). For regional recovery (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 (94%). Combining both techniques (RS-PG-RR), improves the robustness (96%), as well as the precision (80% small or no diff) compared to both individual techniques.

Interestingly, Figure 33 shows little beneficial effects of the bridge parsing method (BP). There is a strong use case for bridge parsing, as it can pick the most likely recovery in case of a syntax error that affects scoping structures. However, the technique is most effective for programs that use deep nesting of blocks, which are relatively rare in Stratego-Java programs. Still, the approach shows no harmful effects. For other languages its positive effects tend to be more pronounced, as we have shown in [Jon+09]. In this previous study, a test set with focus on scope errors is used; showing that bridge parsing improves the results of the permissive grammar technique in 21% of the cases where one or more scope errors occur. The cases where the bridge parser contributes to a better recovery are cases where the

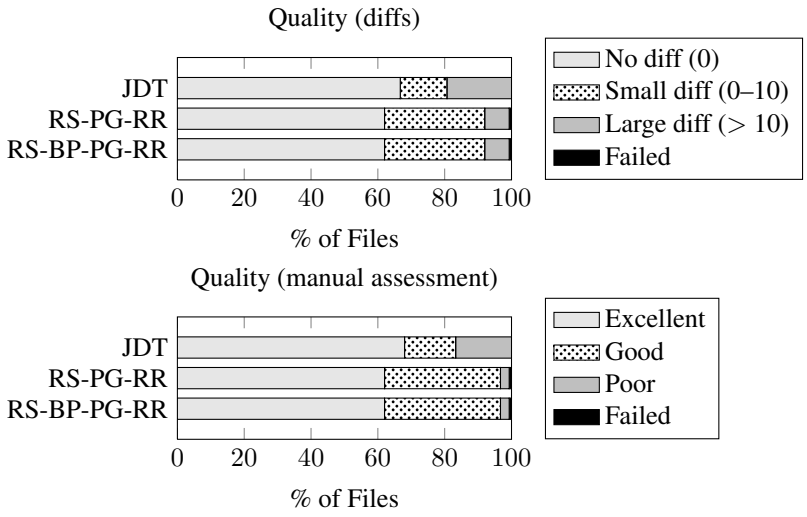


Figure 34: Quality of our approach compared to JDT. **RS** - Region selection, **RR** - Region recovery, **PG** - Permissive grammars, **BP** - Bridge parsing, **JDT** - Java Developer Toolkit.

region selection technique does not detect the erroneous scope as precisely on its own, which is typical for fragments with multiple clustered scope errors.

Overall benchmark

As an overall benchmark, we compare the quality of our techniques to the parser used by Eclipse's Java Development Tools (JDT). 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 [KT].

Both Eclipse and our approach apply an additional recovery technique in the scenario of content completion. Both techniques use specific completion recovery rules that require the completion request (cursor) location as additional information, also, these rules construct special completion nodes 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 34 shows the quality results acquired for the Java test set, using diff counts and applying the criteria of [PD78]. To ensure that all the results are obtained in a reasonable time span, we set a parse time limit of 1 second. 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 most large diff cases only a very small part of the original file was reconstructed, e.g. only the import lines or the import lines plus the class declaration whereby all declarations in the body were skipped. We conclude that our automatically derived recovery technique is at least on par with practical standards.

Cross-language quality and performance

In this experiment we test the applicability of our approach to different languages, using the RS-BP-PG-RR configuration and the WC rule set. 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 35. 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. More specifically, the recoveries for Java-SQL, in general, 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 scalability 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-BP-PG-RR recovery configuration.

To test scalability, we construct 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 are added manually, 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 results, shown as a plot in Figure 36, 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.

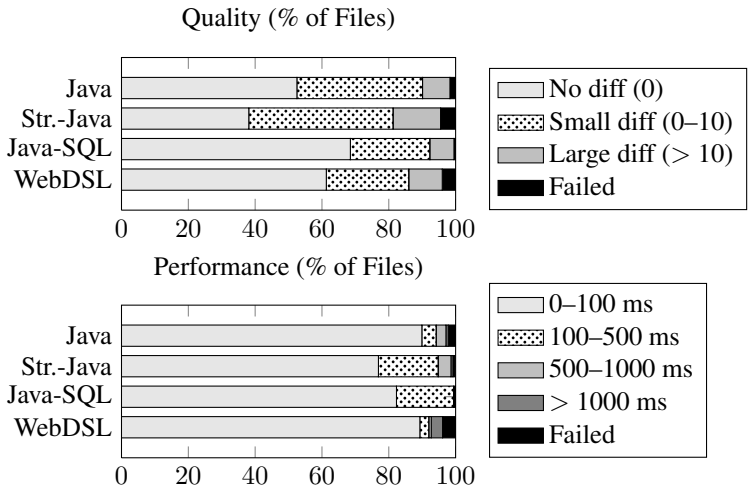


Figure 35: Quality and performance (recovery times) for different languages.

As an additional experiment we study the performance drawbacks in the increased size of a permissive grammar. The extra recovery 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 measure this effect by comparing parse times of the syntactically correct files in the test set, using the standard grammar and the WC permissive grammar. The results show that the permissive grammar has 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.

Content Completion

Error recovery helps to provide editor services on erroneous input. Especially challenging is the content completion service, which almost exclusively targets incomplete programs. In Section 8.5 we discussed the strengths and limitations of our current approach with respect to content 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 (water and insertion rules) serve the purpose of content completion, and how the completion rules improve on this.

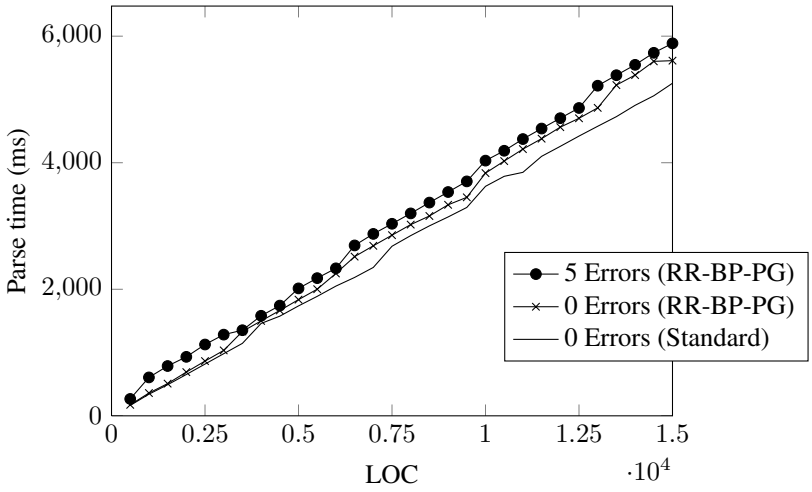


Figure 36: Parse times for files of different length with and without errors. The files are written in the Stratego-Java language and parsed with the RR-BP-PG recovery configuration.

We evaluated completion recovery on a set of 314 test cases that simulate the scenario of a programmer triggering the content completion service. Accurate completion suggestions require that the syntactic context, the tree node 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 37 shows the results for our recovery technique with and without the use of completion recovery. Using the original approach (with the WC rule set), 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, recovering large incomplete constructs can be inefficient since it requires many water and insertion rule applications.

Both problems are addressed by the completion recovery technique, which is specifically designed to handle syntax errors that involve incomplete language constructs. Figure 37 shows the results for the completion recovery strategy of Section 8.5, using a permissive grammar with the WC 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 cursor location is part of the erroneous construct.

A disadvantage of the completion rules is that they significantly increase the

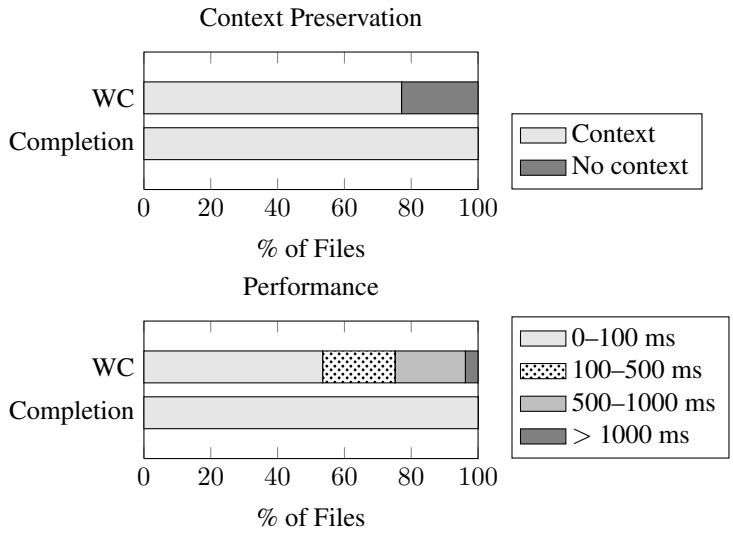


Figure 37: Context preservation and performance (recovery times) of the Stratego-Java grammar extended with completion rules (**Completion**) and extended with recovery rules (**WC**).

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 WC/Completion grammar with parse times for the WC 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. For normal editing scenarios, the completion rules can also be applied as an additional recovery mechanism that is effective only at the cursor location, although we have not focused on this capability in the experiments in this section.

10.3 Summary

In this section we evaluated the quality and performance of different rule sets for permissive grammars, and different configurations for parsing with permissive grammars, region recovery, and bridge parsing. Through experimental evaluation we found that the WC rule set provides the best balance in quality and performance. The three techniques each have their merits in isolation, and work best in combination. Through additional experiments we showed that the recovery quality and performance hold up to the standard set by the JDT, that our approach is scalable, and that it works across multiple languages. In addition, we showed its effectiveness for content completion.

11 Related Work

The problem of handling syntax errors during parsing has been widely studied [Lá71; MF88; PK80; BH82; Tai78; Fis+80; DP95; McK+95; Cor+02]. We focus on LR parsing for which there are several different error recovery techniques [DP95]. 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. Our layout-sensitive regional recovery algorithm can be used in a similar fashion, but selects discardable regions based on layout.

Correcting recovery methods for LR parsers typically attempt to insert or delete tokens nearby the location of an error, until parsing can resume [Tai78; McK+95; Cer02]. There may be several possible corrections of an error which means a choice has to be made. One approach applied by [Tai78] is to assign a cost (a minimum correction distance) to each possible correction and then choose the correction with the least cost. This approach of selecting recoveries based on a minimum cost is related to recovery selection in our permissive grammars, where the number of recovery rules used in a correction decides the order in which recoveries are considered (Section 4).

Successful recovery mechanisms often combine more than one technique [DP95]. For example, panic mode is often used as a fall back method if correction attempts fail. [BF87] 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. In our work we take indentation into account, for the regional recovery technique and for scope recovery using bridge parsing. In addition, by starting with region selection, the performance as well as the quality of the permissive grammars approach recovery is improved.

Regional error recovery methods [Lá71; MF88; PK80; BH82] 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 [PK80], or synchronizing symbols [BH82]), which are language-dependent. In our approach, we assign regions based on layout instead. Layout-sensitive regional recovery requires no language-specific configuration, and we showed it to be 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.

[BH82] presents 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

regions are constructed based on layout. Both approaches have some kind of local repair within phases or regions, and may skip parts of the input.

The LALR Parser Generator (LPG) [Cha91] is incorporated into IMP [Cha+07] and is used as a basis for the Eclipse JDT parser. LPG can derive recovery behavior from a grammar, and supports recovery rules in the grammar and through semantic actions. Similar to our approach, LPG detects scopes in grammars. However, unlike our approach, it does not take indentation into account for scope recovery.

11.1 Recovery 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 [Val07], based on substring parsing [RK91], 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.

[LT93] 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 water recovery 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) [For02; Gri06]. PEGs lack the declarative disambiguation facilities [Vis97c] that SDF provides for SGLR. Instead, they use greedy matching and enforce an explicit ordering of productions. To our knowledge, no automated form of error recovery has been defined for PEGs. However, existing work on error recovery using parser combinators [SD96] may be a promising direction for recovery in PEGs. Furthermore, 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.

11.2 IDE support for Composite Languages

We integrated our recovery approach into the Spoofax [Kat+10a] language workbench. A related project, also based on SDF and SGLR, is the Meta-Environment [Bra+02a; Bra+07]. It 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.

Another language development environment is MontiCore [Kra+07; Kra+08]. Based on ANTLR [PQ95], 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 run-time 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 [Kra+07; Kra+08]. In contrast to MontiCore, our approach is based on scannerless generalized-LR parsing, which supports the full set of context-free grammars, and allows composition of grammars without any restrictions.

11.3 Island Grammars

The basic principles of our permissive grammars and bridge parsing are based on the water productions from island grammars. Island grammars [DK99; Moo01] have traditionally been used for different reverse and re-engineering tasks. For cases where a baseline grammar is available (i.e., a complete grammar for some dialect of a legacy language), [KL03] present an approach of deriving *tolerant grammars*. Based on island grammars, these are partial grammars that contain only a subset of the baseline grammar's productions, and are more permissive in nature. 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, like parsing with GLR*, are limited to skipping content. Our approach supports recovery rules that insert missing literals and provides an extensive set of error reporting capabilities.

More recently, island grammars have also been applied to parse composite languages. [Syn+03] composed island grammars for multiple languages to parse only the interesting bits of an HTML file (e.g., JavaScript fragments and forms), while

skipping over the remaining parts. In contrast, we focus on composite languages constructed from complete constituent grammars. From these grammars we construct permissive grammars that support tolerant parsing for complete, composed languages.

12 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 paper, we presented a flexible, language-independent approach to error recovery to resolve this issue.

We presented three techniques for error recovery. First, permissive grammars, to relax grammars with recovery rules so that strings can be parsed that are syntactically incorrect according to the original grammar. Second, backtracking, to efficiently parse files without syntax errors and to gracefully cope with errors locally. Third, region recovery, to identify regions of syntactically incorrect code, thereby constraining the search space of backtracking and providing a fallback recovery strategy. Using bridge parsing, this technique takes indentation usage into account to improve recoveries of scoping constructs. We evaluated our approach using a set of existing, non-trivial grammars, showing that the techniques work best when used together, and that they have a low performance overhead and good or excellent recovery quality in a majority of the cases.

Acknowledgments This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*. We thank Karl Trygve Kalleberg, whose Java-based SGLR implementation has been invaluable for this work, and Mark van den Brand, Martin Bravenboer, Giorgios Rob Economopoulos, Jurgen Vinju, and the rest of the SDF/SGLR team for their work on SDF.

References

[BH82] David T. Barnard and Richard C. Holt. “Hierarchic syntax error repair for LR grammars”. In: *International Journal of Computer and Information Sciences* 11.4 (1982), pp. 231–258.

[Bra+00] Mark G. J. van den Brand et al. “Efficient Annotated Terms”. In: *Software, Practice & Experience* 30.3 (2000), pp. 259–291.

[Bra+02a] Mark G. J. van den Brand et al. "Compiling language definitions: the ASF+SDF compiler". In: *ACM Trans. Program. Lang. Syst.* 24.4 (2002), pp. 334–368.

[Bra+02b] Mark G. J. van den Brand et al. "Disambiguation Filters for Scannerless Generalized LR Parsers". In: *CC*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 143–158.

[Bra+07] Mark G. J. van den Brand et al. "Using The Meta-Environment for Maintenance and Renovation". In: *CSMR*. Ed. by René L. Krikhaar, Chris Verhoef, and Giuseppe A. Di Lucca. IEEE Computer Society, 2007, pp. 331–332.

[BV04] Martin Bravenboer and Eelco Visser. "Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions". In: *OOPSLA*. Ed. by John M. Vlissides and Douglas C. Schmidt. ACM, 2004, pp. 365–383.

[Bra+06] Martin Bravenboer, Éric Tanter, and Eelco Visser. "Declarative, formal, and extensible syntax definition for AspectJ". In: *OOPSLA*. Ed. by Peri L. Tarr and William R. Cook. ACM, 2006, pp. 209–228.

[Bra+08] Martin Bravenboer et al. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70.

[Bra+10] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. "Preventing injection attacks with syntax embeddings". In: *Science of Computer Programming* 75.7 (2010), pp. 473–495.

[BF87] Michael G. Burke and Gerald A. Fischer. "A practical method for LR and LL syntactic error diagnosis and recovery". In: *ACM Trans. Program. Lang. Syst.* 9.2 (1987), pp. 164–197.

[Cer02] Carl Cerecke. "Repairing Syntax Errors in LR-based Parsers". In: *ACSC*. Ed. by Michael J. Oudshoorn. Vol. 4. CRPIT. Australian Computer Society, 2002, pp. 17–22.

[Cha91] Philippe Charles. "A practical method for constructing efficient LALR(k) parsers with automatic error recovery". PhD thesis. New York, NY, USA: New York University, 1991.

[Cha+07] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton, Jr. "IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse". In: *ASE*. Ed. by R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer. ACM, 2007, pp. 485–488.

[Cor+02] Rafael Corchuelo et al. "Repairing syntax errors in LR parsers". In: *ACM Trans. Program. Lang. Syst.* 24.6 (2002), pp. 698–710.

[DP95] Pierpaolo Degano and Corrado Priami. “Comparison of syntactic error handling in LR parsers”. In: *Journal of Software: Practices and Experience* 25.6 (1995), pp. 657–679.

[DK99] Arie van Deursen and Tobias Kuipers. “Building Documentation Generators”. In: *IEEE International Conference on Software Maintenance*. 1999, pp. 40–49.

[Duc+06] Stéphane Ducasse et al. “Traits: A mechanism for fine-grained reuse”. In: *ACM Trans. Program. Lang. Syst.* 28.2 (2006), pp. 331–388.

[EV06] Sven Efftinge and Markus Völter. “oAW xText: a framework for textual DSLs”. In: *Eclipse Summit Europe, Eclipse Modeling Symposium*. Esslingen, Germany, 2006.

[Fis+80] Charles N. Fischer, D. R. Milton, and S. B. Quiring. “Efficient LL(1) Error Correction and Recovery Using Only Insertions”. In: *Acta Informatica* 13 (1980), pp. 141–154.

[For02] Bryan Ford. “Packrat parsing: : simple, powerful, lazy, linear time, functional pearl”. In: *ICFP*. Ed. by Mitchell Wand and Simon L. Peyton Jones. ACM, 2002, pp. 36–47.

[Fowa] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* `http://www.martinfowler.com/articles/languageWorkbench.html`.

[Fowb] Martin Fowler. *PostIntelliJ*. `http://martinfowler.com/bliki/PostIntelliJ.html`.

[Gri06] Robert Grimm. “Better extensibility through modular syntax”. In: *PLDI*. Ed. by Michael I. Schwartzbach and Thomas Ball. ACM, 2006, pp. 38–51.

[Grö+08] Hans Grönniger et al. “MontiCore: a framework for the development of textual domain specific languages”. In: *ICSE*. 2008, pp. 925–926.

[Hee+89] Jan Heering et al. “The syntax definition formalism SDF”. In: *SIGPLAN Notices* 24.11 (1989), pp. 43–75.

[Hei+09] Florian Heidenreich et al. “Derivation and Refinement of Textual Syntax for Models”. In: *ECMDA-FA*. 2009, pp. 114–129.

[Joh+04] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. “Generalised Parsing: Some Costs”. In: *CC*. Ed. by Evelyn Duesterwald. Vol. 2985. Lecture Notes in Computer Science. Springer, 2004, pp. 89–103.

[JV12] Maartje de Jonge and Eelco Visser. “Automated evaluation of syntax error recovery”. In: *ASE*. Ed. by Michael Goedicke, Tim Menzies, and Motoshi Saeki. ACM, 2012, pp. 322–325.

[Jon+09] Maartje de Jonge et al. "Natural and Flexible Error Recovery for Generated Parsers". In: *SLE*. Ed. by Mark G. J. van den Brand, Dragan Gasevic, and Jeff Gray. Vol. 5969. Lecture Notes in Computer Science. Springer, 2009, pp. 204–223.

[Jou+06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. "TCS: a DSL for the specification of textual concrete syntaxes in model engineering". In: *GPCE*. Ed. by Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen. Portland, Oregon, USA: ACM, 2006, pp. 249–254.

[KV10] Lennart C. L. Kats and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs". In: *OOPSLA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 444–463.

[Kat+08] Lennart C. L. Kats, Martin Bravenboer, and Eelco Visser. "Mixing source and bytecode: a case for compilation by normalization". In: *OOPSLA*. Ed. by Gail E. Harris. ACM, 2008, pp. 91–108.

[Kat+09a] Lennart C. L. Kats, Anthony M. Sloane, and Eelco Visser. "Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming". In: *CC*. Ed. by Oege de Moor and Michael I. Schwartzbach. Vol. 5501. Lecture Notes in Computer Science. Springer, 2009, pp. 142–157.

[Kat+09b] Lennart C. L. Kats et al. "Providing Rapid Feedback in Generated Modular Language Environments. Adding Error Recovery to Scannerless Generalized-LR Parsing". In: *OOPSLA*. Ed. by Gary T. Leavens. Vol. 44. ACM SIGPLAN Notices. Orlando, Florida, USA: ACM Press, 2009, pp. 445–464.

[Kat+10a] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. "Domain-Specific Languages for Composable Editor Plugins". In: *Electr. Notes Theor. Comput. Sci.* 253.7 (2010), pp. 149–163.

[Kat+10b] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. "Pure and declarative syntax definition: paradise lost and regained". In: *OOPSLA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, 2010, pp. 918–932.

[Kat+11] Lennart C. L. Kats et al. *The Permissive Grammars Project*. `http://strategoxt.org/Stratego/PermissiveGrammars`. 2011.

[Kic+97] Gregor Kiczales et al. "Aspect-Oriented Programming". In: *ECOOP*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Vol. 1241. LNCS. Springer, 1997, pp. 220–242.

[KL03] Steven Klusener and Ralf Lämmel. "Deriving tolerant grammars from a base-line grammar". In: *ICSM*. IEEE Computer Society, 2003, p. 179.

[Kra+07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "Efficient Editor Generation for Compositional DSLs in Eclipse". In: *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*. ACM, 2007, pp. 218–228.

[Kra+08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. "MontiCore: Modular Development of Textual Domain Specific Languages". In: *TOOLS (46)*. Ed. by Richard F. Paige and Bertrand Meyer. Vol. 11. Lecture Notes in Business Information Processing. Springer, 2008, pp. 297–315.

[KT] Thomas Kuhn and Olivier Thomann. *Eclipse Corner: Abstract Syntax Tree*. `http://eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`.

[LT93] Alon Lavie and Masaru Tomita. "GLR*-an efficient noise skipping parsing algorithm for context free grammars". In: *Third International Workshop on Parsing Technologies*. 1993, pp. 123–134.

[L71] Jean-Pierre Lévy. "Automatic Correction of Syntax Errors in Programming Languages". PhD thesis. Ithaca, NY, USA: Cornell University, 1971.

[MF88] Jon Mauney and Charles N. Fischer. "Determining the Extent of Lookahead in Syntactic Error Repair". In: *ACM Trans. Program. Lang. Syst.* 10.3 (1988), pp. 456–469.

[McK+95] Bruce J. McKenzie, Corey Yeatman, and Lorraine De Vere. "Error Repair in Shift-Reduce Parsers". In: *ACM Trans. Program. Lang. Syst.* 17.4 (1995), pp. 672–689.

[Moo01] Leon Moonen. "Generating Robust Parsers using Island Grammars". In: *Proceedings. Eighth Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2001, pp. 13–22.

[Moo02] Leon Moonen. "Lightweight Impact Analysis using Island Grammars". In: *Proceedings of the 10th IEEE International Workshop of Program Comprehension*. IEEE Computer Society, 2002, pp. 219–228.

[NN+09b] Emma Nilsson-Nyman, Torbjörn Ekman, and Görel Hedin. "Practical Scope Recovery Using Bridge Parsing". In: *Proceedings of the Internation Conference on Software Language Engineering (SLE 2008)*. Ed. by Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk. Vol. 5452. LNCS. Springer, 2009, pp. 95–113.

[PK80] Ajit B. Pai and Richard B. Kieburtz. "Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Drive Parsers". In: *ACM Trans. Program. Lang. Syst.* 2.1 (1980), pp. 18–41.

[PQ95] Terence John Parr and Russell W. Quong. "ANTLR: A Predicated-*LL(k)* Parser Generator". In: *Softw., Pract. Exper.* 25.7 (1995), pp. 789–810.

[PF11] Terence Parr and Kathleen Fisher. "LL(*): the foundation of the ANTLR parser generator". In: *PLDI*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 425–436.

[PD78] Thomas J. Pennello and Frank DeRemer. "A Forward Move Algorithm for LR Error Recovery". In: *POPL*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM Press, 1978, pp. 241–254.

[RK91] Jan Rekers and Wilco Koorn. "Substring parsing for arbitrary context-free grammars". In: *SIGPLAN Notices* 26.5 (1991), pp. 59–66.

[SC89] Daniel J. Salomon and Gordon V. Cormack. "Corrections to the paper: Scannerless NSLR(1) Parsing of Programming Languages". In: *SIGPLAN Notices* 24.11 (1989), pp. 80–83.

[SC95] Daniel J. Salomon and Gordon V. Cormack. *The disambiguation and scannerless parsing of complete character-level grammars for programming languages*. Tech. rep. TR 95/06. University of Manitoba, Winnipeg, Canada, 1995.

[Sau+06] Stephen Saunders, Duane K. Fields, and Eugene Belayev. *IntelliJ IDEA in Action*. Manning, 2006.

[Sch06] Sylvain Schmitz. *Modular Syntax Demands Verification*. Tech. rep. I3S/RR-2006-32-FR. Laboratoire I3S, Université de Nice-Sophia Antipolis, France, 2006.

[SVW09] August C. Schwerdfeger and Eric R. Van Wyk. "Verifiable composition of deterministic grammars". In: *SIGPLAN Notices* 44.6 (2009), pp. 199–210.

[SD96] S. Doaitse Swierstra and Luc Duponcheel. "Deterministic, Error-Correcting Combinator Parsers". In: *Advanced Functional Programming, Second International School*. Ed. by John Launchbury et al. Vol. 1129. LNCS. Springer, 1996, pp. 184–207.

[Syn+03] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. "Robust multilingual parsing using island grammars". In: *CASCON*. IBM, 2003, pp. 266–278.

[Tai78] Kuo-Chung Tai. "Syntactic Error Correction in Programming Languages". In: *IEEE Trans. Software Eng.* 4.5 (1978), pp. 414–425.

[Tom88] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Vol. 14. Kluwer Academic Publishers, 1988.

[Val07] Ron Valkering. “Syntax Error Handling in Scannerless Generalized LR Parsers”. MA thesis. University of Amsterdam, 2007.

[Vis97a] Eelco Visser. “A Case Study in Optimizing Parsing Schemata by Disambiguation Filters”. In: *International Workshop on Parsing Technology (IWPT 1997)*. Massachusetts Institute of Technology. Boston, USA, 1997, pp. 210–224.

[Vis97b] Eelco Visser. *Scannerless Generalized-LR Parsing*. Tech. rep. P9707. Programming Research Group, University of Amsterdam, 1997.

[Vis97c] Eelco Visser. “Syntax Definition for Language Prototyping”. PhD thesis. University of Amsterdam, 1997.

[Vis02] Eelco Visser. “Meta-programming with Concrete Object Syntax”. In: *GPCE*. Ed. by Don S. Batory, Charles Consel, and Walid Taha. Vol. 2487. Lecture Notes in Computer Science. Springer, 2002, pp. 299–315.

[WY07] Daniel Waddington and Bin Yao. “High-fidelity C/C++ code transformation”. In: *Science of Computer Programming* 68.2 (2007), pp. 64–78.