Building Blocks for Language Workbenches

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op dinsdag 13 december 2011 om 15:00 uur door

Lennart Christopher Leon KATS

doctorandus informatica
geboren te Amsterdam

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. A. van Deursen

Copromotor: Dr. E. Visser

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Dr. E. Visser	Delft University of Technology, copromotor
Prof. dr. P. D. Mosses	Swansea University
Prof. dr. O. Nierstrasz	University of Bern
Prof. dr. P. Klint	University of Amsterdam
Prof. dr. K. G. Langendoen	Delft University of Technology
Prof. dr. ir. R. L. Lagendijk	Delft University of Technology

The work in this thesis has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was financially supported by the Netherlands Organisation for Scientific Research (NWO) project 612.063.512, *TFA: Transformations for Abstractions*.

Cover: *Thor's Formation – Bryce Canyon, Utah* (http://flickr.com/photos/trodel/3599116604) © 2007 Luca Galuzzi, http://galuzzi.it. Creative Commons BY-SA 2.0.

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

ISBN 978-90-8570-780-6

Acknowledgments

This work would not have been possible without the advice and support of many people. First of all, I would like to express my gratitude to my adviser, Eelco Visser, for his guidance and support throughout the years. He has been a mentor and role model to me and gave me the freedom to pursue research directions that I found interesting. I also owe Martin Bravenboer for helping me learn the ropes during my first year at Delft University. I thank my promotor, Arie van Deursen, for his kind support and advice, and his suggestions for improving this manuscript.

I thank the members of the reading committee, Paul Klint, Inald Lagendijk, Koen Langendoen, Peter Mosses, and Oscar Nierstrasz, for reviewing my thesis. I would also like to thank the anonymous reviewers of conferences and journals for always being right – even when contradicting each other – and for providing many useful suggestions. Also, several people voluntarily provided useful feedback on one or more of my papers: Nathan Bruning, Danny Groenewegen, Zef Hemel, Maartje de Jonge, Bernhard Merkle, Nicolas Pierron, Kees Pronk, Glenn Vanderburg, Rob Vermaas, and Sander Vermolen.

All the work of this thesis has been done in collaboration with other people. I thank the co-authors of the included publications for their contributions: Martin Bravenboer, Maartje de Jonge, Karl Trygve Kalleberg, Tony Sloane, Emma Söderberg, Rob Vermaas, and Eelco Visser. I also thank Danny Groenewegen, Zef Hemel, Guido Wachsmuth, and others, for working together on other exciting stuff.

Martin Bravenboer and Eelco Dolstra kindly provided source files that formed a template for this thesis. Alberto González Sánchez provided me the source of his inspiring cover, depicting 'Mount Doctorate,' of which I will soon reach the summit myself. These templates saved me a lot of time during the final preparations of my thesis.

I have Karl Trygve Kalleberg to thank for coming up with the wonderful name "Spoofax" that always seems to make people wonder why the project is named like that, and I'm grateful for his many invaluable contributions to the original Spoofax project and the language workbench. I also thank Maartje de Jonge for her work, in particular for making error recovery fast and robust, and thank Emma Söderberg for helping us make that happen. Other contributors to the Spoofax project are Adil Akhter, Nathan Bruning, Sebastian Erdweg, Ricky Lindeman, Sverre Rabbelier, Vlad Vergu, Rob Vermaas, Tobi Vollebregt, and Guido Wachsmuth; thank you for all your hard work!

The *Software Engineering Research Group* is a wonderful place to do research. Plus it has pretty decent coffee too. Several times a day, we would form a small crowd consisting of loyal coffee and tea addicts including Sander van der

Burg, Eelco Dolstra, Danny Groenewegen, Zef Hemel (my 'bro'), Maartje de Jonge, Rob Vermaas, and Sander Vermolen. We'd arrange these stand-up meetings by sending a short message over an IRC channel (often in the form of a single letter 'c' or the occasional Unicode U+2615 character), and then pretty much talked about life, the universe, and everything. Other current and past group members, unfortunately too many to name here, have contributed to a great working environment. It has been a pleasure to work with you all.

Last but certainly not least, I would like to thank my family. I am grateful to my parents who have supported me these years. And I thank my sister, Merel, for all the fun times and her unwavering encouragement.

Lennart Kats
October 20, 2011,
Delft

Contents

Introduction

1

This dissertation presents research on techniques, methods, and tool support for *domain-specific language engineering*. Domain-specific language engineering is the discipline of designing, developing, and maintaining domain-specific programming languages. The focus of this thesis is the architecture of *language workbenches* their underlying technologies. Language workbenches are tools that make language engineering more efficient by providing an integrated development environment for language engineering tasks. In particular, we introduce the Spoofax language workbench, and describe its techniques for high-level, portable language definitions, language composition, interactive support for defining languages, and language testing.

In this introductory chapter, we first provide background on domain-specific languages and give an overview of domain-specific language engineering approaches and tools. We then discuss open challenges in the area and outline the main research questions that are addressed in this thesis.

1.1 DOMAIN-SPECIFIC LANGUAGES

Programming languages are at the heart of computer science. Fundamentally, programming languages are used to write programs that control the behavior of computer systems. Traditionally, they would do just that: provide a way to control the hardware, by allocating and accessing memory, executing computations, and controlling devices. Over time, high-level programming languages introduced *abstractions* over implementation details and moved from being hardware-oriented to being task-oriented. This evolution of languages has been a fundamental factor in allowing the construction of software of significant scale, complexity, and flexibility.

Domain-specific programming languages (DSLs) introduce abstractions for tasks in a specific domain. Notable early examples of DSLs include languages made in the Unix tradition of Little Languages [Bentley, 1986] that started in the 1970's and spawned specialized languages for technical domains such as shell services (sh), incremental rebuilds (make), lexical analysis (Lex), and parsing (Yacc). Another well-known example is the structured query language SQL [Chamberlin and Boyce, 1974], which was introduced in the same time period, and is still commonly used today for database queries. DSLs are also used in domains such as insurance modeling, real-time financial trading, and scientific computation. Recent uses of DSLs have been extensively documented in surveys such as [Spinellis and Guruprasad, 1997; van Deursen and Klint, 1998; van Deursen et al., 2000; Spinellis, 2001; Mernik et al., 2005].

Two central concepts that make DSLs compelling are *domain-specific notation* and *linguistic abstraction*. Through domain-specific notation, DSLs can adopt specialized words, phrases, and other notational conventions that correspond to the problem domain. Through linguistic abstraction, they allow solutions to be expressed at a higher level. With the combination of these concepts, DSLs provide the following opportunities:

- *Concise, high-level specification* of software by providing specialized notation and abstracting over implementation details, increasing developer productivity and software maintainability and understandability.
- *Encapsulation of domain knowledge* by incorporating domain expertise and technical implementation know-how, ensuring separation of concerns and supporting reuse among multiple DSL programs.
- *Domain-specific analysis, verification, optimization, and parallelization* of DSL programs based on domain concepts that are made explicit in DSLs, aiding in program understanding, robustness, and avoiding the need for manual optimization at the underlying implementation level.
- *Domain-specific tool support* for working with DSLs, using domain-specific analyses in tools such as specialized editors, facilitating DSL program development, understanding, and maintenance.
- *Portability* of DSL programs by abstracting over implementation details of a particular implementation platform and potentially supporting multiple target platforms.

The realization of these opportunities involves two fundamental trade-offs:

- DSLs trade generality for appropriate abstractions, notation, and expressive power in a limited domain.
- The opportunities provided by DSLs must be balanced against the cost of design, development and adoption of a DSL.

The key to effectively applying a DSL approach in a certain domain is to ensure that these are favorable trade-offs. This requires a good understanding of the application domain, in order to select a suitable scope and design appropriate abstractions, and systematic techniques and tools for efficient DSL implementation. DSL engineering techniques and tools aim to facilitate both the design and construction of DSLs by minimizing the development time and ensuring quick turnaround time between design decisions, new domain insights, and new technical developments.

1.2 DOMAIN-SPECIFIC LANGUAGE ENGINEERING

There are two architectural approaches that DSL engineers can follow, determining the techniques and tools involved in the construction of DSLs. With the *internal DSL* approach, DSL engineers rely purely on the use of idiomatic

programming techniques in a general-purpose language, to stylize an application framework as a language in its own right. With the *external DSL* approach, DSL engineers design a custom syntax (notation) and semantics (meaning) for each DSL, usually implemented in the form of specialized interpreters or compilers. We discuss both approaches next.

1.2.1 *Internal Domain-Specific Languages*

Internal DSLs, sometimes called domain-specific embedded languages, rely purely on the syntax and semantics provided by a general-purpose host languages such as Haskell, Scala, or Smalltalk [Hudak, 1998; Fowler, 2011]. Internal DSLs are distinguished from traditional libraries and application frameworks by their use of idiomatic programming techniques. Instead of using specialized tools for language engineering, they use programming techniques such as fluent interfaces [Fowler, 2011] and meta-programming capabilities provided by the host language such as template meta-programming, reflection, and implicits. These features give libraries the look and feel of a new language, while still maintaining full integration and compatibility with the host language. As an example, the Mockito library[1] uses fluent interfaces and reflection to implement an internal DSL in Java:

```
Iterator<String> mockIterator = mock(Iterator.class);
when(mockIterator.next())
  .thenReturn("First")
  .thenReturn("Second")
  .thenThrow(new NoSuchElementException());
```

Using an API that resembles English sentences, it provides an abstraction for the construction of mock objects. In this example, it dynamically creates a class that implements the iterator interface with the behavior described for the `next()` method. As an internal DSL, Mockito adds no keywords or other syntax to the language but relies purely on the syntax and semantics of the Java language.

DSLs are naturally smaller in scope than traditional, general-purpose languages. The approach of internal DSLs can provide a good match for this reduced scope by making it possible to develop a DSL with only modest effort. Still, to fully realize the potential of DSLs, the approach also comes with a number of inherent limitations [Mernik et al., 2005]:

- The syntax and structure of the DSL are constrained by the user-definable notation offered by the host language, limiting the potential for domain-specific notation.

- Appropriate domain-specific constructs and abstractions cannot always be mapped in a straightforward way to functions or objects that can be put in a library. Examples named in [Mernik et al., 2005] are traversals and error handling.

[1] `http://mockito.org/`.

- The opportunities of DSLs for domain-specific analysis, verification, optimization, and parallelization are much harder or unfeasible based on a general-purpose language, since the source code patterns are usually too complex and not well defined. Examples are provided in [Hemel et al., 2011].

These limitations can be overcome by the use of a separate definition of the syntax and semantics of the DSL, allowing the DSL to break the mold of its general-purpose host language and facilitating analyses and transformations on domain-specific language constructs. Such a separate syntactic and semantic definition is the fundamental distinguishing characteristic of *external DSLs*.

1.2.2 *External Domain-Specific Languages*

External domain-specific languages have their own syntax and semantics, independent of any general-purpose language. This means they can provide stronger support for domain-specific notation than internal DSLs, and support increased flexibility in the static and dynamic semantics of the language. This flexibility is achieved by implementing the language syntax and semantics in the form of a tool such as an interpreter or compiler.

Beyond interpreters and compilers, language engineers can implement specialized support for using an external DSL in an integrated development environment (IDE). Modern, graphical user-interface based IDEs provide rich *editor services* that are tailored towards a specific language. Syntactic editor services provide functionality based on the syntax of a language, e.g. syntax highlighting, syntax error marking, code folding, and an outline view. IDEs can also support semantic editor services that correspond to the output of a compiler, marking errors and warnings inside the editor. Modern IDEs even take a step further and provide *deeply integrated semantic editor services* that provide further functionality at the semantic level of a program, such as reference resolving, content completion, and refactoring.

Traditionally, a lot of effort was required for implementing language syntax, semantics, and editor services of external DSLs. Parsers, data structures for abstract syntax trees, traversals, transformations, and so on would be coded by hand for each language. The development of editor services adds to this burden, requiring developers to implement syntax highlighting, outline views, content completion, and all the other language-specific editor services for each DSL. This meant that a significant investment in time and effort was required for the development of a new language. Specialized tools for DSL and IDE engineering aim to address this issue.

1.2.3 *Domain-Specific Language Engineering Tools*

Language engineering tools significantly reduce the effort required for the development of new external DSLs. They are aimed at specific aspects of language engineering, and allow language engineers to write high-level spec-

ifications of DSL components in *meta-languages* rather than develop each component by hand using general-purpose tools and programming languages. The development of language engineering tools has been an active area of research for many decades. We give a brief overview of the different categories of tools.

Tools for syntax definition Writing and maintaining large parsers by hand can quickly become a tedious and laborious job. Instead of writing a parser by hand, the syntax can be described using a grammar – a DSL for syntax definition. The syntax of most programming languages can be described using context-free grammars. For these grammars, it is possible to efficiently *generate* a parser using a parser generator [Grune and Jacobs, 2008].

Parser generators are distinguished by what types of grammars they support and the performance guarantees they provide for generated parsers. Mainstream parser generators such as Yacc [Johnson, 1975] and ANTLR [Parr and Fisher, 2011] generate LR or LL parsers that can only be used for certain types of grammars, e.g. grammars with overlapping productions. Since only the full class of context-free grammars is closed under composition, and not any of its subclasses such as LL and LR, these parsers do not support modularity of syntax definitions. Generalized parsers such as Generalized LR (GLR) [Tomita, 1988] and Earley parsers [Earley, 1970] support the full class of context-free grammars. Using scannerless parsing [Salomon and Cormack, 1989, 1995], Scannerless GLR (SGLR) even realizes this for lexical syntax (keywords, strings, comments, and so on) [Visser, 1997c]. SDF [Heering et al., 1989] is a grammar formalism notable for its use of GLR and subsequent use of SGLR to support modular, declarative syntax definitions.

Tools for static and dynamic semantics The static semantics of a programming language defines restrictions on the structure of valid programs that are hard or impossible to express in a standard syntactic formalism. For example, a restriction can be that variables should be assigned before use. The dynamic semantics of a programming language describes the runtime behavior of programs. In this work we focus on meta-programming tools and frameworks that describe the runtime behavior by transformation to existing languages, in contrast to approaches based on formal semantics, which aim at modelling the observable behavior of programs.

Meta-programming languages and frameworks make it much easier to specify the semantics of a language, by providing a reusable abstraction layer and supporting suitable programming paradigms. Examples based on rule-based transformation systems include ASF+SDF [van den Brand et al., 2002], Stratego/XT [Bravenboer et al., 2008], and TXL [Cordy et al., 1991]. These systems use rewrite rules as a core paradigm for concise specification of transformations. Examples based on attribute grammars include Eli [Kastens and Waite, 1994] and JastAdd [Hedin and Magnusson, 2003]. These systems use equations to concisely specify relations between attributes (or properties) of abstract syntax tree nodes. Other systems include Polyglot [Nystrom et al., 2003], a Java based extensible compiler framework, and Rascal [Klint et al., 2009], a language for analysis and transformations.

Beyond syntax and semantics, developing IDE support is an increasingly important aspect of implementing DSLs. We discuss IDE engineering next, and subsequently the integration of language engineering and IDE engineering.

1.3 IDE ENGINEERING

For many current programming languages, IDEs have been developed separately from a compiler or interpreter of the language. A common approach is to use an *extensible IDE platform* such as Eclipse or Visual Studio, which integrates IDE support for multiple languages using a plugin architecture. Plugins consist of one or more services, such as editor services, which are registered using a component model such as the OSGi Service Platform [OSGi, 2009]. Many plugins already exist for these IDE platforms, providing IDE support for specific languages as well as language-independent facilities such as version control and build management systems. Still, even with extensible IDE platforms, implementing state-of-the-art support for a new language is a daunting undertaking, requiring extensive knowledge of the sometimes complicated and highly interdependent APIs in general-purpose languages such as C, C#, or Java, the extension mechanisms of the plugin framework, as well as an in-depth understanding of the structure and semantics of the subject language.

Specialized tools and frameworks for IDE development significantly simplify the implementation of IDE services. Modern examples are Eclipse IMP [Charles et al., 2007, 2009] and the Dynamic Language Toolkit [DLTK, 2007]. They introduce layers of abstraction and make use of parser generators and code generation techniques to significantly simplify the development of an IDE for an existing language.

A problem with separate development of IDEs and language compilers/interpreters is that there tends to be significant overlap between them. For a fully functional IDE that supports deeply integrated semantic editor services, either an existing compiler should be integrated into the IDE, or the language semantics should be re-implemented. A different approach is to integrate DSL and IDE engineering, as we discuss next.

1.4 INTEGRATED DSL AND IDE ENGINEERING TOOLS

Tools such as Synthesizer Generator [Reps and Teitelbaum, 1989], Centaur [Borras et al., 1989], and Lrc [Kuiper and Saraiva, 1998] were some of the first to integrate the specification of language syntax and semantics for the automatic generation of IDEs and compilers or interpreters. Combining these different aspects of language development had major advantages: a low threshold for making an IDE, co-evolution of components shared between language compiler or interpreter and IDE, and elimination of all redundancy between those components.

A next step for comprehensive language engineering tools was self-application and integration: providing not only IDEs for programmers but also for meta-programmers. The Meta-Environment [Klint, 1993; van den Brand et al., 2001] is notable for realizing this vision. It made it possible for meta-programmers to define languages in a meta-IDE and generate IDEs for those languages. It uses a combination of SDF for syntax definition and ASF to specify language semantics, and provided a graphical interface for working with language definitions and generating new programming environments.

IDE facilities The Meta-Environment and its predecessors generate code editors with syntax highlighting, an error message view, and optional integration with transformations or other tools. However, the generated editors did not yet support deeply integrated semantic editor services, as the tools provided limited means to expose the semantics of a language to be used with such services. In addition, there was much debate at the time as to which editing paradigm to use: free text editing, as applied in plain text editors, or syntax-directed editing, where edit operations are directed by the syntactic structure of the language. Syntax-directed editors were a promising direction for increased language sensitivity in editors at the cost of flexibility [Waters, 1982; Shani, 1983; Khwaja and Urban, 1993]. This design choice helped with the automatic generation of editors from a language specification, but provided an editing environment that is constrained in comparison to current IDEs that parse program text with error recovery as it is typed. In the case of the Meta-Environment, a hybrid approach was chosen, where users could select a part of the program as the "focus" that could be textually edited [van Dijk and Koorn, 1990]. Only text that could be parsed without syntax errors could leave the focus mode, in order to ensure that there was always a syntactically correct snapshot of the edited program.

Language Workbenches Language workbenches are a new generation of language engineering tools that aim to combine language and IDE specification [Fowler, 2005a, 2011]. They are distinguished by their strong focus on IDE support and specialization for DSLs rather than general-purpose languages.

By focusing on the construction and maintenance of new DSLs, rather than on the implementation of existing, general-purpose languages, language workbenches need not be concerned with supporting idiosyncrasies of existing language implementations such as specific preprocessor schemes, multiple language dialects, or irregular type systems. Instead, they aim to provide specialized facilities to increase the cost-effectiveness of DSLs.

The field of language workbenches is still young, but awareness of the need and the benefits of these tools is increasing both in the research community and in industry. Notable language workbenches originating in the former category include EMFText [Heidenreich et al., 2009a] and MontiCore [Krahn et al., 2008]; workbenches in the latter category include MPS [Voelter and Solomatov, 2010] and Xtext [Efftinge and Voelter, 2006].[2] Each provides novel,

[2] A more extensive overview of current language workbenches and a comparison to our work is given in Section 4.7.

unique qualities and innovations. For example, EMFText is notable for its derivation and refinement of concrete syntax; MontiCore for its combination of syntax and editor service specification; MPS for its extensive support for modular language definitions; and Xtext for its wide range of editor services. However, they also share a certain degree of pragmatism in avoiding a number of fundamental language engineering issues. Allegorically, we name seven examples:

- Using mainstream parser generators such as ANTLR, having an excellent implementation and support for error recovery, but being restricted to LL (or LR) grammars and providing no support for composition;
- or, forgoing the parsing challenge altogether and revisiting syntax-directed editing, in the form of projectional editing;
- using a general-purpose language such as Java rather than a DSL to implement the more advanced editor services;
- using a general-purpose language such as Java rather than a DSL for transformations on abstract syntax;
- or, forgoing transformations on the abstract representation and directly generating code;
- using string-based template engines rather than syntactically safe meta-programming systems for code generation;
- providing only general-purpose tools such as JUnit for testing DSL implementations.

Although these examples characterize the state of the practice, there are some exceptions. In particular, MPS provides its own support for syntactic normalization rules for local-to-local transformations, and bases its editor services on projectional editing. Xtext provides the Xtend language for transformations, an imperative language that incorporates most of the Java language and adds features such as string-based template syntax and multiple dispatch. Fully addressing the underlying language engineering issues together poses several research challenges, as outlined next.

1.5 CHALLENGES AND RESEARCH QUESTIONS

In the preceding sections we have argued that DSLs have great potential for productivity gain and described techniques and tools that help realize this potential. Language workbenches integrate multiple techniques and tools for language and IDE specification into a comprehensive, interactive environment. Current implementations lack techniques for applying meta-DSLs for specification of declarative syntax definitions, language semantics, transformations, and language tests, that could further support this combination of specifying languages and IDEs.

In this thesis we focus on techniques to realize a language workbench that facilitates the use of high-level, syntactic and semantic language definitions based on meta-DSLs. The main research goal and subject of this thesis can be formulated as follows:

> *Introduce abstractions for high-level, declarative language definitions, from which extensible, scalable language implementations with IDE support can be generated. Support those abstractions with techniques that realize an integrated, interactive language engineering environment.*

The focus of this work has been on three main research themes:

- Applying **domain-specific languages** for declarative specification of languages and IDEs;
- supporting **declarative syntax definition** for generating a parser-based, interactive development environment;
- and providing **interactive meta-tooling support**, exploring the application of modern IDE technology to DSL engineering.

Below, we discuss these viewpoints and formulate the research questions that drive the work presented in this thesis. In Chapter 9, we revisit these questions and present our conclusions.

1.5.1 *Domain-Specific Languages For Declarative Specification of Languages and IDEs*

Our work follows in a line of research aimed at automatic generation of language implementations. It has been our aim to use as much existing technology as possible, and to determine what programming techniques and idioms, language primitives, and abstractions are needed to employ language implementation technology for high-level specification of both languages and modern IDEs. In particular, our work makes use of the Stratego/XT program transformation suite [Bravenboer et al., 2008], which includes the Stratego program transformation language and SDF [Heering et al., 1989; Visser, 1997c] for syntax definition. Using the *MetaBorg* approach, the tool suite can be effectively applied to composable language specification [Bravenboer and Visser, 2004]. Where previous work emphasized batch tools, we focus on interactive tools instead. In previous work, Stratego has only been applied for generating batch-based, command-line transformation systems before. SDF, however, has been used interactively in the Meta-Environment, but it could only parse and process programs in a fully syntactically correct state.

Language modularity and extensibility Key to the efficient development of languages and IDEs are *abstraction*, to eliminate the accidental complexity of interpreter, compiler, and IDE implementation, *modularity*, to reuse definitions of language and editor components, and *extensibility*, to customize existing

components. To explore these themes, our research begins with a case study in modular language design.

Language extensions can increase the expressivity of programming languages, by introducing abstractions for common programming tasks or for tasks in a certain domain. For instance, an example of a language extension defined using MetaBorg is SWUL, a language extension for constructing user interfaces based on the Swing framework [Bravenboer and Visser, 2004]. MetaBorg provides a general approach called for introducing language extensions, independent of a particular host language, and without restrictions on the syntax or on the context-sensitivity of the language extension.

The MetaBorg approach is based on source-to-source program translation, translating language extensions to the base language. Using source-to-source translation, it is loosely coupled to the base compiler or interpreter. This approach ensures portability of languages extensions and robustness against changes in the base language implementation. It also helps language engineers in rapid prototyping, as they need only basic knowledge of the language structure and semantics, and can abstract over the underlying implementation of the existing compiler or interpreter. A disadvantage of this approach is that source-to-source translators are restricted to only using the surface syntax of a language, and cannot adapt the internal stages and components of a base compiler, for example to support separate compilation or to make use of low-level primitives of the platform. For instance, in the case of Java, the surface syntax of the language does not expose JVM platform features such as jump instructions, unbalanced synchronization, and debugging information. This leads to our first research question:

Research Question 1

> *How can modular language plugin definitions abstract over the implementation architecture of a particular programming language? Can such plugins use lower-level features provided by the target platform?*

To answer this question, Chapter 2 investigates the design and application of an open compiler for the Java language that exposes its platform primitives (i.e., Java bytecode instructions) in the source language. We compare the implementation of language extensions such as traits, partial classes, and iterator generators based on this model to traditional open compiler approaches.

Portability of language definitions Given high-level, modular language definitions, our aim is to generate a combination of artifacts that perform traditional interpretation or compilation tasks *and* artifacts that provide IDE support. The latter entails generating IDE plugins for use in an extensible IDE platform such as Eclipse. Eclipse is based on Java, while our meta-DSL is based on C. As a first step, a study is needed to identify the challenges in porting a meta-DSL and techniques that can efficiently address them.

A central activity in retargeting a DSL to a new platform is usually the introduction of a new compiler back end. The back end translates the DSL to the platform's language or instruction set. Beyond that, operating system-level operations, calls to other executables, performance assumptions, and

uses of the file system, specific to the original platform, must be reconsidered. Unfortunately, these concerns are pervasive in DSLs that an extensive existing code base and a collection of standard libraries that provide functional abstractions for common tasks. The challenge in porting such a DSL lies in minimizing or avoiding the need to manually adapt this existing code base. This leads to the following research question:

Research Question 2

> *How can DSLs be efficiently ported to another platform, taking into consideration their reliance on platform-specific operations and characteristics? Is it possible to do so without changing existing DSL programs and libraries written in the language?*

Chapter 3 identifies idioms for using aspect-oriented programming [Kiczales et al., 1997] to address portability concerns of language implementations. We study the application of these idioms by porting the Stratego language to the Java platform.

Abstractions for combined specification of languages and IDEs Modern IDEs support well over a dozen distinct, language-specific syntactic and semantic editor services. These range from services that rely purely on the lexical syntax of the language, such as automatic comment insertion, to those relying on the context-free syntax, such as syntax highlighting and code folding, to deeply semantic services, such as content completion, which rely on the semantics of the language. Current language definition formalisms do not yet provide the means to specify these editor services, in particular those that rely on deep integration with the semantic definition of the language.

Because of the highly language-specific nature of modern editor services, they depend on a combination of syntactic and semantic properties of languages. Consequently, a risk for the implementation of these services is redundancy between different services and with the language definition. For example, syntactic services such as syntax highlighting and the outline view are often implemented by writing regular expressions, which duplicates functionality also provided by the parser. Another risk is tight coupling to the implementation, i.e. the parser, and potentially even the parse error recovery strategy for robust editor service implementations. In the case of deeply integrated semantic editor services, services depend on properties of the language semantics that are not normally exposed with batch-based tools such as compilers. Even at the implementation level, many compilers work with symbol tables that map name references to types, and are only in memory when considering a certain scope. The full name analysis must be exposed for services such as reference resolving and content completion. Ultimately, abstractions for high-level specifications of these services are needed that balance reuse and separation of concerns, leading to the following research question:

Research Question 3

How can editor service specifications be integrated into syntactic and semantic specifications of DSLs, balancing reuse and separation of concerns? How can language analysis components be structured to expose an interface for use by semantic editor services?

To address this question, Chapter 4 describes the architecture of the Spoofax language workbench and investigates idioms for language definitions that combine syntax, semantics, and editor services. In addition, Chapter 6 introduces language-independent techniques for implementing parser-based editor services.

Abstractions for semantic analyses Semantic analyses are important for understanding and transforming programs. Attribute grammars are a powerful formal specification notation for tree-based computation, particularly for semantic analysis and processing of software languages [Paakki, 1995]. They use declarative equations to specify the functional relationships between attributes (or properties) of abstract syntax tree nodes.

Many extensions of attribute grammars have been proposed that abstract over commonly occurring patterns, in particular supporting attribution patterns with non-local dependencies, such as *copy rules*, collection attributes [Boyland, 2005], and *circular attributes* [Boyland, 1996; Magnusson and Hedin, 2007]. With these extensions, attribute grammars are highly effective for concise specification of classic data-flow analyses such as live variables and reaching definition analysis [Nilsson-Nyman et al., 2008].

Current extensions of attribute grammars are implemented as part of an attribute grammar evaluator. An open issue is to provide abstractions for the specification of attribute grammar extensions rather than provide attribute grammars extensions as part of an attribute grammar evaluator. By providing a set of primitives to specify extensions rather than directly providing the extensions, new abstractions could be quickly implemented as new idioms are identified [Steele, 1999]. Unfortunately, no well-defined set of such primitives has been defined yet, and they are not linguistically exposed in attribute grammar languages. This leads to the following research question:

Research Question 4

Is it possible to generalize over common attribute grammar abstraction mechanisms? What primitives are needed for this generalization? Given these primitives, is it possible to introduce new abstractions for common analyses of DSLs?

Chapter 5 investigates the combination of the attribute grammars and strategic programming to answer this question. We apply the approach to existing idioms from the attribute grammar literature and study its application in grammar analyses and transformations for parse error recovery in Chapter 6.

1.5.2 *Declarative Syntax Definition in Interactive Environments*

The SDF syntax definition formalism allows for declarative, composable syntax definitions. These properties are supported by the scannerless generalized-LR (SGLR) algorithm [Visser, 1997b] used for parsers generated from SDF definitions. Using a combination of scannerless parsing [Salomon and Cormack, 1989, 1995] and generalized-LR parsing [Tomita, 1988; Rekers, 1992] it supports the full class of context-free grammars for both lexical and context-free syntax.

Parsers are at the heart of the implementation of editors in modern IDEs. 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 AST for syntactically invalid programs. Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

Parse error recovery for generalized parsers such as SGLR has been an open issue. The scannerless, generalized nature of SGLR poses challenges for the diagnosis and recovery of errors. 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 fails, 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*. Moreover, based on characters, common error recovery techniques based on token insertion and deletion are ineffective, as many insertion or deletions are required to modify complete keywords, identifiers, or phrases. Together, these two challenges make it harder to apply traditional backtracking error recovery approaches.

For use in a language workbench, an added criterion for an error recovery implementation is that it should require minimal effort from language engineers. IDEs have often used hand-written parsers in the past, or, in the case of Eclipse's Java editor, parsers with manual customization of the recovery behavior. For a language workbench, manual implementation or customization of a parser is an undesirable distraction that breaks the abstraction provided by a syntax definition;

Research Question 5

> *What techniques are needed to efficiently diagnose and recover from syntax errors with scannerless, generalized parsers? Is it possible to support error recovery without breaking the abstraction of pure and declarative syntax definition?*

Chapter 6 introduces techniques for automatic relaxation of grammars to accept syntactically incorrect inputs, in order to diagnose and recover from syntax errors. We investigate the combination of this technique with adaptations

of the parsing algorithm for backtracking and layout-sensitive parsing, in order to efficiently recover from syntax errors in a language-independent fashion.

1.5.3 *Interactive Meta-Tooling Support for Language Engineering*

Meta-programming with concrete object syntax Most meta-programming systems operate on the abstract syntax of an object language, using a structured representation of programs rather than a textual one. A disadvantage of working with abstract syntax is often its verbosity and unfamiliarity compared to the concrete syntax of a language. A solution is provided by *meta-programming with concrete object syntax* [Klint, 1993; Visser, 2002], where the concrete syntax of the language is embedded in the meta language. Quotations of concrete syntax fragments and anti-quotations for meta-level expressions and variables can then be used to manipulate the abstract representation of programs. Visser [2002] described a general architecture for introducing concrete syntax for any object language into any meta language by relying on the compositionality provided by SDF and SGLR parsing.

A usability problem of concrete syntax embedding is that quotations are often ambiguous. For example, a quoted Java code fragment `|[ i = 2 ]|` can either be an assignment expression, part of a local variable declaration, or even an annotation element initializer in abstract syntax. One approach to resolve these ambiguities is to require developers to write explicit tags or type names that indicate the syntactic category, e.g. writing `Expr |[ i = 2 ]|`, using the tag `Expr` to indicate that the quotation contains an expression. An open issue with this approach is discoverability of these names, as they depend on the (grammar) implementation and are not part of the concrete syntax of a language. If a fragment has multiple possible interpretations, and it is not clear from the context which one is correct, then only the developer can resolve it. Based on an automated diagnosis of ambiguities, an IDE could assist developers in resolving ambiguities, leading to the research question:

Research Question 6

> *Can ambiguities in concrete syntax quotations be automatically diagnosed in order to determine the possible syntactic disambiguations? Can an IDE for meta-programming provide unobtrusive, interactive feedback based on such a diagnosis?*

To answer this question, Chapter 7 describes how a generalized parser can be used to recognize all possible interpretations of quotations and investigates how to diagnose these, and how to systematically provide textual disambiguation options. We study the application of this approach based using a set of existing meta programs that use quotations of different object languages.

Abstractions for language definition testing Testing is one of the most important tools for software quality control and inspires confidence in software [Beck, 2003]. Additionally, tests can be used as a basis for an agile, iterative de-

velopment process by applying *test-driven development* [Beck, 2003]. The application of general-purpose testing techniques and frameworks such as JUnit [Hamill, 2004] requires a significant investment in language-specific infrastructure for writing test cases for syntax, static semantics, and editor services. Current research in automatic generation of test suites for language implementations [Boujarwah and Saleh, 1997; Kossatchev and Posypkin, 2005] focuses on testing complete compiler implementations, and is ineffective for testing language definitions as they are developed.

Language workbenches provide language engineers with the ability to interact with a language definition through the use of an editor. A common practice among language engineers is to maintain a "scratch pad" with some example program that focuses on new features that are under development. Language engineers can interact with it in various ways. For example, they can introduce type errors ("does the type checker catch this?"), control-click on an identifier ("does this hyperlink point to the right place?") or generate and run code for the example. An open issue is to provide a *systematic* approach to language testing in language workbenches.

Challenges for low-threshold language testing are to provide an abstraction over the implementation details of a particular language implementation and to provide the same level of interactivity and responsiveness as ad hoc tests in a language workbench. This leads to our last research question:

Research Question 7

> *Is it possible to define a general abstraction for systematic testing of DSL definitions? How can IDEs facilitate the development of DSL tests?*

Chapter 8 describes the design of a parametric testing language that can be instantiated for a specific DSL, investigating how to address testing of different aspects of languages from both a language design and a tooling perspective.

1.6 APPROACH

The core of our research method is to propose new concepts, techniques, and tools, to address open questions in the area of software and language engineering. Our work has a strong emphasis on tools and automation, and has resulted in the development of several open-source tools, in particular Aster, Spoofax, STRJ, and The Dryad Compiler, as well as contributions to the existing JSGLR, Kiama, Stratego/XT, SugarJ, and WebDSL projects.[3]

The aim of our project is to collect techniques, methods, and tool support for domain-specific language engineering that improve the *productivity* of language engineers, based on dimensions such as understandability, maintainability, reliability, portability, compositionality, and ease of development of software language designs. Key to progression along these dimensions is *abstraction* in the form of linguistic constructs that provide increased expressiveness or in the form of tools that abstract over common idioms or address

[3] See `http://lclnet.nl/software/phd/`.

common problems that exist in a domain. Research in programming language abstractions, e.g. the line of research in attribute grammars outlined above [Boyland, 1996, 2005; Magnusson and Hedin, 2007; Nilsson-Nyman et al., 2008] aims at identifying idioms and problem areas and showing for representative case studies how these can be better supported with new abstractions. We use critical discussions on the findings to achieve analytical generalizations of the results and to show how it refines previous work. A summary of how the approaches of the core chapters have been evaluated is given in Section 9.2.

Our research is supported by tool implementations that *realize* the abstractions, and are amenable to quantitative metrics such as performance metrics. In our work we measured the performance of parsing and recovery in Spoofax, attribute evaluation in Aster and Kiama, and compilation in STRJ, giving an indication of practicality of an approach. To evaluate the developed tools, we apply them to projects of a realistic, representative scale. This requires a level of implementation maturity beyond that of prototypes, and in turn, application to real projects that are not throwaway applications. From these studies, new insights into an approach can be gained.

The MoDSE project[4] has been a driving force for the application of tools such as Spoofax in realistic, sizable projects. The WebDSL project, first developed using the Stratego/XT as a batch processor [Visser, 2007], has been a particularly important project, driving research into modular language design [Hemel et al., 2008, 2009; Groenewegen and Visser, 2008, 2011] and eventually static checks in an IDE environment [Kats and Visser, 2010b; Hemel et al., 2011]. The Acoda [Vermolen et al., 2011], mobl [Hemel et al., 2009], and SugarJ projects [Erdweg et al., 2011a,b] were developed primarily using Spoofax. Beyond these projects that use Spoofax as a testbed for research, we attempt to develop communities of users around our tools that work with them and provide feedback to steer further development. Additionally, we apply our tools in education. Spoofax has been used in courses on model-driven software development and compiler construction, where students create a complete compiler and IDE plugin in a single semester. Section 4.6 reports on further experience and other projects that used Spoofax as a testbed for new research and development. This rich application experience resulted in new research directions, following from questions such as: How to support semantic editor services without reimplementing the language semantics in Java? How to get error recovery in the editor? Can we abstract over the manual work needed for writing language definition tests?

1.7 ORIGIN OF CHAPTERS

The core chapters in this thesis are directly based on peer-reviewed publications at conferences or in journals on programming languages and software engineering. As such, each chapter has distinct core contributions, but also

[4] NWO/Jacquard project MoDSE: Model-Driven Software Evolution (MoDSE), 2006–2012, `http://researchr.org/bibliography/modse/`.

contains some redundancy to ensure they are self-contained. This redundancy has not been eliminated to ensure the extended and revised papers can be read independently.

The author of this thesis is the main contributor of all chapters except Chapter 6, being responsible for most of the effort involved in implementing the approach, performing experiments, and writing the text. Chapter 6 is joint work with Maartje de Jonge and Emma Söderberg, incorporating the OOPSLA 2009 paper on error recovery by Kats et al. [2009a], for which the balance of the work was in favor of the thesis author, and the SLE 2009 paper by de Jonge et al. [2009] for which the balance was in favor of De Jonge.

- Chapter 2 is an updated version of the OOPSLA 2008 paper *Mixing Source and Bytecode: A Case for Compilation by Normalization* [Kats et al., 2008a].
- Chapter 3 is a revised and extended version of the SCAM 2010 paper *Encapsulating Software Platform Logic by Aspect-Oriented Programming* [Kats and Visser, 2010a].
- Chapter 4 is an updated version of the OOPSLA 2010 paper *The Spoofax Language Workbench: Rules for Declarative Specification of Languages* [Kats and Visser, 2010b]. This paper won the best student paper award, presented to the best paper that has a (PhD) student as its main author.
- Chapter 5 is an extended version of the CC 2009 paper *Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming* [Kats et al., 2009c].
- Chapter 6 incorporates and extends the OOPSLA 2009 paper on *Providing Rapid Feedback in Generated Modular Language Environments* [Kats et al., 2009a] and the SLE 2009 paper on *Natural and Flexible Error Recovery for Generated Parsers* [de Jonge et al., 2009], and has been submitted to Transactions on Programming Languages and Systems (TOPLAS).
- Chapter 7 is an extended version of the SLE 2010 paper *Interactive Disambiguation of Meta Programs with Concrete Object Syntax* [Kats et al., 2011a].
- Chapter 8 is published as the OOPSLA 2011 paper *Integrated Language Definition Testing: Enabling Test-Driven Language Development* [Kats et al., 2011b].

Mixing Source and Bytecode: A Case for Compilation by Normalization

2

ABSTRACT

Language extensions increase programmer productivity by providing concise, often domain-specific syntax, and support for static verification of correctness, security, and style constraints. Language extensions can often be realized through translation to the base language, supported by preprocessors and extensible compilers. However, various kinds of extensions require further adaptation of a base compiler's internal stages and components, for example to support separate compilation or to make use of low-level primitives of the platform (e.g., jump instructions, unbalanced synchronization, debugging information). To allow for a more loosely coupled approach, we propose an open compiler model based on normalization steps from a high-level language to a subset of it, the core language. We developed such a compiler for a mixed Java and (core) bytecode language, and evaluate its effectiveness for composition mechanisms such as traits, as well as statement-level and expression-level language extensions.

2.1 INTRODUCTION

Programming languages should be designed for growth in order to evolve according to the needs of the user [Steele, 1999]. General-purpose programming languages offer numerous features that make them applicable to a wide range of application domains. However, such languages lack the high-level abstractions required to adequately cope with the increasing complexity of software. Through the introduction of *language extensions*, it is possible to increase the expressivity of a language, by turning programming idioms into linguistic constructs. Language extensions allow for static verification of correctness, security, and style constraints. Language extensions may be *domain-specific* in nature, such as embedded SQL, or may be *general purpose* in nature, such as traits or the enhanced for loop, enumerations, and other features added in Java 5.0.

The mechanisms available for realizing language extensions determine the quality of extension implementations and the effort needed for their construction. If language extensions can be realized with relative ease, then building new abstraction mechanisms can be used in the software development process to incorporate programming idioms and avoid boilerplate code. The quality of a language extension comprises robustness (are error messages reported on the extended language? is code generation complete?), composability (does

the extension combine with other extensions?), and the quality of the generated code. Finally, separate compilation, i.e. binary distribution of compiled, statically verified (library) components, is an important feature to reduce compilation time.

The creation of a language extension implies the reuse of an existing implementation of the base language. Generally, such reuse can be categorized as either black box reuse of a compiler by adding a preprocessor, or white box reuse by means of a deep integration with the compiler implementation.

Language extension by preprocessing A preprocessor transforms a program in an extended language into a program in the base language. Examples of preprocessors include annotation processors such as XDoclet [XDoclet, 2000] and Java's APT [Sun Microsystems, 2004], SQLJ [Melton and Eisenberg, 2000] an embedding of SQL in Java, and StringBorg [Bravenboer et al., 2010], a generic approach for embedding languages such as SQL. As they are employed as a separate tool, rather than requiring integration into a compiler, preprocessors are highly portable. By avoiding direct compiler extension, their implementation only requires basic knowledge of a language's structure and semantics, not of its compiler. Thus, the compiler is considered as a black box; only its published interface — the syntax of the programming language — is used, and the internal architecture of the compiler is hidden. This separation of concerns makes preprocessors well suited for rapid implementation or prototyping of language extensions.

While preprocessors are an attractive, lightweight solution to language extensibility, they are not considered a mature solution for production implementation of languages. Production of a parser that is exactly compatible with the base language is not always a trivial undertaking. The lack of a (complete) source level semantic analyzer results in error messages by the base compiler about code fragments generated by the preprocessor, rather than the source code written by the programmer. Separate compilation is only possible if the compilation units of the extended language align well with those of the base language. This fails in the case of new modularity abstractions. In other words, considering the base compiler as a black box condemns the preprocessor implementer to reimplement the front end of the base compiler.

Modifying an existing compiler To avoid reimplementation efforts, language extensions are often implemented by extension of the front end of a compiler. This level of integration ensures that existing compiler components, such as a parser and semantic analysis, can be reused in the extended compiler. By generating code in the front end language, it can then be further compiled using the base compiler. Traditional monolithic compilers are typically not designed for extensibility, and adding a new feature may require extensive refactoring of the implementation. Since such refactorings are not incorporated upstream, this effort needs to be repeated with each release of the compiler. Extensible compilers, such as Polyglot [Nystrom et al., 2003], ableJ [Van Wyk et al., 2007], and the JastAdd extensible Java Compiler [Ekman and Hedin, 2007], are designed for extensibility with the principle that

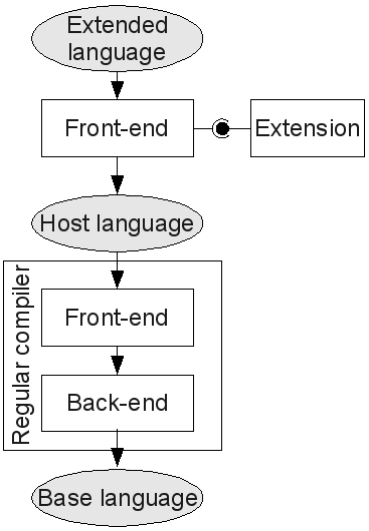


Figure 2.1 The front end extension pattern, applied by many conventional extensible compilers.

the implementation effort should be proportional to the size of the language extension. This *front end extension pattern* is illustrated in Figure 2.1. However, even these systems do rely on white box extension, by exposing their internal structure.

Extending a compiler purely by transformation to the base language is sometimes inadequate. Compilers typically consist of multiple stages, parsing and gathering semantic information in the early stages (the *front end*), and generating and optimizing code in later stages (the *back end*) [Aho et al., 2006]. This strict separation of stages is imposed to ensure straightforward modularization and reuse of compiler components. As such, some compilers for different languages share a common intermediate language and associated back end. Only the final stage or stages of a compiler back end actually output the target-machine code.

Integration into the *back end* makes it possible to also manipulate compiled code. This can be necessary to output specific instruction sequences or clauses not otherwise generated by the base compiler. For example, for Java, the front end does not expose a 'goto' operation, unbalanced synchronization primitives, or means of including debugging information. The back end of a compiler also forms an essential participant in composition of source and compiled code. Consider language extensions aimed at modularity, such as *traits* [Ducasse et al., 2006] and *aspects* [Kiczales et al., 1997]. To support separate compilation, such extensions require integration into a compiler's back end. Separate compilation enables distribution of modules in compiled form, or weaving of code into compiled classes. Using a classic, multi-staged compiler, implementing such extensions is a demanding task that requires

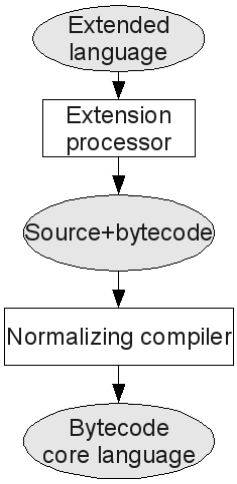


Figure 2.2 Modular extension of a normalizing compiler.

in-depth understanding of the compiler. Extensions that span across multiple compilation stages get tangled throughout the different components of a compiler, and create a large dependency on its implementation.

Mixing source and bytecode Summarizing, the decision to extend a compiler using a simple, front end based approach, or a more deeply integrated approach, comes down to a choice between black box or white box reuse of a base compiler, each with their own drawbacks. Rather than dismissing preprocessors, we want to embrace their simplicity and modularity and propose a compiler architecture that deals with their flaws.

In this chapter, we propose an open compiler architecture based on *mixing source and bytecode* in order to enable *compilation by normalization*. That is, the base language is a combination of the high-level source language (Java) and the low-level bytecode core language. This means that it is possible to use bytecode primitives, such as the *goto* instruction, directly from Java, as in the statement

```
if (condition) `goto label;
```

where the backtick (`) operator is used to distinguish between the syntax of the two constituent languages. Similarly, source code expressions and statements can be embedded in bytecode, nested to arbitrary depth. The compiler for this mixed base language *normalizes* an input program in the combined language to a program in the bytecode core language. Thus, the language produced as output of the compiler is a subset of the input language.

This architecture combines the light weight construction of extensions using a preprocessor, with the access to the compiler back end of a compiler extension. Without knowing the internals of the compiler, a preprocessor can generate code in the core language where needed, as well as the extended input language where possible (see Figure 2.2). In other words, an extension

preprocessor extends the base compiler by adding transformations from the extended language into a mix of low-level and high-level code, as is convenient for the definition of the particular extension. The idea of transformation-based compilers is not new. In particular, Peyton Jones and Santos [1998] applied the idea in the implementation of the Glasgow Haskell Compiler GHC in which compiler optimizations are formulated as transformations on core language constructs. The front end of the compiler consists of type analysis and simple desugarings. However, in GHC the core language is not a subset of the compiler's input language. As a result it is not possible to feed the output of the compiler back into the compiler. In our approach the *complete compiler* can be used as a normalization tool, which allows the construction of pipelines of preprocessors, and the implementation of separate compilation for new abstraction mechanisms. By keeping track of origin information when generating code, error messages produced later in the chain can refer to the original source input.

To evaluate the notion of compilation by normalization, we have created a prototype implementation of a Java compiler based on this principle. This prototype, the *Dryad Compiler*, unifies the Java language with the underlying bytecode instruction language. Extending a regular Java typechecker, we provide full typechecking for this combined language. Using the collected type information, we introduce *overloaded instructions* to the bytecode language that can be normalized to regular instructions, and facilitate code generation.

The contributions of this chapter are as follows:

- The introduction of a compiler architecture based fully on normalization steps rather than multiple, independent stages.
- An implementation of this compiler architecture for a major programming language.[1]
- Case studies that show how the approach can be used for compiler extensibility.
- A compact, polymorphic set of core bytecode instructions.

Outline We proceed as follows. In Section 2.2 we discuss the design of a number of language extensions, evaluating how they benefit from compilation by normalization when implemented as an extension of the Dryad Compiler. We describe the syntax and semantics of the mixed Java/bytecode language and the architecture of its compiler in Section 2.3. In Section 2.4 we discuss how normalization rules incrementally transform the Java/bytecode language and its extensions to the core language. In Section 2.5 we offer a discussion of the architecture of the Dryad compiler, and compilation by normalization in general. We present related and future work in Section 2.6, and finally conclude in Section 2.7.

[1]The Dryad Compiler, an open source project available from `http://strategoxt.org/Stratego/TheDryadCompiler/`.

2.2 EXTENSIONS BASED ON THE DRYAD COMPILER

In this section, we discuss a number of compiler extensions for the Dryad Compiler. We first discuss extension at the *class level*, with partial and open classes in Section 2.2.1 and traits in Section 2.2.2. In Section 2.2.3 we discuss how the principle of compilation by normalization can be applied at the *statement level* for iterators, and in Section 2.2.4 we show how it can benefit the implementation of *expression-level* extensions.

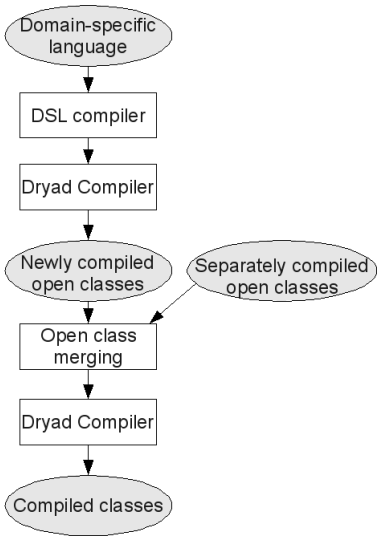
2.2.1 *Extension with Partial and Open Classes*

In Java, classes are defined in a single source file. *Partial classes* enable the distribution of class members over multiple source files. Partial classes can be used for separation of concerns, for example dividing GUI-related and event-handling code, or as a way of modifying existing classes included in a third-party application or library. Another application of partial classes is merging generated code fragments, such as code generated from partial models [Warmer and Kleppe, 2006] or from modular transformations generating code for specific aspects. Partial classes can be implemented relatively easily as an extension of an existing compiler, by merging them to regular classes.

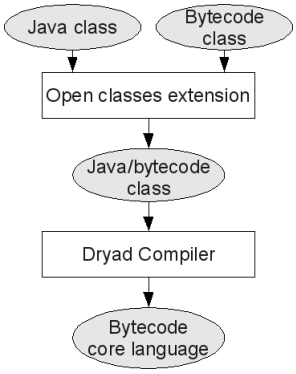
Open classes extend the notion of partial classes by allowing the extension of compiled classes, rather than just classes in source code form. One implementation of open classes for Java is provided by MultiJava [Clifton et al., 2000, 2006]. MultiJava is derived from the Kopi Java Compiler, a fairly small open-source implementation written in Java itself. This makes it a relatively accessible candidate for such an extension. MultiJava uses different stages that add to the existing stages of the Kopi compiler (i.e., parsing, importing definitions, different typechecking stages, grouping of multi-methods, bytecode generation). This required a significant part of the development effort to be spent in understanding the base compiler. Clifton notes that this was in fact the case, attributing this to a lack of documentation on Kopi [Clifton et al., 2000].

Existing implementations of partial and open classes only allow the definition of classes that augment others in source form, merging them into either source or compiled classes. A third scenario, merging together multiple classes in compiled form, is not supported. This scenario can be applied to use open classes for supporting separate compilation in compilers that target the Java platform. Consider Figure 2.3a, where open classes facilitate merging of newly and previously compiled class fragments. In this architecture, a compiler for a domain-specific language (DSL), for example producing GUI-related code, is implemented as an extension of the Dryad Compiler. The code it produces is processed by another extension, which merges the open classes for final compilation.

The mixed source/bytecode language allows us to think of source and bytecode classes as instances of a single language; there is no fundamental difference in the merge process required for them. Figure 2.3b shows the



(a) Using open classes for incremental compilation.



(b) Extending the Dryad Compiler to support open classes.

Figure 2.3 Open classes.

architecture of the open classes extension. It merges together members of class fragments, either in source or bytecode form. The resulting Java/bytecode class (see Figure 2.4) is passed to the Dryad Compiler, which provides the support for compilation of these composed fragments. This design allows a straightforward implementation of the extension, no longer requiring implementation-level knowledge of the open compiler it builds on. Compilation of the merged classes is handled by the Dryad compiler; the extension only provides the composition mechanism. Using the technique of source tracing, on which we elaborate in Section 2.3.4, it maintains location information relative to the original source files for compile-time errors and debugging information.

2.2.2 *Extension with Traits*

Traits are primitive units of reusable code that define a set of methods that can be imported into regular classes. Using a set of operators, traits can be composed and manipulated. For instance, the `with` operator composes two or more traits, and is also used to extend a class with the methods defined in a trait (see Figure 2.5). Traits were originally introduced by Schärli et al. in the context of Smalltalk [Ducasse et al., 2006]. They have since been ported to statically typed languages, such as Java, C#, and Scala. To support traits with separate compilation, they must explicitly specify their *required methods*, i.e. all methods that are referenced but not provided by a trait.

```
class Calculator {
  // From Calculator_Operations.java
  public void add() {
    operation = new Add(getDisplay());
    ⋮
  }

  // From Calculator_Gui.class
  `private setDisplay(int number : void) [
    iload number;
    ⋮
  ]
  ⋮
}
```

Figure 2.4 Open classes merged to a single Java/bytecode definition.

```
class Shape with TDrawing {
  Vertices getVertices() { ... }
  ⋮
}

trait TDrawing {
  void draw() { ... }

  require Vertices getVertices();
}
```

Figure 2.5 Example of a class `Shape` importing a trait `TDrawing`.

To the best of our knowledge, only Scala – which supports the feature natively rather than in the form of an extension – supports separate compilation of traits [Odersky et al., 2008]. This allows for statically verified, binary distribution of traits in libraries, but requires a significantly different compilation model than the source-to-source transformation commonly applied by implementations of traits.

To enable separate compilation to binary class files in our traits extension, we designed a compilation scheme translating traits to regular, abstract classes. This way, required methods can be mapped to abstract methods. Although we have no intention for the Java Virtual Machine (JVM) to load these classes at run-time – traits are merely compile-time entities – this mapping enables us to use the base compiler's verification and compilation facilities for abstract classes.

After traits are compiled, the resulting class files can be composed according to the composition operators. For the `with` operator this means that a trait's methods are added to a client class. Similarly, the `minus` operator removes methods from a trait. The `rename` operator renames a method declaration and all occurrences of it in the core language invocation constructs. Unlike in Java, the names used in these constructs are fully qualified and unambiguous, making this a straightforward operation. The composition operations are followed by a basic consistency check that confirms that all

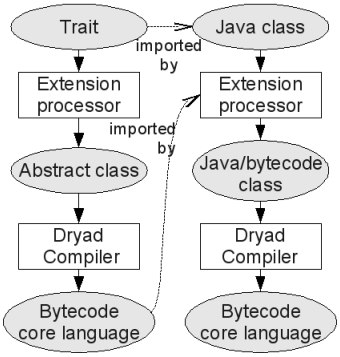


Figure 2.6 Separate compilation of a trait (left) and a class importing it (right).

required methods are defined and that there are no duplicate definitions. More extensive typechecking is performed by the base compiler. Consider Figure 2.6, which illustrates the architecture of this extension. By leveraging the Java/bytecode language for inclusion of compiled code, the extension remains loosely coupled from the base compiler. Our implementation of traits currently spans 104 lines of code[2], making it a relatively lightweight implementation. Still, it includes support for separate compilation, consistency checking, and source tracing. As it does not require implementation-level knowledge of the base compiler, but only of its input language, the focus shifts to these issues rather than on direct adaptation of the existing compilation process.

2.2.3 *Extension with Iterator Generators*

Java 5.0 introduced the *enhanced for loop*, a language feature that allows programmers to iterate over collections. The Java compiler treats this as a form of syntactic sugar, and translates it to a regular `while` loop that uses the `java.lang.Iterable` and `java.util.Iterator` interfaces (see Figure 2.7). As such, the enhanced for loop can be used to conveniently iterate over any type that implements the `Iterable` interface.

Implementing the iterator interfaces is somewhat involved and follows a common pattern with considerable boilerplate code. A complementary language extension that deals with this problem is that of *iterator generators*, a feature supported by other languages such as Python and Sather [Murer et al., 1996]. Like the enhanced for loop, this feature abstracts away from the underlying interfaces. Consider Figure 2.8, which uses it to define an iterator method that splits a string at every space character. It loops over the results of a call to `String.split()`, and uses the `yield` statement to return all sub-

[2]Not included is the syntax definition. Implemented as a stand-alone program, this figure does include 47 lines of I/O code, imports, and comments.

```
for (String s : getShortWords("foo bar")) {
  System.out.println(s);
}
```

```
Iterator<String> it = getShortWords("foo bar").iterator();
while (it.hasNext()) {
  String s = it.next();
  System.out.println(s);
}
```

Figure 2.7 The enhanced for loop (top) and its desugared form (bottom).

strings with less than four characters as elements of the resulting iterator. The iterator method operates as a coroutine: control is "yielded" back to the client of the iterator at every `yield` statement. When the client requests the next element, the iterator method is resumed after the last `yield` statement.

In earlier work, we implemented the `yield` statement as a source-to-source transformation, abstracting over regular Java control flow statements.[3] The `yield` statement introduces a form of *unstructured control flow*: the method can be entered at any arbitrary point it is used. To express this in a legal Java program, the desired control-flow graph must be transformed to make use of Java's *structured* control flow statements, a non-trivial problem, also faced when decompiling bytecode [Miecznikowski and Hendren, 2001]. We accommodated for this by using a `switch` statement and `case` labels at every point in the control flow graph. Since the Java `switch` statement can only be used in a structured fashion (i.e., it disallows `case` labels inside blocks of code nested in it), all existing control flow statements in the method must be rewritten to become part of the `switch`. This turned out to require significant effort, essentially re-implementing a considerable part of the existing Java language.

The infomancers-collections library [Dov, 2008] aims to avoid the complications of source-to-source transformation. It effectively hides the language extension from the Java compiler, by using a dummy `yield()` method that can be invoked from anonymous classes that implement the iterator interfaces. A regular Java compiler can then be used to compile the code, unaware of the special semantics of these invocations. The resulting bytecode is then altered by the library, replacing the dummy invocations with an actual implementation, modifying the (unstructured) control flow of the method. This is a rather intricate process that requires the use of a bytecode manipulation library to generate a new class. Since the Java compiler is oblivious to the special semantics of the dummy invocations, it is unable to perform proper control flow analysis during the initial compilation, which may lead to unexpected results. In particular, compilers may optimize or otherwise generate code that violates the stack-neutrality property of statements (see Section 2.3.3), which can result in invalid programs after inserting new jump instructions.

In our approach, we treat the language extension as a form of syntactic sugar that can be projected to the base language (i.e., Java/bytecode), just like

[3] As part of the java-csharp project, available from `http://strategoxt.org/Stratego/JavaCSharp/`.

```
public Iterable<String> getShortWords(String t) {
  String[] parts = t.split(" ");
  for (int i = 0; i < parts.length; i++) {
    if (parts[i].length() < 4) {
      yield parts[i];
    }
  }
}
```

Figure 2.8 Iterator definition with the `yield` statement.

```
class ShortWords implements Iterator<String> {
  int _state = 0;
  String _value;
  boolean _valueReady;

  String _t;
  String[] _parts;
  int _i;

  private void prepareNext() {
    if (_valueReady || _state == 2) return;
    if (_state == 1) `goto afterYield;

    _parts = _t.split(" ");
    for (_i = 0; _i < _parts.length; _i++) {
      if (_parts[_i].length() < 4) {
        _state = 1;
        _valueReady = true;
        _value = _parts[_i];
        return; // yield value
afterYield:
      }
    }
    _state = 2;
  }

  public String next() {
    prepareNext();
    if (!_valueReady)
      throw new NoSuchElementException();
    _valueReady = false;
    return _value;
  }

  public boolean hasNext() {
    if (!_valueReady) prepareNext();
    return _valueReady;
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

Figure 2.9 Iterator definition, generated from Figure 2.8.

the enhanced for loop. For this, the `yield` statement is rewritten to a `return` statement to exit the iterator method, in conjunction with a small, bytecode-based jump table at the beginning of the method for re-entry. Based on a finite state machine model of the method, a field `_state` is maintained to indicate the next method entry point. Consider Figure 2.9, which shows the class that will be generated local to the original `getShortWords()` method. It highlights all lines of code that directly correspond to the original method. The `prepareNext()` method is central to this implementation, and includes an adapted version of the original method body. Its control flow is adapted by the addition of unstructured control flow primitives, in the form of bytecode instructions. In this example, we implement this using a jump table and `goto` instructions, embedded in Java using the backtick (`) operator. Alternatively, a `tableswitch` instruction could be used to form an unstructured switch statement.

It is not possible to statically determine if the iterator will return a value or not for a given state, and we cannot make assumptions on the order of invocation of the `Iterator.next()` and `Iterator.hasNext()` interface methods. Therefore, a call to `prepareNext()` is used in both these methods, caching the value as necessary. To ensure persistence of local variables during the lifetime of the iterator, all local variables must be replaced or shadowed by fields.

The amount of code required to define a custom iterator by implementing the iterator interfaces (Figure 2.9) illustrates the increase in productivity that can be gained by abstracting over this using iterator generators (Figure 2.8). The projection to Java/bytecode that realizes this abstraction is relatively straightforward, as it maintains the structure of the original method, unlike in our earlier source-to-source approach. On the other hand, this approach also avoids the complexity of the low-level, purely bytecode-oriented approach, eliminating the need for special libraries and using the convenience and familiarity of the Java language for most part of the implementation.

2.2.4 *Assimilating Expression-Level Extensions*

Embedded domain-specific languages add to the expressivity of general-purpose languages, combining domain-specific constructs with the general-purpose expressivity of the host language. Examples include embedded database queries, or integrated regular expressions. We have explored DSL embedding with Java as a base language and described MetaBorg [Bravenboer and Visser, 2004], a general approach for DSL embeddings. In that context, we have coined the term *assimilation* for the transformation that melds the embedding with its host code. Assimilation preserves the semantics of language extension constructs, while making it possible to compile them using the base language compiler.

Small, statement- or expression-level language extensions are especially well-suited for assimilation. They can often be assimilated locally to an implementation in the host language (often API-calls), without disturbing the surrounding code. However, for specific kinds of language extensions this is

```
System.out.println(e1 ?? e2);
```

(a) The ?? operator, with operands of type *T*.

```
System.out.println(
  {| T lifted = e1; // evaluate e1 only once
     if (lifted == null) lifted = e2;
   | lifted
   |});
```

(b) In-line assimilation to an expression block.

```
T lifted = e1;
if (lifted == null) lifted = e2;
System.out.println(lifted);
```

(c) Lifted to pure Java.

Figure 2.10 Assimilation using statement-lifting.

not possible. One class of such extensions is that of language extensions that take the form of *expressions* but require assimilation to *statements* in the host language (e.g., to use control flow constructs or declare local variables, which is not possible in expressions). In-place assimilation to an expression does not suffice in these cases, because Java and similar languages do not allow nesting of statements in expressions. One technique to overcome this problem is using an intermediate syntax in the form of *expression blocks* [Bravenboer and Visser, 2004]. These enable the use of statements in expressions, facilitating in-line expression assimilation.

Expression blocks take the form

```
{| statements | expression |}
```

where the statements are executed before the evaluation of the expression, and the value of the expression block is the value of the embedded expression. A separate, generally applicable normalization step in the form of *statement-lifting* can be used to *lift* the statement component of the expression to the statement level [Bravenboer and Visser, 2004]. Consider for example the C# coalescing operator:

```
e1 ?? e2
```

This operator returns the value of expression *e1* if it is non-null, or otherwise the value of *e2*. It could be used in conjunction with Java's nullable (or "boxed") types and its support for auto-unboxing, providing a default value when converting to a non-nullable (i.e., primitive) type.

Consider Figure 2.10, which shows how the coalescing operator can be assimilated to regular Java statements. In Figure 2.10a, the operator is used in a method call. Using an expression block, it is straightforward to translate it in-line to regular Java statements, as seen in Figure 2.10b. Finally, in Figure 2.10c, the expression block is lifted to the statement level.

Unfortunately, proper statement-lifting is not trivial to implement: it requires *syntactic knowledge* of every language construct that an expression may

```
Iterator<Integer> it = ...;

for (int i = i0; it.hasNext(); i = it.next() ?? 0) {
   ...
}
```

(a) The coalescing operator in a loop.

```
Iterator<Integer> it = ...;

Integer lifted = next();
if (lifted == null) lifted = 0;
for (int i = i0; it.hasNext(); i = lifted) {
   ...
}
```

(b) Incorrect program after statement-lifting.

Figure 2.11 Statement-lifting in a `for` loop.

```
System.out.println(`[
  push `e1;
  astore lifted;
  ifnull else;
    push `e2;
    goto endif;
  else:
    aload lifted;
  endif:
]);
```

Figure 2.12 Assimilation of the `??` operator to bytecode.

appear in, including various statements, other expressions, and possibly other language extensions. Furthermore, it requires *semantic knowledge* of the language, as it is not always sufficient to simply move statements to the syntactic statement level. For instance, the simple lifting pattern from [Bravenboer and Visser, 2004] cannot be applied if the expression is the operand of short-circuiting operators (e.g., `||` in Java) or the conditional of a `for` loop (see Figure 2.11). In these cases, simply lifting the translated statements to the statement level changes the semantics of the program.

In the bytecode core language, there is no statement level or expression level. This distinction only exists in Java source code, and is simply a consequence of the Java syntax. Thus, we can overcome this limitation by assimilating the operator directly to the bytecode core language, using instructions in place of the original operator (see Figure 2.12).

Given that the core language is enriched with constructs of the more convenient Java language, we can also apply the complete, mixed language to synergistic effect and assimilate the operator in a more elegant fashion. Consider Figure 2.13, which assimilates the coalescing operator to Java statements, embedded in a bytecode block. In addition to these statements, we use a `push` pseudo-instruction to place the `lifted` variable on the stack (we elaborate on the role of the stack and the `push` instruction in Section 2.3.1). This value forms the result of the expression, and is used as the argument of the call

```
System.out.println(`[
  `T lifted = e1;
  `if (lifted == null) lifted = e2;
  push `lifted;
]);
```

Figure 2.13 Assimilation to bytecode-embedded Java.

to `System.out.println`. Like expression blocks, the bytecode fragment can contain any number of statements and a single resulting expression, making normalization of the expression block extension to a bytecode block trivial. As such, this pattern effectively does away with the complications associated with statement-lifting, and thereby simplifies the implementation of expression-level language extensions using statements.

2.3 REALIZATION OF THE BASE COMPILER

The Dryad Compiler operates by normalization of the mixed Java/bytecode language to its core (bytecode) language. We implemented a parser for the language as an extension of the regular Java language, using the modular syntax definition formalism SDF [van den Brand et al., 2002]. The language is normalized through normalization rules expressed in the Stratego [Bravenboer et al., 2008] program transformation language. In the remainder of this section, we give an overview of the design of the language and its compiler.

2.3.1 *Language Design*

Key to compilation by normalization is the notion of a core language that is – often incrementally – extended with new abstractions to form a rich, high-level language. For the Dryad compiler, we build upon the existing bytecode language, an assembly-like core language, and mix it with the standard Java language [Gosling et al., 2005]. The two syntax forms are integrated using the *backtick operator* `, which toggles the syntax form of a fragment of code. Figure 2.14 gives an overview of the syntax of the language. In this figure we use italics to refer to other symbols in the syntax, and an overline to indicate lists of symbols (e.g., $\overline{T\ x}$ indicates a list of type/identifier pairs). For brevity, we left out some of the more advanced Java language constructs. For mixing of class, method, and field declarations, the ` notation is optional, and was also left out from this overview.

The bytecode assembly language we use shares similarities with existing languages such as Jasmin [Meyer and Downing, 1997] and its derivatives. It provides a somewhat abstracted representation of the underlying binary class format. For instance, it allows the use of symbolic names rather than relative and absolute *offsets* for locals and jump locations. Like Jasmin, our representation allows constants to be used in-line, eliminating the need to maintain a separate constant pool with all constants and types used in a class. Still, our representation remains close to the underlying binary bytecode instructions.

General

p	::= $\overline{cd}$	Program (start symbol)
T	::= C | void | int | long | double	Types
	| float | boolean | char | byte	

Java

cd	::= class C extends C { $\overline{fd}$ $\overline{md}$ $\overline{cd}$ }	Class declaration
md	::= C ($\overline{T\ x}$) { super ($\overline{e}$); $\overline{s}$ } | T m ($\overline{T\ x}$) { $\overline{s}$ }	Method/constructor decl.
fd	::= $T\ f$;	Field declaration
s	::= { $\overline{s}$ }	Statement block
	| e;	Expression statement
	| $C\ x = e$;	Local variable declaration
	| if (e) s else s | while (e) s | for (e; e; e) s	Control flow
	| return; | return e;	Return statement
	| throw e;	Throw exception
	| synchronized (e) { $\overline{s}$ }	Synchronization
	| try { $\overline{s}$ }catch($C\ x$) { $\overline{s}$ }	Exception handling
	| l:	Label
e	::= $x = e$ | x	Local variables
	| (T) e	Cast
	| e + e | e - e	Basic operators
	| $e.m$ ($\overline{e}$)	Method invocation
	| new C ($\overline{e}$)	Object creation

Bytecode

cd	::= classfile C extends C fields $\overline{f}$ methods $\overline{md}$	Class declaration
md	::= C ($\overline{T\ x}$: T) [$\overline{I}$] | <init> ($\overline{T\ x}$: C) [$\overline{I}$]	Method/constructor decl.
fd	::= f : T	Field declaration
I	::= catch ($\overline{l : C}$) [$\overline{I}$]	Exception handling
	| *(see Figure 2.15)*	Bytecode instruction

Mixing

s	::= `$I$ | `[$\overline{I}$]	Bytecode statement
e	::= `$I$ | `[$\overline{I}$]	Bytecode expression
I	::= `$s$ | `[$\overline{s}$] | push `e	Embedded Java

Tracing

s	::= trace (o) [$\overline{s}$]	Statement trace
e	::= trace (o) [e]	Expression trace
I	::= trace (o) [$\overline{I}$]	Instruction trace
md	::= trace (o) [$\overline{md}$]	Method trace
o	::= $\overline{s}$ @ loc | e @ loc | $\overline{I}$ @ loc | $\overline{md}$ @ loc	Trace originating code
loc	::= $path$:i:i	Location specification

Figure 2.14 Syntax for the mixed Java/bytecode language.

Most notably, it preserves the *stack machine*-based nature of the instruction set. More abstract representations are for example provided by Soot [Vallée-Rai et al., 1999], but this does not match our design goal of exposing the complete core language functionality. Instead, we embrace the operand stack to provide low-level operations, and use it for interoperability between the two languages. Figure 2.14 shows the basic elements of the bytecode language, while Figure 2.15 gives an overview of the instruction set. Note that this figure shows a reduced bytecode instruction set, using overloaded instructions, on which we elaborate in Section 2.4.2. For compatibility, we also support the full standard set of instructions, which can be mapped to the reduced set.

Arithmetic	**Stack**	**Control flow**	**Arrays**
add	ldc *c*	ifeq *l*	aload
div	ldc2_w *c*	ifne *l*	astore
mul	new *C*	goto *l*	arraylength
neg	pop	*l*:	newarray *T*
rem	pop2	athrow	multianewarray *T n*
sub	dup	return	
shl	dup_x1	xreturn	**Fields**
shr	dup_x2	lookupswitch $\overline{n : l}$ default: *l*	getstatic $C.f : T$
ushr	dup2	tableswitch *n* to *n*: $\bar{l}$ default: *l*	putstatic $C.f : T$
xor	dup2_x1		getfield $C.f : T$
and	dup2_x2	**Invocations**	putfield $C.f : T$
or	swap	invokevirtual $C.m(\overline{T} : T)$	
inc		invokestatic $C.m(\overline{T} : T)$	**Miscellaneous**
dec	**Conversions**	invokestatic $C.m(\overline{T} : T)$	instanceof *C*
	x2i	invokeinterface $C.m(\overline{T} : T)$	monitorenter
Comparison	x2l	invokespecial $C.m(\overline{T} : T)$	monitorexit
lt	x2d		breakpoint
gt	x2f		nop
le	i2b		
ge	i2s		
eq	i2c		
ne	checkcast *C*		

Figure 2.15 The (reduced) bytecode instruction set.

Interoperability between Java and Bytecode Java and bytecode have many of the same basic units of code, e.g. classes, methods, and fields. In our mixed language, the Java and bytecode representations of these units can be combined arbitrarily. How to combine the two languages at a finer level of granularity, i.e. inside method bodies, is less obvious. In Java, a method body is a tree-like structure of statements that may contain leafs of expression trees. Bytecode methods consist of flat, scopeless lists of instructions. Perhaps the most elementary form of mixing at this level is to allow bytecode fragments in place of statements, forming *bytecode statements*, and vice versa. This – among other things – allows statement-level separate compilation and basic insertion of source code into compiled methods. At the statement level, state is maintained through local variables and fields. These can be shared between Java and bytecode statements, as the language uses a common, symbolic representation for both fields and local variables.

Expressions can be used as the operands of statements or other expressions, passing a value to the enclosing construct. *Bytecode expressions* are fragments of bytecode that can be mixed into expressions. They are conceptually similar to bytecode statements and share the same syntax for embedding. However, as expressions, they must produce a resulting value. At the bytecode level, such values are exchanged using the *operand stack*. For example, the *load constant* instruction (`ldc`) pushes a value onto the stack, and the `add` instruction consumes two values and pushes the addition onto the stack. Such instructions can be used to form a legal bytecode expression:

```
int i = `[ ldc 1; ldc 2; add ];
```

```
void locals() {
  { // Java blocks introduce a new  scope
    `[ ldc 1; store var ];
    System.out.println(var);
  }

  // 'var' is out of scope and can be redefined
  int var = 2;
}
```

Figure 2.16 Local variable scoping.

Vice versa, the `push` pseudo-instruction places the value of a Java expression onto the stack:

```
push `"Java" + "expression";
```

2.3.2 *Name Management and Hygiene*

Local variables shared between Java and bytecode follow the standard Java scoping rules. Bytecode has no explicit notion of declaration of variables, only of assignment (using the `store` instruction). In the mixed language, the first assignment of a local variable is treated as its declaration, and determines its scope (see Figure 2.16). In regular bytecode there exists no notion of local variable scoping; all scopes are lost in the normalization process. To ensure proper hygiene during normalization, this means that all local variable identifiers – both intermediate and user-defined – need to be substituted by a name that is unique in the scope of the entire method. For the example in Figure 2.16, two unique variables can be identified in separate scopes. After normalization, these variables get a different name.

2.3.3 *Typechecking and Verification*

Typechecking is an essential part of the compilation of a statically typed language. The Java Language Specification specifies exactly how to perform name analysis and typechecking of the complete Java language [Gosling et al., 2005]. The analyses provide type information required for the compilation (e.g., for overloading resolution) and give feedback to the programmer in case of errors. We employ a Stratego-based typechecker for Java, which is implemented as a tree traversal that adds semantic information to the abstract syntax tree.

The typechecker for the mixture of Java source and bytecode is a modular extension of a typechecker for Java source code. The source code typechecker is designed to handle language extensions by accepting a function parameter that is invoked for extensions of the Java source language. The extension can inspect or modify the environment of the typechecker, recursively apply the typechecker, or completely take over the traversal of the abstract syntax tree. For the mixed Java/bytecode language, the extended language constructs are the mixing constructs of Figure 2.14, where bytecode is embedded in Java.

If any of these constructs are encountered, the extension of the typechecker takes over the traversal by switching to the bytecode verifier. The current typechecker environment is passed to the verifier, ensuring all variables and other identifiers are shared with the surrounding code. The verifier in turn returns the resulting operand stack of the bytecode fragment. For bytecode expressions, these must consist of a single type, which is passed back to the Java typechecker. Vice versa, the bytecode verifier invokes the source code typechecker for any embedded Java constructs, using it to resolve the types of embedded Java expressions and variables declared in Java statements.

Stack-Consistency of mixing constructs The bytecode verifier ensures correct stack behavior and type safety of a program. In the mixture of Java and bytecode, we impose specific rules on the stack behavior to ensure safe interoperability and natural composability between Java and bytecode fragments. These are verified by the bytecode verifier.

One restriction we impose on bytecode expressions is that they must leave a single value on the stack, just like Java expressions.[4] Leaving no value on the stack is considered an error. Likewise, leaving more than one value on the stack is considered illegal as this can lead to a stack overflow when it is not properly cleaned up.

Unlike Java expressions, statements do not leave a value on the stack; they compile to a sequence of *stack-neutral* bytecode instructions. That is, they may use the stack for intermediate values during evaluation, but must restore it to its original height afterwards. This ensures that all Java statements can safely be placed in a loop construct, without the risk of a stack overflow or underflow. Even more so, the JVM actually *requires* that methods have a fixed maximum stack height, disallowing loops with a variable stack height [Lindholm and Yellin, 1999]. For compound statements (e.g., `for`, `while`), stack-neutrality extends to the contained statements: the input stack of all nested statements must be the same as the stack outside the containing statement. This restriction goes hand in hand with the JVM restriction that at any point in a method, the types of the values on the stack must statically be known, and must be the same for all incoming edges of the control flow graph. Any other jumps to a location are considered illegal. The restriction can always be satisfied on the statement level based on the property of stack-neutrality. A jump from one statement to another is therefore always legal. To preserve this property in the mixed language, we place the same restriction of stack-neutrality on bytecode statements: only bytecode sequences that restore the stack height are legal bytecode statements. This ensures that fragments of Java and bytecode can be composed naturally without risk of stack inconsistencies, and ensures support for arbitrary control flow between Java and bytecode statements.

Verifier implementation The JVM specification [Lindholm and Yellin, 1999] includes a description of how a bytecode verifier should operate. It describes

[4] Actually, `void` method invocations are an exception to this, but these cannot be nested into other expressions.

the process as a fix-point iteration, where a method's instructions are iterated over a number of times until all possible execution paths have been analyzed. Because of restrictions on the form of a method and its instructions, this is a straightforward process. One restriction is that for all instructions the effect on the stack can be statically determined. For instance, for a method invocation instruction, this means that it must specify the arguments it takes from the stack and what the return type is. This also means that the verification can be done in an intraprocedural setting, using a static, global environment. This allows it to be tightly integrated into the Java source code typechecker, as it can be used to verify individual fragments at a time.

We implement our analysis using a monotone framework [Kam and Ullman, 1977; Aho et al., 2006]. This representation allows a generic formulation of such analyses using a specific set of operators and constants for each framework instance. Bytecode verification is a forward data-flow analysis, and assumes an empty stack at the beginning of a method. The operators that determine the types on the stack through fix-point iteration are defined as follows:

- *The transfer function* determines the resulting stack of an instruction, given the input stack. For instance, for an `ldc` instruction, a single value is loaded onto the stack.

- *The join operation* merges the stack at branch targets (i.e., labels and exception handlers), unifying the types of the stack states if they are compatible (e.g., `String` and `Integer` unify to the `Object` type).

2.3.4 *Source Tracing*

During compilation, code often undergoes multiple normalization steps. If there are any errors in any of these steps, they should reflect the originating source code, and not the intermediate code. To maintain this information, we introduce *source tracing* information that explicitly indicates the source of a fragment of code, in the form of a path and location of the originating file and an abstract syntax tree reference. Source tracing is exposed as a language feature, using the `trace` keyword (see Figure 2.14). This ensures maximal, source-level interoperability with other tools that may exist in the compilation chain. Consider Figure 2.17, which shows a class file compiled using the traits extension. In addition to language-level support, we provide an API to transparently include this information in the result of normalization rules in Stratego. Using this facility, extensions of the compiler can fall back on the error reporting and checking mechanisms already provided by the base compiler, and may catch these errors to improve the user experience.

Source tracing information is also used to generate debugging information in the produced Java class file. This takes the form of a table that maps instruction offsets to original source file locations, and enables stepping through the code using a debugger, as well as accurate position information in run-time exceptions.

```
classfile Shape
methods
  trace (void draw() {...} @ TDrawing.java:2:2) [
    public draw(void) [ ... ]
  ]
  ⋮
  trace (... getVertices() {...} @ Shape.java:9:2) [
    public getVertices(void) [ ... ]
  ]
```

Figure 2.17 Compiled methods with source tracing.

2.3.5 *Data-Flow Analysis on the Core Language*

Leveraging the bytecode verifier and the source tracing facility, we implemented analyses such as unreachable code analysis and checking for definite assignment at the bytecode level. By performing these analyses on the core language, we can formulate them as a monotone framework instance, or make use of the information already provided by the verifier. Furthermore, dealing with a normalized form of the language, these analyses have fewer constructs to deal with, reducing their complexity.

Reachability of statements can be determined by use of the regular bytecode verifier: it returns a list of stack states, leaving the stack states of all unreachable instructions uninitialized. Using source tracing information, the Java statements that generated such code (which is illegal in Java) can be reported as unreachable. For testing definite assignment, we formulated another monotone framework instance that maintains a list of all assigned variables, matching against the `load` and `store` instructions. Through iteration it determines whether or not variables are assigned before use, and if variables marked `final` are not assigned more than once. While the core language has no direct notion of `final` variables, this information can be retrieved using the method's source tracing information.

In addition to verification, we applied these analysis techniques for optimizations on the core language, including dead code elimination and peephole optimizations. Similarly, other optimizations such as constant propagation can be implemented at this level.

2.4 NORMALIZATION RULES FOR CODE GENERATION

Using normalization rules, high-level language constructs can be rewritten to lower-level equivalents, e.g. from constructs of language extensions to the Java/bytecode language, and then to the core bytecode language. We express these rules as Stratego rewrite rules, which take the form

L : $p_1 \rightarrow p_2$ **where** s

where L is the name of the rule, p_1 is the left-hand side pattern to be matched against, and p_2 is the result of the rewrite rule. The `where` clause s may specify additional conditions for the application of the rule, or may declare

```
normalize-finally:
 |[ synchronized (e) { bstm* } ]| →
 |[ Object locked = e;
    try {
      `[ push `locked; monitorenter ];
      bstm*
    } finally {
      `[ push `locked; monitorexit ];
    }
  ]|
```

Figure 2.18 Normalization of the `synchronized` statement.

variables to be used in the result. Using the technique of *concrete syntax embedding*, Stratego rewrite rules may use the concrete syntax of the language being transformed as patterns [Visser, 2002; Bravenboer et al., 2008]. These concrete syntax fragments are parsed at compile-time and converted to equivalent match or build operations using the abstract syntax tree. These patterns are typically enclosed in "semantic braces":

```
normalize-if:
  |[ if (e) s ]| →
  |[ if (e) s else ; ]|
```

this rule normalizes the basic, single-armed `if` statement to the more general two-armed `if` statement, with the empty statement in the `else` clause. This normalization ensures that other rules only have to deal with the latter form.

2.4.1 *Mixed Language Normalization*

Rather than directly normalizing from high-level language constructs to bytecode, this is often done through a series of small normalization steps. Often, these are rules that produce a mixture of Java and bytecode, which is further normalized in later steps. Iterative rule application and leveraging the primitives made available in the mixed Java/bytecode language make it possible to normalize the complete language using relatively small steps that focus on a single aspect.

Consider Figure 2.18, which demonstrates a normalization rule for the standard Java `synchronized` statement. It is rewritten to a mix of more low-level Java statements and the `monitorenter` and `monitorexit` unbalanced synchronization instructions. The resulting fragments can in turn be normalized to core language constructs themselves.

We apply normalization rules in both the core compiler as well as in the language extensions. For example, in Section 2.2.2 we discussed the extension of Java with traits, mapping trait classes to Java abstract classes for separate compilation. Consider Figure 2.19, which illustrates this mapping by means of a normalization rule. This rule makes use of a `where` clause, and depends on two helper functions: `trait-name`, which determines and registers a new name for the resulting abstract class, and `trait-methods`, which determines the set of methods to be included. Given the mapping of traits to regular

```
normalize-trait:
  |[ trait x trait* { method1* } ]| →
  |[ abstract class y { method2* } ]|
  where
    y        := <trait-name> x
    method2* := <trait-methods> (method1*, trait*)
```

Figure 2.19 Normalization of traits to abstract classes.

Java	Reduced instruction set	Regular bytecode
intVar++	load intVar	iload_1
	dup	dup
	inc	iconst_1
		iadd
	store intVar	istore_1
longVar++	load longVar	lload_1
	dup	dup2
	inc	lconst_1
		ladd
	store longVar	lstore_1

Figure 2.20 Instructions generated for identical Java expressions of type `int` and `long`.

Java classes, the core normalization rules can proceed to normalize the result further down to bytecode.

2.4.2 *Pseudo-Instruction Normalization*

The JVM supports over two hundred instructions, ranging from very low-level operations, such as manipulation of the stack, to high-level operations, such as support for synchronization and arrays. Many of these instructions are specialized for specific types, such as the `iadd` and `fadd` instructions that respectively add two integers or two floats. Rather than requiring the code generator to select the proper specialization of `<T>add`, we introduce overloaded pseudo-instructions to defer this to the final step of the compilation. This simplifies the implementations of specific language extensions, as they do not have to reimplement this specialization process.

The pseudo-instructions form a reduced set of only 67 essential instructions (see Figure 2.15). These are normalized to regular bytecode instructions, based on the type information provided by the verifier. For this, the verifier is extended with a transformation operator, which uses the type information provided by the transfer function of the verifier to replace all overloaded instructions with type-specific, standard bytecode instructions.

Consider Figure 2.20, which illustrates how very different bytecode instructions must be generated for identical Java expressions if they operate on different types. Instructions such as `iload` and `iadd` have a type prefix, and the `dup` or `dup2` instruction depend on the number of words that a value occupies (i.e., two for `long` values). In the reduced set, identical instructions can be used for many such patterns, thus simplifying normalization rules from Java (or another language) to the instruction set, and reducing their total number.

Compilation by normalization in practice From an external view, the Dryad Compiler has no discernible stages, and simply normalizes the input code to resulting code that can be normalized again. At each normalization step, the transformed code forms a valid Java/bytecode program. This design enables extensions – as well as built-in features – to make use of a wider range of language constructs, and prevents scattering their implementation across different compilation stages. Furthermore, it allows for different (separate) compilation scenarios than possible with conventional open compilers.

Still, the internal architecture of the Dryad Compiler does employ separate, discernible components. For instance, it employs a (global) semantic analysis phase, based on the Dryad typechecker component. As such, it does not conform to what may be an idealized image of how a normalizing compiler should work: by pure, iterative application of declaratively defined normalization rules. As it is, the Dryad Compiler uses strategies to maintain a degree of control over the application order of normalization rules. To simplify their implementation, the rules are formulated without special logic for preserving the semantic information collected in the analysis. This means that in some cases, the analyzer must be reapplied. While this may not be ideal, this architecture does not hinder the applications presented here: for the extensions, the Java/bytecode language acts as the interface of the compiler. The internal implementation of the compiler, how this language is normalized, and how this design may (and likely *will*) change in the future, is of no concern to the developer of an extension.

Core Language-Based Analysis We perform a number of data-flow analyses and optimizations on programs after they have been normalized to the core language. By doing these at this level they can be applied independently of the source language. The analyses have fewer constructs to deal with than for source code, reducing their complexity. Still, reducing a program to the bytecode form can also mean a loss of precision, due to the missing high-level structure and the reduced size of the code window used by transfer functions. Logozzo and Fähndrich [2008] offer a discussion comparing the relative completeness of either approach.

Composition of language extensions Modular definition of language extensions aids in the composability with other extensions, possibly developed by different parties. Ideally, these can be imported into a language without requiring any implementation level knowledge of the extensions. At the syntax level, this can be achieved using a modular syntax definition formalism such as SDF [van den Brand et al., 2002]. On the semantic level, the primary determinant of the compositionality of extensions is the type of analysis and transformations that are performed (global or local). For global transformations, an ordering of application must be determined, which requires implementation-level knowledge. Thus, composable language extensions should use primarily local transformations; composition of global transformations is an orthogo-

nal issue. Using small normalization steps, facilitated by expression blocks, pseudo-instructions, and the increased expressivity of the general Java/bytecode language, many extensions can be expressed using local transformations.

2.6 RELATED AND FUTURE WORK

Compiler extension Extensible compilers, such as the JastAdd Extensible Java compiler [Ekman and Hedin, 2007], ableJ [Van Wyk et al., 2007], Polyglot [Nystrom et al., 2003], and OpenJava [Tatsubori et al., 1999] provide a foundation for compilation of language extensions. Polyglot, ableJ, and OpenJava follow the front end extension approach; they offer an extensible semantic analysis stage and aid in projection to the base language. JastAdd on the other hand provides its own, modular back end. Our approach does not preclude the use of these tools, but can add to them by offering at the same time the Java language for projection and direct use of the underlying bytecode primitives or inclusion of compiled class fragments.

Forwarding, as supported by ableJ, avoids the need to implement full semantic analysis for a language extension [Van Wyk et al., 2002, 2007]. Instead, forwarding allows the meta-programmer to define the projection of a language construct to the base language, and by default applies semantic analysis over the forwarded program fragment. Our work is related to forwarding, as we similarly define a mapping from a source language to a base language, using normalization rules. Semantic analysis can be performed on the projected code, and any errors can be reported in reference to the user code using source tracing. Unlike in ableJ, we introduce core language constructs into the source language, to increase its expressivity and to facilitate normalization to this core language form.

Macro-based systems, such as JSE [Bachrach and Playford, 2001] and the Jakarta Tool Suite [Smaragdakis and Batory, 2002] implement direct projection to a base language, forgoing semantic analysis. This can lead to confusing error messages in references to locations and constructs in the generated code. We support semantic analysis on the source language using an extensible typechecker and verifier, and provide source tracing facilities to ensure any errors on code resulting from transformations can be traced back to user code.

Source tracing is a technique related to origin tracking [van Deursen et al., 1993], which maintains the origins (i.e., source terms and positional information) of terms during rewriting. We extend this notion by defining language constructs to maintain and share this information *explicitly* at the source level, to help interoperability between tools and simplify the internal representation. In Section 4.4.4 we revisit the topic of maintaining position information across transformations, describing an implementation of (implicit) origin tracking for Stratego.

Language composition A similarity can be drawn between the mixed Java/bytecode language presented here and existing languages such as C++ and Ada that allow inline assembly code languages. These too can be used for optimizations or obfuscation and provide access to low-level operations. As-

sembly code, however, is a representation of instructions for a specific CPU architecture, and therefore is much more low-level and less safe than bytecode. This makes it more difficult to use it for composition of source and compiled code.

The Java/bytecode language also shows similarities with the Jeannie language [Hirzel and Grimm, 2007], which integrates Java with C code, compiling to code that makes use of the Java Native Interface (JNI) [Liang, 1999], the JVM's standard foreign function interface. The C language can be used for accessing platform functionality or (legacy) libraries, but does not actually form the core language of the Java platform. Similar to the Dryad Compiler, Jeannie performs semantic analysis on the combined language, and introduces a number of bridging constructs for conversions and interoperability between the two constituent languages.

Different platforms We have applied the *compilation by normalization* architecture for the Java platform. Java provides a safe, mature, and ever-evolving platform for developing applications, and has hosted many new languages and language extensions [Hardwick and Sipelstein, 1996; Melton and Eisenberg, 2000; Seymour and Dongarra, 2001]. A similar architecture can be realized for other platforms based on a bytecode language. For instance, .NET may be an interesting case given it was designed from the ground up to host multiple languages, and includes support for *unsafe code*, which allows direct manipulation of object pointers. Other platforms provide a more high-level intermediate language, such as the Glasgow Haskell Compiler Core language [Peyton Jones and Santos, 1998]. Core is not a strict subset of the Haskell language and cannot be directly compiled [Tolmach, 2001], but this and similar languages make good candidates for use as a core language that is grown to a more extensive, (not necessarily existing) high-level language, by introducing new abstractions, while preserving the core functionality.

Other language extensions We presented a number of compiler extensions, demonstrating how the Dryad Compiler can facilitate their implementation. Future work could include other extensions, such as adding full support for aspect-oriented programming (AOP) [Kiczales et al., 1997], building upon the implementation of open classes (i.e., intertype declarations in AOP) and composition of code at the statement and expression level. Full aspect weaving can be performed by composition of compiled classes with aspect source code, or vice versa. A related design is used for the AspectBench Compiler (abc) [Avgustinov et al., 2005], which applies aspect weaving on the Jimple language, a three-address (stackless) representation of bytecode. The abc compiler uses Soot [Vallée-Rai et al., 1999] to (de)compile Java and bytecode to Jimple. Thereby they avoid some of the complexities associated with bytecode-level weaving. Using the Java/bytecode language instead would enable the direct insertion of regular Java code into class files for both advice and any dynamic checks or other wrapping code that accompanies it. By providing typechecking and verification for the combined language, as well as the guarantee of stack-neutrality of inserted statements, we provide the same level of safety

as is possible with weaving using Jimple. Similar techniques can be used in the implementation of annotation processing tools (such as Java APT [Sun Microsystems, 2004]), which typically operate purely based on source code.

There is a growing interest in running dynamic programming languages on the JVM. These compilers, such as Jython [Pedroni and Rappin, 2002] for the Python language, typically compile to Java source code. Using the Dryad Compiler as a basis, such compilers can make use of specific bytecode features and optimizations, such as the newly considered dynamic invocation instruction.[5] Similarly, other Java code generators could be retrofitted to generate selected bytecode features. The F2J Fortran compiler, for instance, used to generate Java source code, before it was re-engineered to directly generate bytecode instead, to support `goto` instructions and specific optimizations [Seymour and Dongarra, 2001]. JCilk also generates Java code, to assimilate an extension with fork-join primitives and exception handling for multithreaded computations [Danaher et al., 2006]. Similar to the problems with implementing the `yield` statement as a source-to-source transformation (Section 2.2.3), this significantly complicates the control flow semantics. However, rather than directly generating bytecode for the complete JCilk language, the current implementation depends on a modified version of the GNU Compiler for Java as its back end. It would be straightforward to use the Dryad Compiler instead, making use of the mixed language and introducing support for source tracing.

Java's source model of single inheritance with interfaces does not always match that of a given language that is intended to target the JVM. For instance, a restriction of Java interfaces is that only one method with a given signature may exist. This renders interfaces incompatible if they define methods with the same signature but with a different return type. For generated code, such as interfaces generated from annotations, this can be a restricting or incalculable factor. At the bytecode level, this restriction does not exist. Additionally the `synthetic` modifier, used for instance on bytecode methods of inner classes, can be used to mark methods inaccessible from user code (JVM Spec. [Lindholm and Yellin, 1999], §4.7.6). It can be used to hide multiple methods with the same signature, and can enable friend class semantics for generated code.

2.7 CONCLUSION

To increase programmer productivity, language extensions, both domain-specific and general-purpose, have been and continue to be developed. These may generate source code or bytecode; either approach has its advantages. Mixing source code and bytecode, a new language can be formed that has a synergistic effect, resulting in a language that at once provides the low-level expressivity of bytecode and the convenience and familiarity of Java. The combined language allows rapid development of language extensions through normalization steps that can remain loosely coupled from the base

[5] JSR 292: http://www.jcp.org/en/jsr/detail?id=292.

compiler. Using intermediate forms such as expression blocks and pseudo-instructions, these steps remain relatively small and maintainable.

Mixing source and bytecode opens the doors for new, more fine-grained forms of separate compilation, providing a foundation for composition of source and bytecode classes up to instruction- and expression-level precision. By use of source tracing, these composed fragments of code can be traced back to their original source files, enabling accurate location information for error messages and debugging.

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 the anonymous reviewers of OOPSLA 2008 for providing useful feedback on an earlier version of this chapter.

Using Aspects for Language Portability

ABSTRACT

Software platforms such as the Java Virtual Machine and the CLR .NET virtual machine have their own ecosystem of a core programming language or instruction set, libraries, and developer community. Programming languages can target multiple software platforms to increase interoperability or to boost performance. Introducing a new compiler back end for a language is the first step towards targeting a new platform, translating the language to the platform's language or instruction set. Programs written in modern languages generally make extensive use of APIs, based on the runtime system of the software platform, introducing additional portability concerns. They may use APIs that are implemented by platform-specific libraries. Libraries may perform platform-specific operations, make direct native calls, or make assumptions about performance characteristics of operations or about the file system. This chapter proposes to use aspect weaving to invasively adapt programs and libraries to address such portability concerns, and identifies four classes of aspects for this purpose. We evaluate this approach through a case study where we retarget the Stratego program transformation language towards the Java Virtual Machine.

3

3.1 INTRODUCTION

Programming languages form layers of abstraction over low-level machine code. New languages with higher-level abstractions can be created by introducing a new layer, targeting a lower-level programming language and an API in that language. Such high-level languages no longer target a particular hardware platform, but rather target a *software platform*. A software platform consists of one or more programming languages, application frameworks, and libraries, and can be used on one or more hardware platforms. By targeting software platforms in high-level languages, their design and implementation can benefit from the abstractions available on such a platform [Stahl et al., 2006]. Examples of software platforms are the Java and .NET platforms, LAMP[1], and C with the POSIX library. Each platform has its own ecosystem of programming languages, libraries, and developer community.

Language portability across software platforms For programming language designers, targeting multiple software platforms can be appealing for many reasons. Doing so may allow the language to run on different hardware; can make integration with existing software such as the Eclipse IDE on Java pos-

[1] Linux, Apache, MySQL, and Perl/PHP/Python.

sible; it may improve performance, as seen with JRuby [Nutter, 2008] and IronPython [Hugunin, 2006] at different points in time. A new platform may also attract a different developer community. Altogether, targeting a language to a new platform can be a rewarding endeavor.

Programming languages are typically implemented using a front end/back end architecture that aids in retargetability: a new back end can be added to generate code using a language or (bytecode) instruction set supported by the platform. However, this only addresses portability concerns at the bottom layer of the technology stack provided by a software platform.

Programming languages can also abstract over the layer of software (libraries and frameworks) provided by a platform, in particular to provide domain-specific abstractions in domain-specific languages (DSLs) [Stahl et al., 2006; Mernik et al., 2005]. For DSLs it is especially common to use frameworks in their respective technological domain, such as frameworks for web applications, data persistence, mobile applications, and so on. Libraries can be highly platform-dependent: they may perform platform-specific operations, make direct native calls, or make assumptions about performance characteristics of operations or about the file system. Some libraries are only available for selected platforms. For example, if a DSL for administrative web applications uses the Java-based Hibernate framework in its implementation, some equivalent is required to implement the language on another platform. These platform dependencies are exposed to programs either directly through the runtime system of the language, or indirectly through standard functions or libraries written in the language. Such dependencies can lead to programs being tied to a particular platform, regardless of the front end/back end separation in the compiler architecture.

Another area that requires attention when considering language portability is interoperability with existing applications and libraries on the platform. Care should be taken to address platform idiosyncrasies such as event handling models and exception handling. For example, in C when some error condition arises, an application could log a message to the console and quit with a non-zero exit code. But for other platforms throwing an exception instead could increase interoperability with other applications. For example, when embedded in a GUI application, a popup could be shown.

Addressing language portability Portability concerns at the level of programs and libraries written in a language can be addressed by modifying their sources, introducing changes to match the new platform or to abstract over the original platform. To avoid having to fork the sources and having to maintain multiple copies, *conditional compilation* can be used. Using conditional compilation, sources can be statically configured by a set of compiler flags that enable or disable logic for specific platforms. The most straightforward way of conditional compilation is to use a text-based preprocessor such as the C preprocessor (cpp). These preprocessors directly manipulate text and ignore the base language's syntax rules. This practice makes it much harder for various tools – IDEs, code analyzers, etc. – to process the code. Alternatively, true static conditional language constructs can be added, but they still complicate

reasoning about the language and supporting it in tools. Moreover, using any form of conditional compilation also leads to tangling and scattering of platform logic throughout the base source code.

Instead of conditional compilation, this chapter proposes to use aspect weaving [Kiczales et al., 1997] to address library-level portability concerns of programming languages. Using aspects, portability concerns can be expressed separately, rather than scattering them across the base source code. By using load-time aspect weaving, the base source code can be separately packaged and compiled and invasively adapted with platform-specific changes.

To evaluate this approach, we report on our experience in retargeting the Stratego program transformation language [Bravenboer et al., 2008] to Java. The language has originally been compiled to C and is associated with the GNU/Linux operating system; we evaluate how aspects can be used to address portability concerns when targeting the Java platform instead. Previous experience with the Stratego interpreter for Java has shown that targeting the platform has a number of appealing applications, such as integration with compiler front ends written in Java to support transformations [Kalleberg and Visser, 2007a], and integration into the Eclipse IDE with Spoofax. The present work improves the level of integration with such tools and allows Stratego programs that originally targeted the C platform – rather than specialized programs making use of Java-based libraries – to be used on Java and in Eclipse.

Stratego does not support aspect-oriented programming out of the box. Few languages do. In general, some form of aspect weaving must be added to the language for our approach to be effective. For our study, we introduced aspect weaving facilities to the language. Our results indicate that a minimal, lightweight set of aspect weaving features suffices to address portability concerns. Using load-time or run-time weaving, these features are straightforward to implement and support separate compilation.

Using aspect-oriented programming rather than conditional compilation, we can adapt the Stratego standard library without having to directly change the original code; the C and Java implementations remain truly separate. Existing Stratego programs that were designed for use on the C platform can be compiled to Java without requiring their sources to be changed. Java-specific adaptations are generally only needed in Stratego libraries, and can be expressed entirely using aspects.

This chapter studies the use of aspects to address language portability concerns. To this end, we:

- identify *four classes of aspects to address portability concerns*: glue code, migration, integration, and optimization aspects;
- describe a minimal, lightweight form of aspect weaving required to implement these aspects; and
- evaluate our approach through the implementation of a Stratego-to-Java compiler[2], showing instances of all four classes of portability aspects.

[2]STRJ, the Stratego-Java compiler, an open source project available from `http://www.strategoxt.org/Stratego/STRJ/`.

Outline We begin this chapter by discussing general objectives, design principles, and typical challenges faced in porting a language to multiple software platforms. We then discuss classes of aspects that can be used to meet these challenges. In Section 3.3, we describe the Stratego language, the current compiler design, and introduce our extension of Stratego with aspects. Section 3.4 shows portability problems in retargeting Stratego, and how these can be addressed with aspects. We discuss our results in Section 3.5.

3.2 TARGETING MULTIPLE SOFTWARE PLATFORMS

When targeting a different software platform with a language, there are a number of general goals that guide the design and implementation of such an effort. First, existing applications should run on the new platform with no or minimal changes. Second, any libraries written in the language should be reused. Unlike normal applications, libraries – especially those bundled with the language – often use low-level, primitive operations. If that is the case, a number of platform-specific changes must be made that are not exposed in the API. Third, integration with other applications, frameworks, and languages on the target software platform is a key part of the retargeting effort.

A well-known design principle for portable language implementations is the use of a front end/back end architecture [Aho et al., 2006], separating parsing, analyses and platform-independent transformations from platform-specific code generation. Another best practice is to define a fixed set of primitives provided by the platform, used by the language implementation or its standard library. For example, for Stratego primitives are defined for traversals over trees, for parsing, for file system access, and so on, each with a clearly defined interface. By separation of the platform-specific part of the compiler and maintaining a clear set of primitives, these components can be replaced when a different platform is targeted.

3.2.1 *Language Portability Concerns*

In theory, with the design principles outlined above in hand, we can port languages to a new platform by simply creating a new compiler back end and runtime system for it. However, in practice, a new back end and runtime system do not automatically constitute language portability. Many applications, libraries, and sometimes the compiler itself, have been written with a particular platform in mind, making certain assumptions and showing behavior only appropriate for that particular platform. These library-level portability concerns cannot be addressed by the compiler and runtime alone, and require changes to the code base written in the language. In this section, we outline different sorts of these concerns. We revisit them in Section 3.4, showing concrete cases for the Stratego language.

Platform-specific libraries High-level languages, particularly those that target a specific domain, are often implemented by providing a linguistic abstraction over high-level frameworks or libraries. These libraries are tied to a particular

software platform. For example, consider the Hibernate framework for Java. There are many popular object-relational mapping (ORM) frameworks for other platforms, but their semantics differ slightly.

When a library is not supported on a particular platform, a similar, alternative library may be available that can be used instead. Using *glue code*, it may be feasible to adapt it to the same basic interface of the reference library. Glue code is code that does not directly contribute any functionality towards meeting the program's requirements, but serves solely to "glue together" different parts of code, helping programs and libraries interoperate.

Platform escapes and native calls Some languages allow escapes to code written in the host language (e.g., C). These can introduce portability concerns as the code escaped to typically does not run on another platform. For other platforms, an alternative must be implemented. Ideally, code in the retargeted language – in our case study, Stratego – should be used instead to ensure portability. If this is not possible, the escaped code should be added to the set of primitives of the language, allowing different platforms to provide their own definitions.

Similar to escapes to platform code, native calls to other programs that run on the platform introduce a number of portability problems, as they add dependencies to external programs that may not be available on other hardware architectures or operating systems. Some platforms simply may disallow the use of native calls, or do not bundle required external programs, making deployment more difficult. Rather than making direct native calls, a more portable approach is to use libraries where possible. They have an interface that is less dependent of the operating system, file system, and path configuration used for the software platform.

Interoperability and integration with platform applications A strong motivation for targeting a particular platform can be to integrate the programs in the retargeted language with applications and libraries that run natively on the target platform (e.g., Eclipse on Java). Key to integration is a good interface for interoperability between the different languages. The public API of a generated application should be human-readable and easy to use. Advanced language features should be mapped to corresponding features on the platform. For instance, the Scala language targets the JVM and exposes advanced language features such as traits as standard Java interfaces for interoperability [Odersky et al., 2008]. In addition to the basic API interface, important notions for interoperability are event models and exception handling. Events and exceptions are particularly important when embedding programs written in a language traditionally used for batch processing, such as Stratego.

Performance and stack behavior Software platforms may use different libraries, a different language, and a different runtime system (e.g., the JVM), leading to different performance characteristics. On some platforms, performing an operation one way may be faster, while on other platforms, doing it another way is faster. For low-level operations, such issues can be addressed in the compiler back end, which can generate code most appropriate for a given

platform. High-level operations cannot be easily optimized by the compiler, as they may need to be changed at the algorithmic level.

Perhaps even more so than general performance trade-offs, a pressing concern in retargeting languages is that of memory limitations. Many platforms have a set limit to the heap and stack memory sizes that can be used by applications. Particularly software platforms that target hardware with limited resources (such as mobile devices) may have severe restrictions. While we do not address heap size limitations in this chapter – which, at least in principle, could be partially addressed in the same way as general performance concerns – we do pay particular attention to stack size restrictions.

Stack size constraints of the JVM are a notorious problem for running functional languages on the JVM, as they typically do not have explicit looping constructs but use recursion to perform loops [Schinz and Odersky, 2001]. Our case study of Stratego forms no exception in this regard. The stack consumption of recursive looping is linear to the amount of loops performed, whereas it remains constant for iterative loop constructs such as the "for" loop, typically used in Java programs. The maximum stack size cannot be changed at run-time, only when a new JVM is created, which may not be possible or desirable in all environments. A common approach to avoid the linearly increasing stack consumption of recursion is to rewrite recursive loops to iterative ones. One automatic approach to do this is by rewriting recursive tail calls into jumps to the start of a function [Schinz and Odersky, 2001]. A limitation of this approach is that it only applies to tail calls: many functions exhibit recursive or indirect recursive behavior that cannot be directly rewritten to use tail calls. Some common functional programming functions such as *map* and *filter* can only be written in tail recursive style by reversing the output list, incurring additional performance overhead. Baker [1995] proposed a more general approach, turning all calls into tail calls by converting the program into continuation-passing style. Unfortunately, for the JVM, Schinz and Odersky [2001] performed a similar experiment, which showed that a simple Fibonacci function in continuation-passing style was about 20 times slower than the standard version. Clearly, there is no silver-bullet solution in compilers for the stack size problem.

3.2.2 *Aspects to Address Language Portability Concerns*

We identify four classes of aspects to address portability concerns such as those listed in the previous subsection:[3]

- **Glue code aspects** add glue code to help compatibility with platform-specific libraries. Glue code written in the retargeted language itself is often more concise and high-level than wrappers at the platform level. Glue code aspects may be *transitional*; as libraries are ported to a platform, some additional glue code may be used to help compatibility.

[3]Note that there is a relation with the concerns in the previous subsection, but they do not necessarily map one-to-one to the aspects.

- **Migration aspects** help developers in retargeting their applications to a different platform. Typical use cases include platform escapes and native calls, which hinder portability. These aspects may either *passively* display warnings or errors for developers that use operations not (fully) supported on a platform, or they may *actively* aid the developer, redirecting such operations to alternatives that are supported on a platform but may not be fully compatible with the original operations. For example, if a language relies on Java serialization, on another platform it may implement serialization by use of an alternative API.
- **Integration aspects** aid in interoperability and integration with other languages on the platform. For example, they can add exception handling or throwing to functions written in the retargeted language, or they can add application-specific hooks for use by other software on the platform. Integration aspects can be *application-specific*, pertaining to a particular application; or *generic*, adapting the runtime system used for any application.
- **Optimization aspects** address run-time performance and scalability concerns. They change definitions in applications or libraries in order to achieve better performance for a particular platform, or to avoid platform restrictions such as stack size overflows resulting from uses of deep recursion on the JVM.

After giving an overview of the Stratego language and our extension with aspects in the following section, we show concrete example use cases of these aspects in Section 3.4.

3.3 MODULARITY AND ASPECTS IN STRATEGO

In this section we first briefly introduce the Stratego language and its implementation architecture. Stratego is a *bootstrapped* language, which means that it is implemented in itself. For our case study, this gives us the opportunity to explain a bit about the language itself and show how aspect-oriented programming (AOP) can be used with Stratego. As the current version of Stratego provides no support for AOP itself, we will also present the design and implementation of an extension of Stratego with a minimal set of aspect weaving facilities.

Stratego is a domain-specific language for program transformation, used to build tools such as compilers, static source code analyzers, and interpreters. Together with the XT set of tools, Stratego/XT can be used for building comprehensive, stand-alone program transformation tools. The most important of these tools are the ATerm exchange format [van den Brand et al., 2000] and SDF/SGLR [Visser, 1997c], both of which have been reimplemented in Java. For a comprehensive description of the Stratego/XT language and toolset, we refer the reader to [Bravenboer et al., 2008].

At its base, Stratego is a term rewriting language. Using a first-order term representation of (abstract syntax trees of) programs of a given language, pro-

grammers can define *rewrite rules* to transform programs. As a basic example, the following is the definition of a single rewrite rule that desugars a one-armed "if" statement with a condition *e* and a body *stm* to a two-armed "if" statement:

```
desugar-java:
  If(e, stm) →If(e, stm, Empty())
```

Whereas most term rewriting engines use a fixed (innermost) strategy for rewriting, Stratego allows the definition of custom rewriting *strategies*. Strategies are a generic description of how a rule or series of rules should be applied [Visser et al., 1998]. Stratego provides a few basic combinators for composing transformations from rules. For example, the combinator `s1 ; s2` produces the *sequential composition* of the transformation strategies `s1` and `s2`, and the combinator `s1 <+ s2` produces the *deterministic choice* of `s1` and `s2`. More complex strategies can be constructed from these basic combinators. For example, the strategy definitions

```
try(s)    = s <+ id
repeat(s) = try(s; repeat(s))
```

are part of the Stratego Standard Library. The strategy `try(s)`, tries to apply transformation `s`, but succeeds by producing the original term when `s` fails. The strategy `repeat(s)`, repeatedly applies a transformation `s` until it fails.

Just as it is a term rewriting language, Stratego is also a functional language. Both rules and strategies are (not necessarily pure) functions at heart: rules typically rewrite one term to another, while strategies typically correspond to higher-order functions. These are conventions that are not enforced by Stratego; in fact, many strategies in the Stratego standard library correspond to first-order functions or procedures that interoperate with the software platform.

3.3.1 *Modularity and Extensible Definitions*

Stratego's module system uses hierarchical *modules* that correspond to files relative to the set of import paths. Stratego uses a flat namespace, and allows multiple definitions of rules and strategies with the same name. These can be defined within a single module or across multiple different modules. For example, one module may define multiple `desugar-java` rules related to control-flow features, while another module may define desugaring rules for other language features. All definitions for a rule or strategy are merged together, in a fashion similar to open classes or multimethods: when called, the first successful matching definition is dispatched. The traditional, whole-program compilation scheme of Stratego does not support extension of definitions that have been compiled separately.

Together, a set of modules can form a Stratego *library*, which can be separately compiled and reused in other Stratego components. Each library introduces its own partition to the namespace: while modules can freely extend strategy or rule definitions within the same library, it is a compile-time error to add definitions that already exist in another library. There are two reasons

for this restriction; the first being that it avoids accidental name clashes, while the second is more pragmatic: since libraries are separately compiled, the current compilation scheme simply does not allow definitions in other libraries to be extended.

3.3.2 *Introducing Aspect-Oriented Programming to Stratego*

For introducing portability aspects it is necessary to have aspect weaving support in the implementation language of the compiler. For many languages, there are out-of-the-box AOP implementations, such as AspectJ [Kiczales et al., 2001] for Java, but not so for Stratego. As a programming language, Stratego offers a powerful mechanism for extending existing Stratego programs, but this mechanism is not sufficient to express the portability aspects we described in Section 3.2.2. It cannot be used to adapt definitions in external Stratego libraries without having to "recompile the world," and can only be used to introduce new definitions that are *independent* of the existing definitions. For example, a new rule may be added that desugars the "for" statement, but the rule for the "if" statement cannot be adapted without directly changing the original rule.

In previous work, Kalleberg and Visser [2006] introduced AspectStratego, an extension of Stratego with a full-featured aspect language. However, since it was based on source-level aspect weaving, it cannot be used to weave into separately compiled libraries. Weaving into separately compiled libraries can only be supported by weaving into compiled code or through load-time or run-time weaving. Load-time and run-time may be the options of choice for the purpose of addressing portability concerns as they require comparatively little implementation effort.

In this chapter we introduce a new extension of Stratego based on load-time weaving, supporting basic before/after/around advice. In our extension we define a new set of modifiers that refine Stratego's definition extension mechanism and give control over definitions in separately compiled libraries. These modifiers are `extend`, which extends an external definition; `override`, which overrides an external definition; and `internal`, which indicates that a definition should be closed to extension and must not be exported to other libraries. We also introduce the `proceed` keyword, familiar to that of some more conventional aspect languages, which allows advice to return control back to the intercepted code.

As an introductory example, we can add – perhaps a cliché – logging messages to the `desugar-java` rule from the beginning of this section:

```
override desugar-java =
  log(|Info(), ["Desugaring: ", <id>])[4];
  proceed
```

Note that we do not syntactically separate the definition of the join point and the advice. This particular aspect specifies a join point matching any strategy

[4]Note that in Stratego, function signatures use a vertical bar (`|`) to indicate value arguments; function `log` is called with two value arguments rather than with two function arguments.

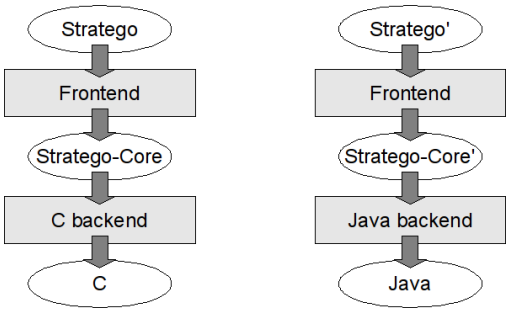


Figure 3.1 The basic architecture of the Stratego-to-C and Stratego-to-Java compilers, each compiling to core Stratego and then to the target language. The new Java compiler supports Stratego with aspects (Stratego′).

or rule named `desugar-java` with zero arguments. When needed, join points can use wild cards: a pattern `desugar-*` would match any rule with the prefix `desugar-`. The aspect also specifies "before" advice that prints a message with the current term (i.e., `<id>`) for every invocation of `desugar-java`. We do not distinguish between "before" and "after" advice in the language: instead, we use the standard Stratego operators such as sequential composition and conditional choice to combine the definitions.

3.3.3 *Implementation of Aspects in Stratego*

To implement aspect weaving in Stratego, we must add the new syntactic constructs to the front end, and adapt the back end to generate code for the new features (Figure 3.1). We implement aspects by generating standard Java source code. Since our join point model is relatively simple, this requires only a modest extension of the compiler. In general, generating code to a language that directly supports aspects – such as AspectJ [Kiczales et al., 2001] – may be easier, but in this case we want to avoid introducing a dependency to the AspectJ compiler.

The Stratego front end merges all rules and strategies with the same signature, as described in Section 3.3.1. The Java back end then compiles each definition to a single class that inherits from the `Strategy` class (Figure 3.2). This class defines a number of overloads of the `invoke()` method with different parameters, and a `dynamicInvoke()` method for invoking strategies with an unanticipated number of parameters. Figure 3.3 shows the class that implements the `desugar-java` rules. It overrides the `invoke()` method, which takes context information about the runtime[5] and the current term to which the rule is applied as its arguments. Overloads of `invoke()` with more parameters are not overridden; their default implementation in the superclass throws an illegal argument exception.

[5]The runtime context contains all information relevant for the current instance of a compiled program, including definitions of primitives and the Stratego stack.

```
public abstract class Strategy {
  // Invokes a strategy with no parameters
  public IStrategoTerm invoke(Context context, IStrategoTerm term) {
    throw new IllegalArgumentException();
  }

  // Invokes a strategy with one parameter
  public IStrategoTerm invoke(Context context, IStrategoTerm term,
      Strategy s) {
    throw new IllegalArgumentException();
  }

  (...overloads of invoke() with more arguments...)

  // Invokes a strategy with a variable number of arguments
  public IStrategoTerm invokeDynamic(Context context,
       IStrategoTerm term, Strategy[] s, IStrategoTerm[] t) {
    Based on the number of arguments, call the corresponding invoke()
    method or throw an IllegalArgumentException
  }
}
```

Figure 3.2 The Strategy base class.

```
public class desugar_java extends Strategy {
  public static desugar_java instance =
    new desugar_java();

  @Override
  public IStrategoTerm invoke(Context context, IStrategoTerm term) {
    IStrategoConstructor cons0 = term.getConstructor();
    if (cons0 == DesugarJava._consIf2) {
      Desugar the "if" construct
    } else if (...) {
      Apply other definitions of the desugar-java rule
    } else {
      return null; // rule application failed
    }
  }
}
```

Figure 3.3 The strategy class for `desugar-java`.

Each strategy is implemented as a *singleton class*, and has a mutable `instance` field. This field is used to invoke the strategy or to pass it as the argument to a higher-order function, such as the strategy call `topdown(desugar-java)` that applies `desugar-java` to a tree in a top-down fashion.

For strategies that are adapted by aspects, the `instance` field is assigned to the instance of the adapted strategy. For example, Figure 3.4 adds the logging advice from the previous subsection to the class for `desugar-java`. Note in particular that the `proceed` call in this definition is defined by copying the `instance` field of the original `desugar_java` class. When the library that contains the override class is initialized, it simply reassigns the existing `instance` field:

```
desugar_java.instance =
  new desugar_java_override();
```

Multiple libraries may add advice to the same definition in this fashion, following the order in which they were imported. The `proceed` field always

```
class desugar_java_override extends desugar_java {
  private final desugar_java proceed =
    desugar_java.instance;

  @Override
  public IStrategoTerm invoke(Context context, IStrategoTerm term) {
    term = log.instance.invoke(context, ...);
    if (term == null) return null;
    return proceed.invoke(context, term);
  }
}
```

Figure 3.4 A class overriding the `desugar-java` definition.

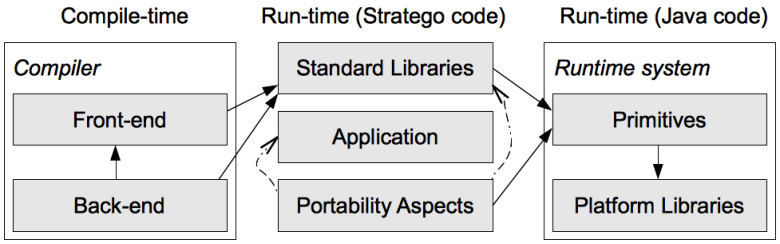


Figure 3.5 The role of aspects and the interaction between components of Stratego on Java. Dependencies are indicated by solid lines; weaving is indicated by a dashed line.

refers to the definition that preceded the new advice, allowing multiple advice rules to be combined.

3.4 ENCAPSULATING PLATFORM LOGIC WITH ASPECTS

In this section we elaborate on the concrete cases where aspects can be used to express platform logic, following the same structure as Section 3.2.1. In total, we created seven libraries with 44 aspect definitions to address portability concerns in different areas.

Figure 3.5 illustrates the overall architecture and interaction between the compiler and runtime components. At the left-hand side, the compiler consists of a shared front end and a platform-specific back end. The middle part of the figure shows the Stratego part of the runtime system; libraries and applications written in Stratego. Platform-specific portability aspects weave into these and may make use of the Java part of the runtime system. This last part consists of a fixed set of primitives that wrap libraries that run on the software platform.

3.4.1 *Platform-Specific Libraries*

On the C platform, Stratego uses the ATerm library [van den Brand et al., 2000] to represent terms, and the scannerless generalized-LR SGLR parser [Visser,

1997b] for parsing. While a Java implementation of the ATerm library exists, we use a more flexible library that allows custom term library implementations based on a fixed interface, making it possible to operate on arbitrary (wrapped) Java objects [Kalleberg and Visser, 2007a]. It supports the same basic operations of the ATerm library, but has different performance characteristics.

To parse files, we use JSGLR, a Java port of the SGLR parser. At the time of writing, JSGLR supports most functionality of SGLR, and – for the most part – is successful in strictly following the SGLR semantics. However, JSGLR currently has a number of differences that affect how it can be used: it operates on strings rather than streams[6], it has a built-in mechanism to create abstract syntax trees rather than parse trees, and it supports position information in abstract syntax trees. To support this new interface, we wrote new strategies to control the JSGLR parser, and added *glue code aspects* to support applications that may not have been designed with JSGLR in mind.

Figure 3.6 shows glue aspects related to the JSGLR parser library. The first two affect the standard API for parsing from streams: here, we first read the stream to a string using the `read-text-from-stream` strategy, and then call the JSGLR parser to parse that string. The next two already operate on strings, but are overridden with calls to the JSGLR-based API. Note that the `parse-stream-pt` and `parse-string-pt` strategies produce a parse tree, while the `parse-stream` and `parse-string` strategies produce a compacted abstract syntax tree. The default implementation of the latter two strategies uses a Stratego transformation to create the compact tree, while the JSGLR alternatives simply request the parser to create the compact tree. The last definition in the figure is a migration aspect for the `asfix-anno-location` strategy, which adds position information annotations to parse trees. On the C platform, this is implemented using a C function that transforms the parse tree to add position annotations. The `jsglr-asfix-anno-location` is a Java implementation of this function, but we also report a warning to indicate that developers should use the more efficient, native position information support of JSGLR instead.

3.4.2 *Platform Escapes and Native Calls*

The Good Stratego uses a runtime system with a well-defined set of primitives, as advocated in Section 3.2, to implement primitives in the standard library that cannot otherwise (or not efficiently) be implemented directly in Stratego. These primitives can be invoked using the `prim` language construct. For example, the standard library utility strategy `concat-strings` concatenates strings, and is implemented as follows:

```
concat-strings =
  prim("SSL_concat_strings", <id>)
```

[6]While a stream-based wrapper API is available, strings are used since they work better with backtracking for error recovery as applied in Chapter 6, the construction of token lists, used in Spoofax editors (Section 6.8.4), and with GWT, which doesn't support streams.

```
module jsglr-parser-compat

imports
  jsglr-parser

rules

  override parse-stream-pt(...) =
    read-text-from-stream;
    jsglr-parse-string-pt(...)

  override parse-stream(...):
    read-text-from-stream;
    jsglr-parse-string(...)

  override parse-string-pt(...) =
    jsglr-parse-string-pt(...)

  override parse-string(...) =
    jsglr-parse-string(...)

  override asfix-anno-location =
    warn-msg(|"Deprecated feature, use origin-location instead");
    jsglr-asfix-anno-location
```

Figure 3.6 Glue code aspects for compatibility with parsing on Java.

That is, the strategy calls a primitive by the name `SSL_concat_strings` and passes it the input term `<id>` of the strategy. In turn, `SSL_concat_strings` is implemented by a C function or a Java class in the runtime system.

Stratego has a large collection of well over a hundred of these primitives, most of which are straightforward to implement in Java. A compiler (back end) for a given software platform can recognize the `prim` constructs and translate them to function calls for that particular platform.

The Bad In addition to the `prim` construct, Stratego also supports a special `external` modifier to directly call custom, native C functions, not unlike the native methods used in a language like Java. Some Stratego applications use these as optimizations or to interface with native libraries. There are also some cases where escapes to platform code are made from the standard libraries and compiler. Because of the ad hoc nature of these functions – any application can define and use their own native functions – native function calls cannot be uniformly translated by a compiler.

As an example of an external strategy, some Stratego programs, including the C back end of the Stratego compiler and WebDSL [Groenewegen et al., 2010], use a native, C-based pretty-printer. Such a pretty printer can be generated from a pretty printing table and is typically faster than a pure Stratego-based implementation that reads the table at runtime. To use such a pretty printer, the pretty printing program must be linked to the Stratego program, and the name of the function must be declared in Stratego:

```
external pp-java(|)
```

Unfortunately, when Stratego runs on Java, this external C function is not available. It is not part of the fixed set of primitives, which means that an alternative must be provided at the program-level rather than at the level of the

runtime system. Using a runtime check or through conditional compilation at the call sites, an appropriate alternative could be called. Using aspect weaving instead, such changes can be implemented without changing the original code – which is undesirable for third-party libraries. All aspects dealing with native function calls on Java can be collected together rather than scattered throughout the code. For the case of `pp-java`, we can simply add advice that calls the table-based pretty printer instead:

```
override pp-java =
  pp-java5-to-abox; box2text-string(|80)
```

The Ugly Stratego programs have traditionally been based on XTC, a library for creating monolithic transformation programs by composition of smaller tools, such as a parser or a pretty-printer [Bravenboer et al., 2008]. XTC's function in the Stratego world has been to reconcile the philosophy of the Unix platform of "tools that do one thing, and do it well" with the need for comprehensive transformation tools. XTC maintains its own component model in a customizable location in the file system, called the XTC repository. The repository is used to store and retrieve file system paths to shared tools. Using the `xtc-command` strategy, any of these tools can be invoked with a given set of command-line options. XTC also provides a `xtc-find` strategy that has been used to find the paths of parse and pretty-printing tables used for an application, and for locating library headers for use by the Stratego compiler.

Lately, XTC is being phased out in favor of libraries [Bravenboer et al., 2008], which are more efficient than forking a new process for specific tasks. In addition to performance concerns, XTC – and direct invocations of executables in general – also hinders portability of applications that use it. On the Java platform, invoking tools in this fashion is often not possible, and using the file system for resources forms a mismatch with the light-weight deployment system of using JAR files for applications.

Figure 3.7 shows a number of *migration aspects* for running Stratego on the JVM. The first definition extends `xtc-command` to redirect any calls to the command-line `pp-java` tool to more portable strategies. Multiple, independent extensions like it can be added for other invoked tools, aiding in the portability of existing Stratego applications that have not yet made the transition to libraries. We also override the `xtc-find` strategy, which normally returns the absolute path of a file using the XTC repository. This is an example of a transitional migration aspect: we only display a warning and do not call `proceed`, allowing legacy applications to continue in case the requested file simply exists in the current directory or if it is an executable on the path.

Native executables can also be directly invoked using the `call` strategy. Normally, `call` is implemented using the `fork` strategy to fork the current process, but on Java no such notion exists (although it could be simulated). Therefore, as additional migration aspects, in Figure 3.7 we also override `fork` to print an error, and redefine `call` using a new Java-specific primitive, print-

```
override xtc-command(tool) =
  if tool ⇒"pp-java" then
    directly pretty-print using pretty-printing library
  else
    proceed
  end

override xtc-find =
  warn-msg(|["XTC used to find non-local file ", <id>]);
  id // don't proceed with the original xtc-find

override fork(child) =
  fatal-err(|"Not supported on this platform")

override call =
  ?(program, args);
  log(|Info(), ["Calling external tool ", program]);
  prim("SSL_EXT_call", program, args) ⇒0
```

Figure 3.7 Migration aspects related to native calls and XTC usage.

ing an informational message that reminds developers they are calling a native executable.

The Stratego compiler normally uses library definitions that use XTC to find header files for imports. On Java, headers should be distributed inside the compiler's JAR file instead. They can be automatically embedded using Stratego's `import-term` construct. The `import-term` construct is also available on the C platform, but has significantly different performance characteristics, inflating compilation times and executable sizes as they are serialized to array literals in C. Using separate aspect definitions ensures that only on Java `import-term` is used to retrieve headers.

Figure 3.8 shows the aspects that introduce `import-term` calls to the `pack-stratego-parse-stratego`. The first aspect fetches headers of the standard libraries. These are all tied to the version of the compiler and cannot be overridden. The second aspect retrieves the headers for the Java-Front library, which may be overridden with a different version by the user. The third aspect in Figure 3.8 ensures that any third-party library headers are not loaded using XTC, but only from the directories specified with the "-I" command-line option.

3.4.3 *Interoperability and integration with Java applications*

So far, we have already shown a number of ways in which direct interoperability between compiled Stratego code and Java can be achieved. Java code is used for the implementation of the platform primitives, and can be used to implement custom term libraries. By encapsulating Java code as primitives, Stratego programs can also directly invoke Java code or trigger event hooks. While applications may add their own application-specific hooks into Stratego programs, in this section we show a more generic case where we add Java code to the Stratego standard library to help integration with Java applications.

```
extend pack-stratego-parse-stratego:
  (IncludeFromPath(name), includes) → ("", ast)
  where
    switch !name
      case "libstratego-lib": import-term(headerfile)
      case "libstratego-xtc": import-term(headerfile)
      case ...
    end ⇒ ast

extend pack-stratego-parse-stratego:
  (IncludeFromPath("libjava-front"), includes) → result
  where
    if not(proceed ⇒ result) then
      result := ("", <import-term(libjava-front.rtree)>)
    end

override strc-get-include-dirs =
  <get-config> "-I"
```

Figure 3.8 Migration aspects for XTC usage for Stratego compilation.

Traditional Stratego programs have been batch applications such as compilers. They output a number of informational or error messages to the console, may write resulting files to disk, and then exit. When integrating such an application into interactive Java applications, such as the Eclipse IDE, a different way of presenting feedback to users is required, particularly for error reporting. This can be addressed using an *integration aspect* to improve this behavior by adapting the `fatal-err` strategy, used to report fatal errors (as seen in Figure 3.7). Normally, this standard library strategy is defined as follows:

```
fatal-err(|msg) =
  log(|Critical(), msg, <id>);
  <exit> 1
```

This standard definition prints an error to the standard error output, and then exits the application using `1` as the exit code. On the JVM, when the exit strategy is called, the `SSL_exit` primitive is invoked, which throws a `StrategoExit` exception containing the exit code. Unfortunately, this exception does not provide the reason *why* the application exited. This piece of context information was still available, though, when `fatal-err` was called. Thus, if we redefine that strategy to throw a more specific exception, `StrategoErrorExit` – which subclasses `StrategoExit` – we can make this information available to any Java application that invoked the Stratego code. We can refine the error handling behavior by throwing an informative Java exception instead, using a primitive called `SSL_EXT_fatal_err`:

```
override fatal-err(|msg) =
  log(|Critical(), msg, <id>);
  prim("SSL_EXT_fatal_err", msg, <id>)
```

Applications such as the most recent version of Spoofax can use the thrown exception to present the user with a pop-up in case of errors, referring the user to the complete error log for more details.

Besides exception handling, there are other examples of uses of aspects for integration with other applications. For example, hooks into the logging and

assertion strategies of Stratego to integrate with a graphical user interface or the Eclipse debugging API. Strategies that read and write text can also use the Eclipse API to manage encoding, which is maintained in metadata for Eclipse projects. More application-specific aspects may hook into strategies to provide feedback as a transformation runs, or to interact with the user.

3.4.4 *Performance and Stack Behavior*

Since a different software platform – different libraries, language, and the JVM runtime – is used for Stratego on Java, it has different performance characteristics than Stratego does on C. Using aspects, performance-critical sections of code can be replaced with new definitions that better suit the platform.

Using the excellent Java profiler, we were able to identify only a few bottlenecks in normal Stratego programs. Most could be addressed by ordinary, general compiler optimizations (e.g., caching the result of `getConstructor()` in a local variable, as seen in Figure 3.3). One strategy that really stood out in the profiler results was the `read-text-from-stream` strategy. It is implemented directly in Stratego by reading a stream character by character, constructing a string from the results. Still, the strategy was never a bottleneck for typical C-based Stratego applications. However, recall that we used this strategy in Figure 3.6, which means that it is now used for almost all parser invocations. We can override the strategy with a new definition

```
override read-text-from-stream =
  prim("SSL_EXT_read_text_from_stream")
```

where we introduce a new Java primitive that efficiently reads the stream using a block buffer instead of characters.

Optimization aspects can also be applied to adapt performance-sensitive areas of specific applications. One particular difference between Stratego on C and Stratego on Java is that they use a fundamentally different internal representation of terms: on C, maximal sharing is employed; on Java, any object that implements the Stratego term interfaces can be treated as a term. The latter allows for custom terms to be used for specific applications, or for storing additional information in terms. We adapted the runtime system of the Aster attribute grammar language that runs on Stratego (Chapter 5) to make use of Java objects to store parent pointers. On the C version of Stratego, where terms are represented using maximal sharing, annotations were used for this purpose. The use of parent points makes the Java implementation, which integrates with Spoofax, significantly faster.

Stack behavior As a functional programming language, Stratego uses recursion to express loops. On the Java platform, the stack is an expensive commodity, and using deep recursion quickly uses it up. We found that many Stratego applications, such as the compiler itself, often resulted in a stack overflow exception because of this restriction.

Stratego has many strategies that rely on left-to-right traversal of terms, producing new, immutable left-to-right encoded terms. To rewrite these recursive strategies to use tail recursion would mean that additional, intermedi-

```
override filter(s) =
  prim("SSL_EXT_filter", s | <id>)

override map(s) =
  is-list;
  all(s)

override getfirst(s) =
  is-list;
  one(where(s; ?x)); !x
```

Figure 3.9 Optimization aspects for stack-intensive strategies.

ate data would have to be maintained on the heap. The performance overhead rewriting key strategies such as `map` and `filter` in this fashion may not be acceptable. As an alternative approach, we can redefine these strategies using imperative Java code. Similar to what can be done with mixed imperative/-functional programming languages such as Scala [Odersky et al., 2008], selective use of imperative code in key library definitions allows for better stack behavior using mutable data structures and iterative looping.

Figure 3.9 illustrates some of the *optimization aspects* that help avoid stack overflows. In the figure, we redefine `filter` using a new primitive, implemented in Java; `map` using the `all` operator for lists; and `getfirst` using the `one` operator. Internally, each Strategy is based directly on imperative Java code: `filter` uses a primitive implemented using an array and a "for" loop (not shown here for reasons of space), and the other strategies use the primitive `all/one` operators. These `all/one` operators are standard strategic programming operators (described in [Bravenboer et al., 2008]) that apply their argument to *all* or *one* of the subterms of the current term. Since the operators are implemented directly in Java, we can use them here to avoid deep recursion on long lists. Similarly, we redefined or extended eight other key library strategies that use recursion. As a result, stack overflow conditions for virtually all applications are now avoided, without having to change the JVM's default stack size. For those applications that use custom, stack-intensive strategies, similar measures can be taken.

3.5 DISCUSSION

Aspects for portability concerns Our case study of Stratego for Java has shown instances of aspects of all four classes of glue code, migration, integration, and optimization aspects. We were able to neatly separate these concerns from the base source code, grouping them together by association. Using conditional compilation instead, they would be scattered throughout the existing Stratego code base.

Aspects allowed us to implement a new Stratego back end without "polluting" the existing code base with Java-related concerns. Since they can be woven into compiled code – using load-time aspect weaving – there was no need to recompile the base Stratego compiler components and libraries with every Java-related change. This resulted in a shorter development cycle.

For our case study of porting Stratego to Java, we introduced a modest extension of the Stratego language to support portability aspects. The implementation of the extension was rather straightforward as we made use of the standard composition operators that Stratego already provided, combined with runtime support by means of an `instance` field that simply indicates the actual version of a strategy. When applied to other languages, aspect-oriented facilities of the retargeted language and its host language may be leveraged to further simplify the approach.

Perspective for future platform support In our case study, we developed a new Java back end, runtime system, and portability aspects, so that Stratego now supports both C and Java as a platform. It can still be attractive to consider supporting other platforms as well. In particular, we are currently investigating the feasibility of adding Javascript support, in order to make it possible to use Stratego in a web browser, and to integrate it into client-side web applications. With its dynamic characteristics and sandbox-like execution environment, Javascript is comparatively closer to Java than it is to C, which means that we may be able to reuse some of the Java portability aspects.

Once another platform is added and there is already an established set of aspects, we expect that there are many opportunities for reuse, in part directly as some aspects will be the same for multiple platforms, and in part indirectly for aspects that are woven into the same code but with different advice. Once support for multiple platforms is established, e.g., C, Java, and Javascript, it becomes possible to pick and match aspects that correspond to a new platform. For example a new web-based or mobile platform likely has commonalities with Java and Javascript, so it could use some of the aspects defined for those platforms.

Stratego on the JVM The Stratego-to-Java compiler currently passes all standard Stratego unit tests, and can be used to compile large projects such as PIL [Hemel and Visser, 2009], WebDSL [Groenewegen et al., 2010], and the Aster attribute-grammar extension of Stratego (Chapter 5). These applications run without Java-specific changes except for disabling the native Java pretty printer of WebDSL (as discussed in Section 3.4.2). Any features that do not translate well to the Java platform, such as native calls using XTC, are generally handled by migration aspects, helping developers migrate to newer, platform-independent APIs.

By supporting the Java platform and deep integration with other Java applications – in part made possible by the integration aspects – this work has been an important step in the realization of a fully fledged interactive development environment for Stratego in the form of the Spoofax language workbench.

Performance In our implementation of aspects for Stratego, every strategy call is implemented as a virtual method invocation. Most normal Java programs also use a large number of virtual methods. To reduce the performance overhead associated with these calls, the JVM can speculatively inline them [Detlefs and Agesen, 1999]. (For this reason, we gave the `instance` fields of strategy classes the exact type of the class, rather than the more

general `Strategy` type.) Because of this optimization, the overhead of our implementation strategy is relatively low. Initial measurements show that typical Stratego applications compiled to Java are no more than two times slower than when they are compiled to C. For example, consider compilation of the core Stratego compiler itself to Java, on a 2.4 GHz machine. This takes about 30 seconds when the compilation runs on the C platform, and about 56 seconds on Java. Performance is probably better when it is not run as a batch process but as a long-running process, such as an interactive environment, where the overhead of class loading and JIT compilation is smaller. Future optimizations, particularly in the term library and the I/O runtime system, may make this gap smaller. Another possible optimization may be to change all strategy calls into guarded, non-virtual calls in the code generator. Still, our initial performance assessments indicate the flexibility provided by aspects (and pluggable term libraries [Kalleberg and Visser, 2007a]) does not lead to a prohibitive amount of overhead.

3.6 RELATED WORK

While there has been previous work proposing to apply aspects in fields where traditionally conditional compilation has been applied, none of these works have applied aspects to address language or compiler portability concerns. Adams et al. [2009] and Reynolds et al. [2008] systematically studied different patterns of conditional compilation uses in, respectively, the Parrot virtual machine and the Linux kernel. Both studies concluded that for a large part of these uses, aspect weaving is a feasible alternative, but that in many cases preparation of the base source code was required to expose additional join points. Lohmann et al. [2006] did a quantitative analysis of the performance overhead of using aspects for configuration of the eCos kernel, and found the overhead to be acceptable. Likewise, they concluded that preparation of the base source code was required. C-CLR [Singh et al., 2007] is a tool that shows different views of source code, hiding disabled conditional parts. Through clone detection techniques, it allows mining of aspects.

All the above works have studied configuration of *systems software*, using the C language. In contrast, we studied the use of aspects for portability of high-level *programming languages*. In our case study, we did not find the need to prepare the base source code to expose join points or to use potentially fragile statement-level join points (as suggested by [Lohmann et al., 2006]). In part, this was due to high-level and concise nature of Stratego definitions and in part because of the nature of the aspects. As such, only a modest aspect-oriented extension of Stratego was required, which meant only a small, acceptable startup cost was required for using aspects for portability.

Another area where aspects have been used for language engineering is in the construction of compilers [de Moor et al., 1999; Avgustinov et al., 2008], decomposing different crosscutting concerns in analysis, transformation, and generation of code. These techniques are complementary to our approach, as we still rely on a modular front end/back end definition of a compiler

to target a particular platform. In contrast to these techniques we use aspects outside the compiler, to address portability concerns that cannot be effectively addressed inside the compiler.

A notably different approach to language portability is taken by the Scala compiler. It supports both the Java and .NET platforms, but its primary platform is Java: the standard Scala framework borrows types and methods from Java, and, at this point, the .NET back end has not been updated to the most recent version of the language. Instead of using conditional compilation for the Scala standard library, the .NET back end addresses the Java-based nature of Scala by redirecting a fixed set of Java-specific method calls to compatible .NET methods [ScalaNet, 2008]. For example, calls to `Object.hashCode()` are redirected to their .NET equivalent, `Object.GetHashCode()`. As the two platforms are closely related, this strategy suffices for operations on standard types such as strings and objects. However, the approach reduces separation of concerns as the compiler must encode library-specific logic. In contrast, we use separate libraries of aspects. For platforms with a greater set of differences, where simple redirects do not suffice, these libraries can be used to encode further platform-specific logic. Encoding platform logic in the compiler approach is also less flexible as it cannot be used to adapt third-party libraries or applications.

There have been many proposals of intermediate languages to address compilation to multiple host languages, starting with languages such as the Universal Computer Oriented Language UNCOL [Steel, 1961] to more recent works such as C-- [Jones et al., 1999] and PIL [Hemel and Visser, 2009]. These languages form an excellent complementary technique to our approach, eliminating much of the work required in implementing back ends for multiple platforms. However, by themselves, they do not address portability concerns such as uses of native calls, platform-specific libraries, or platform-specific performance concerns, as discussed in Section 3.2.1.

As a comparatively high-level intermediate language that supports object-orientation, the PIL language can also be used to implement complete application libraries [Hemel and Visser, 2009], thus potentially addressing the problem of platform-specific libraries. While that approach involves a high upstart cost – requiring the implementation of application libraries by hand – it also a high payoff: libraries implemented in this fashion can in principle be used with any platform that PIL supports. In our present work we take a different approach, relying on existing application libraries, and using glue code and migration aspects to ensure portability of existing code.

Aspect languages In previous work, Kalleberg and Visser introduced AspectStratego [Kalleberg and Visser, 2006], which extends Stratego with support for aspects, showing how they can be used to address concerns such as format checking, adaptable algorithms, and traceability. AspectStratego uses source-level weaving and supports a more extensive join point model than that of the present work. In particular, it can be used to define pointcuts at match and build operations, rather than just at the level of strategy definitions and calls. In contrast, for the present work we used load-time weaving, which was

essential to allow weaving into compiled libraries, and ultimately allowed for a more straightforward implementation. As such, it gave a good indication of how a minimal, lightweight aspect weaving addition can be used for language portability,

Aspect languages that integrate with object-oriented programming languages, such as AspectJ [Kiczales et al., 2001], typically have a more elaborate join point model than the one we presented here or that of AspectStratego. They may support pointcuts for specific packages, classes, and parameter types. In contrast, aspects for domain-specific languages generally have a more restrictive join point model. In the case of Stratego, which is not object-oriented, lacks package names, and has definitions that span multiple modules, such a model arose naturally, and formed a good match with the classes of aspects shown in this chapter.

Other mechanisms for modularization Dynamic languages such as JavaScript, Smalltalk, and Ruby allow extensions and modifications of existing objects and classes with new methods that support the needs of particular applications or libraries. This practice is informally known as *monkey patching* [Bracha, 2008]. In our approach, all changes are performed at load time, and are statically checked; missing join points are reported by the compiler. Still, when systematically applied, a fully dynamic approach to address portability concerns as presented here is certainly feasible.

Language modularity extensions such as MultiJava [Clifton et al., 2006] and eJava [Warth et al., 2006] add support to Java for extending methods. Stratego supports a similar concept by supporting multiple definitions of one rule or strategy in one or more modules. The present work adds support for extending strategies across library boundaries. A particularly interesting feature of MultiJava is *lexical scoping* of method extensions. As we weave in extensions dynamically at load-time, we do not support lexical scoping. However, an extension of our work could be to add dynamic checks to scope extensions, using the current execution context.

3.7 CONCLUSION

This chapter proposes to use aspect-oriented programming to address portability concerns with regard to languages that target multiple platforms. To this end, we identified four general classes of aspects to address such concerns: aspects that add glue code to platform-specific libraries, aspects that help developers migrate to a new API, aspects that help integrate with other applications on a platform, and those performing platform-specific optimizations. We showed instances of these classes in retargeting Stratego to the Java platform. In this case study, we successfully used aspects to neatly encapsulate concerns that would otherwise have spanned many different modules, in the compiler, library, and runtime system. It is our expectation that the same techniques can be used to help in separation of portability concerns for other high-level languages.

Future work with regard to aspects for portability concerns relates to software product lines: once more software platforms are targeted by languages, are there aspects that can be applied to multiple platforms? What are their dependencies on other aspects?

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 for his work on Stratego/J that formed the basis for the compiler presented here, and thank the anonymous reviewers of SCAM 2010 for providing useful feedback on an earlier version of this chapter.

The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs

4

ABSTRACT

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. Spoofax integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment. It uses concise, declarative specifications for languages and IDE services. In this chapter we describe the architecture of Spoofax and show how language semantics can be described using rewrite rules, showing how analyses can be reused for transformations, code generation, and editor services such as error marking, reference resolving, and content completion. The implementation of these services is supported by language-parametric editor service classes that can be dynamically loaded by the Eclipse IDE, allowing new languages to be developed and used side-by-side in the same Eclipse environment.

4.1 INTRODUCTION

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain [van Deursen et al., 2000; Mernik et al., 2005]. They provide linguistic abstractions over common tasks within a domain, so that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. DSLs have a concise, domain-specific notation for common tasks in a domain, and allow reasoning at the level of these constructs. This allows them to be used for automated, domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [Mernik et al., 2005].

For developers to be productive with DSLs, good integrated development environments (IDEs) for these languages are essential. Over the past four decades, IDEs have slowly risen from novelty tool status to becoming a fundamental part of software engineering. In early 2001, IntelliJ IDEA [Saunders et al., 2006] revolutionized the IDE landscape [Fowler, 2005b] with an IDE for the Java language that parsed files as they were typed (with error recovery in case of syntax errors), performed semantic analysis in the background, and provided code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as programmers were typing, and the ability to transform the program based on the abstract

representation (refactorings). The now prominent Eclipse platform, and soon after, Visual Studio, quickly adopted these same features. No longer would programmers be satisfied with code editors that provided basic syntax highlighting and a "build" button. For new languages to become a success, state-of-the-art IDE support is now mandatory. For the production of DSLs this requirement is a particular problem, since these languages are often developed with much fewer resources than general purpose languages.

There are five key ingredients for the construction of a new domain-specific language. (1) A parser for the *syntax* of the language. (2) Semantic *analysis* to validate DSL programs according to some set of constraints. (3) *Transformations* manipulate DSL programs and can convert a high-level, technology-independent DSL specification to a lower-level program. (4) A *code generator* that emits executable code. (5) Integration of the language into an *IDE*.

Traditionally, a lot of effort was required for each of these ingredients. However, there are now many tools that support the various aspects of DSL development. Parser generators can automatically create a parser from a grammar. Modern parser generators can construct efficient parsers that can be used in an interactive environment, supporting error recovery in case of syntax-incorrect or incomplete programs. Meta-programming languages [Bravenboer et al., 2008; van den Brand et al., 2002; Cordy et al., 1991; Hedin and Magnusson, 2003; Klint et al., 2009] and frameworks [Nystrom et al., 2003; WALA, 2006] make it much easier to specify the semantics of a language. Tools and frameworks for IDE development, such as IMP [Charles et al., 2007, 2009] and the Dynamic Language Toolkit (DLTK) [DLTK, 2007], simplify the implementation of IDE services. Other tools, such as the Synthesizer Generator [Reps and Teitelbaum, 1989], Centaur [Borras et al., 1989], and Lrc [Kuiper and Saraiva, 1998] can even generate a working structured or visual code editor from a language description.

Language workbenches With a wealth of language construction tools, a need arose for comprehensive tools that integrated these different solutions and guided the development of languages. Pleban [1984] suggested the notion of a "language designer's workbench" to address this issue: an toolset that integrates support for all aspects of programming language design and implementation. This idea was elaborated upon by Lee [1989], and also reflected by [Heering and Klint, 2000] who proposed the notion of language design assistants that would support the language designers by providing design choices and performing consistency checks during the design process.

More recently, Fowler [2005a, 2009, 2011] described the trend of integrating the development and use of DSLs into a single IDE environment, and introduced the term *language workbenches* for these tools. In contrast to earlier work, these workbenches focus on language-oriented programming, domain-specific languages, and integration of meta-programming and programming in modern IDEs. In [Fowler, 2009] he described this development as follows:

> "Whereas external and internal DSLs have been around for longer than I've been programming, language workbenches are a much newer ani-

mal. These tools support DSL creation not just in terms of parsing and code generation but also in providing a better editing experience for DSL users."

Fowler studied a number of practical, modern examples of language workbenches that allow developers to define and use text-based DSLs, including the Meta Programming System (MPS) [Voelter and Solomatov, 2010] and Intentional Programming [Simonyi, 1995]. In his article he also spoke of visual editor environments such as DSL Tools [Cook et al., 2007], but as these have a very different programming model and do not support text-based languages, we will not discuss them here. Fowler described that language workbenches greatly increase the cost-effectiveness of developing a new language, perhaps even to the point that they can be developed for a single application as sometimes strived for in language-oriented programming [Ward, 1994; Fowler, 2005a]. Rather than using a pure text representation, the workbenches Fowler described store the abstract representation of a DSL program, and use syntax-directed (or projectional) editing to manipulate this representation directly. Based on an abstract representation of a program, these workbenches can analyze a DSL program, perform transformations on it, and may show different views.

While it is important to maintain an abstract representation of a program to enable IDE features such as those made popular by IntelliJ, this does not imply that it should be the principal *storage* representation of programs, certainly given the disadvantages of that approach. Fowler noted the need to be able to store incomplete and contradictory information in the abstract representation, which is not trivial in this model. Other disadvantages include the lack of free text editing; incompatibility with standard, text-based version control systems and issue trackers; and having no way to import artifacts from other (possibly legacy) tools or to edit programs with other tools (leading to vendor lock-in). A free text editing approach, based on modern parser generators, seems much more attractive, since it avoids these problems without precluding the advantages of a language workbench.

Requirements For a language workbench based on freely editable, textual languages, we identify the following requirements:

(1) It must provide an integrated environment for both defining languages and using generated editors.

(2) Conversely, it must be possible to deploy generated editors separately from the workbench for use by "end developers," who may not be interested to work in a meta-programming environment.

(3) The environment must provide state-of-the-art IDE facilities. It should provide a substantial number of modern, language-specific editor services such as automatic indentation and bracket insertion, on-the-fly error markers, reference resolving, and content completion. Many of these services require an abstract representation of a DSL program; the editor should schedule parsing and semantic analysis in the background.

(4) The environment should support efficient, agile language definition, through incremental and selective development of IDE services, which requires separation of concerns between language specifications and pure IDE logic.

Related work The Meta-Environment [Klint, 1993; van den Brand et al., 2001] was one of the first tools that, while not focusing on DSLs, could be described as a language workbench (avant la lettre), combining language specification using ASF+SDF [van den Brand et al., 2002] and generated editors for using these languages (1). While it supported the construction of editors for "end developers," these could never really escape the meta-environment (2). Conceived in the early nineties, it did not yet support modern IDE features (3) based on real-time parsing and semantic analysis as programs are edited; error recovery was unavailable for the generated GLR parser at the time. Rather, it required developers to save a file in syntactically correct state to get a list of errors in a separate view. Based on the SEAL language [Koorn, 1993], the Meta-Environment supported separation of concerns in the user interface design and the language definition (4).

More recent endeavors, which fuse language specification and the construction of modern, interactive IDE components (3), are EMFText [Heidenreich et al., 2009a], MontiCore [Krahn et al., 2008], TEF [Scheidgen, 2010], and Xtext [Efftinge and Voelter, 2006].

These approaches all follow the same general architecture. They define their own language for the description of grammars. They may allow annotations in the grammar for the description of syntactic editor services [Krahn et al., 2008], or, by annotating lexical use-def relations, basic semantic editor services [Efftinge and Voelter, 2006; Heidenreich et al., 2009a]. From this grammar they generate a new, separate Eclipse plugin project (2). However, rather than providing a truly integrated environment, they require a second Eclipse instance to load this plugin (1). For IDE support beyond the basic services that can be derived from the grammar, the workbenches allow developers to write fragments of Java code to customize the generated plugin (4). The workbenches either use generated Java classes or an Eclipse Modeling Framework (EMF) metamodel [Budinsky et al., 2004] for the abstract syntax. Transformations are carried out using Java visitors or external, EMF-based tooling (4). The tools include string template engines for code generation.

Spoofax In this chapter we present Spoofax, a language workbench that enables efficient, agile development of software languages with state-of-the-art IDE support based on concise, declarative specifications.

Spoofax is an integrated environment for the specification of languages and accompanying IDE support in Eclipse. Generated editors can be *dynamically loaded* into the 'meta' Eclipse instance enabling smooth switching between development *of* the language and development *with* the language under construction. Spoofax also supports the generation of a stand-alone plugin for the language under construction that can be deployed to "end developers" without exposing the meta-programming facilities.

Spoofax supports a wide range of editor services based on tightly integrated, real-time application of syntactic and semantic analyses. Analyses are based on the structured abstract representation provided by a live parse of the text in the editor, which uses a parser scheduled in a background thread. Error recovery ensures that editor services function even in the presence of (multiple) syntactic errors, as described in Chapter 6. Origin tracking [van Deursen et al., 1993] techniques are used to relate the results of analysis back to the text in the editor without requiring preservation of layout information in the specification of analyses and transformations. These and other techniques for the implementation of editor services have been factored into language-parametric components, allowing language developers to focus purely on the language-specific parts of a compiler and IDE.

Spoofax supports language definition *with* declarative domain-specific languages. The modular, declarative syntax definition formalism SDF [Heering et al., 1989; Visser, 1997c] is closed under composition, ensuring support for language extensions and embeddings [Bravenboer and Visser, 2004]. The Stratego transformation language provides a unified formalism for *concise* specification of analysis, transformation, and code generation, enabling reuse of analysis rules for multiple purposes, including dynamic rules [Bravenboer et al., 2008] for context-sensitive analysis and transformation. We have developed idioms for language specification based on rewrite rules that can be used in batch compilation as well as in interactive editor services. Editor descriptor DSLs provide the bridge between specification of syntax and semantics and the language-parametric editor service components, providing a pluggable interface supporting the language engineer in adding new operations to the editor.

To show that our approach is practical, we describe the specification of a web language and report on practical experience with the implementation of other languages and integration with external tools.

The Spoofax language workbench is an open source project, and is available from `http://spoofax.org/`.

Previous work The Spoofax project started in 2007 with the development of Eclipse editors dedicated to Stratego and SDF [Kalleberg and Visser, 2007b]. In order to provide IDE support for languages built *with* Stratego and SDF, we developed a prototype of a new Spoofax environment built from scratch, described in [Kats et al., 2009b], where we showed how DSLs can be used to define presentational editor services, and how such definitions can be derived from a grammar. We also sketched an interface for error markers and reference resolving. This chapter shows how a single semantic description based on rewrite rules can be used for both compilation and interactive editor services such as error markers, reference resolving, and content completion. The new Spoofax environment comes with full-featured, "bootstrapped" IDE support for the meta-languages used for language specification, as well as meta-programming features such as the ability to apply transformations directly from the environment.

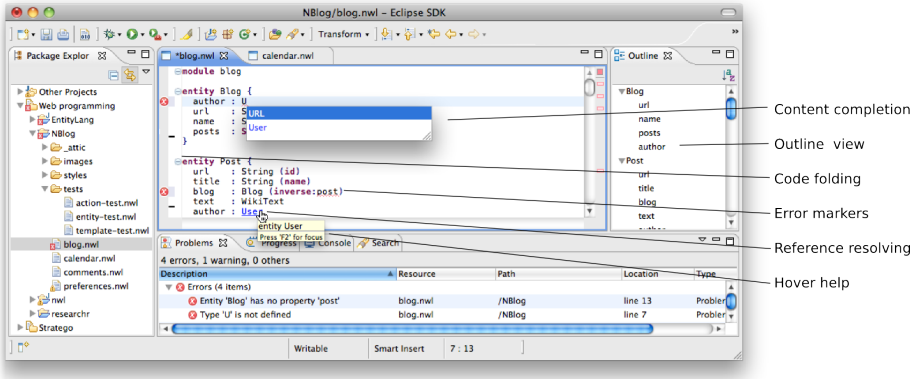


Figure 4.1 Editor services for a web language.

Outline We proceed as follows. We first describe the architecture of Spoofax and the general anatomy of Spoofax language definitions in Section 4.2. In Section 4.3 we discuss syntax definition and the specification of syntactic editor services. In Section 4.4 we discuss the definition of language semantics: analysis, transformations, code generation, and editor services based on these techniques. We report on experience with language development using Spoofax in Section 4.6. In Section 4.5 we elaborate on the implementation of the language workbench. Finally, we discuss related work and directions for future work in Sections 4.7 and 4.8, and conclude in Section 4.9.

4.2 AN OVERVIEW OF SPOOFAX

In this section we give an overview of Spoofax from the point of view of three categories of software developers. "End developers" of a Spoofax IDE work with the editor services specialized to the their (domain-specific) language. The developers of Spoofax itself maintain its architecture and language-parametric components. Language engineers use Spoofax to develop a language definition, i.e. the language-specific elements of an IDE.

4.2.1 *Editor Services*

Modern IDEs provide a wide variety of language-specific editor services, which are based on tightly integrated, real-time application of syntactic and semantic analysis. Editor services are separate entities that provide a service that is directly exposed in the editor or is shared across multiple components. Figure 4.1 illustrates a selection of editor services.

The editor checks the syntax of the program text, marks syntactic errors inline, and highlights text elements based on the syntactic structure *as the developer types*. The syntactic state of the parser at the cursor is used for editor services such as syntax completion, automatic bracket insertion, bracket

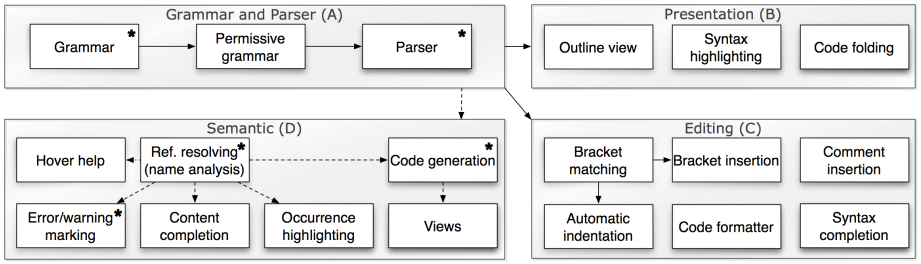


Figure 4.2 Relations between IDE components. Dependency flow is indicated with arrows; generative dependencies are indicated with a solid line. Components with an asterisk are generally also part of traditional batch compiler implementations.

highlighting, automatic indentation, and comment insertion. The abstract representation provided by the parser enables code folding, the outline view, and navigation using the quick outline feature.

Based on live semantic analysis of the abstract representation produced by the parser, the editor displays error and warning markers in the code. Program navigation and understanding is supported by reference resolving, occurrence highlighting, and hover help, which use semantic analysis to reveal relations between elements of a program. Content completion shows the developers the valid ways to complete the current construct. Transformation and code generation, using the results of semantic analysis, can be triggered each time the editor is saved, or on demand through the "Transform" drop down menu or using context menus.

In addition to the language-specific editor services provided by a Spoofax plugin, Eclipse is an extensible environment that offers many language-generic development facilities such as plugins for version control, build management, and issue tracking, and the package explorer view (left of Figure 4.1) that gives an overview of all projects and is used for resource management.

4.2.2 *Component Architecture*

Traditionally software languages are developed first as stand-alone compilers and IDEs are later added, typically requiring a significant reimplementation of many of the ingredients of the compiler to realize the implementation of editor services. The components of a compiler – parser, semantic analysis, transformations, and code generation – also play a central role in editor services based on the abstract syntax and semantic analysis of a program. Spoofax has been designed to factor out language independent implementation knowledge into the generic Spoofax libraries. Furthermore, language-specific definitions are defined such that they can be reused in several IDE components. Figure 4.2 gives an overview of basic compiler components (marked with an asterisk) and editor services in an IDE. The dependencies between these components

can be characterized as *generative dependencies* – a component can be automatically derived from another – and *usage dependencies* – a component calls another.

The grammar and parser are at the root of the dependency graph in Figure 4.2 (A), since the syntactic structure of programs is the basis for the implementation of all other services. In particular, the services for presentation in Figure 4.2 (B) and editing in Figure 4.2 (C) are automatically derived from the grammar. These services can then be customized, or re-written from scratch, as desired. Key for the derivation of functionality from a grammar is the use of a declarative syntax formalism. Semantic actions or escapes to external functions, which are sometimes used with parser generators, make it hard to reason about the structure of a grammar for other purposes. In the implementation of Spoofax we use SDF [Heering et al., 1989; Visser, 1997c]. Another essential component for an editor is error recovery, which ensures that editor services based on the structure of the program keep working in the presence of syntax errors. In Chapter 6 we describe how we can derive error recovery rules for a grammar to ensure good error recovery even for complex grammars composed of multiple embedded languages or extensions.

The semantic services cannot be derived from the grammar, since they depend on an interpretation of the syntactic structure of programs. *Name analysis* is a central component, which is reused in all other semantic editor services. Name analysis resolves the declaration of names in a program according to the scope rules of the language.

4.2.3 *Structure of a Language Definition*

A Spoofax language definition is an Eclipse project that defines the language-specific elements of an IDE, reusing the language-parametric components from the Spoofax infrastructure.

Figure 4.3 gives an overview of the default structure of a language definition project. Each of the three main components – syntax, service descriptors, and semantics – is defined in a number of modules. Developers are free to organize these how they wish, but the default layout separates the different concerns into different files, allowing developers to quickly familiarize themselves with the components and interfaces of a language definition.

An important design principle in the combination of derived and handwritten files has been to clearly indicate in the file name which files are generated. These files are regenerated every time the project is rebuilt and should not be edited by the language developer. To ignore specific rules in the generated file, they can be disabled or redefined in the accompanying handwritten file. If the generated file is not used at all, it can simply be removed from the list of imported descriptor files.

The *syntax* is defined using SDF [Heering et al., 1989; Visser, 1997c]. The default project comes with a skeletal language with four production rules (shown in Figure 4.4), and a module `Common.sdf` with default rules for comments and lexical patterns such as strings and identifiers.

Custom	Generated
Syntax definition	
Lang.sdf	Common.sdf
Editor service descriptors	
Lang.main.esv	
Lang-Builders.esv	Lang-Builders.generated.esv
Lang-Colorer.esv	Lang-Colorer.generated.esv
Lang-Completions.esv	Lang-Completions.generated.esv
Lang-Folding.esv	Lang-Folding.generated.esv
Lang-Outliner.esv	Lang-Outliner.generated.esv
Lang-References.esv	Lang-References.generated.esv
Lang-Syntax.esv	Lang-Syntax.generated.esv
Semantic definition	
lang.str check.str generate.str	

Figure 4.3 Language definition components.

Editor services are defined using declarative, rule-based editor descriptor languages. These can be used to define presentation or editing services, and can describe the interface of semantic editor services (describing what transformations to use for which service, and which views can be shown for a language). Derived services are maintained in separate `.generated.esv` files, and provide basic functionality (or, at the very least, examples) for these services based on the grammar. Not all services can be derived, but these files are also a source of documentation and examples.

Semantic definitions are specified using Stratego [Bravenboer et al., 2008], which provides an integrated solution for analysis, transformation, and code generation rules. Spoofax separates editor service specifications and the transformations that implement them. Editor service descriptors specify *which* transformations to apply, while the Stratego specifications specify *what* these should do. This design ensures flexibility in the implementation of services and allows for possible future integration with other meta-programming languages and frameworks.

We discuss the three categories of definitions and their relations in more detail in the following sections.

4.2.4 *Agile Language Development*

The architecture of the Eclipse platform is based on the OSGi Service Platform [OSGi, 2009], in which each plugin is (usually) a JAR containing Java classes, a plugin manifest, optional descriptor files, and auxiliary resources, such as images. The descriptors specify which parts of the Eclipse framework a given plugin extends, and which parts of the plugin may be extended by other plugins. The plugin model used by Eclipse implies distributing plugins as static JARs. The normal workflow cycle for plugin developers is to declare

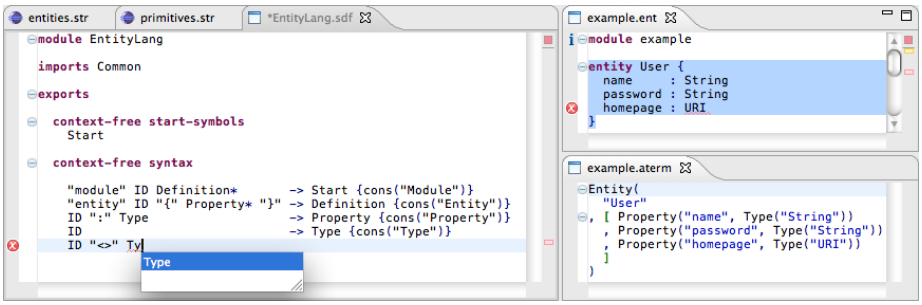


Figure 4.4 Multiple editors, side by side, in the same Eclipse IDE instance: the definition of an entity language (left), an editor for the entity language itself (upper right), and the abstract syntax of the selected entity (lower right).

new extensions in the `plugin.xml` descriptor file, implement these in Java, and test the plugin in a second instance of Eclipse, which is detrimental to a rapid development process.

One IDE instance Language definitions in Spoofax are based on the Eclipse plugin project model: each language definition includes a plugin manifest and descriptor files that allow it to be distributed to "end developers" as a normal Eclipse plugin. However, to enable agile language development we use a very different workflow model than that of standard Eclipse plugin development. By using language-parametric editor services that dynamically load and update language-specific service specifications (described in more detail in Section 4.5.1), we can use generated editors for a language in the same environment (Eclipse instance) in which we edit the language definition itself. Figure 4.4 illustrates how a grammar (left) and a generated editor (upper right) can be used side by side. The editor is fully functional and includes semantic services also defined in the same environment. The lower right editor illustrates an abstract syntax view for the selection in the generated editor that is updated in real-time as the selection is edited.[1] The same view can be used to inspect (intermediate) results of transformations.

Inductive design Rather than designing a complete DSL "on paper," before its implementation, it is good practice to incrementally introduce new features and abstractions through a process of evolutionary, inductive design [Visser, 2007; Fowler, 2005a]. In the context of a language workbench, this means that DSL programs and the DSL itself evolve together. This enables quick turn-around time for the development of the DSL and the subsequent gradual extension as new applications are developed, and new insights into the domain are acquired.

[1] While a graphical abstract syntax view can be visually appealing, we opt for an automatically formatted textual view instead, as it is much more concise, conveys the same information, and benefits from standard, textual editor services. Moreover, the same textual representation is also used in (and can be copy-pasted to) specifications of analyses and transformations.

The Spoofax environment assists in the initial creation of a new language using a wizard that simply takes the name of the language, the file extension it uses, and a package name. The wizard then creates a new Eclipse project with a skeletal language definition. From this point, new language features can be added through an iterative development process. New language constructs can be added to the grammar. These features can be directly used in the editor for the language.

A new language project created by the wizard includes standard Eclipse plugin configuration files (these are typically not changed by language developers), as well as specification files native to the Spoofax environment. Developers can then define editor services and define semantics for these features. Some editor services are automatically derived from the grammar; their specification can be adapted as desired.

An important aspect of the Spoofax architecture is that it allows for selective development of editor services. Developers can freely select what services to implement: the editor can also be used with a subset of all features. For example, developers may forgo sophisticated semantic analyses and transformations, and simply define code generation by a very direct mapping of abstract syntax to target code using string templates. Reuse is key for efficient language development, which means there are some dependencies between services (as seen in Figure 4.2), but most can be completed individually, allowing the language and IDE to be evaluated and used at every stage of development. In Chapter 8 we discuss how tests can be used to drive an incremental DSL development process.

Language understanding with views The specifications of most editor services, in particular those for semantic services based on analyses and transformations, are defined in terms of a textual abstract representation of programs. Using the abstract syntax view (Figure 4.4), developers can inspect the abstract syntax of a text selection or file.

The abstract syntax view of Figure 4.4 is not a built-in Spoofax feature, but it is a view that is defined with the default, skeletal language definition. Views show the results of transformations (as indicated by the relation in Figure 4.2). When a view is opened, it is automatically placed to the side of its source file, allowing developers to view both at a glance. Views are implemented using standard, textual editors (either for DSLs or languages such as Java that live in the Eclipse environment). They show either the abstract syntax of a transformation or the concrete syntax (e.g., standard Java code). The default view for showing the abstract syntax is defined by showing the result of the "identity" transformation, i.e., code is only parsed, not transformed.

Views are an important aspect of our architecture and a requirement for agile language development: they are essential for awareness of the abstract representation of a language, and can be used to show (intermediate) results of analyses and transformations independent of other editor services.

4.2.5 *Example Domain-Specific Language*

In the following sections we use the NWL language[2], a subset of WebDSL [Groenewegen et al., 2010], to illustrate key points of language definition. NWL covers several aspects of web programming i.e. entity declarations (data modeling), properties with inverse relations, parameters and variables, expressions, template definitions, page navigation, and several types of template elements. However, in this chapter we focus only on definitions of entities and actions (which are analogous to data type definitions and functions in other languages).

4.3 SYNTAX

The central implementation artifact for any textual language is the parser, which can be generated from a grammar.

In Spoofax, the grammar has the following roles:

1. it specifies the concrete syntax (keywords etc.)
2. it specifies the abstract syntax (the data structure used for analysis and transformation of programs written in the language)
3. it is used to derive editor services for presentation and editing that can be customized by the developer

We use SDF [Heering et al., 1989; Visser, 1997c] to define grammars. SDF grammars are declarative, highly modular, combine lexical and context-free syntax into one formalism, and can define concrete and abstract syntax together in production rules.

SDF productions take the form $p_1 \ldots p_n \rightarrow s$ and specify that a sequence of strings matching symbols p_1 to p_n matches the symbol s. Productions can be annotated with a constructor name n to uniquely identify them in the abstract syntax using the {**cons**(*n*)} annotation. Other annotations include {**left**} and {**right**} to specify the associativity of operators. Spoofax also supports {**deprecated**(*e*)} to mark obsolete language constructs. When a file is parsed that uses these constructs, they are marked with a warning in the editor. Optionally the editor can report an explanation e to instruct the user to use alternative syntax.

Figure 4.5 shows an abbreviated SDF grammar for the NWL language. The grammar extends the basic entity language of Figure 4.4 with additional features. NWL modules consist of a module name and a list of `Def` definitions. Definitions can be entity declarations, import declarations, or actions. Actions have a comma-separated list of `Param` parameters and a list of `Stat` statements.

[2]The complete definition of NWL is available at `http://strategoxt.org/Spoofax/NWL/`.

```
module NWL
imports Common
exports

context-free start-symbols
  Start

context-free syntax
  "module" ID Def*          → Start {cons("Module")}
  "import" ID               → Def   {cons("Import")}
  "entity" ID "{" Prop* "}" → Def   {cons("Entity")}
  "action" ID "(" {Param ","}* ")" "{" Stat* "}"
                            → Def   {cons("Action")}

  ID ":" Type         → Param {cons("Param")}
  ID ":" Type         → Prop  {cons("Property}
  ID                  → Type  {cons("SimpleType")}
  "Set" "<" Type ">" → Type  {cons("SetType")}

  Exp ":=" Exp ";"    → Stat  {cons("Assign")}
  "for" "(" ID ":" Type ")" "{" Stat* "}"
                      → Stat  {cons("ForAllEntity")}
  "for" "(" ID ":" Type "in" Exp ")" "{" Stat* "}"
                      → Stat  {cons("ForAll")}

  "all" "(" Type ")" → Exp   {cons("ForAllExp")}
  STRING             → Exp   {cons("StringLit")}
  ID                 → Exp   {cons("Var")}
  Exp "." ID         → Exp   {cons("PropAccess")}
```

Figure 4.5 An abbreviated grammar for entities and actions in NWL. The full NWL grammar also includes annotations for entity properties and additional statements.

Mapping between abstract and concrete syntax The abstract syntax, used in the specification of editor services, can be represented as first-order terms of the form

```
t ::= "..."      // string literals
  | c(t1,...,tn) // constructor applications
  | [t1,...,tn]  // lists of terms
```

As an example, consider the first production of the NWL grammar:

```
"module" ID Def* → Start {cons("Module")}
```

This production has three elements: the literal "module", an identifier name, and a list of definitions. For analyses and transformations we're usually not interested in literals and layout, so only the name and list of definitions are included in the abstract representation:

```
Module("example", [Entity("User", [...])])
```

which corresponds to the abstract syntax of the module at the upper right of Figure 4.4.

Spoofax generates a parser from the grammar, which produces the abstract representation of a file every time the user presses a key and a short delay passes. After the parser completes, all editor services that depend on the abstract representation are updated automatically. Internally, the abstract representation is stored efficiently in memory as Java objects, and maintains full layout and position information for use in services that need it.

Editor services related to presentation and editing can be based directly on the syntax definition (as indicated by the relation in Figure 4.2). These services can be fully specified using declarative editor service descriptor specifications. Rather than give an exhaustive overview of these descriptors and their features (available online at `http://spoofax.org`), we show some examples in this section to give an impression of how a declarative descriptor DSL can concisely describe these services.

Syntax highlighting Default syntax highlighting behavior is derived based on the literals and lexical syntax in the grammar. The colors used for this derived behavior are specified in the generated colorer descriptor, shown in the lower half of Figure 4.6. It specifies a color for keywords (alphanumeric literals in the grammar), operators (non-alphanumeric literals), strings (lexicals that allow spaces), numbers (lexical numeric patterns), and identifiers (other lexicals). The default colorization works well, but can be customized in the `NWL-Colorer.esv` file. The top half of Figure 4.6 illustrates custom coloring rules for the `Type` symbol, with specific colors for the `SimpleType` and the `SetType` constructor. Other coloring rules can override the colors for literals and lexicals, and can specify background colors, colors for regions of code rather than single productions, and more.

Code folding and outline view Code folding and the outline view are specified by selecting grammar productions that should be made foldable or shown in the outline view. Figure 4.7 illustrates some folding rules for the NWL language. Spoofax uses heuristics to automatically derive a generated folding descriptor, based on the logical nesting structure of the language. Currently, productions rules that have an identifier lexical and a list of child elements are included in this descriptor. While not perfect, the heuristic provides a good starting point for a new folding definition. Any undesired definitions in the generated file can be disabled by using the `(disabled)` annotation in the custom specification. The `(folded)` annotation can be used for constructs that should be folded automatically.

Bracket highlighting and insertion By describing pairs of matching brackets and the comment constructs of a language, the bracket highlighting, bracket insertion, and comment insertion features can be enabled for an editor (Figure 4.8). Bracket pairs are also used to supplement the automatic indentation specification (not shown): the cursor is automatically indented one level if a newline is entered after an opening bracket.

Syntax completion We distinguish syntactic and semantic content completion (the latter is discussed in the next section). Syntactic content completion provides users with completion suggestions based purely on static, syntactic templates. For example

```
completion template:
  "entity " <e> " {\n\t\n}"
```

```
module NWL-Colorer
imports NWL-Colorer.generated
colorer
  Type.SimpleType : cyan
  Type.SetType    : gray


module NWL-Colorer.generated
colorer
  keyword    : magenta bold
  identifier : default
  string     : blue
  ...
```

Figure 4.6 Syntax highlighting rules for NWL.

```
module NWL-Folding
imports NWL-Folding.generated
folding
  Start.Module
  Definition.Entity
  Definition.Action
```

Figure 4.7 Folding rules for NWL.

```
module NWL-Syntax
language
  line comment  : "//"
  block comment : "/*" * "*/"
  fences        : ( ) { }
```

Figure 4.8 Comment and bracket definition rules for NWL.

is a syntactic completion rule for entity definitions. Completion rules are composed of static strings and placeholder expressions. Static strings allow for precise control of the presentation of completions and are enclosed by double quotes. They can use `\n` for newlines or `\t` for one indentation level (following the user's tab/space configuration). Placeholder expressions are indicated by angular brackets. The editor automatically moves the cursor to these expressions once the user selects a completion proposal, allowing the expressions to be filled in as the user continues typing.

4.4 ANALYSIS AND TRANSFORMATION

Semantic analysis has two key roles in the implementation of programming languages. First, the analysis checks if programs program are (type) consistent, reporting errors if they are not. Second, it provides semantic information for use by compilers, IDEs, and other language-specific tools.

In IDEs, semantic analysis forms the basis for all semantic editor services. There are two forms of semantic analysis that are particularly important for IDEs: name analysis and type analysis. Name analysis binds each identifier occurrence to its declaration. Name analysis is exposed directly in the IDE in the form of reference resolving: press and hold Control and hover the mouse cursor over an identifier to reveal a blue hyperlink that leads to its declaration.

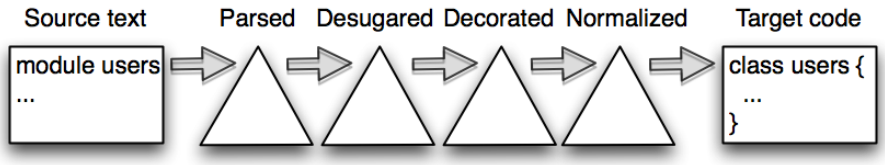


Figure 4.9 Staged compilation: source code is parsed, transformed, and finally printed to target code.

Type analysis determines the type of expressions and is important for reporting errors and for context-dependent code generation. Other analyses such as data-flow analysis and pointer analysis may also have a role in marking errors and warnings or for optimization of the generated code, but in this chapter we focus on name and type analysis because of their central and crucial role in both compilation of languages and for editor services.

Like many traditional compilers, we employ a staged architecture of analyses and transformations, as illustrated in Figure 4.9. First programs are parsed, then syntactic sugar is eliminated, and then they are analyzed, creating abstract syntax trees decorated with semantic information. Semantic editor services such as reference resolving and error marking operate on these decorated trees. After the analysis, the tree can be further normalized to a core form, and finally code generation rules can generate resulting code.

In the remainder of this section, we first introduce the Stratego transformation language, and then show idioms for using Stratego rewrite rules to concisely and declaratively specify analyses and transformations for use by editor services and for code generation.

4.4.1 *Stratego*

Spoofax uses the Stratego program transformation language [Bravenboer et al., 2008] to describe the semantics of a language. The Stratego language is based on the paradigm of term rewriting with programmable rewriting strategies introduced by Visser et al. [1998]. Basic transformations are defined by means of conditional *term rewrite rules* of the form

```
r : t1 → t2 where s
```

with `r` the name of the rule, `t1` and `t2` first-order terms, and `s` a *strategy expression*. A rule applies to a term when its left-hand side `t1` matches the term, and the condition `s` succeeds, resulting in the instantiation of the right-hand side pattern `t2`. Otherwise the application *fails*. Unconditional rules have no `where` clause, others may have multiple clauses that must all be satisfied.

In addition to checking applicability constraints, the `where` clause of a rule can perform computations that may be used in the right-hand side of the rule. For example, in the rule schema

```
r : t1 → t2 where t3 := <s> t4
```

the term `t4` is transformed by application of a strategy or rule `s`, matching against and binding variables in the pattern `t3`. Term `t4` may use variables from the left-hand side `t1`, and right-hand side `t2` may use `t3`.

Rewrite rules can concisely express small transformations based on the abstract representation of a program. Using content completion of terms, based on the syntax of a language, and by providing views of the abstract syntax and the results of transformations, the Spoofax environment assists in writing these rules.

More complex transformations can be created by composing rules using *strategies*. Many strategies can be compared to visitors in object-oriented programming in that they guide the application of rules in a tree. A strategy is essentially a partial function from terms to terms. If a strategy is not defined on a term it is said to *fail*. Failure arises from the failure of rewrite rules to apply to terms. Strategies are composed from basic combinators such as the identity transformation `id`, sequential composition `s1; s2` and deterministic choice `s1 <+ s2`. The Stratego standard library provides a number of strategies for general use, such as `map(s)` that applies a strategy expression `s` to each element of a list, and `alltd(s)` that descends down the branches of a term up to the points where `s` can be successfully applied, returning the complete term after transformation.

Context-sensitive transformations can be expressed by means of *dynamic* rewrite rules [Bravenboer et al., 2006b], which are instantiated at run-time, as illustrated by the following schema:

```
r : t1 → t2
where rules(dr : t3 → t4)
```

The dynamic rule `dr` is defined when `r` is applied to a term matching `t1`. Any variables that `t3` and `t4` share with `t1` are then inherited by the instantiation of `dr` (concrete examples follow below).

4.4.2 *Desugaring*

Desugaring rules simplify the abstract representation of programs, transforming higher-level constructs to more general, lower-level constructs, or mapping constructs with multiple forms to a single, canonical form. They can also be used to migrate deprecated language constructs to newer alternatives. This way, later analysis and transformation stages only need to focus on a subset of all language constructs.

The full WebDSL language supports various different iteration expressions and statements with optional filters and ordering clauses [Groenewegen et al., 2010]. In our NWL subset we provide three basic iteration constructs without filtering or ordering, shown in the grammar of Figure 4.5. The `ForAllExp` expression and the `ForAllEntity` statement iterate over all instances of a given entity type. The `ForAll` statement is more general and iterates over a given expression. To avoid having to write typing, transformation, and code generation rules for each of these variations, we use desugaring rules to transform similar constructs to the most general form.

```
desugar-top = innermost(desugar)

desugar:
  ForAllEntity(x, t, s*) →
  ForAll(x, t, ForAllExp(t), s*)

desugar:
  Call(x) → CallArgs(x, [])
```

Figure 4.10 Simple desugaring rules.

Figure 4.10 shows a `desugar` rule that transforms the `ForAllEntity` statement to the more general form. Another definition of the rule transforms action calls without arguments to action calls with an empty list of arguments (omitted in the syntax definition). More sophisticated, context-sensitive transformations, such as type inference for the `ForAll` loop, can be applied after semantic analysis of the program. The `desugar` rules are exhaustively applied in an innermost fashion by the `desugar-top` strategy.

4.4.3 *Reporting Errors and Warnings*

There are a number of concerns in checking a program for errors and reporting them to the user:

- *Context*: identifying points in the code to check
- *Assumptions*: only report an error if certain assumptions hold (validating the context and avoiding spurious errors)
- *Constraints*: checking for constraints at the context
- Formulating an *error message*
- *Attribution* of the error to a particular character range in the source text (usually, only part of the context is marked, such as the name of an erroneous method)

In Spoofax, error checking is a transformational process: it transforms an abstract syntax tree to a list of errors (also a tree, containing messages and attributed tree nodes). We use regular Stratego rewrite rules to transform parts of the tree to errors. These *check rules* encapsulate all of the above concerns, and adhere to the following idiom:

```
check:
  context → (target, $[error message])
  where assumption
  where require(constraint)
```

The rule checks whether for the given `context` (the term under scrutiny) the `constraint` is satisfied, given that the `assumption` holds. The `require(s)` operator is sugar defined as follows:

```
require(s) = not(s)
```

```
check-top = collect-all(check)

check: Var(x) → (x, $[Variable [x] is not declared])
  where require(type-of)

check: SimpleType(x) → (x, $[Undefined type [x]])
  where require(is-simple-type)

check: PropAccess(e2, p) → (p, $[[t] has no property [p]])
  where t := <type-of> e2
  where require(type-of)

check: ForAllExp(t) →
       (t, $[For loop requires entity type argument])
  where <is-simple-type> t
  where require(<is-entity-type> t)

check: SetType(t) →
       (t, $[Set requires entity type argument])
  where <is-simple-type> t
  where require(<is-entity-type> t)
```

Figure 4.11 Consistency checking rules.

That is, the check rule *succeeds* (producing the specified `error message`), if the `assumption` succeeds but the `constraint` *fails*.

The right-hand side is a pair consisting of target term and an error message. The `target` is (typically) a subterm of the `context` and denotes the term to which the error message should be attributed. The error message should explain that the `constraint` does not hold. The error message is a string typically composed using *string interpolation*, i.e. `$[...]` is a string consisting of all literal characters between the quotes, except for escapes between `[...]`. For example, if `t` is bound to `"Blog"` and `p` to `"size"` then `$[[t] has no property [p]]` evaluates to the string `"Blog has no property size"`.

Figure 4.11 gives a selection of check rules for the NWL language. The `check-top` strategy controls the application of these rules: it collects a list of all tuples resulting from successful applications of the check rules. For the specification of assumptions and constraints, these rules use a number of helper rules that are defined by the name and type analysis (which we describe in the next subsection). The `type-of` helper is a rule that returns the type of an expression. If the type cannot be resolved (indicating an error in the program), `type-of` *fails*. For example, the first check rule of Figure 4.11 requires that `type-of` succeeds for the context. If it does not, an error is reported. Another helper is `is-simple-type`, used in the second check rule: it only succeeds if the given type is a declared entity type (as opposed to a set type). This helper is also used to distinguish the case where constructors are undefined or just non-entity types.

4.4.4 *Binding Transformations to Editor Services*

Transformation with origin information For the check rules it is important to maintain the relation between the original source location and the term an error is attributed to. Figure 4.12 illustrates the different steps of the error

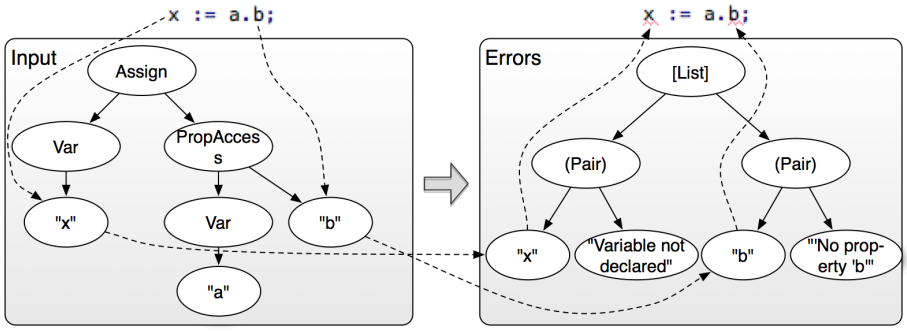


Figure 4.12 Transformational error checking with origin information.

checking strategy. First, the source code is parsed, desugared, and analyzed, producing the abstract syntax tree to the left. Then the `check-top` strategy is applied, producing the tree to the right, with pairs for all attributed terms and error messages. Throughout the desugaring, analysis, and error checking process, the relations between the original source positions and the terms are maintained, as shown by the dashed lines. Using these relations, all errors can be reported in the source code locations that correspond to the attributed terms.

The origin relations are maintained automatically by the Spoofax environment, allowing language developers to write concise, position information-agnostic transformations rather than keeping track of the original position manually. Position information is implicitly passed along with transformations through a form of *origin tracking* [van Deursen et al., 1993]. Origin tracking is a general technique that can be applied in term rewriting systems to implicitly maintain a link to the originating term after it has been rewritten to a new term.

Origin tracking as introduced in [van Deursen et al., 1993] has originally been used to trace back errors to their original source, much like we do in our error checking transformation. However, as we show in the remainder of this section, the technique can also be used to for high-level specifications of analyses and transformations used by editor services.

Control rules The transformation of an abstract syntax tree to a list of errors is controlled by the analysis *control rule*. Control rules are regular Stratego rewrite rules with a fixed signature that form the interface between the IDE and the Stratego transformation specification.

The control rule for semantic analysis and error reporting is shown in Figure 4.13. The rule is given information about the file that is being analyzed, in the form of a tuple with the abstract syntax tree of the file to be checked, its project-relative path, and the file system path of the project itself. The rule transforms this input to a new abstract syntax tree, decorated with semantic information, and lists of errors, warnings, and informational notes to be

```
editor-analyze: (ast, path, project-path) →
                (ast'', errors, warnings, notes)
  where editor-init;
        ast'     := <desugar-top> ast;
        ast''    := <declare-top> ast';
        errors   := <check-top> ast'';
        warnings := <check-warning-top> ast'';
        notes    := <check-note-top> ast''
```

Figure 4.13 Control rule for semantic analysis.

marked in the program. In the `where` clause, the rule first resets the current analysis environment using the built-in `editor-init` strategy. Corresponding to Figure 4.9, the rule then desugars the input abstract syntax tree and analyzes it, by calling the `declare-top` strategy (given below). It then collects all markers for the resulting (decorated) abstract syntax tree using the check strategies.

Semantic editor descriptors Editor descriptor files determine which control rule is used for which service. For example, the `NWL-Builders.esv` file specifies:

```
observer: editor-analyze
```

which means that `editor-analyze` is used as the `observer` control rule for the semantic analysis and error reporting. Control rules can also be specified for when files are saved (`on save`), for when a view is opened (`builder`), for reference resolving (`reference`), and for each of the other semantic services of Figure 4.2.

4.4.5 *Name and Type Analysis*

Name analysis binds each identifier occurrence to its declaration. Based on name analysis, type analysis identifies the types of expressions. Name analysis is also used for semantic editor services and code generation, as illustrated in Figure 4.2. Together, name and type analysis form the basis for consistency checking rules as those seen in Section 4.4.3.

As name analysis and type analysis are generally used together, it is common practice for compilers to combine the two analyses. Rather than locating the definition site for identifier occurrence, the name analysis then directly collects a mapping of names to types, and the type analysis determines the types of nested expressions.

In Spoofax we use the name analysis independently from the type analysis in semantic editor services. For this reason it is important to separate it from the type analysis. In this section we present an idiom for separate specification of name and type analysis using rewrite rules and dynamic rules. For simplicity, we divide the name analysis in two stages: first we collect all globally visible names, and then we analyze scoped names.

Unscoped name analysis Globally visible names can be stored directly as dynamic rules that map an identifier to its declaration site. For NWL, we use

```
declare-top =
  declare-primitive-types;
  alltd(declare);
  rename-top

declare-primitive-types =
  rules(
    PrimitiveDef: "String" → "String"
    PrimitiveDef: "Int"    → "Int"
    ...
  )

declare: d@Entity(x, p*) → d
  where rules(EntityDef : x → d)

declare: d@Action(x, param*, stat*) → d
  where rules(ActionDef : x → d)

declare: Import(x) → Import(x)
  where open-import(
          resolve-import
        , parse-file
        , declare-top
        )

resolve-import: Import(x) → $[[<project-dir>]/[x].nwl]
```

Figure 4.14 Name analysis for globally visible names.

the `EntityDef` and `ActionDef` dynamic rules for global names (Figure 4.14). The `declare-top` strategy controls the stages of the name analysis. It first declares all primitive types, then applies the `declare` rule to the outermost definitions, and finally calls `rename-top` to analyze scoped names (shown below). The `declare` rules match entities and actions, and define dynamic rules that map each name `x` to its definition site `d`.

For import declarations, Spoofax provides a generic `open-import` strategy (used in Figure 4.14, bottom). This strategy ensures that files are cached after they are parsed and avoids cycles in the dependency graph. The `open-import` strategy takes three strategy expressions as its arguments, that respectively resolve the filename of an import (`resolve-import`), parse the file (`parse-file`), and analyze the file (`declare-top`).

Scoped name analysis For scoped names, we base our analysis on the notion of consistent renaming [Waite and Goss, 1984], which is the task of renaming all names in a program such that they are unequal to all other names that do not correspond to the same declaration site. However, rather than directly changing the names in the tree, we add annotations that satisfy this uniqueness requirement. This way, the abstract syntax tree remains the same modulo annotations, aiding in debugging and traceability of analysis and code generation.

As an example, Figure 4.15 shows how an action definition in concrete syntax (top), abstract syntax (middle), and as abstract syntax with renaming annotations (bottom). After renaming, each local name combined with its annotation is globally unique: for instance, we can distinguish `"s"{"a_0"}` from any other identifier `s` defined elsewhere.

```
action add(s : Set<User>) {
  for (u : User in s) { ... }
}
```

```
Action("add",
  [Param("s", SetType(SimpleType("User")))],
  [ForAll("u", SimpleType("User"), Var("s"), ...)]
)
```

```
Action("add",
  [Param("s"{"a_0"}, SetType(SimpleType("User")))],
  [ForAll("u"{"b_0"}, SimpleType("User"),
          Var("s"{"a_0"}), ...)]
)
```

Figure 4.15 Consistent renaming of an action definition.

```
rename-top = alltd(rename)

rename: d@Param(x, t) → Param(y, t)                                    (A)
  where y := x{<new>};
        rules(
          VarDef   : y → d
          RenameId : x → y
        )

rename: Var(x) → Var(y)                                                (B)
  where y := <RenameId> x

rename: Action(f, p1*, s1*) → Action(f, p2*, s2*)                      (C)
  where {| RenameId:
           p2* := <rename-top> p1*;
           s2* := <rename-top> s1*
         |}

rename: d@ForAll(x, t, e, s1*) → ForAll(y, t, s2*)
  where {| RenameId:
           y := x{<new>};
           rules(
             VarDef   : y → d
             RenameId : x → y
           );
           s2* := <rename-top> s1*
         |}
```

Figure 4.16 Name analysis using renaming rules.

Figure 4.16 shows *renaming rules* that add consistent naming annotations to local variables. The first definition of `rename` (Figure 4.16 (A)) operates on action parameters. It replaces the name `x` of a parameter with the annotated name `x{<new>}`, where `<new>` generates a fresh, globally unique name. The `VarDef` dynamic rule is defined to map the annotated name to the definition site. The `RenameId` dynamic rule records the renaming for the current scope: for each following occurrence of `x`, it should given the name `y` (Figure 4.16 (B)). Any names that are not declared will not be renamed, and are reported as errors in the check stage as `VarDef` is never defined for them.

To respect the scopes of the language, we use *dynamic rule scopes* [Bravenboer et al., 2006b] to reflect scoping construct of the language. This Stratego feature takes the form `{| r: ... |}` and ensures that any definition of the

```
type-of: StringLit(x)        → SimpleType("String")
type-of: Entity(x, prop*)    → SimpleType(x)
type-of: Property(_, t, _)   → t
type-of: Param(x, t)         → t
type-of: ForAll(x, t, _, _)  → t
type-of: ForAllExp(t)        → SetType(t)
type-of: Var(x)              → <type-of> (<VarDef> x)
type-of: PropAccess(e, f)    → t
  where p := <lookup-property(|f)> (<type-of> e);
        t := <type-of> p
```

Figure 4.17 Typing rules.

```
lookup-property(|f):
  Entity(y, p*) → <fetch-elem(?Property(f,_,_))> p*

lookup-property(|f):
  SimpleType(y) → <lookup-property(|f)> (<EntityDef> y)

is-simple-type = is-primitive-type <+ is-entity-type

is-primitive-type: SimpleType(x) → <PrimitiveDef> x

is-entity-type: SimpleType(x) → <EntityDef> x
```

Figure 4.18 Helper rules for error checking.

dynamic rule `r` made directly or indirectly within the scope is no longer visible after the scope is exited. The rename rule for actions (Figure 4.16 (C)) uses this feature to scope variables declared within an action. As the `RenameId` for these variables is undefined at the end of the scope, the variables are no longer visible in any following actions. The last definition of `rename` combines local renaming and scoping for the `for` statement.

Type analysis Type analysis can be specified using rules that project a language construct to its type. The `type-of` rule maps language constructs to their type (Figure 4.17). Typing rules for literals generally specify a constant type (e.g., for `StringLit`). Typing rules for declarations can fetch the type from one of the subterms (e.g., for entities or properties). Other typing rules use name analysis rules to fetch a definition and then apply the basic `type-of` rules (e.g., for `Var`). For compound expressions such as `PropAccess`, we fetch the property definition corresponding to the expression and return its type. To fetch the property `f` of a given type we use the `lookup-property(|f)` helper rule, where the pipe (`|`) indicates that it receives a *term argument* `f` (as opposed to a strategy argument).

Other projection rules can test for various properties based on the name analysis. Figure 4.18 illustrates additional projection rules that are used in the `check` rules. Of these, the `lookup-property` rule is of particular interest. It uses the Stratego standard library strategy `fetch-elem` to try and fetch the property `f` from an entity definition. A second definition fetches the property given a type instead of an entity.

```
editor-resolve:
  (selected, pos, ast, path, project-path) → target
  where target := <decl-of> selected

decl-of: Var(x)        → <VarDef> x
decl-of: SimpleType(x) → <EntityDef> x
decl-of: Submit(x)     → <ActionDef> x
```

Figure 4.19 Rules for reference resolving.

4.4.6 *Reference Resolving and Occurrence Highlighting*

Reference resolving and occurrence highlighting are specified using a control rule that given an identifier, respectively looks up its declaration or all occurrences of that identifier. For reference resolving, this rule is shown in Figure 4.19. The rule is given the selected node in the tree, its position (specified as a list of child offsets), the decorated abstract syntax tree, and the file system paths. The same tuple is given to the control rules of all other semantic editor services (with the exception of the analysis and error checking control rule). Reference resolving is implemented by simply applying the dynamic rules from the name analysis stage to the selected identifier, using a `decl-of` helper rule to fetch the appropriate declaration. Occurrence highlighting can be specified in a similar fashion, but is omitted here for reasons of brevity. Both services rely fully on the name analysis specification and only require a small addition to implement the editor service.

4.4.7 *Content Completion*

Semantic content completion (sometimes called content assist) provides completion proposals based on the syntactic and semantic context of the expression that is being edited. Content completion can provide suggestions for globally visible names as well as locally scoped names. For this reason, we integrate content completion into the name analysis of Section 4.4.5.

An important aspect of our design for the content completion service is its interface to the semantic analysis. 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. One approach (as taken by IMP [Charles et al., 2009], for example) is to require the language developer to add special productions to the grammar that map these incomplete expressions to an abstract representation. Unfortunately, this means the developer has to do extra work and has to account for all cases where content completion may be expected in the grammar.

In our approach, we opt for a generic representation of incomplete syntactic expressions: we introduce an artificial `COMPLETION` term to the abstract representation at the point of the cursor. For example, if a variable is expected at the cursor, a `Var(COMPLETION("p"))` is placed at the corresponding point in the abstract representation, with `p` the (possibly empty) identifier prefix

```
rename-top = alltd(propose-completion <+ rename)

propose-completion:
  Var(COMPLETION(p)) → all-vars
  where all-vars := <all-keys-RenameId>;
        rules(ContentProposals: () → all-vars)

propose-completion:
  PropAccess(e, COMPLETION(p)) → p2*
  where Entity(x, p*) := <EntityDef>;
        p2*           := <map(property-name)> p*;
        rules(ContentProposals: () → p2*)

editor-complete:
  (selected, position, ast, path, project-path) → p*
  where editor-init;
        ast' := <declare-top> (<desugar-top> ast);
        p*   := <ContentProposals> ()
```

Figure 4.20 Content completion rules.

that was already typed in. Similarly, compound expressions can take a form like `PropAccess("blog",COMPLETION("p"))`.

In Figure 4.20 we define content completion for variables and property access expressions by extending the name analysis from Figure 4.16. We replace the definition of `rename-top` with one that applies `propose-completion` before it applies `rename`. The first definition of `propose-completion` provides completion suggestions for variables, by taking all *keys* of the `RenameId` dynamic rule. Stratego provides the derived function `all-keys-RenameId` to do this. As the rule is evaluated within the context of the analysis, this results in a list of all variables in the current scope, and thus all valid suggestions if a variable is expected at the cursor location. The `ContentProposals` dynamic rule stores the list. The second `propose-completion` rule provides completion suggestions for property access expressions, by fetching all property names and recording them in the dynamic rule. The control rule for content completion is called `editor-complete`. It first performs name analysis (extended with content completion support) and then fetches the value of the `ContentProposals` rule.

The editor descriptor specifies which control rule to use for content completion, and may specify a character sequence that automatically triggers content completion. For example, for NWL, content completion should be automatically triggered when the user types '.' or ':'.

4.4.8 *Transformations, Code Generation, and Views*

Based on the Stratego language, Spoofax supports transformations and code generation using concrete object syntax [Visser, 2002], allowing rules to be written using the concrete syntax of the language that is transformed as patterns. These patterns can be syntactically checked against the syntax of the language. The Stratego compiler converts the patterns to abstract syntax. Concrete syntax can be very effective for writing modular transformations that are divided into a series of small, separate transformation steps [Hemel

```
generate-java:
  (selected, position, ast, path, project-path) →
  ($[[path].java], <to-java> selected)

to-java:
  Entity(x, p*) →
  $[ class [x] {
        [p2*]
    }
   ]
  where p2* := <to-java> p*

to-java:
  Property(x, Type(t)) → $[
    private [t] [x];
    public [t] get_[x] { return [x]; }
    public void set_[x] ([t] [x]) { this.[x] = [x]; }
  ]
```

Figure 4.21 Code generation rules.

et al., 2009]. After transformation using concrete syntax, a pretty printer can produce text output from the result. We elaborate on concrete object syntax in Chapter 7.

Concrete object syntax can be contrasted to template engines, where no further transformation steps are possible after code is rewritten (unless the resulting text is re-parsed). Stratego also offers language support for template engine-like transformations, by providing indentation-safe string interpolation as a "quick and dirty" code generation solution. For the small NWL language we opted for this approach to code generation: after some desugaring and normalization, NWL is printed to Java, and no subsequent transformations are applied. Figure 4.21 shows code generation rules using string interpolation and the control rule for generating Java.

Transformations and code generation can be used as a basis for views. Views can be described using descriptors, and can have a title, a rule name, and a number of annotations:

```
builder: "Generate Java code" =
  generate-java (openeditor) (realtime)
builder: "Show abstract syntax" =
  generate-aterm (openeditor) (realtime) (meta)
```

This specification defines views to show Java code and the abstract syntax of a selection. The **(openeditor)** annotation specifies that an editor should be opened for the view (rather than just generating code and saving it a file). The **(realtime)** annotation specifies that the view should be updated in real-time as the source file is changed. Finally, the **(meta)** annotation specifies that the view should only be available to language engineers, not to end developers.

4.5 IMPLEMENTATION

The implementation of Spoofax is based on the Eclipse platform, allowing it to be used together with other, language-independent Eclipse plugins such as build management tools and other language-specific plugins.

For the definition of generic IDE components we make use of the IMP framework [Charles et al., 2007, 2009], which greatly simplified the construction of these components by abstracting over the Eclipse API and guiding us through the development process using wizards. Languages developed in our environment maintain compatibility with IMP. In addition to the service descriptor DSLs, it is also possible to use Java-based implementations using the IMP framework to implement components of a language.

The language workbench integrates the language-parametric editor service components based on IMP with the Spoofax libraries for parsing (JSGLR) and running Stratego (Stratego/J)[3]. As Spoofax/IMP – the official name of the workbench implementation – integrates nearly all of the libraries of the Spoofax project, the workbench is also simply called the Spoofax language workbench.

4.5.1 *Language-parametric Editor Services*

From an implementation point of view, only a small portion of an IDE or IDE plugin implementation is actually language-specific. A great part of the implementation of IDE components deals with accidental complexity that has little or no relevance to a particular language. IDE development frameworks such as IMP [Charles et al., 2007, 2009] and the DLTK [DLTK, 2007] simplify the implementation of these components by abstracting away from many of these details.

In Spoofax we implemented IDE components in a language-parametric fashion. Each component is written in Java, and interacts with the APIs for abstract syntax trees, tokens, text editor widgets, the file system, the parser and error recovery, etc. However, it does not include any specific knowledge of the language it is used for. Instead, the components interpret an editor descriptor that configures these language-specific parts. For each editor service, a factory class that reads the editor descriptor for that service, and instantiates the language-parametric implementation class for it. These classes use the standard Java collection classes such as hash tables to efficiently store and access the language configuration. For example, for the syntax highlighting service, we maintain hash tables that specify which formatting style to apply to which kind of character sequence. For parsing, we use a generated parse table that is dynamically loaded into the environment.

We use proxy classes to implement the Eclipse and IMP extension points for editor services, satisfying the static Eclipse/OSGi component model for loading plugins. These proxy classes invoke the corresponding factory class as services are first used, loading the parametrized implementation classes. When a language definition is changed, the proxy classes seamlessly load new services from the descriptors.

Parser technology and IDE components Key for rapid feedback in interactive environments is a parser that supports error recovery in order to parse files

[3] Available from `http://strategoxt.org/`. The architecture of the Java implementation of Stratego is described in Chapter 3.

with syntax errors and incomplete programs as a programmer edits the text. As an example, consider the file at the left of our screenshot in Figure 4.4. This file is not syntactically correct, since the last production does not have a symbol it matches to. To still provide content completion, the parser still has to produce a sensible abstract representation for the editor services to operate on. In Chapter 6 we describe language-generic error recovery techniques that we use in the Spoofax language workbench to address this issue.

Most code editors and IDEs implement syntax highlighting using regular expressions or a scanner. In Spoofax, we use a scannerless parser, and take a different approach: we construct a stream of artificial tokens *after* parsing rather than before parsing, based on the lexical structure of parsed sentences. We elaborate on the implementation of this approach in Section 6.8.4.

A number of editor services directly interact with the syntactic state of the parser. For instance, the bracket insertion service needs to determine if the cursor is inside a string literal or a comment. If that is the case, and the user types an opening bracket `{`, the service should not automatically insert the closing bracket `}`. One way to determine whether the cursor is inside a string or comment is simply having a language-specific "blacklist" of syntactic constructs where bracket insertion should be disabled. However, to do this in a language-agnostic way, we had to take a different approach. Instead, we analyze the active production at the selected character, which can be determined from the parse tree. If it is a lexical production, we can examine the lexical character class that can be parsed by it. If the character class includes the bracket that would be inserted (e.g., `}`), then that indicates that the bracket would become part of the lexical. For these cases, closing bracket insertion is disabled.

Another editor service that interacts directly with the parser is content completion. As we discussed in Section 4.4.7, programs are often in a syntactically incorrect state when a completion proposal is demanded. For example, an expression '`e.`' may be a property access expression in the making. If content completion is triggered at that point, we insert a placeholder identifier at the point of the cursor, forming an expression '`e.placeholder`', and parse the file again. This ensures that the expression is still parsed as a compound expression. Any missing semicolons, parentheses, etc. can be picked up by the regular error recovery rules. The placeholder also allows us to easily add an artificial `COMPLETION` constructor at the point where the placeholder appears, which is used for the content completion analysis of Section 4.4.7.

Interpretation versus compilation Using language-parametric editor services does not enforce an interpretative model. It is also possible to use language-parametric editor services with compiled Java classes that are dynamically loaded using classloaders. In fact, we use classloaders to load compiled Stratego specifications for semantic services. Using a compiled model can lead to improved runtime performance, but comes at the cost of compilation speed. For Stratego, developers can choose whether to interpret or compile specifications, depending on whether they want more agile development or if they want to distribute the plugin to "end developers." A similar approach could

also be used for syntactic editor services, but we feel that the trade-off of compilation time versus performance for those services may be unacceptable: we have never seen performance problems for those services based on an interpretative model.

4.5.2 *Semantic Services and Rewrite Rules*

The Stratego language was originally compiled to C, but we developed a more flexible back end that compiles it to Java and interoperates with Eclipse, described in Chapter 3. Where the C-based Stratego operates only on the ATerm data type of the ATerm library [van den Brand et al., 2000], the Java implementation is more flexible and can transform any tree structure that implements the `IStrategoTerm` interface. Using adapter classes, any Java data type can be made to implement it. This approach is called the program object model adapter (POM) approach. Previous experience with a Stratego interpreter on Java showed that this approach allows rule-based transformations to interoperate with Java-based frameworks such as a compiler front ends [Kalleberg and Visser, 2007a].

We use the POM adapter approach to transform trees that maintain position and layout information. By using a custom *factory class* used to construct new `IStrategoTerm` instances, we implemented a form of origin tracking [van Deursen et al., 1993] for Stratego. Origin tracking is key to concise, position-agnostic specification of transformations and analyses in Spoofax.

The term factory class is used to construct new terms by compiled and interpreted Stratego programs. Term factories have methods to *create* new terms given a constructor name/and or children or by parsing them from files [Kalleberg and Visser, 2007a]. We added methods to *replace* subterms of a term. Terms are functional, immutable data types, so by default these methods simply return a new term with the given subterms. In Spoofax, we return a new term that maintains a link to the original term and its children. We changed the implementation of the Stratego traversal operators `all`, `some`, and `one` to use the new "replace" methods. This way, all strategies that use these combinators (such as `alltd` and `collect-all`, shown in this chapter) support origin tracking.

4.5.3 *Editor Extensibility and Customization*

Language workbenches provide an integrated development environment for building a language and IDE support, abstracting over low-level IDE implementation details. Still, it can be useful to have a "backdoor" to escape the environment and add features that (we) the developers of the workbench had not anticipated.

Spoofax language definitions can be extended using the standard Eclipse extension model. New components can be added to plugins simply by declaring them in the `plugin.xml` descriptor file and implementing them in Java. Since we base our implementation on IMP, helpful IMP wizards can even

generate skeletal Java implementation for common editor services. Once declared, Java implementations of services can even override the Spoofax implementations. Drawbacks of adding services this way are that they do not provide the same abstractions as DSLs would, and that they tie developers to the standard Eclipse way of plugin development: custom, Java-based services can only be used in a secondary Eclipse instance.

4.6 EXPERIENCE

We have substantial experience in using Spoofax and languages created with Spoofax. Several languages are part of the standard distribution, and are used in the development of new languages:

- The SDF language. While this language was implemented before the Spoofax workbench, we specified the static semantics of SDF using the idioms in this chapter to support editor services such as views on syntax productions and content completion.
- The ESV editor descriptor language of Section 4.3 and 4.4.
- The Stratego language. We described the static semantics to provide full-featured Stratego IDE support.
- The ATerm language [van den Brand et al., 2000], used for abstract syntax. In addition to the Spoofax specification, we added special support for views based on the source file of an ATerm file (e.g., allowing "generate Java code" to be applied to an ATerm created from an NWL file).

Spoofax has also been used, by ourselves and others, to develop a variety of new languages covering different domains. Some have been developed independently of our environment before Spoofax was available, and have used Spoofax to describe the syntax and static semantics for IDE integration. A selection:

- Acoda [Vermolen et al., 2011] is a tool set that uses DSLs to aid in data migrations as data models evolve. Given different versions of a data model, it can automatically derive textual, editable evolution traces that can be executed to perform data migration. The data model editor uses views to generate these traces and supports the full range of editor features for editing them.
- PIL [Hemel and Visser, 2009], a Java-like, object-oriented programming language that targets multiple software platforms.
- Mobl [Hemel and Visser, 2011], a DSL for web-based mobile phone applications. It generates HTML/Javascript code when files are saved, allowing them to be directly tested in a browser.
- SugarJ [Erdweg et al., 2011a,b], an extensible language based on Java, Stratego, and SDF.

- A language for testing Spoofax language definitions, described in Chapter 8. This language syntactically and semantically embeds a language under test and evaluates test cases in the editor as they are edited.

- An assembly language for Java bytecode (based on the language of Chapter 2), used mainly as a pedagogical tool for inspecting and creating Java class files, or to directly generate bytecode from other Spoofax languages.

- NWL (shown in this chapter), and WebDSL [Groenewegen et al., 2010], a DSL for the Web. Hemel et al. [2009, 2011] describe the implementation of WebDSL and its support for static consistency checking.

- Aster (described in Chapter 5), an attribute grammar language based on strategies for description of attribute propagation patterns.

Observations To describe a few key lessons learned from our experience, Spoofax allows for a much more agile development model than was previously possible with separate tools for syntax definition, meta-programming, and especially IDE development. As languages are developed they are directly usable in the IDE from the point that a syntax is defined. From there, developers can incrementally add additional functionality, from presentational editor services to semantic analyses and code generation. Not all languages listed above have yet reached the level of maturity where they incorporate the full set of services from Figure 4.2, but they still provide an improved user experience compared to a standard text editor. Using Spoofax also changed the deployment model of the languages: they can now be distributed as Eclipse plugins instead of separate command-line compilers, which tends to be much more appealing to "end developers."

In our experience the Spoofax workbench significantly lowers the bar for creating a new language by providing an interactive environment for development and playing with it as it evolves. We have developed various experimental languages to try out new language concepts, and have been using it for the past two years in courses on compiler construction and model-driven software engineering. While meta-programming naturally comes with a learning curve, Spoofax allows students to quickly experiment with language features and experience all aspects of language development in a unified environment. For the course on compiler construction, we use the bytecode language to allow students to experiment with stack architectures and abstractions over a low-level language. Based on the testing language of Chapter 8, they can do test-driven language development, and checking assignments can be partially automated.

Customization Based on the Eclipse platform, Spoofax language plugins are open to various forms of extensions and integration with other plugins. In the WebDSL plugin we particularly made use of that, and integrated the plugin with the Eclipse Web Tools Platform for testing and deploying web applications. The plugin also provides its own wizard that configures database and

deployment configuration. These customizations have been implemented as custom Java code, directly embedded in the plugin itself. The WebDSL plugin also integrates with the Acoda plugin: Acoda can extract data models from web applications, show these in a view, and then derive evolution traces by comparing data models.

Evolution As languages evolve with new insights, new constructs may be added and others removed, which can lead to incompatibilities with older programs. One way Spoofax assists in migrating older DSL programs to newer revisions of a language is through the notion of deprecated syntax. Syntactic constructs marked deprecated are displayed with a warning in the editor, and may be transformed to updated constructs using desugaring rules (Section 4.4.2) or by a migration transformation that processes these constructs and produces a migrated program. An area of future work is to provide tool support to assist in these migrations, much like Acoda can do for data models.

4.7 DISCUSSION AND RELATED WORK

Spoofax is a language workbench for textual domain-specific languages. Textual languages benefit from integration with standard, text-based version control systems and issue trackers, easy importing and exporting of files from other tools (avoiding vendor lock-in), files that are editable with other tools, and free text editing. Notable other tools for creating and using textual DSLs are the Meta-Environment [Klint, 1993; van den Brand et al., 2001], EMFText [Heidenreich et al., 2009a], MontiCore [Krahn et al., 2008], TCS [Jouault et al., 2006], TEF [Scheidgen, 2010], and Xtext [Efftinge and Voelter, 2006]. A thorough comparison of a number of these tools has recently been provided by Goldschmidt et al. [2008] and by Pfeiffer and Pichler [2008]. In our discussion we contrast these tools to Spoofax.

The Meta-Environment is a platform for language development, source code analysis, and source code transformation [Klint, 1993; van den Brand et al., 2001]. It includes SDF, the ASF term rewriting language, and provides an IDE framework written in Java. The Meta-Environment derives basic syntax highlighting from SDF grammars. ASF tree-traversal may also be used to annotate the AST with coloring directives. ASF is also used to specify the typing rules of the language, and may include custom error messages, presented in a separate view. The original implementation of the Meta-Environment supported the SEAL language [Koorn, 1993], allowing editors to be extended with additional widgets. While these could interact with the editor, they could only do so if it was in a syntactically consistent state, lacking the interactivity of modern parser-based editor services.

Compared to the Stratego language, ASF has limited expressive power. ASF is a pure term rewriting language, whereas Stratego adds strategies [Visser et al., 1998] and dynamic rules [Bravenboer et al., 2006b], which have been key for concise, reusable analyses in Spoofax. In Spoofax we use origin tracking for position-agnostic specification of editor services. Origin tracking was

originally implemented as a prototype for ASF [van Deursen et al., 1993], but is no longer used in the current version.

The EMFText [Heidenreich et al., 2009a], TEF [Scheidgen, 2010], and TCS [Jouault et al., 2006] workbenches take a different approach to syntax development than we do in Spoofax. Rather than combining the specification of concrete and abstract syntax into a single grammar, they start with the construction of a metamodel. TCS and EMFText have a generic concrete syntax that can be derived from a metamodel, at the cost of domain-specific notation. In EMFText, developers also have the option to write their own grammar with productions that map to elements of the metamodel. On the one hand, this approach allows the use of existing EMF metamodels for building new languages, which can be useful in certain cases. On the other hand, the approach is also less agile than using a syntax formalism that integrates abstract and concrete syntax: it introduces redundancies in the grammar, and requires the syntax to be maintained in two places. The three workbenches provide error markers, reference resolving, and content completion that can be customized using Java, but none of the presentation and editing services of Figure 4.2.

Xtext [Efftinge and Voelter, 2006] is also based on EMF metamodels, but integrates the specification of concrete syntax and metamodel into a single grammar specification. Xtext was originally part of openArchitectureWare [Efftinge et al., 2008], and used the OCL-like Xtend language for model transformations and for the specification of problem markers using constraints and for reference resolving. Recently, Xtext has been reimplemented based on TMF[4]. It now relies on grammar annotations and uses Java code to describe the static semantics of languages and all but the most basic editor services.

Abstract representation The abstract representation of programs in Spoofax is based on trees, where dynamic rules superimpose graph structures. In contrast, EMF is based on graphs. EMFText and Xtext allow annotations in grammar productions to specify simple use-def relations between productions that introduce back-edges to the abstract syntax tree, effectively producing graphs. This approach of syntactic name resolution only works for names with a global scope (or lexical scopes in the case of EMFText). It is inadequate for compound expressions (e.g., `blog.author`). EMFText and Xtext resort to imperative Java code to specify these relations, or, in the latest version of Xtext, the Java-like Xtend language. In Spoofax, the name analysis is fully specified in a rule-based Stratego specification. An interesting future step for Spoofax could be to map abstract syntax trees and dynamic rules to EMF models, and express constraints over such models with Stratego.

MontiCore [Krahn et al., 2008] generates custom Java classes to represent the abstract syntax. It supports basic presentational services, which are specified as grammar properties. Syntax coloring is specified as lists of keywords to highlight. Pre-defined (Java-style) comments are supported. Folding is specified by a list of non-terminals. For semantic editor services, MontiCore

[4] `http://www.eclipse.org/modeling/tmf/`.

grammars specify events, which may be specialized with user-defined Java classes.

Parsing Interestingly, with the exception of the Meta-Environment and TEF, all tools that we described generate ANTLR parsers. TEF uses a the RunCC parser generator. ANTLR's LL(k) or LL(*) parsers cannot cope with left recursion in grammars. Likewise, RunCC's LR(k) parsers are limited to a subset of the context-free grammars. This means that they are not closed under composition, which means that adding extensions to a grammar or reusing grammars can introduce conflicts in the parser [Kats et al., 2010]. These parsers also rely on using a single, separate scanner, which means that adding extensions with a different lexical syntax is not possible. In contrast, Spoofax and the Meta-Environment use SDF to specify grammars and generate scannerless generalized LR (SGLR) parsers. Based on SGLR parsing, SDF grammars can be freely composed, allowing for embedded languages and language extensions.

Dynamically loading editor services One area in which Spoofax excels compared to other tools is in supporting agile language design and development. Languages can be incrementally developed and changes can be dynamically loaded into the environment. Spoofax automatically derives syntactic editor services from the syntax definition. Editor services can then be selectively customized as desired. In contrast, the other tools often provide generic syntax highlighting for keywords and hardcoded default symbols such as strings, but do not derive highlighting or other services based on analysis of the grammar. Of the other tools, only the Meta-Environment and TCS have the ability to dynamically reload language definitions without requiring the editor environment to be relaunched. In the case of the Meta-Environment, this comes at the cost of the idiosyncrasies of a custom IDE environment rather than integration into a common language platform such as Eclipse. TCS integrates into Eclipse, but is primarily designed for adding a textual syntax to languages that have a model-based principal representation. While it allows free text editing, it does not allow the same flexibility in language design as other workbenches where language engineers can write their own grammar.

We use the Stratego language for analyses, transformations, and code generation. Other workbenches use Java classes for analysis, sometimes combined with grammar annotations, Java visitors for transformations, and template engines for code generation. Based on strategies and rewrite rules, Stratego concisely specify analyses and transformations. Providing a choice of concrete object syntax (ensuring syntactic correctness) [Visser, 2002] and string interpolation, Stratego also provides a flexible solution for code generation.

Meta-programming languages Other notable meta-programming languages include ASF+SDF [Klint, 1993], JastAdd [Hedin and Magnusson, 2003], Rascal [Klint et al., 2009], and TXL [Cordy et al., 1991]. They provide limited IDE support at this point, although the developers of JastAdd and Rascal are actively working on IDE support dedicated to their meta-programming languages. However, much like the original Spoofax Stratego editor [Kalleberg

and Visser, 2007b], they do not provide a language workbench solution for languages developed with them. Like Stratego, ASF+SDF and TXL support a form of concrete object syntax, but they do not support string interpolation. Only Rascal supports both forms of code generation. JastAdd relies on Java visitors on the abstract syntax tree to generate code.

Homogeneous language embeddings Our work has focused on heterogeneous formalisms for language definition, where a meta-language is used to define or extend a separate object language. In contrast, with homogeneous language embedding systems, the meta-language is used to extend itself [Hudak, 1998; Tratt, 2008]. Examples include Converge [Tratt, 2008], a multi-stage dynamically typed language; Template Haskell [Sheard and Jones, 2002], a Haskell-based meta-programming system based on compile-time generative programming; and Helvetia [Renggli et al., 2010], a Small-talk based system. Of these, Helvetia is notable for providing extensible IDE support in addition to an extensible language.

On the one hand, homogeneous approaches can help interoperability between DSLs by ensuring they share a common basis. On the other hand, DSLs developed are also limited by their host language, in terms of its type system (or lack thereof), representation of runtime values, the execution model, and the runtime platform. For example, a language such as mobl [Hemel and Visser, 2011], which targets a combination of JavaScript and HTML5 and introduces its own type system, cannot be expressed using a homogeneous approach.

4.8 OPEN ISSUES AND FUTURE WORK

In this chapter we showed that the Stratego language can be used for concise specifications of analysis, transformations, and code generation. Still, many other meta-programming languages exist that each have their own merits and uses. We believe that Spoofax has the potential to become a common, open platform for hosting multiple meta-programming languages. Spoofax defines a lightweight, technology-agnostic interface between editor services and semantic analyses. Analysis specifications in other meta-programming languages could follow the same interface, and to some degree may follow the same idioms for reuse, allowing them to implement the same set of editor services. One constraint is the representation of the abstract syntax, which may require marshaling to another form. Another is our use of origin tracking in the specification of analyses. Without origin tracking, semantic specifications would have to explicitly ensure that analysis results contain position information.

Another meta-language for analyses and transformations that we have used in Spoofax is the Aster attribute grammar language (Chapter 5). As the language is based on Stratego, Aster can be used in Spoofax based on the same interface and data structures. Still, more experience is needed to identify patterns in the paradigm of decorated attribute grammars for efficient specification of editor services and complete DSLs.

Another area of future work is in providing tool support for first-class modular language components. Based on the modular SDF syntax formalism and SGLR parsing, it is possible to decompose languages into separate, reusable components [Bravenboer and Visser, 2004]. As an example, WebDSL [Groenewegen et al., 2010] reuses the syntax definition of HQL for queries. Such compositions still require much manual effort in terms of managing and building project sources, as Spoofax provides only limited tool support for these forms of reuse. One ideal scenario may be that developers of a language could just pick and match the language features they need, and that the environment composes them. Composition at the semantic level poses additional challenges. Based on a uniform type system and a single host language, it is possible to combine homogeneous language components. Notably, Helvetia [Renggli et al., 2010], MPS [Voelter and Solomatov, 2010], and SugarJ [Erdweg et al., 2011a,b] rely on these properties. However, if a different host language or type system is used, reuse is currently limited to the syntactic level.

4.9 CONCLUSION

Modern IDEs increase developer productivity by incorporating many different kinds of editor services specific to the syntax and semantics of a language. They assist developers in understanding and navigating through the code, they direct developers to inconsistent or incomplete areas of code, and they even help with editing code by providing automatic indentation, bracket insertion, and content completion. As a consequence, developers that have grown accustomed to these services are growing less accepting of languages that do not have solid IDE support.

To efficiently develop languages with IDE support, Spoofax supports selective, incremental development of editor services that can be dynamically loaded, evaluated, and tuned in the same environment. Using high-level languages to specify the syntax and semantics of a language, it provides a language development solution that greatly increases productivity of language engineers in building language and IDE components compared to using handwritten components or separate language engineering tools.

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 would like to thank Karl Trygve Kalleberg for his many contributions to the Spoofax project; the development teams of IMP, SDF, (J)SGLR, and Stratego/XT for their valuable efforts; Maartje de Jonge and Emma Söderberg for their contributions to error recovery with JSGLR; Rob Vermaas, Bernhard Merkle, and the anonymous reviewers for suggestions for an earlier version of this chapter; and finally our users for providing continuous feedback and coming up with interesting new ideas.

Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming

5

ABSTRACT

Attribute grammars are a powerful specification formalism for tree-based computation, particularly for software language processing. Various extensions have been proposed to abstract over common patterns in attribute grammar specifications. These include various forms of copy rules to support non-local dependencies, collection attributes, and expressing dependencies that are evaluated to a fixed point. Rather than implementing extensions natively in an attribute evaluator, we propose *attribute decorators* that describe an abstract evaluation mechanism for attributes, making it possible to provide such extensions as part of a library of decorators. Inspired by strategic programming, decorators are specified using generic traversal operators. To demonstrate their effectiveness, we describe how to employ decorators in name, type, and flow analysis.

5.1 INTRODUCTION

Attribute grammars are a powerful formal specification notation for tree-based computation, particularly for software language processing [Paakki, 1995], allowing for modular specifications of language extensions and analyses. At their most basic, they specify declarative *equations* indicating the functional relationships between *attributes* (or properties) of a tree node and other attributes of that node or adjacent parent and child nodes [Knuth, 1968]. An *attribute evaluator* is responsible for scheduling a tree traversal to determine the values of attributes in a particular tree.

Attribute grammars are nowadays employed in a wide range of application domains and contexts. To extend their expressivity for use in particular domains, and to abstract over commonly occurring patterns, basic attribute grammars have been extended in many ways, in particular supporting attribution patterns with non-local dependencies. For example, *remote attribution* constructs allow equations that refer to attributes of nodes arbitrarily far above or below the node for which they are defined [Boyland, 2005; Kastens and Waite, 1994]. *Chain attributes* express a dependence that is threaded in a left-to-right, depth-first fashion through a sub-tree that contains definitions of the chain value [Kastens and Waite, 1994]. *Self rules* provide a local copy of subtrees, which may be adapted for tree transformations [Baars et al., 2003].

More generally, *collection attributes* enable the value of an attribute of one node to be determined at arbitrary other nodes [Boyland, 2005; Magnusson et al., 2007]. A different kind of remote attribute is provided by *reference attribute grammars* that allow references directly to arbitrary non-local nodes and their attributes [Hedin, 2000], allowing for attributes that look up a particular node or collection of nodes. Finally, some attribute grammar systems support equations with *circular dependencies* that are evaluated to a fixed point [Boyland, 1996; Magnusson and Hedin, 2007].

All of these extensions aim to raise the level of abstraction in specifications, by translation into basic attribute grammars or by using an extended evaluator. Unfortunately, each of these extensions has been designed and implemented separately and is hardwired into a particular attribute grammar system. Potential users may find that a particular system does not provide the set of demanded extensions. Adding new abstractions is non-trivial, since it requires modification of the attribute evaluation system itself. For example, it can sometimes be useful to thread attribute values from right-to-left (e.g., when computing backward slices or use-def relations between variables). In a system with only left-to-right chained attributes, this dependence must be encoded using basic attribute equations, despite the similarity of the abstractions.

In his OOPSLA'98 invited talk, "Growing a Language" [Steele, 1999], Guy Steele argued that "languages need to put the tools for language growth in the hands of the users," providing high-level language features that abstract over various extensions, rather than directly providing language features to solve specific problems. To this effect, we propose *attribute decorators* as a solution for the extensibility problem of attribute grammar specification languages. A decorator is a generic declarative description of the tree traversal or evaluation mechanism used to determine the value of an attribute. Decorators augment basic attribute equations with additional behavior, and can provide non-local dependencies or a form of search as required. For instance, a decorator can specify that the value of an attribute is to be sought at the parent node (and recursively higher in the tree) if it is not defined at the current node. Decorators can also enhance the usability of attribute equations for specific domains, separating the generic behavior from specific equations such as type checker constraints or data-flow equations, supported in other systems through specialized extensions.

In this chapter, we present Aster, a system for *decorated attribute grammars*. Decorators are sufficiently powerful to specify all of the attribute grammar extensions listed above, avoiding the need to hardwire these into the system. A library of decorators suffices to cover common cases, while user-defined, domain-specific decorators can be used for specific applications and domains.

Decorators are inspired by *strategic programming*, where generic traversal strategies enable a separation between basic rewrite rules defining a tree transformation and the way in which they are applied to a tree [Visser et al., 1998; Lämmel et al., 2003]. In our case, local attribute equations define the core values of a tree computation, while decorators describe how those values are

combined across the tree structure. The Aster specification language is built as an extension of the Stratego strategic programming language [Bravenboer et al., 2008]. We reuse the generic traversal operators of Stratego for the specification of decorators, and its pattern matching and building operations as the basis for attribute equations.

The contributions of this chapter are as follows.

- The decorated attribute grammars programming paradigm, using decorators for application-level extensibility of attribute grammars.
- The identification of primitives for the specification of decorators to define abstract evaluation strategies for attributes.
- The implementation of Aster, a decorated attribute grammar system.[1]
- A collection of decorators for functionality provided by existing AG extensions and new abstractions specific to name, type, and flow analyses.

Outline We begin this chapter with background on attribute grammars and introducing our basic notations. Section 5.3 defines decorators, showing how they augment basic equations and capture common patterns. In Section 5.4 we present typical language engineering applications, demonstrating how decorators can be effectively applied in this area. In Section 5.5, we study the application of Aster to a tool for grammar analyses and transformations. We briefly outline our implementation in Section 5.6. In this extended version of the original paper [Kats et al., 2009c], Section 5.4.4 elaborates on data-flow analysis and circular attribute evaluation.

5.2 ATTRIBUTE GRAMMARS

As they were originally conceived, attribute grammars (AGs) specify dependencies between attributes of adjacent tree nodes [Knuth, 1968]. Attributes are generally associated with context-free grammar productions. For example, consider a production `X ::= Y Z`. Attribute equations for this production can define attributes for symbols `X`, `Y` and `Z`. Attributes of `X` defined at this production are called *synthesized*, as they are defined in the context of `X`. They can be used to pass information upwards. Conversely, attributes of `Y` and `Z` defined in this context can be used to pass information downwards, and are called *inherited* attributes.

5.2.1 *Pattern-Based Attribute Grammars*

In this chapter we adopt a notational variation on traditional AGs in which attribute equations are associated with tree or term patterns instead of grammar productions [Farnum, 1992; Boyland and Graham, 1994]. Trees can be denoted with prefix constructor *terms* such as `Root(Fork(Leaf(1), Leaf(2)))`. Tree

[1] Available from `http://strategoxt.org/Stratego/Aster/`, and provided as part of Spoofax at `http://spoofax.org`.

```
1  eq Root(t):
2     t.global-min := t.min
3     id.min       := t.min
4     id.replace   := Root(t.replace)

5  eq Fork(t1,t2):
6     t1.global-min := id.global-min
7     t2.global-min := id.global-min
8     id.min        := <min> (t1.min, t2.min)
9     id.replace    := Fork(t1.replace, t2.replace)

10 eq Leaf(v):
11    id.min     := v
12    id.replace := Leaf(id.global-min)
```

Figure 5.1 An attribute grammar specification for *repmin* in pattern major form.

```
1  eq min:
2    Root(t)     → t.min
3    Fork(t1,t2) → <min> (t1.min, t2.min)
4    Leaf(v)     → v

5  eq global-min:
6    Root(t).t      → id.min
7    Fork(t1,t2).t1 → id.global-min
8    Fork(t1,t2).t2 → id.global-min

9  eq replace:
10   Root(t)     → Root(t.replace)
11   Fork(t1,t2) → Fork(t1.replace, t2.replace)
12   Leaf(v)     → Leaf(id.global-min)
```

Figure 5.2 An attribute grammar specification for *repmin* in attribute major form.

patterns for matching and construction are terms with variables (indicated in italics throughout this chapter), e.g. `Fork(`t_1`,`t_2`)`.

Basic attribute equations have the form

eq p`:` r`.`a `:=` v

and define equations for a term that matches pattern p, where attribute a with a relation r to the pattern has value v. The relation r can be a subterm of p indicated by a variable or the term matched by the pattern itself, indicated by the keyword `id`. An equation for a pattern p can include multiple attribute relation definitions of the form r`.`a `:=` v.

As an example, consider the transformation known as Bird's *repmin* problem [Bird, 1984], which can be well expressed as an AG, as illustrated in Figure 5.1. In this transformation, a binary tree with integer values in its leaves is taken as the input, and a new tree with the same structure and its leaves replaced with the minimum leaf value is produced as the output. For example, the tree `Root(Fork(Leaf(1),Leaf(2)))` is transformed to `Root(Fork(Leaf(1),Leaf(1)))`.

In the specification of Figure 5.1, the local minimum leaf value in a subtree is computed in the synthesized attribute `min` (lines 3, 8 and 11). At the top of the tree, the minimum for the whole tree is copied to the inherited `global-min` attribute (line 2), which is then copied down the tree to the leaves

(lines 6 and 7). Finally, the `replace` attribute constructs a tree where each leaf value is replaced by the global minimum (lines 4, 9, 12).

Attribute equations are often defined in sets that share a common pattern, but may also be grouped to define a common attribute, which can make it easier to show the flow of information at a glance. Consider Figure 5.2, which is equivalent to the specification in Figure 5.1, but organizes the equations per attribute instead. Equations can be defined in separate modules, across different files, and are automatically assembled into a complete specification. Thus, language definitions can be factored per language construct and/or per attribute to support modular, extensible language definitions [Hedin and Magnusson, 2003; Van Wyk et al., 2006].

Using patterns helps separation of concerns when specifying a syntax and AG analyses. However, it can still be useful to use the concrete syntax of a language. Aster supports this using the generic approach of *concrete object syntax embedding* as described in [Visser, 2002]. For example, instead of a pattern `While(`*e*`,`*s*`)`, we can use a concrete syntax pattern, which is typically enclosed in "semantic braces":

```
eq |[ while (e) s ]|:
  id.condition = e
```

Concrete syntax patterns are parsed at compile-time, and converted to their abstract syntax equivalents. Section 5.4 includes further examples of this technique.

5.2.2 *Copy Rules*

In theory, basic attribute equations with local dependencies are sufficient to specify all non-local dependencies. Non-local dependencies can be encoded by passing context information around using local inherited and synthesized attributes. In the repmin example, this pattern can be seen in the definition of the global minimum value, which is defined in the root of the tree. This information is passed down by means of so-called *copy rules*, equations whose only purpose is to copy a value from one node to another.

To accommodate for the oft-occurring pattern of copying values through the tree, many AG systems provide a way to *broadcast* values, eliminating the need for tedious and potentially error-prone specification of copy rules by hand. For example, the repmin example can be simplified using the `including` construct of the GAG and LIGA systems [Kastens and Waite, 1994], which provide a shorthand for specifying copy rules. Using this construct, the copy rules in Figure 5.1, lines 6 and 7 could be removed and line 12 replaced by `id.replace := Leaf(including Root.global-min)`, specifying that the value is to be copied downward from the `Root` node.

5.3 DECORATORS

While constructs such as `including` provide notational advantages for some specifications, they cannot be used if the desired pattern of attribution does

```
def down       global-min
def up         min
def rewrite-bu replace

eq Root(t):
  t.global-min := id.min

eq Fork(t1,t2):
  id.min       := <min> (t1.min, t2.min)

eq Leaf(v):
  id.min       := v
  id.replace   := Leaf(id.global-min)
```

Figure 5.3 Repmin using decorators.

not precisely fit their definition. These notations are built into AG systems, and as such a developer is faced with an all-or-nothing situation: use a nice abstract notation if it fits exactly or fall back to writing verbose copy rules if there is no suitable shorthand. This section proposes attribute decorators as a more flexible alternative to building these shorthand abstractions into the AG system. Decorators can be defined to specify how attribute values are to be propagated through the tree. Common patterns such as `including` can be provided in a decorator library, while user-defined decorators can be written for other cases.

To define high-level attribute propagation patterns, we draw inspiration from strategic programming [Visser et al., 1998; Lämmel et al., 2003]. This technique allows the specification of traversal patterns in a generic fashion, independent of the structure of a particular tree, using a number of basic, generic traversal operations.

5.3.1 *Basic Attribute Propagation Operations*

Consider the specification of Figure 5.3. It specifies only the principal repmin equations, avoiding the copy rules. The flow of information is instead specified using decorators (at the top of the specification). For instance, `global-min` uses the `down` decorator, which specifies that values should be copied downwards. Before we elaborate on the decorators used in this example, let us first examine the unabbreviated set of equations and reduce them to a more generic form that uses elementary propagation operations. After this, we will show how these operations can be used in the specification of decorators.

Downward propagation of the `global-min` attribute, first defined at the root of the tree (as seen in Figure 5.3), was originally achieved by

```
eq Fork(t1,t2):
  t1.global-min := id.global-min
  t2.global-min := id.global-min
```

Another reading of this specification says that 'the `global-min` of any non-root term is the `global-min` of its parent.' Thus, if we can reflect over the tree structure to obtain the *parent* of a node, we can express this propagation as

```
eq Fork(t1,t2):
  id.global-min := id.parent.global-min

eq Leaf(v):
  id.global-min := id.parent.global-min
```

This notation makes the relation to the parent node's attribute value explicit, rather than being than implied by the context. It forms the basis of specifying the downward propagation in a more generic way: `id.parent.global-min` could be used as the default definition of `global-min`, used for nodes where no other definition is given (here, all non-root nodes). This is essentially what the `down` decorator in Figure 5.3 does.

A different form of propagation of values was used in the `replace` attribute:

```
eq replace:
  Root(t)     → Root(t.replace)
  Fork(t1,t2) → Fork(t1.replace, t2.replace)
```

Here we can recognize a (common) rewriting pattern where the node names remain unchanged and all children are replaced. We abstract over this using the `all` operator:

```
eq replace:
  Root(t)     → all(id.replace)
  Fork(t1,t2) → all(id.replace)
```

`all` is one of the canonical *generic traversal operators* of strategic programming [Visser et al., 1998; Lämmel et al., 2003]. It applies a function to all children of a term. Other generic traversal operators include `one`, which applies a function to exactly one child, and `some`, which applies it to one or more children. In this case, we pass `all` a reference to the `replace` attribute. This reveals an essential property of attribute references in Aster: they are *first-class citizens* that can be passed as the argument of a function in the form of a closure. The expression `id.replace` is a shorthand for a closure of the form $\lambda t \rightarrow (t.\texttt{replace})$. It can be applied to the current term in the context of an attribute equation or in a sequence of operations, or to a term t using the notation $<f> t$.

5.3.2 *Attribute Propagation using Decorators*

We implement attribute definitions using functions that map terms to values. Parts of such a function are defined by attribute equations. Some attribute definitions form only a partial function, such as those in Figure 5.3. In that figure, copy rules are implicitly provided using decorators. Decorators are essentially higher-order functions: they are a special class of attributes that take another attribute definition (i.e., function) as their argument, forming a new definition with added functionality. This means that the declaration `def down global-min` and the accompanying equations for the `global-min` attribute effectively correspond to a direct (function) call to decorator `down`:

```
eq Root(t):
  t.global-min := id.down(the original global-min equations)
```

```
  decorator down(a) =
(1) if a.defined then
(2)   a
    else
(3)   id.parent.down(a)
    end

  decorator up(a) =
    if a.defined then
      a
    else
      id.child(id.up(a))
    end
```

```
decorator rewrite-bu(a) =
  all(id.rewrite-bu(a))
; if a.defined then
    a
  end

decorator down_at-root(a) =
  if not(id.parent) then
    a
  else
    fail
  end
```

Figure 5.4 Basic decorator definitions.

A basic decorator d decorating an attribute a is specified as follows:

```
decorator d(a) = s
```

The body s of a decorator is its evaluation strategy, based on the Stratego language [Bravenboer et al., 2008]. It provides standard conditional and sequencing operations. Using *generic traversal operators*, the evaluation strategy can inspect the current subtree. These operators are agnostic of the particular syntax used, making decorator definitions reusable for different languages. In this chapter, we introduce the notion of *parent references* as an additional generic traversal operator, in the form of the `parent` attribute. Furthermore, we provide a number of *generic tree access attributes* that are defined using these primitives, such as the `prev-sibling` and `next-sibling` attributes to get a node's siblings, and `child(c)` that gets the first child where a condition c applies. Finally, we introduce *reflective attributes* that provide information about the attribute being decorated. These include the `defined` attribute, to test if an attribute equation is defined for a given term, and the `name` and `signature` attributes to retrieve the attribute's name and signature.

To illustrate these operations, consider the definition of the `down` decorator, which defines downward propagation of values in the tree (see Figure 5.4). This decorator automatically copies values downwards if there is no attribute equation defined for a given node. It checks for this condition by means of the `defined` reflective attribute (1). In case there is a matching equation, it is simply evaluated (2). Otherwise, the decorator acts as a copy rule: it "copies" the value of the parent. For this it recursively continues evaluation at the parent node (3). Conversely, the `up` decorator provides upward propagation of values. If there is no definition for a particular node, it inspects the child nodes, eventually returning the first successful value of a descendant node's attribute equation.

The `rewrite-bu` decorator provides bottom-up rewriting of trees, as we did with the `replace` attribute. Using the `all` operator, it recursively applies all defined equations for an attribute, starting at the bottom of the tree. Rewrites of this type produce a new tree from an attribute, which in turn has attributes of its own, potentially allowing for staged or repeated rewrites.

```
Program    ::=  Function*
Function   ::=  function ID(Arg*) { Stm* }
Stm        ::=  { Stm* }
             |  if (Expr) Stm else Stm
             |  while (Expr) Stm
             |  var ID :  Type | ID := Expr
             |  return Expr
Type       ::=  IntType | ...
IntType    ::=  int
Arg        ::=  ID : Type
Expr       ::=  Int | Var | ID(Expr*) | Expr + Expr | Expr * Expr
Int        ::=  INT
Var        ::=  ID
```

Figure 5.5 The "while" language used in our examples.

In the next section we provide some examples of more advanced decorators. At their most elaborate, these may specify a pattern p, can be parameterized with functions $a*$ and values $v*$, and may themselves be decorated ($d*$):

```
decorator d* [p .] name (d [, a* ] [| v* ]) = s
```

Note in particular the vertical bar '`|`', used to distinguish function and value arguments; in a call `f(|x)`, x is a value argument, in a call `f(x)` it is a function. The same convention, based on the Stratego notation, is supported for attributes. Furthermore, note that decorators can import other decorators $d*$. Such decorators are said to be *stacked*, and provide opportunity for reuse. To illustrate this, consider the `at-root` decorator of Figure 5.4. It evaluates attribute equations at the root of a tree, where the current node has no parent. Using the `down` decorator, the result is propagated downwards. Effectively, applying this stacked decorator results in a function application of the form `id.down(id.at-root(a))`. Stacking can also be achieved by declaring multiple decorators for an attribute. For example, we can add a "tracing" decorator to the `global-min` attribute, logging all nodes traversed by the `down` decorator:

```
def down trace global-min
```

5.4 APPLICATIONS

In this section we discuss a number of common idioms in AG specifications, and show how attribute decorators can be used to encapsulate them. We focus on language processing, a common application area of AG systems. As a running example we use a simple "while" language (see Figure 5.5). We demonstrate different language analysis problems and how they can be dealt with using high-level decorators that are agnostic of the object language. As such, they are reusable for more sophisticated languages and other applications.

```
decorator node.collect-all(a) =
  let results =
    node.children.map(id.collect-all(a))
  ; concat
  in if <a> node then // add to results
       ![<a> node | <results>]
     else
       results
     end
  end
```

Figure 5.6 The `collect-all` decorator.

5.4.1 *Constraints and Error Reporting*

A fundamental aspect of any language processing system is reporting errors and warnings. We define these as declarative *constraints* using conditional attribute equations. These equations specify a pattern and a conditional `where` clause that further restricts the condition under which they successfully apply:

```
eq error:
  |[ while (e) s ]| → "Condition must be of type Boolean"
  where not(e.type ⇒ BoolType)

  |[ e1 + e2 ]|     → "Operands must be of type Int"
  where not(e1.type ⇒IntType; e2.type ⇒IntType)
```

Each equation produces a single error message string if the subexpression types do not match `IntType` or `BoolType`. Rather than having them directly construct a list, we can collect all messages using the `collect-all` decorator (see Figure 5.6). It traverses the tree through recursion, producing a list of all nodes where the attribute succeeds. Note that this decorator does not test for *definedness* of the equations (using *a*`.defined`), but rather whether they can be successfully applied. Using `collect-all` with the `error` attribute, we can define a new `errors` attribute:

```
def collect-all errors :=
  id.error
```

This notation both declares the decorators and a default equation body, which refers to `error`.

To provide usable error messages, however, the error strings need further context information. We can define a new, application-specific decorator to add this information before they are collected, and use it to augment the `error` attribute:

```
decorator add-error-context(a) =
  <conc-strings> (a," at ",id.pp," in ",id.file,":",id.line)

def add-error-context error
```

With this addition, the `errors` attribute now lists all errors, including a pretty-printed version of the offending construct (provided a `pp` attribute is defined), and its location in the source code (given a `file` and `line` attribute).

5.4.2 *Name and Type Analysis*

Type analysis forms the basis of static error checking, and often also plays a role in code generation, e.g. for overloading resolution. Types of expressions typically depend on local operands, or are constant, making them well-suited for attribute equations. Moreover, an AG specification of a type analysis is highly modular, and may be defined across multiple files. Thus, let us proceed by defining a `type` attribute for all expressions in our language to perform this analysis:

```
eq type:
  Int(i)          → IntType
  |[ e1 + e2 ]|   → IntType where e1.type⇒IntType()
                              ; e2.type⇒IntType()
  Var(v)          → id.lookup-local(|v).type
  |[ f(args) ]|   → id.lookup-function(|f, args).type
```

Variable references and function calls require non-local *name analysis* to be typed. This can be done using parameterized *lookup attributes* that given a name (and any arguments), look up a declaration in the current scope [Hedin, 2000]. In the example we reference the local `type` attribute of the retrieved node, but lookup attributes can be used to access arbitrary non-local attributes for use in various aspects of the system. The actual lookup mechanism is provided by means of reusable decorators: to do this for a particular language, it suffices to select an appropriate decorator and define the declaration sites and scoping constructs of the language. Our lookup attributes are defined as follows:

```
def lookup-ordered(id.is-scope) lookup-local(x) :=
  id.decl(|x)

def lookup-unordered(id.is-scope) lookup-function(|x, args) :=
  id.decl(|x, args)
```

Figure 5.7 shows the prerequisite `decl` and `is-scope` attribute definitions for the name analysis, specified as arguments of the above attributes. Again, these are highly declarative and each address a single aspect. Declaration sites are identified by the `decl` attribute, which is parameterized with an identifier name x and optionally a list of arguments. It only succeeds for matching declarations. All declarations also define a `type` attribute. Similarly, the `is-scope` attribute is used to identify scoping structures. Note in particular the equations of the "if" construct, which, for the purpose of this example, defines scopes for both arms, similar to try/catch in other languages.

Languages employ varying styles of scoping rules. In our language we have two kinds of scoping rules: C-like, *ordered* scoping rules, and Algol-like, *unordered* scoping rules. In many languages, local variables typically use the former, while functions typically use the latter. We define the `lookup-ordered` and `lookup-unordered` decorators to accommodate for these styles (see Figure 5.8). They traverse up the tree, inheriting the behavior of the `down` decorator, thus giving precedence to innermost scopes. Along this path, the `lookup-ordered` decorator visits the current node (1). If no declaration is found there (i.e., `fetch-decl` fails), the `<+` combinator specifies that it should

```
eq |[ var x : t ]|:
  id.type     := t
  id.decl(|x) := id

eq |[ x : t ]|: // function parameters
  id.type     := t
  id.decl(|x) := id

eq |[ function f(params) : t stm ]|:
  id.type          := t
  id.decl(|f, args) := id where params.map(id.type).eq(|args.map(id.type))

eq is-scope:
  |[ function f(params) : t { stm* } ]| → id
  |[ if (e) s1 else s2 ]|.s1 → s1
  |[ if (e) s1 else s2 ]|.s2 → s2
  |[ while (e) s ]|          → id
  |[ { s* } ]|               → id
```

Figure 5.7 Attributes for name analysis and types of declarations.

```
  decorator down lookup-ordered(fetch-decl, is-scope) =
(1) fetch-decl
(2) <+ id.prev-sibling(lookup-outside-scopes(fetch-decl, is-scope))

  decorator down lookup-unordered(fetch-decl, is-scope) =
    (id.is-root <+ is-scope) // only look in scoping structures
(4) ; lookup-in-scope(fetch-decl, is-scope)

  lookup-in-scope(fetch-decl, is-scope) =
    fetch-decl
    <+ id.child(lookup-outside-scopes(fetch-decl, is-scope)) // enter scope

  lookup-outside-scopes(fetch-decl, is-scope) =
    fetch-decl
(3) <+ not(is-scope) // do not enter scope subtrees
     ; id.child(lookup-outside-scopes(fetch-decl, is-scope))
```

Figure 5.8 Lookup attributes and decorators.

proceed at (2), visiting any preceding siblings using the `lookup-outside-scopes` helper function. This function performs a local lookup for declarations in these nodes, respecting the scoping rules by avoiding traversal of scoping constructs (3). In contrast, `lookup-unordered` follows a straight path to the root of the tree, doing a search in encountered scopes (4).

5.4.3 *Control-flow Analysis*

Control-flow analysis forms the foundation of data-flow analysis, which is prerequisite to various compiler optimizations, refactorings, and static checks for bug patterns or security violations. A recent paper by Nilsson-Nyman et al. [2008] demonstrated how AGs can be employed for modularly specifying such analyses, ensuring separation of concerns and reusability with different data-flow analyses.

We take an approach similar to that of the JastAdd project, using reference attributes [Hedin, 2000] to declaratively define the control flow graph. Consider Figure 5.9, which defines a `succ` attribute, providing a reference to all

```
  def down succ-enclosing:
    Program(_)            → []
(2) [s1, s2 | _].s1       → [s2]
(3) |[ while (e) s ]|.s → [id]

(4)def default(id.succ-enclosing) succ:
    |[ { s; s* } ]|       → [s]
(1) |[ if (e) s1 else s2 ]| → [s1, s2]
    |[ return e ]|        → []
    |[ while (e) s ]|     → [s | s.succ-enclosing]

(5)decorator default(a, default) =
    if a then
      a
    else
      default
    end
```

Figure 5.9 Specification of the control flow.

```
def contributes-to(id.succ) stm:
  id.pred := stm
```

Figure 5.10 Specification of the reverse control flow.

successors of a statement. For instance, for the "if" statement, the successors are the "then" and "else" branches (1). A helper attribute, `succ-enclosing`, determines the default successors based on the enclosing block. For sequences of statements, the successor is the next statement in the sequence (2). The "while" statement overrides this behavior, by setting the successor of the enclosed block to itself (3). For any non-control flow statements, we specify `succ-enclosing` as the default successor `succ` (4), using the `default` decorator (5).

The specification of the `succ` attribute allows for a natural, declarative way of specifying the forward control flow of a language. However, a number of data-flow analyses depend on the *predecessors* of a statement. To avoid specifying these by hand, it is possible to use *collection attributes* [Boyland, 2005; Magnusson et al., 2007; Boyland, 1996] to *derive* the reverse flow graph. Collection attributes introduce a *"contributes to"* clause, allowing nodes to contribute values to collections in other nodes. Using this technique, we can define the predecessor graph in a single equation, by contributing each statement to its successors, as shown in Figure 5.10.

Figure 5.11 defines the `contributes-to` decorator. Note that for clarity, we use fragments of pseudocode in lieu of more advanced Stratego constructs. The complete, 20-line source is available from `http://strategoxt.org/Stratego/Aster/`. This decorator operates in two phases: the first time any collection attribute is evaluated, it enters the *survey phase* (1), where the complete tree is traversed, adding all contributing nodes to a list maintained for each node contributed to. This is done only once, rather than for every collection attribute retrieved. After this phase completes (2), referenced collections only require the application of any attribute equations associated with it (for `pred`, `stm` is returned). Note that all required bookkeeping op-

```
  decorator contributes-to(a, targets) =
    if not(completed survey phase) then
(1)   mark survey phase complete
    ; id.root
    ; in a top-down fashion:
        for a node x, apply targets and add them to the list of
        contributions for x
    end
(2); apply a to the list of contributions for the current node
```

Figure 5.11 The `contributes-to` decorator, contributing values to a list of nodes.

erations (i.e., storing contributions and whether the survey phase completed) are performed in the context of the current attribute: they are stored in tables associated with the attribute's unique signature and its argument values (i.e., `id.signature`).

5.4.4 *Data-flow analysis*

The control flow graph, specified by the `succ` and `pred` attributes, forms the foundation of any data-flow analysis. As such a graph may have cycles in it, data-flow analyses have the peculiar property that their equations may involve circular dependencies. This makes them unsuitable for traditional AGs. However, by extending the formalism with *circular attributes* [Magnusson and Hedin, 2007; Boyland, 1996], it becomes possible to use declarative AG equations to specify such analyses [Nilsson-Nyman et al., 2008]. Circular attribute equations can be solved by fixed point iteration, as long as their underlying data forms a lattice. We implemented this in a decorator that evaluates circular attributes.

A typical data-flow analysis is that of *reaching definitions*, which forms the basis for a constant propagation optimization. Its definition is as follows [Aho et al., 2006]:

$$RD_{out}(b_i) = gen(b_i) \cup (RD_{in}(b_i) \backslash kill(b_i))$$
$$RD_{in}(b_i) = \bigcup_{x \in pred(b_i)} RD_{out}(x)$$

These equations define the "out" and "in" sets of *definitions* for each statement b_i, respectively defining all variable definitions that reach just after and just before a statement. At the beginning of each statement, the definitions that reach it are defined by the union of all reaching definitions of the predecessor statements *pred* of Figure 5.9. The "out" set is maintained by means of a so-called "gen" set and a "kill" set that indicate for each statement b_i the statements to be respectively added to it and removed from it. In our following example, we use these sets as a basis for the propagation of constant definitions: statements may *add* or *remove* (i.e., invalidate) constant definitions. For example, an assignment statement `x := 3` *adds* a new constant definition for variable x when it completes. Therefore, this definition is added to the "out" set of the statement.

While there is an obvious correspondence between the equations above and regular attribute equations, a typical characteristic of data-flow analyses

```
  def circular(|[]) stm:
(1) id.constant-out :=
      <id.table-union> (
        id.constant-genset,
        <id.table-diff> (id.constant-in, id.constant-killset)
      )

(2) id.constant-in := <id.isect> id.pred.map(id.constant-out)

(3) def default(![]) |[ var : t := e ]|:
      id.constant-genset  := [(var, e.constant-value)]
      id.constant-killset := [(var, ())] where not(e.constant-value)

(4) def constant-value:
      Int(i)          → i
      Var(v)          → id.constant-before.lookup(|v)
      |[ e1 * e2 ]|   → <add> (e1.constant-value, e2.constant-value)
      |[ e1 + e2 ]|   → <mul> (e1.constant-value, e2.constant-value)
```

Figure 5.12 Constant propagation.

is that their definition involves *circular dependencies*: the equations are defined in terms of each other. Given a control flow graph that has cycles in it, a traditional AG evaluator cannot solve these equations. Instead, an iterative process is required. Nilsson-Nyman et al. [2008] recently demonstrated how such equations can be naturally implemented for a liveness analysis in an attribute grammar system, by extension with *circular attributes* [Magnusson and Hedin, 2007; Boyland, 1996]. They implemented the extension itself in Java, as part of the JastAdd evaluator [Magnusson and Hedin, 2007]. Another AG system that supports data-flow analysis is Silver [Van Wyk et al., 2010]. However, rather than extending the AG evaluation system, it uses an external model checking tool to perform the analysis [Van Wyk et al., 2007].

We provide a mechanism for circular attributes as part of the attribute grammar specification (or a library) rather than its evaluation system, in the form of the `circular` decorator. Instead of using the standard memoized evaluation scheme for attributes (see Section 5.6), this decorator provides an alternative evaluation strategy based on fixpoint iteration.

Consider Figure 5.12, which defines a constant propagation analysis using circular attributes and the `circular` decorator. In this definition, the `constant-out` and `constant-in` equations (1, 2) correspond to the "out" and "in" sets seen before. Note again the dependency between the two equations. The `circular` decorator is declared for both attributes, where the empty list `[]` is the initial value. The constants removed or added by each statement are defined by the "gen set" and "kill set" definitions (3). These sets are empty by default, but are specifically defined for assignments. In turn, these are defined by use of the by the `constant-value` attribute (4), which provides the constant value of an expression if one is currently available (or fails if an expression is not constant).

Following the basic algorithm of Magnusson and Hedin [2007], the `circular` decorator we define here only allows a single fixpoint iteration to be active at a time. Any circular attributes on nodes referenced during its iteration are

```
  def plain node.circular(a|initial) =
    (get-cache(|a.signature, node) <+ initial.init <+ !EVAL_FAILED())
  ; if ?FINISHED(value) + ?BUSY(value) then
(1)   !value // currently finished or busy; return last value
    else
(2)   fixpoint(a|a.signature, node, id)
    end
  ; not(EVAL_FAILED) // fail if failure placeholder encountered

  fixpoint(a|signature, node, oldvalue) =
(3) recompute(|signature, node, oldvalue)
  ; if not(fixpoint is running) then
      globally mark fixpoint running
(4) ; while marked changed:
        clear old list of participants
      ; apply recompute, with the latest result as its oldvalue argument
    ; mark all participants FINISHED
    ; globally unmark fixpoint running
    end

  recompute(a|signature, node, oldvalue) =
(5) node := <put-cache(|signature, node, BUSY(oldvalue))>
  ; (s <+ !EVAL_FAILED()) // evaluate or use placeholder for failure
  ; if id.eq(|oldvalue) then
(6)   register node as a participant
      put-cache(|signature, node, oldvalue) // no longer BUSY
    else
      id.init
(7) ; put-cache(|cache, node, id)
(8) ; globally mark changed
    end

  get-cache(|table, key) = retrieve a value from the cache
  put-cache(|table, key, value) = store a value in the cache
```

Figure 5.13 The circular attribute evaluation decorator.

said to be its *participants*. Once the iteration has reached a fixpoint, all participants are marked `FINISHED`, and do not require a new fixpoint iteration. In practice, this means that for an intraprocedural analysis, a single fixpoint loop is performed per method. During the iterative process, any attribute encountered is marked `BUSY`, which indicates that it should not be recomputed until the next step of the iteration is entered.

Consider Figure 5.13, which defines the `circular` decorator and a number of helper functions. The main `circular` decorator determines how to evaluate an attribute based on its current state: for attributes marked `FINISHED` or `BUSY`, the current value is returned (1). If not (2), the `circular-fixpoint` helper (re)computes the attribute equation value (3). It also starts a new fixpoint iteration as required, which runs while any of the participants records a change (4).

To recompute the value of equations, the `recompute-circular-def` helper performs the required bookkeeping operations: any active equation is registered as `BUSY` (5), all participants of the iteration are added to a list (6), any changes to values are recorded (7), and the newly computed value is stored using the `put-cache` operation (8). Like for the `contributes-to` decorator of Section 5.4.3, all bookkeeping operations are performed in the context of

the current attribute signature, with the exception of the global "fixpoint running" property.

Since the `circular` decorator evaluates equations multiple times, it is necessary to disable the default memoization behavior of equations (outlined in Section 5.6). We do this by importing the `plain` decorator, a built-in decorator that indicates that this behavior must be disabled for the decorator and any attributes that are transitively decorated by it. Thus, it forms the basis for attributes equations that require a customized caching scheme. In the `circular` decorator, we use the `put-cache` and `get-cache` helper functions to control the standard caching operations. The `plain` decorator also disables automatic term initialization, i.e., annotating a term with unique keys (see Section 5.6). This allows us to do this in a more fine-grained fashion for this decorator, using the `init` attribute to explicitly initialize result terms as necessary. In the future, as we transition to an implementation that no longer uses these annotations, this operation should no longer be required.

In this chapter we have shown how circular attributes can be implemented as a reusable library or application component, using strategic programming primitives to specify decorators, avoiding adaptation of the base AG system itself. For a more elaborate discussion of the uses of and evaluation algorithms behind circular attributes in general, we kindly refer the reader to [Magnusson and Hedin, 2007] and [Boyland, 1996].

5.5 CASE STUDY: GRAMMAR ANALYSES AND TRANSFORMATIONS

Beyond applications in typical attribute grammar idioms, we have applied Aster for the derivation of parse error recovery rules from grammars. Chapter 6 describes how these rules are constructed and the criteria by which they are selected. In this section we describe our experience with the construction of the `make-permissive` tool that performs the derivation, and its design and implementation.

The `make-permissive` tool is a batch processor tool that takes a given grammar, analyzes it, and generates a new grammar that includes derived recovery rules. The tool has a total of 544 SLOC spread over eight modules:[2]

- *make-permissive*: The main module, handling I/O, control, and command-line options (155 SLOC).
- *sdf-analysis*: Equations for constructing a graph-based model and analyzing SDF definitions (76 SLOC).
- *sdf-injections*: Equations for reasoning over injections based on the graph-based model (20).
- *sdf-heuristics*: Heuristic analyses over the graph-based model and injection analysis (48 SLOC).

[2]The full source code and example outputs are available from `http://strategoxt.org/Stratego/PermissiveGrammars`.

- *water-sections*: Equations for generating "water" recovery rules (Section 6.4.2) based on conditions that use the analyses above (28 SLOC)

- *insert-sections*: Equations for generating "insertion" recovery rules (Section 6.4.3) based on conditions that use the analyses above (191 SLOC).

- *optimize*: Optimizes the resulting grammar (20 SLOC).

- *pp-commented-sdf*: Handles output of SDF modules with comments about the recovery rules (8 SLOC).

In these modules, we define a total of 43 attributes consisting of 64 equations. Out of the 43 attributes, 30 use decorators in their definition. The analysis performed by the tool corresponds to one as seen in many compilers or optimizers, and uses a number of the decorators shown in this paper, demonstrating their applicability for building such a tool.

The tool uses the name analysis decorators of Section 5.4.2 (in 2 attributes) to analyze the name bindings in SDF. The `contributes-to` decorator of Section 5.4.3 is used (in 1 attribute) to collect references from productions to other productions. The `collect-all` decorator of Section 5.4.1 (used in 3 attributes) and the `down` decorator of Section 5.3 (used in 9 attributes) are used for collecting and distributing various sorts of information. In addition, the tool uses decorators for assertions, to control caching behavior. and to define a single new decorator for analyses on the graph structure formed by grammar productions.

An interesting observation that can be made about the `make-permissive` tool is that its ratio of equations to attributes is rather low. An explanation may be that the equations manage by exception, and that the decorators specify their general strategy.

The `make-permissive` tool is part of the standard Spoofax distribution of Chapter 4. It is compiled using the Java version of Stratego of Chapter 3, and distributed in the form of a 497 KB jar file that inlines the Aster runtime and standard library. Even for large grammars, such as the testing language of Chapter 8, which embeds the Stratego language, it can generate an output grammar in less than 3 seconds on a 2.8 Ghz Intel Core 2 Duo laptop.

5.6 IMPLEMENTATION

The Aster language is built as an extension of the Stratego strategic programming language [Bravenboer et al., 2008], which natively supports the canonical generic traversal operators. The Aster compiler is implemented in standard Stratego, using only a (bootstrapped) AG specification for error reporting (using constraint rules similar to those in Section 5.4.1). It compiles AG specifications to regular Stratego programs through a series of normalization steps. The normalization process starts by grouping attribute equations together,

forming separate strategies for each attribute and decorator. As illustrated in Section 5.3.2, attribute equations and decorators are implemented as functions with generic traversal operations (called *strategies* in strategic programming). Inherited attributes are defined at the parent of a node; therefore, their implementation uses the `parent` primitive. Attribute references and imported decorators are converted to strategy calls. For decorator calls, static reflective data is added for reflective attributes such as `signature`. Finally, a memoization mechanism is added to cache all attribute and decorator calls. We elaborate on these normalization steps in the technical report that accompanies this chapter [Kats et al., 2008b].

Using memoization, attributes are evaluated at most once, thus achieving optimal evaluation. Similar memoization-based dynamic evaluation has been used before in many other systems, e.g. by Jalili [1983] and recently in JastAdd [Hedin and Magnusson, 2003]. In Aster, memoization can be selectively disabled and overridden with custom behavior using decorators. For example, we disable it for the data-flow analysis of Section 5.4.4.

5.6.1 *Performance*

Our current, experimental implementation has not been tuned for performance. One constraining factor is currently the ATerm library used to represent trees, which forms an integral part of Stratego. It is optimized for a maximally shared representation of terms, where identical subtrees occupy the same space in memory [van den Brand et al., 2000]. This makes it less suitable for storing additional, dynamic information in tree nodes, in our case parent references (for `id.parent`) and memoized attribute values. We worked around this by annotating tree nodes with unique keys, and use these to store the added information in separate tables. In the future, we would like to adapt or replace the underlying implementation to better accommodate for this. Regardless, preliminary performance measurements indicate promising results.

We compared our compiler against JastAdd [Hedin and Magnusson, 2003], a mature AG system that uses an evaluation mechanism conceptually very similar to our own. We used the repmin program of Figure 5.1 as a test case and compared it against the repmin program of `http://jastadd.org/`. Over an average of fifty runs, JastAdd took 51 ms to replace all leaves in a large tree with 2^{16} leaves. Our system took 150 ms, or 180 ms for the version of Figure 5.3 where decorators are used in place of manual copy rules. Further testing confirms an unfortunate, but constant overhead of about a factor three in the base performance level, due to the expensive memoization and term initialization operations. Still, the results indicate a low overhead of the decorator mechanism. Furthermore, both our specifications, especially when using decorators, are more concise than the version implemented in JastAdd. Where the JastAdd specification uses 21 SLOC, ours use respectively 15 and 8 SLOC, including abstract syntax and attribute declarations.

The general principle behind attribute decorators shares similarities with the Decorator design pattern, which describes how to add functionality to objects at run-time [Gamma et al., 1995]. Variations of this idea exist in languages such as Python, which features decorators for functions [van Rossum, 2000]. In our case, we augment basic attribute definitions with either propagation of values from other nodes or with higher-level behavior such as a circular evaluation scheme. This kind of augmentation is similar to code weaving used in many forms of aspect-oriented programming [Kiczales et al., 1997].

Although considerable research has been devoted to various special-purpose extensions of AGs (as illustrated in the preceding sections), rather less attention has been paid to extensibility of AG systems. Two systems that do aim at different degrees of extensibility are first-class attribute grammars [de Moor et al., 2000; Viera et al., 2009] and Silver [Van Wyk et al., 2010].

In first-class AGs, attribute equations are *first-class citizens*, allowing them to be combined and manipulated using the language itself. Using function combinators, basic up, down, and chain copy rules can be defined [de Moor et al., 2000; Viera et al., 2009]. These combinators show similarities with decorators, although they are purely defined in terms of functional dependencies, and lack the reflective and traversal primitives that form the building blocks of decorators. The paper does not indicate that they could be used to implement more sophisticated forms of propagation and manipulation of equations, such as the collection and circularity decorators. Based on the Haskell type checker, first-class AGs prevent errors where the use of an attribute does not match its type. Errors due to cyclic dependencies or a mismatch between attribute equations and grammar productions are not reported. Our system is based on Stratego, which is largely untyped (but could be typed [Lämmel, 2003]). Further complicated by the use of parent node references, it currently does not provide a fully typed system, other than basic static pattern coverage checking. In practical Aster programs, such as the `make-permissive` tool of Section 5.5, we selectively use type assertions (in the form of decorators) that are dynamically checked to alleviate this limitation.

Silver supports extension with automatic copy rules as well as more advanced features such as collection attributes in a relatively accessible manner [Van Wyk et al., 2010]. Implemented in itself, the Silver language can be used to modularly implement such extensions. While adding extensions of this kind is made easier through facilities such as forwarding for local transformations [Van Wyk et al., 2002] and higher-order attributes, it is hard to imagine a regular Silver user building such an extension. Moreover, it is difficult to encapsulate these extensions in a single application or library, as they must be integrated in the base AG system. In contrast, many decorators are light-weight so they can be developed quickly and easily as needed.

A system that particularly inspired our design has been JastAdd [Hedin and Magnusson, 2003], which extends traditional AGs in a number of interest-

ing ways.[3] JastAdd uses reference attributes [Hedin, 2000], which we also use in a number of decorators. Its extensions include collection attributes [Magnusson et al., 2007] and circular computations [Magnusson and Hedin, 2007]. These are built into the JastAdd implementation; there is no user-level mechanism to define similar extensions. As demonstrated in Section 5.4, decorators can be used to define these same features at a higher level. Admittedly, we would not expect users to define relatively complex features like this very often, but building on the high-level framework provided by decorators is likely to be much easier than modifying the underlying implementation of an AG evaluation system. JastAdd is designed to be used in conjunction with hand-written code, particularly using visitors. As such, it provides a way to write traversals that interoperate with declarative attribution. In theory, this facility could be used to implement something similar to decorators, but this would require the addition of generic traversal on top of the Java implementation of trees, essentially duplicating the Stratego platform we use.

In related work, we introduced the Kiama system [Sloane et al., 2009], based on the Scala language. Kiama takes a pure embedding approach [Hudak, 1998] to its implementation, and is written as a Scala library. Based on the Scala language, it provides part of the flexibility of Aster by providing functional abstractions for copy rules and circular attributes. However, it lacks the notion of decorators or the full set of primitives used to specify decorators.

5.8 CONCLUSIONS AND FUTURE WORK

We propose decorated attribute grammars as a formalism for application-level extensibility of AG systems. To this end, we have identified primitives for the specification of decorators to define abstract evaluation strategies for attributes. By means of a prototype implementation and by employing decorators in different language engineering applications, we demonstrated the feasibility of using decorators to implement common abstractions over basic attribute grammars. These can be provided in the form of a library, and may be extended with user-defined decorators, where decorator stacking can be applied to reuse existing definitions.

In the future, we would like to explore further applications of decorated attribute grammars, in particular in the domain of implementing domain-specific languages and modular language extensions. For this we want to build upon the rewriting capabilities of the Stratego transformation language, the foundation of Aster. As such, we aim to take the best of both worlds; rewriting with Stratego and declarative analysis with attribute grammars.

Another direction for Aster is full integration with Spoofax. Aster is currently distributed as part of Spoofax, and can be used to define analyses for Spoofax language plugins, but additional experience is needed to identify patterns for specifying editor services and complete languages with Aster. In the future we would particularly like to explore decorators that encapsulate logic

[3] For the purposes of this chapter, we focus on the attribute grammar features of JastAdd, ignoring its support for rewriting trees during evaluation [Ekman and Hedin, 2004].

for typical editor service components, incremental compilation concerns, and related aspects.

Acknowledgments This research was supported by NWO projects 638.001.610, *MoDSE: Model-Driven Software Evolution*, 612.063.512, *TFA: Transformations for Abstractions*, and 040.11.001, *Combining Attribute Grammars and Term Rewriting for Programming Abstractions*. We would like to thank Nicolas Pierron for the discussions on attribute grammar systems and their implementation. We thank the anonymous reviewers of CC 2009 for providing useful feedback on an earlier version of this chapter.

Error Recovery for Generated Modular Language Environments

6

ABSTRACT

Integrated development environments (IDEs) increase programmer productivity, providing rapid, interactive feedback based on the syntax and semantics of a language. To support modular language definitions, language extensions, and embedded languages, constituent IDE plugin modules and their grammars can be composed. 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 chapter introduces a novel error recovery mechanism, which allows it to be used with files with syntax errors – common in interactive editing. 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. We evaluate the recovery quality and performance of our approach using a set of composed languages, based on Java and Stratego.

6.1 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 a programmer with rapid, interactive feedback based on the syntactic structure and semantics of the language. High expectations with regard to IDE support place a heavy burden on the shoulders of developers of new languages.

Parsing in IDEs One burden in particular for textual languages is the development of a parser. The parser forms the foundation of all language-specific editor services in an IDE. By parsing the source code text displayed in the editor, the parser constructs a structured representation in the form of abstract syntax trees (ASTs). These are used for presentational editor services, such as syntax highlighting, code folding, and outlining, and semantic editor services, such as cross-referencing, checking for semantic errors, and code completion.

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

programs [Degano and Priami, 1995]. Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

The development costs of a complete parser with error recovery support by hand are often prohibitive. Parser generators are an indispensable tool for rapid language development. They automatically generate a working parser from a grammar definition, significantly reducing the development time of the parser and the turnaround time for changing the parser as a language design evolves. 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 [Bravenboer and Visser, 2004].

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. In this chapter we show how all three can be achieved, by using a *generalized parser* that supports the complete set of context-free grammars with strict time complexity guarantees[1], and showing how to support error recovery in such a setting.

Error recovery for generalized parsers Generalized parsers such as generalized LR support the full class of context-free grammars, which is closed under composition. By using scannerless GLR (SGLR) [Visser, 1997b], even scanner-level composition problems such as reserved keywords are avoided. Parse error recovery for generalized parsers such as SGLR has been a long-standing open issue.

Challenges 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 recov-

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

ery solutions and makes it harder to diagnose syntax errors and identify the offending substring.

Approach overview In this chapter 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 6.3). We show how these rules can be specified and automatically derived (Section 6.4) and used with minimal changes to the parsing algorithm (Section 6.5). Since our recovery rules are normal parse production rules, the approach preserves the generalized and scannerless capabilities of the parser and its worst-case time complexity. We further use a layout-sensitive algorithm to select recovery suggestions for scoping structures in languages (Section 6.6) and use layout to constrain the search space for recovery rule applications (Section 6.7).

Contributions This chapter integrates and updates our work on error recovery for scannerless, generalized parsing [Kats et al., 2009a; de Jonge et al., 2009] and draws on our work on bridge parsing [Nilsson-Nyman et al., 2009]. We implemented our approach based on the modular syntax definition formalism SDF [Heering et al., 1989; Visser, 1997c] and JSGLR [Spoofax, 2011], a Java-based implementation of the SGLR parsing algorithm. In addition, this chapter 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 give a balanced selection and evaluation of recovery techniques.

6.2 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 [Ducasse et al., 2006] or aspects [Kiczales et al., 1997, 2001] 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. Such an embedding both increases the expressiveness of the host language and facilitates static checking of queries. Figure 6.1 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

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

Figure 6.1 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 6.2 Program transformation using embedded object language syntax.

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. Transformation languages such as Stratego [Bravenboer et al., 2008] and ASF+SDF [van den Brand et al., 2002] allow fragments of a language that undergoes transformation to be embedded in the specification of rewrite rules. Figure 6.2 shows a Stratego rewrite rule that rewrites a fragment of code from a domain-specific language to Java. The rule uses meta-variables (written in *italics*) to match "action" constructs and rewrites them to Java methods with a similar signature. SDF supports meta-variables by reserving identifier names in the context of an embedded code fragment.

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 6.2. In the embedded Java language, `void` is a reserved keyword. For the enclosing Stratego language, however, this name is a perfectly legal identifier. This difference in lexical syntax is essential for a clean and safe composition of languages. It is undesirable that the introduction of a new language embedding or extension invalidates existing, valid programs.

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

be elegantly addressed [Bravenboer et al., 2006a]. The *Scannerless Generalized-LR* (SGLR) parsing algorithm [Visser, 1997b] realizes scannerless parsing by incorporating the generalized-LR parsing algorithm [Tomita, 1988]. GLR supports the full class of context-free grammars, which is closed under composition, unlike subsets of the context-free grammars such as LL(k) or LR(k). 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.

The syntax definition formalism SDF [Heering et al., 1989; Visser, 1997c] integrates lexical syntax and context-free syntax supported by SGLR as the parsing algorithm. Undesired ambiguities in SDF2 definitions can be resolved using declarative *disambiguation filters* [van den Brand et al., 2002]. Implicit disambiguation mechanisms such as 'longest match' are avoided. Other approaches, including PEGs [Ford, 2002], language inheritance in MontiCore [Krahn et al., 2008], and the composite grammars of ANTLR [Parr and Fisher, 2011], implicitly disambiguate grammars by forcing an ordering on the alternatives of a production – the first (or last) definition overrides the others. Enforcing explicit disambiguation allows undesired ambiguities to be detected, and explicitly addressed by a developer. For non-trivial grammars, in particular composed, independently developed grammars, this characteristic is of vital importance.

Non-determinism in grammars can negatively affect parser performance. With traditional backtracking parsers, this would lead to exponential execution time, but with GLR the parser runs in cubic time in the worst case. GLR also has the attractive property that it can parse the deterministic LR class of grammars that it extends in linear time, while gracefully coping with any ambiguities. While scannerless parsing tends to introduce additional non-determinism, the implementation of parse filters during parsing rather than as a pure post-parse filter eliminates most of this overhead [Visser, 1997a].

SDF has been used to define various composite languages, often based on mainstream languages such as C/C++ [Waddington and Yao, 2007], PHP [Bravenboer et al., 2010], and Java [Bravenboer and Visser, 2004; Bravenboer et al., 2006a]. The example grammar shown in Figure 6.3 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 $p_1 \ldots p_n \rightarrow s$ 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.

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

Figure 6.3 Syntax of Java with embedded SQL queries, adapted from [Bravenboer et al., 2010].

6.3 ISLAND GRAMMARS

Island grammars [van Deursen and Kuipers, 1999; Moonen, 2001, 2002] 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 [Moonen, 2001]. In this chapter 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 [van Deursen and Kuipers, 1999; Moonen, 2001]. The integration of lexical and context-free productions of SDF allows island grammars to be written in a single, declarative specification that includes both lexical syntax for the definition of water and context-free productions for the islands. Although SGLR did not support error recovery per se, a parser using an island grammar behaves similar to one that implements a noise-skipping algorithm. It can skip over any form of noise in an input file. However, using an island grammar, this logic is entirely encapsulated in the grammar definition itself.

Figure 6.4 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 series 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.

```
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 6.4 An island grammar for extracting calls from a legacy application; adapted from [Moonen, 2001].

The parse tree produced for an island is constrained using disambiguation filters. First, the `{avoid}` annotation on the `WATER` rule specifies a disambiguation filter for these productions, indicating that the production is to be avoided: 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:

```
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.[2] Internally, the parse tree includes the complete character stream, all productions used, and their annotations. In this chapter, we focus on abstract syntax trees (derived from the parse trees) where only the `{cons(`*name*`)}` constructor labels appear in the tree. Figure 6.5 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.

6.4 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

[2]Note that parse forests are efficiently represented using the ATerm library [van den Brand et al., 2000], which employs hash-consing to achieve maximal sharing of subtrees, ensuring that any identical leaves and branches occupy the same space in memory.

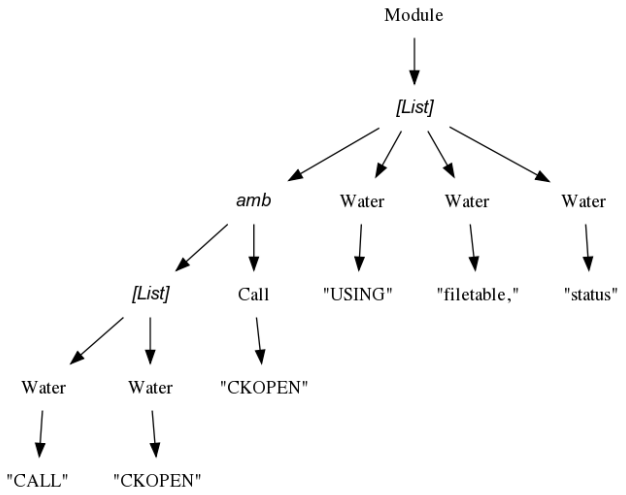


Figure 6.5 The unfiltered abstract syntax tree for a COBOL statement, constructed using the ExtractCalls grammar.

```
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 6.6 Part of the standard Java grammar in SDF; adapted from [Bravenboer et al., 2006a].

case an input sentence cannot be parsed, in the latter case, the parser algorithm is adapted to do so. This observation suggests that the basic principle behind island grammars may be adapted for use in recovery for complete, well-defined grammars. In contrast, the technique of island grammars is targeted only towards partial grammar definitions.

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. Without loss of generality, we focus many of our examples on the familiar Java language. Figure 6.6 shows a part of the SDF definition of the language. Indeed, Java can be parsed without the use of SGLR, but SGLR has been invaluable for extensions and embeddings based on Java such as those described in [Bravenboer and Visser, 2004; Bravenboer et al., 2006a].

```
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 6.7 Chunk-based recovery rules for Java.

6.4.1 *Chunk-Based Water Recovery Rules*

Whereas island grammars have an underlying "chunk" structure, this structure is lacking in complete, well-defined grammars. Rather, these grammars typically 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 6.7 extends the standard Java grammar with a coarse-grained chunk structure at the statement level. In this grammar, every `Stm` symbol is considered a "chunk," which can be parsed as either a statement or as water, effectively skipping over any noise that may exist within method bodies. Note that the standard Java grammar, as shown in Figure 6.6, already uses an `{avoid}` annotation to explicitly avoid the "dangling else problem," a notorious ambiguity that occurs with nested if/then/else statements. Therefore, in our recovery rules we use `{recover}` rather than `{avoid}` to distinguish between the two concerns of disambiguation and recovery.

We can extend the grammar of Figure 6.7 to introduce a chunk-like structure at other levels in the hierarchical structure formed by the grammar, e.g. at the method level or at the class level, in order to cope with syntax errors in different places. 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 by counting all occurrences of the `{recover}` annotation in ambiguous branches, and selecting the variant with the lowest count. In case of multiple branches with the same count, the first can be picked.

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 and 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 is already moderately effective in recovery from syntax errors, 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.

6.4.2 *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 and use it as a basis for our general approach to error recovery. Note throughout this section we use only the standard, unaltered SDF specification language, adding only the `{recover}` annotation and 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 6.3, any form of layout, which may include comments, is implicitly interleaved in the patterns of concrete syntax productions. The parser, in a way, 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. Context-free syntax in SDF is a convenient way to define context-free productions without having to worry about the interleaving of layout. Only in the *kernel syntax* that lies at the heart of SDF, does the production explicitly include the layout:

```
syntax
  "if" <LAYOUT?-CF> "(" <LAYOUT?-CF> <Expr-CF>
       <LAYOUT?-CF> ")" <LAYOUT?-CF> <Stm-CF>
   → <Stm-CF> {cons("If")}
```

The parse table generator for SDF automatically converts context-free productions to this form. (The production above was derived from the `If` production in Figure 6.6). Expressed in kernel syntax, the symbol names in the rule above use angle brackets and explicitly state that they are related to context-free (CF) syntax. The optional layout symbols `<LAYOUT?-CF>` are not considered for the construction of the abstract syntax tree (and may be stored as annotated data instead).

```
module Java-15-Permissive-WaterOnly
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 6.8 Water recovery rules.

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 6.8, which combines elements of the comment definition of Figure 6.6 and the chunk-based recovery rules from Figure 6.7. 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. This ensures that large strings, consisting of multiple words and special characters each, 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();
```

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 can be inspected instead.

The adapted grammar of Figure 6.8 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.

```
module Java-15-Permissive-InsertionsOnly
imports Java-15
exports
lexical syntax
   → ")" {recover, cons("INSERT")}
   → "]" {recover, cons("INSERT")}
   → "}" {recover, cons("INSERT")}
   → ">" {recover, cons("INSERT")}
   → ";" {recover, cons("INSERT")}
lexical syntax
  INSERTSTARTQ StringPart* INSERTENDQ
                        → StringLiteral {cons("INSERTEND")}
  "\""                  → INSERTSTARTQ  {recover}
  "\n"                  → INSERTENDQ
lexical syntax
  INSERTSTARTC CommentPart* INSERTENDC
                        → Comment       {cons("INSERTEND")}
  "/*"                  → INSERTSTARTC {recover}
  EOF                   → INSERTENDC
```

Figure 6.9 Literal-insertion recovery rules.

6.4.3 *Literal-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 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 actually "insert" the particular literal into the parse stream. SDF actually allows us to simulate this using separate productions that "insert" literal symbols. We illustrate this in Figure 6.9. Consider the first lexical syntax section, which lists a number of basic *literal-insertion recovery rules*, each inserting a closing bracket or other literal that ends a production pattern.

Literal-insertion rules have an empty pattern, indicating that they match the empty string. That is, for each of these literals specified in the grammar, an empty string may be matched against instead. 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 it is defined in lexical syntax context, 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 6.1 and Figure 6.2. 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.

On Literals, Identifiers, and Reserved Words Literal-insertion rules can also be used for literals that are not *reserved words*. For example, for the combined Stratego-Java language, a good insertion rule is:

```
lexical syntax
  → "end"
```

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

```
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. 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:[3]

```
syntax
  [e] [n] [d] → "end"
```

The literal-insertion rule simply adds an additional derivation for the `"end"` symbol, providing the parser with an additional way to parse it. As such, the rule does not change how identifiers (`ID`) are parsed. This is an important property when considering composed languages in general. In many cases, some literals in one sublanguage may not be reserved words in another. With a naive recovery strategy that inserts tokens into the stream, this could result in keywords being inserted in place of identifiers (e.g., `end` in Java). But since the insertion rules only apply when a literal is expected, these effects are avoided with our approach.

[3] Actually, in fully normalized kernel syntax form, the character codes `[\101] [\110] [\100]` are used.

Insertion Rules for Lexical Symbols The lower sections of Figure 6.9 specify insertion rules for terminating the productions of the `StringLiteral` and `Comment` symbols, first seen in Figure 6.6. Both rules have a `{recover}` annotation on their starting literal. Alternatively, the annotation could be placed on the complete production, but this formulation is beneficial for the runtime behavior of our adapted parser implementation, ensuring that the annotation is considered before construction of the 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. In contrast, 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 file, the number of different possible interpretations is severely reduced, thus reducing the overall runtime complexity of the recovery.

Insertion rules can also be used to insert lexical symbols such as identifiers. Missing identifiers generally indicate an error in the enclosing context-free construct. Identifier insertion is a feasible approach to recover from these kinds of errors, but requires the introduction of placeholder identifiers that adds to the complexity of tools that process the AST. Still, for certain use cases such as content completion in an IDE, this form of recovery can be useful. We revisit the topic in Section 6.8.

6.4.4 *Combining Different Recovery Rules*

The water recovery rules of Section 6.4.2 and the insertion rules of Section 6.4.3 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-WaterOnly
  Java-15-Permissive-InsertionsOnly
```

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.

6.4.5 *Automatic Derivation of Permissive 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, we don't want the editor to suggest to insert a "try" or "synchronized" keyword. In those cases,

discarding some substrings instead may be a safer alternative. The decision whether to consider particular keywords for insertion may depend on their semantic meaning and importance [Degano and Priami, 1995]. To take this into account, expert feedback on a grammar is vital. Since we have aimed at maintaining language independence of the approach, our main focus is on more generic, structure-based properties of the grammar.

In this section we have identified and focused on four different distinct, general *classes of literals* that commonly occur in grammars:

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

Each has its own particular kind of insertion rule, and each follows its own particular definition pattern. By analysis of a grammar, using heuristic rules to recognize these patterns, we derive water recovery rules and recovery rules for insertions of the above categories. Thereby, our system maintains language independence by providing a generic, automated approach towards the introduction of recovery rules.

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.

SDF specifications are fully declarative. It is this nature that is essential for automated analysis and transformation of a grammar specification. It is not feasible to do so for other syntax formalisms that use semantic actions to construct ASTs and may maintain state or call external functions (e.g., to determine operator priorities).

We formulated a set of heuristic rules for the detection of different production patterns based on our experience with different grammars. For instance, *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. And finally, the last literal is not used as the starting literal of any other production.

The heuristic rules for the other categories involve a larger set of conditions. 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. Finally, for the fourth category we derive rules for matching starting and ending literals in `LAYOUT` productions. Note that we found that some grammars (notably the Java grammar of [Bravenboer et al., 2006a]) use kernel syntax for `LAYOUT` productions to more precisely control

```
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 6.10 A selection of context-free productions that appear in the Java grammar.

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 6.10. Looking at the first production, and using the heuristic rules above, we can recognize that `}` qualifies as a closing literal. Likewise, `)` satisfies the conditions we have set. By programmatically analyzing the grammar in this fashion, we collected the set of closing literal insertion rules of Figure 6.9.

Note that none of the generated inserted closing literals of Figure 6.9 ever occurs as an opening literal in the grammar. We only derive rules from brackets that appear in a balanced fashion with another (possibly different) 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. We exclude lexical productions that define strings and comments, for which we only derive more restrictive insertion rules.

From the productions of Figure 6.10 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.[4] This follows from the "void" class literal production of Figure 6.10. The `class` keyword is never used as a starting literal of any production (as seen in the same figure, not even so for class headings), and therefore satisfies our set of rules. In practice, we have found that these anomalies are relatively rare and harmless, or sometimes even beneficial.

We evaluated our set of heuristic rules using Java, Java-SQL, Stratego and Stratego-Java grammars, as outlined in Section 6.9. For these grammars, a total number of respectively 19, 43, 37, and 47 insertion rules were generated, along with a constant number of water recovery rules as outlined in Figure 6.8. The source code of the derivation tool and the complete set of derived rules for these grammars are available from `http://strategoxt.`

[4]Note that for narrative reasons, we did not include an insertion rule for this keyword in Figure 6.9.

`org/Stratego/PermissiveGrammars`. The implementation of the tool is described in Section 5.5.

6.4.6 *Customization of Permissive Grammars*

A good error recovery mechanism is not only *language independent*, but is also *flexible* [Degano and Priami, 1995]. That is, it allows grammar engineers to use their experience with a language to improve recovery capabilities. Our system, while remaining within the realm of the standard SDF grammar specification formalism, delivers both of these properties. Language engineers can add their own recovery rules using SDF productions similar to those shown earlier in this section.

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. 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 four deletions and two insertions. Other alternatives, less close to the original intention of the programmer, might be preferred by the recovery mechanism. Based on this observation, a grammar engineer can add *substitution recovery rules* to the grammar:

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

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

Modular Definition of Customizations 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.

Besides composition, SDF also provides a mechanism for subtraction of languages. The `{reject}` disambiguation annotation filters all derivations for a particular set of symbols [van den Brand et al., 2002]. Using this filter, it is possible to disable some of the automatically derived recovery rules. Consider for example the insertion rule for the `class` keyword, which arose as an anomaly from the heuristic rules of the previous subsection. Rather than directly removing it from the generated grammar, we can disable it by extending the grammar with a new rule that rejects this recovery. Figure 6.11 illustrates this with a grammar that imports the generated permissive grammar, and disables the `class` insertion rule.

```
module Java-15-Permissive-Customized
imports
  Java-15-Permissive
exports
  lexical syntax
     → "class" {reject}
    ...
```

Figure 6.11 A customized permissive grammar.

```
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 6.12 Interpretations of `i=f(x)+1;` with insertion recovery rules (underlined) and water recovery rules.

6.5 PARSING PERMISSIVE GRAMMARS

When all recovery rules are taken into account, permissive grammars provide many different interpretations of the same code fragment. As an example, Figure 6.12 shows the possible interpretations of the string `i=f(x)+1;`. The interpretations can be obtained by applying additional 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 a very 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 parallel, which significantly increases the workload for the parser, even if there are no errors to recover from.

```
void methodX() {
   if (true) // missing {
      foo();
   }
   int j = 0;
   while (j < 8)
      methodY(j++);
}
```

Figure 6.13 The missing opening bracket is detected at the `while` keyword.

6.5.1 *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). For normal grammars, they are less practical as backtracking parsers exhibit exponential behavior in the worst case [Johnstone et al., 2004]. Moreover, as they only explore one possible interpretation at a time, they do not allow declarative disambiguation. Still, when applied to error recovery, these issues are less problematic. For pathological cases with repetitive backtracking, the parser can be aborted, and (ideally) a secondary, non-correcting, recovery technique can be applied. 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 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.

6.5.2 *Selecting Choice Points for Backtracking*

An important challenge error recovery techniques must address is the difference between the actual location of the error and the point of detection [Degano and Priami, 1995]. Consider for example the code in Figure 6.13. Because of the missing opening bracket of the `if` statement, the closing bracket after the enclosed `foo();` statement is misinterpreted as closing the method. 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.

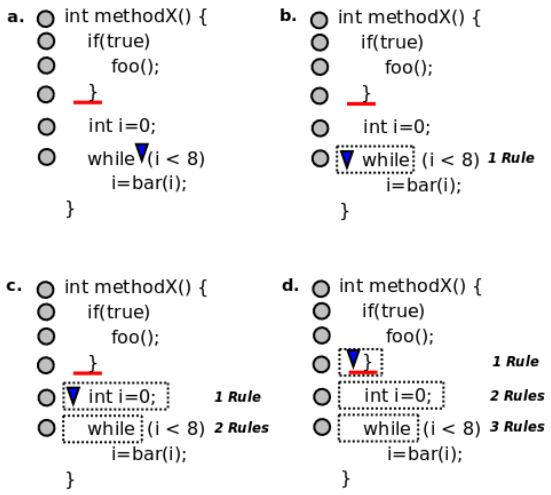


Figure 6.14 Applying error recovery rules with backtracking. The initial point of failure and the start of the recovery search space is indicated by a triangle. The entire search space is indicated using dashed lines, where the numbers to the side indicate the number of recovery rules can be applied at that line.

As generalized LR parsers process different interpretations in parallel, they use a more complicated stack structure than regular LR parsers. Instead of a single, linear stack, they use a graph-structured stack (GSS) that efficiently stores the different interpretation branches, which are discarded as input tokens or characters are shifted [Tomita, 1988]. This poses a challenge for applying backtracking, since all the discarded branches must be stored in case the old state is revisited. We found that it is prohibitive (in terms of performance) to maintain the complete stack state for each shifted character. Therefore, we only selectively record the stack structure to minimize the overhead introduced. In the current implementation, we only create one backtracking choice point for each line of the input file.

6.5.3 *Applying Recovery Rules*

Figure 6.14 illustrates how to apply backtracking to recover the Java fragment of Figure 6.13. 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 6.14a 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 6.14b) 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 6.14c, 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 6.14d, 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 fragment. Once the original line where the error was detected can be successfully parsed, normal parsing continues.

6.5.4 *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 for each newline character. And finally, if all branches are discarded and no accepting state is reached, the parser enters recovery mode. Once the recovery is successful, normal parsing resumes with the newly constructed stack.

Figure 6.15 shows the recovery algorithm in pseudo code. The `Recover` function controls the iterative search process described in Section 6.5.3. The function starts with some initial configuration (line 3–5); it enables the recovery productions that are ignored in normal parsing mode, selects the most recent choice point, and initializes the `candidates` variable. The choice points from the point of failure are then visited in reverse order (line 6–9). Using the `RecoverParse` function, choice points are iteratively explored (line 7), attempting to find a valid interpretation of the program by applying recovery rules. Once a valid interpretation is found, the set of stacks (GSS structure) of the parser is non-empty (line 9). During each iteration, the algorithm also maintains *candidate stacks* that represent a partial interpretation of the program that may lead to a recovery in a later iteration of the algorithm, as the search space is increased.

The `RecoverParse` function attempts to construct a valid interpretation by re-parsing the fragment starting from the choice point, optionally revisiting previous candidate stacks that were previously cut off. After the parser is reset to the state stored by the choice point (line 18–20), characters are consumed from the choice point location, until the original point of failure is

```
RECOVER(parser)
 1  ▷ Input: parser - A parser instance
 2
 3  parser.ignoreRecoverProductions ← false
 4  choicePoint ← Most recent choice point
 5  candidates ← {}
 6  do
 7     candidates ← RECOVERPARSE(parser, candidates, choicePoint)
 8     choicePoint ← Choice point before choicePoint or choicePoint if none
 9     until |parser.stacks| > 0
10  parser.ignoreRecoverProductions ← true

RECOVERPARSE(parser, candidates, choicePoint)
12  ▷ Input:
13     parser - The parser instance
14     candidates - Previous candidate stacks
15     choicePoint - The starting point for recovery
16  ▷ Output: New candidates created by one extra recovery production
17
18  parser.stacks ← choicePoint.stacks
19  parser.offset ← choicePoint.offset
20  newCands ← {}
21  while parser.offset ≤ parser.failureLocation
22  do
23     offsetCands ← { c | c ∈ candidates ∧ c.offset = parser.offset }
24     parser.stacks ← parser.stacks ∪ offsetCands
25     parser.parseCharacter()
26     createdCands ← Stacks of parser created using a recovery production
27     parser.stacks ← parser.stacks \ createdCands
28     newCands ← newCands ∪ createdCands
29  return newCands
```

Figure 6.15 A backtracking algorithm to apply recovery rules.

reached (line 21). With each iteration, previously cut-off candidate stacks are added to the set of stacks of the parser, if the offset of the candidate matches the offset of the parser at that point (line 23–24). After that, a single character is parsed (line 25), which can result in stacks being popped (interpretations failing) or new stacks being added. Any new candidate stacks found that used a recovery rule are excluded from further exploration and will be revisited in the next iteration (line 26–28). If eventually all stacks are popped, `RecoverParse` leaves the *parser* instance with an empty set of stacks and only returns the candidate stacks for further examination.

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.

6.6 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 [Charles, 1991]. Omitting brackets (or other character sequences marking scopes) of scopes is one of the most common errors made by programmers. These errors can be addressed by common parse error recovery 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. A challenge is to select the most appropriate position.

As an example, consider the Java fragment in Figure 6.16. 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. Bridge parsing [Nilsson-Nyman et al., 2009] provides an algorithm to improve error recovery based on indentation. Based on a set of rules that describe the typical relation between scopes and layout for Java, it can correctly recover cases such as the example above.

```
class C {
  void m() {
    int y;
  int x;
}
```

Figure 6.16 Missing `}`.

A bridge parser can be configured to work for any given language, and works independently of a particular parser technology. Bridge parsing shares its inspiration in island grammars [van Deursen and Kuipers, 1999; Moonen, 2001, 2002] with our permissive grammars approach: A bridge parser employs a scanner that only recognizes tokens that make up scoping structures ("islands") and important tokens for determining how those islands should be connected ("reefs"). All other tokens ("water") are skipped. Given a list with these kinds of tokens and a set of constraints, the bridge parser constructs a *bridge model*, which captures the scopes in the input.

Figure 6.17 shows an example of a token list and a bridge model for the program fragment of Figure 6.16. At the top of the figure, the token stream is shown and the mapping to islands, water, and reefs. Reefs are have a number indicating the indentation level, which is key for the identification of scope structures in the form of matching islands. The bottom of the figure shows

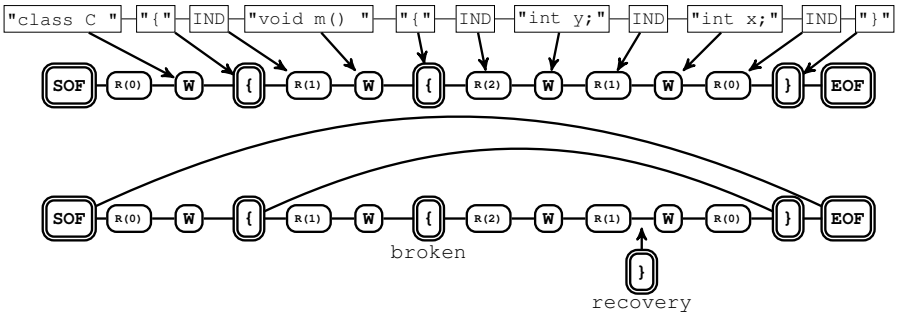


Figure 6.17 A bridge parser model. Top: tokenization; middle: mapping to islands (double edges), water (`W`), and reefs (`R(`n`)`); bottom: bridges between matching islands.

the complete bridge model with matching islands connected by bridges. It also shows the identification of a broken scope, revealed by an island without a bridge, and its recovery, identified by matching the island with the reef that follows it. A comprehensive description of the algorithm, incrementally constructing multiple bridges, is given in [Nilsson-Nyman et al., 2009].

A bridge parser builds its bridge model using tokens created by a scanner. This may appear contrary to the scannerless nature of SGLR, but practical experience has shown that a bridge parser is most time and memory-efficient when independent from a specific grammar and focusing just on the scoping structures of the language. Based on an accurate (scannerless) lexical analysis, additional reefs can be identified using the keywords of a language, but previous results showed that doing so only marginally improves recovery quality [de Jonge et al., 2009]. For this reason and for simplicity, the bridge parsers used in this chapter only include scope tokens and layout reefs. A limitation of this approach is that for embedded languages, it is possible that tokens 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.

A bridge parser is generated from a bridge parser specification (or bridge grammar), specifying tokens such as islands and reefs, along with constraints for matching and recovering of scopes. Bridge parser specifications are compositional and can be extended in several steps. Generic behavior such as "closest match recovery" or "layout-based recovery" are defined in a generic module that can be reused and redefined by other grammars. For specific grammars, additional behavior can be added, e.g. taking into account keywords and layout conventions for expressions. Using heuristics such as those described in Section 6.4.5, instance grammars can be automatically derived.

Bridge parsing forms a supplementary recovery approach that can be used together with permissive grammars. It also formed the inspiration for a layout-sensitive region that we discuss next.

6.7 LAYOUT-SENSITIVE REGIONAL RECOVERY

In this section we describe a layout-sensitive region recovery algorithm that ensures 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 6.4.3 discussed an example of this, 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 suggestions to a particular part of the file, *region selection* improves the efficiency as well as the quality of the recovery, avoiding suggestions that are spread out all over the file. In some cases it is better to ignore a small part of the input file, rather than to try and fix it using a combination of insertions and discarded substrings. As a second application of the regional approach, *region skipping* is used as a fallback recovery strategy that discards the erroneous region entirely in case a detailed analysis of the region does not lead to a satisfactory recovery.

6.7.1 *Nested Structures as Regions*

Language constructs such as "while" statements and class bodies form suitable regions for regional error recovery. They form free standing blocks, in the sense that they can be omitted without influencing the interpretation of other blocks. Erroneous free standing blocks can simply be skipped, providing a coarse recovery that allows the parser to continue. A typical technique to select such regions is to look for certain marker tokens in the context of an error, such as the fiducial tokens of Pai and Kieburtz [1980]. These tokens depend on the language used. For example, for Java, keywords such as `while` and `class` could be used. We will take a more language-independent approach.

The bridge parsing technique discussed in Section 6.6 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,

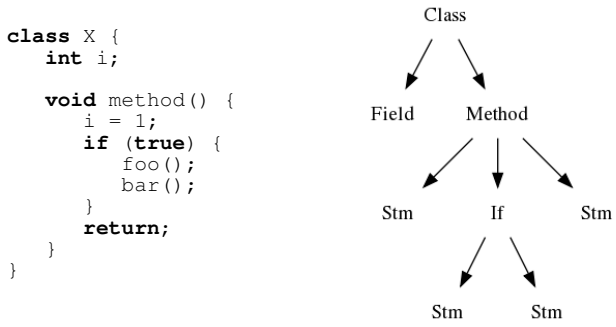


Figure 6.18 Indentation closely follows the hierarchical structure of a program.

uses indentation to detect erroneous structures. Indentation typically follows the logical nesting structure of a program, as illustrated in Figure 6.18. 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, but 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 regional recovery can lead to better and faster recovery results.

6.7.2 *Layout-Sensitive Region Selection*

We follow an iterative process to select an appropriate region that encloses a syntax error. 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. We show example scenarios in Figure 6.19 and 6.20.

Figure 6.19 shows a syntax error and the point of detection, indicated by a triangle (left figure). A candidate region is selected based on the alignment of the `void` keyword and the closing bracket (middle figure). The candidate is then validated by discarding the region. Since the parsing of the remainder of the fragment is successful (right figure), the region is accepted as erroneous. Figure 6.20a shows a different example, where a candidate region is rejected. Based on the point of detection, an obvious candidate region is the `m2` method (middle figure). However, an attempt to parse the succeeding construct leads

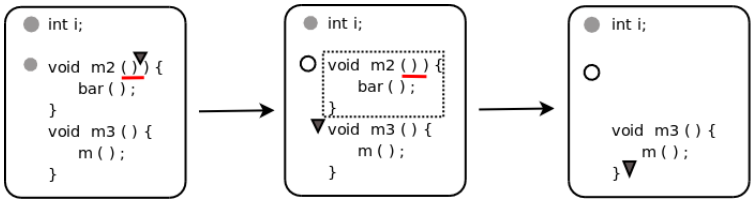
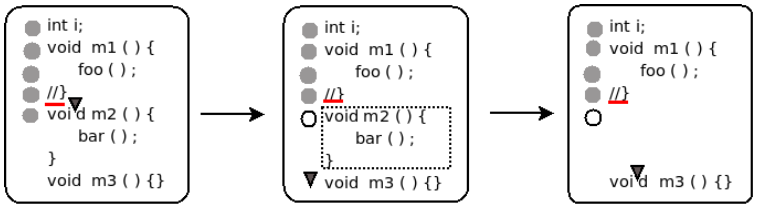
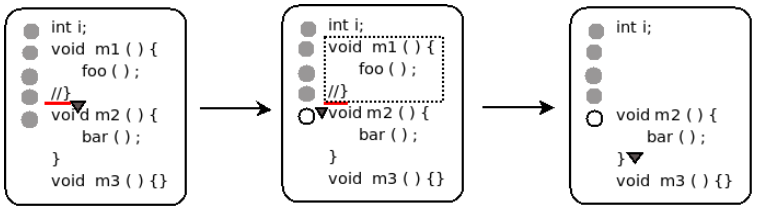


Figure 6.19 A candidate region is validated and successfully discarded.



(a) A candidate region is rejected.



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

Figure 6.20 Iterative search for a valid region.

to a premature parse failure, therefore the region is rejected. In Figure 6.20b an alternative candidate region is selected, this time one preceding the point of detection. This region is successfully validated.

The region validation criteria should balance the risk of selecting the wrong candidate, which may lead to spurious errors, and the risk of rejecting a correct candidate region. The latter typically occurs in the context of multiple errors, in which a new, unrelated error causes the parser to fail again. Both cases lead to large regions, which should be avoided. We currently consider a region valid if the two lines of code succeeding it parse correctly, which has shown good practical results.

6.7.3 *Selection Schemata*

The candidate regions are explored in an ordered fashion, with the aim to find the smallest fragment enclosing the error. In case the erroneous region is not found, the whole file is marked as erroneous region and further inspected by a correcting technique.

Current structure The first candidate region is the construct starting from the error detection location. The region is recognized by a forward skip until the end of the construct is found. In the example to the right, the parser fails after reading the mistakenly inserted second bracket. Discarding the entire `while` statement resolves the error.

```
while(true)) {
    foo();
}
```

Previous structure The second candidate is the structure preceding the error detection location. The region is detected by a backwards skip, using the indentation information stored in the choice points. Typical problems that are solved by discarding the previous structure are incomplete lines and scope errors caused by a missing closing bracket. The error in the example to the right is detected after the `bar();` statement, while the preceding line caused the error.

```
void methodX() {
    foo(
    bar();
}
```

Siblings Regions that are mutually dependent should be discarded as a whole. A typical example is shown at the right. The unclosed "then" clause cannot be discarded, because the "else" clause cannot exist in isolation. The "sibling-procedure" deals with this situation. The procedure starts with the current structure as discarded region. Then it successively includes the prior sibling and the next sibling, until a valid erroneous region is found or all siblings have been considered.

```
if (c) {
    foo();
else
    bar();
```

Parent The next region considered is the parent structure, identified through a forward and backward search for a decrease in indentation. The example shows a method that accidentally has two closing brackets, which causes the parser to fail at the return keyword. Discarding the method region solves the problem. The parent selection adds some robustness to the selection scheme. Small deviations in the use of indentation, or nearby, unrelated errors might be captured in the parent region. If the parent region can not be discarded, we continue recursively with the parent of the parent and so on.

```
int foo() {}
    int i = 1;
    return i;
}
```

6.7.4 *Practical Considerations*

Brackets Similar to indentation conventions, conventions for using *brackets* can vary among programmers and among programming languages. The placement and whether or not to use brackets for block structures varies, as illustrated in the figure. The two code fragments have the same indentation characteristics, but have a different decomposition in regions. To address this issue, our implementation adapts the selection schemata to cover these cases specific for a generic notion of brackets or other block-enclosing tokens.

```
bar();
while(true)
    foo();
doX();

while(true)
{
    foo();
}
```

Separators Separators and operators that may reside between language constructs are another practical consideration. The last line in this Stratego fragment has an error, however removing the erroneous line does not lead to a valid parse. The operator <+ that connects the erroneous construct with the preceding construct must be removed as well. We have extended the region selection schema for last siblings with a candidate region consisting of the original region plus the lexical token at the end of the preceding sibling.

```
EvalExp =
  EvalAdd <+
  EvalSubt <+
  EvalMul <
```

Multi-line comments and strings The selection schemata 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 large, the region selection schema can fail to locate the error. A particularly problematic case commonly seen in practice are unclosed lexicals 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 file is reached. As a consequence, a missing block comment ending is typically detected lately. 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 structure (/*). If this structure is recognized, a candidate region is selected from the start of the reduction, making it possible to cope with flat structures such as block comments that do not follow the indentation.

```
/* Comments ...
int foo() {
   ...
}
...
EOF
```

6.7.5 *Integrating Recovery Techniques*

We combine the different techniques described in this chapter in a multiply staged recovery approach. Region selection is applied first to detect the erroneous region. In the second stage, the erroneous region is inspected by one of the correcting techniques, bridge parsing or permissive parsing. 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 Section 6.5 is used, where backtracking is restricted to the erroneous region. In case both correcting techniques fail, the erroneous region is skipped as a fallback recovery strategy.

6.8 APPLYING ERROR RECOVERY IN AN IDE

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. By supporting error recovery, they can provide these services even when the program has syntactic errors, which is very common

when source code is edited interactively. 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.

6.8.1 *Efficient Construction of Languages and Editor Services*

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, and improve the productivity of language engineers by providing high-level languages, frameworks, and tools for efficient language engineering [Fowler, 2005a].

6.8.2 *Guarantees on Recovery Correctness*

Using regional recovery, bridge parsing and permissive grammars, 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 according to the grammar. That is, it will only produce ASTs with tree nodes that conform to the structure imposed by production rules in the grammar. Even for cases where a region of code is skipped by the parser, this property is maintained as only well-formed trees can be constructed using a grammar's production rules, even when augmented with derived recovery rules as described in Section 6.4:

- Water recovery rules (Section 6.4.2) consume parts of the input but do not directly contribute AST nodes.
- Insertion recovery rules for context-free production rules (Section 6.4.3, 6.4.5) do not directly contribute tree nodes.
- Insertion recovery rules for lexical production rules (Section 6.4.3, 6.4.5) only contribute lexical tree nodes that correspond to the recovered lexicals.
- Bridge parsing insertions do not directly contribute tree nodes
- Finally, regional recovery (Section 6.7) only consumes input and does not contribute tree nodes.

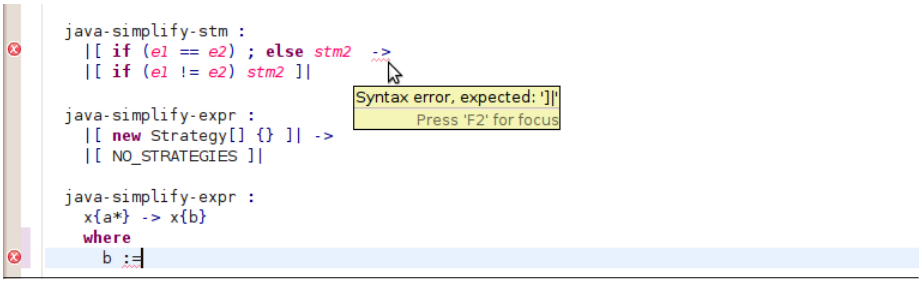


Figure 6.21 An editor for Stratego with embedded quotations of Java code.

The property of well-formedness of trees significantly simplifies the implementation and specification of editor services, 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 that we discuss next.

6.8.3 *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 6.4.5). 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 was 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 6.21 shows a screenshot of an editor for Stratego with embedded Java and 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.

6.8.4 *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.

6.8.5 *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

```
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 6.22 Java `ForEach` production and its derived completion rules.

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 6.4 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 6.22 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 marking terminals as optional patterns. For best results, we generate rules that use placeholder patterns that reflect the signature of the original production, such as an empty list pattern `[]`. By analyzing injection chains in the grammar, it is possible to find sensible placeholder patterns for most non-terminals. For example, for the second rule of Figure 6.22, a pattern `Block([])` can be used in place of the `NULL()` placeholder. Production rules that do not use the `NULL()` placeholders are also suitable for normal error recovery, since they preserve the wellformedness property.

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 comple-

tion 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.

6.9 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.

6.9.1 *Setup*

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.[5]
- *The Dryad compiler*: An open compiler for the Java platform (Chapter 2) written using Stratego-Java.
- *The StringBorg project*: A tool and grammar suite that defines different embedded languages [Bravenboer et al., 2010], providing Java-SQL code.
- *JSGLR*: A Java implementation of the SGLR parser algorithm [Spoofax, 2011].

Syntax error seeding

To produce a test set with syntax errors, we use error seeding techniques based on files from these projects. 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

[5] `http://www.yellowgrass.org/`.

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 selected to represent typical editing errors. We distinguish the following categories for seeded errors:

- *Incomplete construct*, language constructs that miss one or more suffix symbols, e.g. a Java method call `index(`.
- *Erroneous context*, a special case of incomplete constructs, where a subexpression of a construct can not be parsed because of a broken surrounding context, e.g. the assignment in `for (i = 0`.
- *Misplaced construct*, a syntactically valid construct that appears at a point in the program where it is not expected, e.g. the assignment in `class X { int i; i = 3; }`.
- *Missing*, *incorrect* or *superfluous* symbols, parts of language constructs that are either missing or unexpected at a certain location, e.g. the missing `void` keyword in a method declaration `public foo() {}`.
- *Combined* is the case where two or more errors, from the above mentioned categories, appear in the source code. These errors are randomly distributed over the code.

We selected five representative base files from each project, and generated test files using the error seeding rules. 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 297 Stratego-Java test cases, 190 WebDSL test cases, 195 Java-SQL test cases, and 301 Java testcases.

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 14 erroneous Stratego-Java files of increasing size in the interval of 1000–7000 LOC.

Test oracle

We compare the recovery result of each generated syntax error to the AST of the base file. For some files in the *Incomplete construct* and *Broken context* categories, the base files do not realistically reflect the expected result, as information is lost in the test file. 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 these cases we construct an expected result, a priori, by completing these constructs with the minimal amount of symbols possible. In the case of the "for" loop that would be `for (x = 1; x; ) {}`.

Quality Measuring

To measure the quality of a recovery, we compare each recovered test file against the base file or expected file. We use two methods to compare the results. First, we do a manual inspection of the pretty-printed results, following the quality criteria of Pennello and DeRemer [1978]. Following these criteria, an *excellent* recovery is one that is exactly the same as the intended program, a *good* recovery is close to this result, and a *poor* recovery introduces spurious errors. Since this method is arguably a subjective comparison, 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 [van den Brand et al., 2000], 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 a "small diff" (1–10 lines of abstract syntax) roughly corresponds to the *good* qualification with a one or two line change in the concrete syntax source code, and a "large diff" ($>$ 10 lines) corresponds to a *poor* qualification. After a selection of recovery techniques and recovery rule sets, we show both metrics together in a comprehensive benchmark in Section 6.9.2.

Performance Measuring

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 file from the parse time of the incorrect file.

To evaluate the scalability of the technique, we compare the parse times for erroneous and correct files of different size in the interval 1000–7000 LOC.

For all performance measures included in this chapter, an average, collected after several 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.

6.9.2 *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

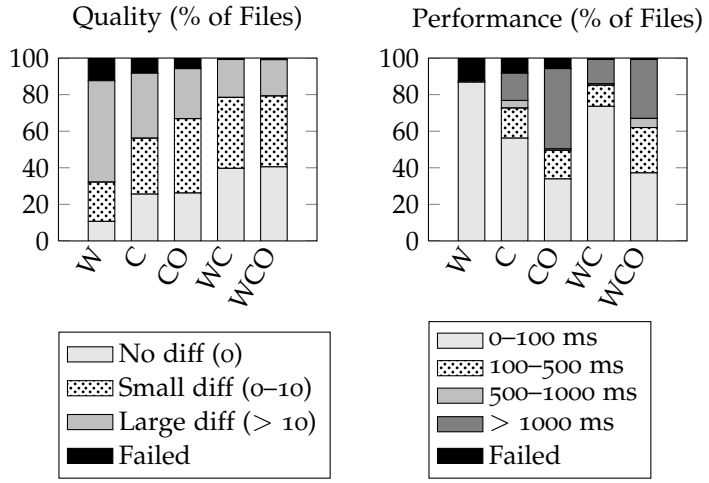


Figure 6.23 Quality and performance (recovery times) using a permissive grammar with different recovery rule sets for Stratego-Java. **W** - Water, **C** - Insertion of closing brackets, **O** - Insertion of opening brackets.

use the Stratego-Java language – 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. For the permissive grammars approach of Section 6.4, 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 6.23. The figure includes results for W, C, CO, WC and WCO for a Stratego-Java grammar, where PG is used in combination with region selection, described in Section 6.7. 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), on the other hand, does not appear to have a large positive effect on the recovery quality, which follows from comparing C to CO, and WC to WCO. Moreover, the insertion rules for opening brackets prove to be costly with respect to performance. 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. The performance diagram shows that this leads to objectionable parse times in certain cases (13% $>$ 1.0 second for WC). For these cases, a practical im-

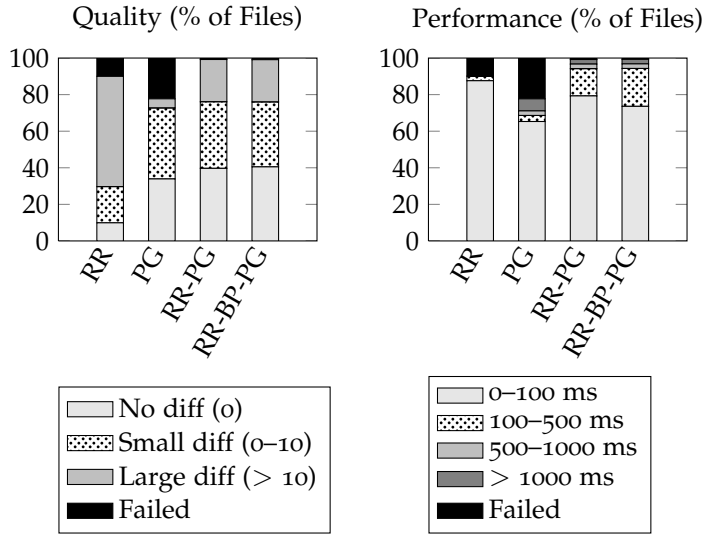


Figure 6.24 Quality and performance (recovery times) using combinations of techniques for Stratego-Java. **RR** - Regional recovery, **PG** - Permissive grammars, **BP** - Bridge parsing.

plementation 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 500 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 chapter: the permissive grammars and backtracking approach of Section 6.4 and 6.5 (PG), regional recovery of Section 6.7 (RR), and bridge parsing of Section 6.6 (BP). We use the WC recovery rule set of Section 6.9.2. 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 both regional recovery and permissive grammars (RR-PG), and then the combination of all three techniques together (RR-BP-PG). The results from the experiment are shown in Figure 6.24.

As shown in the Performance part of Figure 6.24, all techniques give reasonable performance. Consequently, we focus on quality to find the best combination. Considering the Quality part of Figure 6.24 and the results of PG, we see that it has the largest number of failed recoveries (22%), but regardless of this fact it still leads to reasonable recoveries ($<$ 10 diff lines) in the majority of cases (73%). 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 (90%). Combining both techniques (RR-PG), improves the robustness (99%), as well as the precision (76% small or no diff) compared to both individual techniques.

Interestingly, Figure 6.24 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, but it is most effective for programs that use deep nesting of blocks. These 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 [de Jonge et al., 2009]. In this previous study a test set with more focus on scope errors is used, while in this study we use a test set where this kind of error is not so dominant. Unsurprisingly, in this evaluation the cases where the bridge parser contributes to a better recovery are cases where the region selection technique does not detect the erroneous scope as precisely on its own.

Overall benchmark

As an overall benchmark, for this experiment 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 [Kuhn and Thomann, 2006]. In case the JDT parser is unable to recover the entire body of a method, Eclipse uses a secondary parsing approach that analyzes these method bodies for the purpose of content completion. Because of its specialized nature, we have not included it in our experiments. Figure 6.25 shows the quality results acquired for the Java test set, applying the criteria of Pennello and DeRemer [1978] and using diff counts. 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. 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 RR-BP-PG 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., all syntax error categories except *Incomplete construct* and *Broken context* (as described in Section 6.9.1). This allows for a fully automated comparison of erroneous and intended parser output. The results of the experiment are

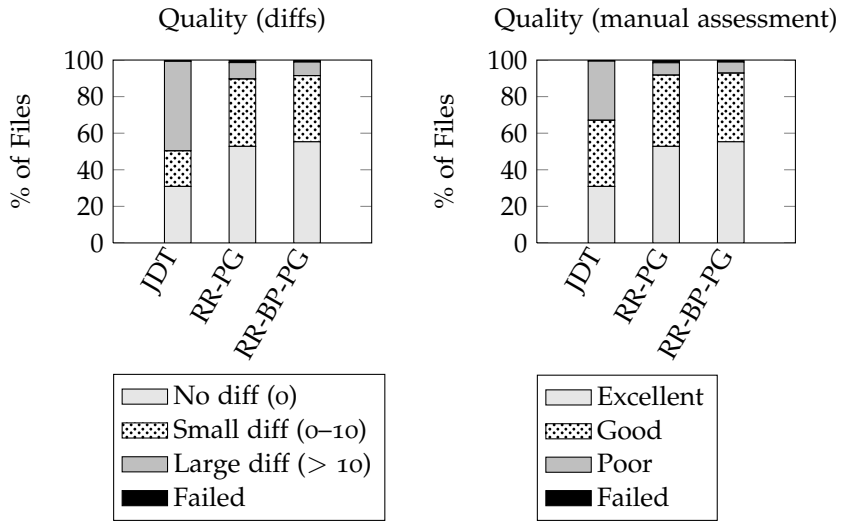


Figure 6.25 Quality of our approach compared to JDT. **RR** - Regional recovery, **PG** - Permissive grammars, **BP** - Bridge parsing, **JDT** - Java Developer Toolkit.

shown in Figure 6.26. 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 RR-BP-PG recovery configuration. To test scalability, we construct a test set consisting of files of different size in the interval 1000–7000 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. By parsing erroneous input of varying size we can observe the time complexity of the parser. We measure parse times as a function of input size, both for syntactically correct files and files with syntax errors. The results, shown as a plot in Figure 6.27, 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.

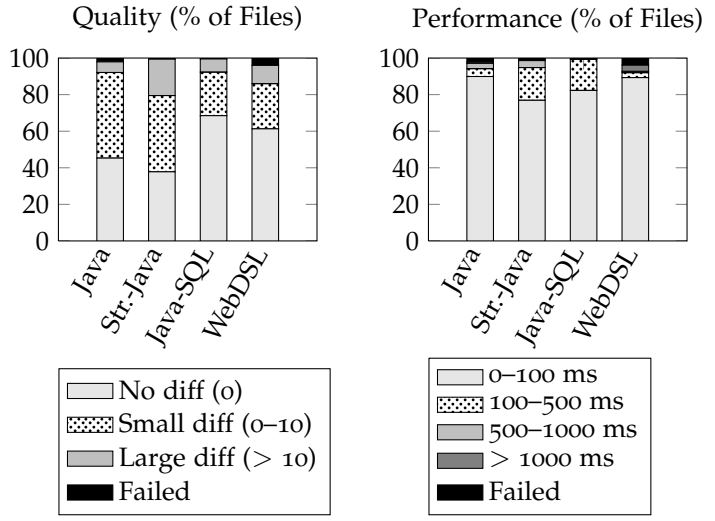


Figure 6.26 Quality and performance (recovery times) for different languages.

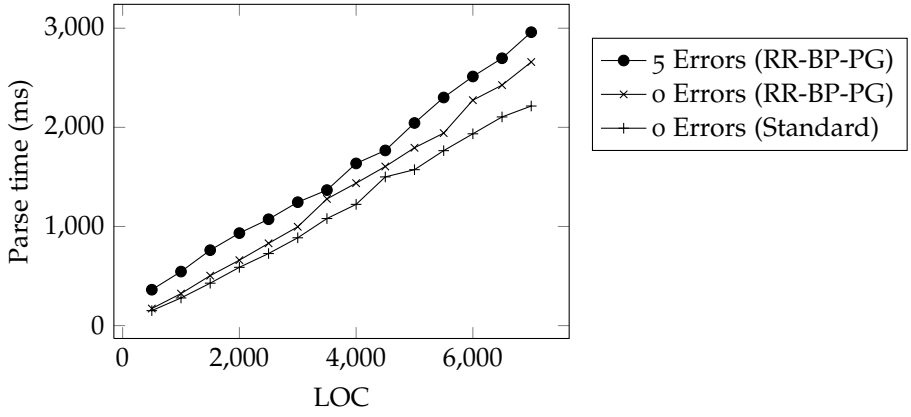


Figure 6.27 Parse times for files of different length with and without errors.

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 linearly increases parse times of syntactically correct files with a factor of about 1.15. 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 ac-

ceptable 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 6.8.5 we discussed the strengths and limitations of our current approach with respect to context completion and introduced special completion rules to overcome these limitations. 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.

We evaluated completion recovery on a set of 314 test cases that simulate the scenario of a programmer triggering the context 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 6.28 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 6.28 shows the results for the completion recovery strategy of Section 6.8.5, using a permissive grammar with the WC rule set and 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 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.

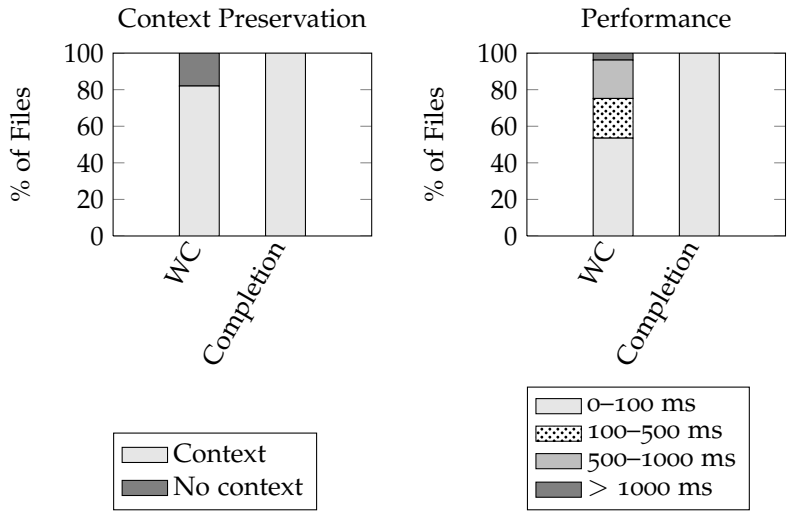


Figure 6.28 Context preservation and performance (recovery times) of the Stratego-Java grammar extended with completion rules (**Completion**) and extended with recovery rules (**WC**).

6.9.3 *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.

6.10 RELATED WORK

There are several different forms of error recovery techniques for LR parsing [Degano and Priami, 1995]. These techniques can be divided in *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. Then, 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. Successful recovery mechanisms often combine more than one technique [De-

gano and Priami, 1995]. For example, panic mode is often used as a fall back method if correction attempts fail.

Burke and Fisher [1987] present a correcting method based on three phases of recovery. The first phase looks for simple correction by the insertion or deletion of a single token. If this does not lead to a recovery, one or more open scopes are closed. The last phase consists of discarding tokens that surround the parse failure location. 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 [Lévy, 1971; Mauney and Fischer, 1988; Pai and Kieburtz, 1980] select a region that encloses the point of detection of an error. Typically, these regions are selected based on nearby marker tokens (also called fiducial tokens [Pai and Kieburtz, 1980]), 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.

The LALR Parser Generator (LPG) [Charles, 1991] is incorporated into IMP [Charles et al., 2007] 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.

6.10.1 *Recovery for Composable Languages*

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

Lavie and Tomita [1993] developed GLR*, a noise skipping algorithm for context-free grammars. Based on traditional GLR with a scanner, their parser determines the maximal subset of all possible interpretations of a file by sys-

tematically 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) [Ford, 2002]. PEGs lack the disambiguation facilities [Visser, 1997c] 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 [Swierstra and Duponcheel, 1996] 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.

6.10.2 *IDE support for Composite Languages*

We integrated our recovery approach into the Spoofax language development environment (Chapter 4). A related project, also based on SDF and SGLR, is the Meta-Environment [Klint, 1993; van den Brand et al., 2001]. 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 provides no error recovery.

Another language development environment is MontiCore [Krahn et al., 2007, 2008], which uses 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 [Krahn et al., 2007, 2008]. 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.

6.10.3 *Island Grammars*

The basic principles of our permissive grammars and bridge parsing are based on the water productions from island grammars. Island grammars [van Deursen and Kuipers, 1999; Moonen, 2001] 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), Klusener and Lämmel [2003] 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 extended set of error reporting capabilities.

More recently, island grammars have also been applied to parse composite languages. Synytskyy et al. [2003] 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.

6.11 CONCLUSION

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

We presented four 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, bridge parsing, to improve the recoveries of scoping constructs by taking indentation usage into account. Fourth, region recovery, to identify regions of syntactically incorrect code, thereby constraining the search space of backtracking and providing a fallback recovery strategy. We evaluated our approach using a set of existing, non-trivial grammars, showing that the techniques work best when used together, and that they lead to 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.

Interactive Disambiguation of Meta Programs with Concrete Object Syntax

7

ABSTRACT

In meta-programming with concrete object syntax, meta programs can be written using the concrete syntax of manipulated programs. Quotations of concrete syntax fragments and anti-quotations for meta-level expressions and variables are used to manipulate the abstract representation of programs. These small, isolated fragments are often ambiguous and must be explicitly disambiguated using quotation tags or types, using names from the nonterminals of the object language syntax. Discoverability of these names has been an open issue, as they depend on the (grammar) implementation and are not part of the concrete syntax of a language. Based on advances in interactive development environments, we introduce *interactive disambiguation* to address this issue, providing meta-programmers with real-time feedback and proposing quick fixes in case of ambiguities. In this chapter we describe a general architecture for automatic generation of mixin grammars that combine meta languages with object languages, and give an algorithm for determining disambiguation suggestions for ambiguities in the combined language. The approach is fully language independent.

7.1 INTRODUCTION

Meta programs analyze, transform, and generate programs. Examples include compilers, interpreters, and static analysis tools. Most frequently, meta programs operate on the abstract syntax of an object language, using a structured representation of programs rather than a textual representation of their source code. Using a structured representation ensures well-formedness, enables compositionality of transformations, and makes it easier to support type safety and hygiene.

In principle, any general purpose programming language can be used as a host for meta-programming. Abstract syntax can be represented and manipulated using data structures and operations available in the host (meta-)language. Unfortunately, the syntactical notation used for these structures and operations is usually verbose and very different from the concise, well-known concrete syntax of the object language. Manipulating the abstract syntax through an API can get tedious, and larger structures are often hard to recognize.

Meta-programming with concrete object syntax as a surface syntax for the abstract representation is, for a great number of situations, a best of both

worlds between a textual and an abstract syntax representation: the meta program is written using the familiar concrete syntax of the object language, while at the meta level, all operations are done on a structured representation of the object program. Concrete object syntax can be syntactically checked as meta programs are compiled. This technique is now supported by many meta-programming systems. Examples include syntax macro systems such as <bigwig> [Brabrand et al., 2002], code generators such as Jak [Batory et al., 1998] and Repleo [Arnoldus et al., 2007], and program transformation systems such as ASF+SDF [van den Brand et al., 2002], Meta-AspectJ [Huang et al., 2008], Rascal [Klint et al., 2009], Stratego/XT [Bravenboer et al., 2008], and TXL [Cordy et al., 1991]. Visser [2002] describes a general architecture for introducing concrete syntax for any object language into any meta language. The approach employs modular syntax definition in SDF and Scannerless Generalized-LR (SGLR) parsing for defining the syntax and parsing the combined meta and object language [Bravenboer and Visser, 2004; Visser, 1997c].

A prevailing problem with embedding concrete object syntax inside a meta-language is that the syntax of the combined meta-and-object language is usually highly ambiguous when the embedding employs a single pair of quotation and anti-quotation symbols. For example, a quoted Java code fragment `|[ i = 2 ]|` can either be an assignment expression, part of a local variable declaration, or even an annotation element initializer.

Two approaches have been proposed to address ambiguity in meta programs, each with their own trade offs and limitations. Perhaps the most straightforward approach is to use tagged quotation and anti-quotation symbols, e.g. writing `Expr |[ i = 2 ]|` using the tag `Expr` to indicate that the quotation contains an expression. The other approach is to use type information of the meta-programming language to attempt to select the intended interpretation of a concrete syntax quotation [Bravenboer et al., 2005; Vinju, 2005]. For example, for an embedding of Java in Java, a statement `Expr assign = |[ i = 2 ]|;` can be disambiguated based on the declared type `Expr`, while the quotation itself does not have to be explicitly tagged.

Unfortunately, both approaches of disambiguation have their drawbacks. On the one hand, using explicit tags adds additional syntactic overhead, especially in cases where the host language is explicitly typed or when the tags are not strictly needed for disambiguation (e.g., for `|[ if (c) m(); ]|`). On the other hand, using type-based disambiguation is inadequate when multiple possible interpretations are type correct. In those cases tagged quotations [Bravenboer et al., 2005] or heuristic filters [Vinju, 2005] must be used instead. Furthermore, type-based disambiguation is highly dependent on, and usually tightly coupled to, both the type system of the meta-programming language and its abstract syntax representation. Adding support for ambiguous quotations and anti-quotations requires rather invasive changes in the existing type checker or to a type-based preprocessor that disambiguates the meta program. For many host languages it may not be practical, nor possible, to modify or extend the type checker.

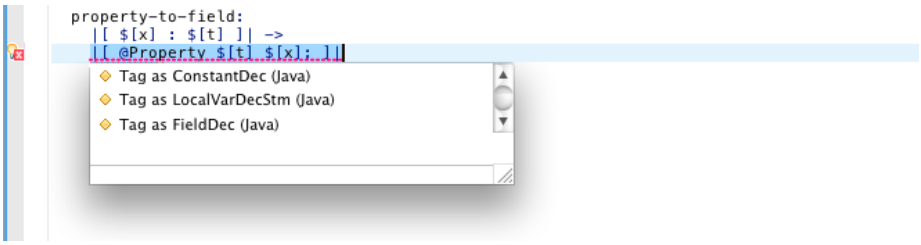


Figure 7.1 Screenshot of a quick fix dropdown menu, listing three possible tags to disambiguate a Java quotation. The menu can be triggered by clicking on the error icon shown in the left margin, or by using a keyboard shortcut. Selecting a suggestion fixes the ambiguity.

A pressing problem that both approaches share is a lack of *discoverability* of quotation tags and types. Meta-programmers may be intimately familiar with the concrete syntax of a language, but may not be well-grounded in the specific names of non-terminals in the syntax definition and the corresponding tag and type names. Having to know these names adds to the learning curve of meta-programming. Furthermore, as object languages evolve, or as additional object languages are added to a meta program, new ambiguities can be introduced for existing code that has not yet, or insufficiently, been explicitly disambiguated. Neither of the two approaches provides developers with adequate feedback if the developer must decide how to fix such an ambiguity.

In this chapter we propose *interactive disambiguation* as a *complementary* approach to tag-based and type-based disambiguation that addresses the concern of discoverability. Our work builds on advances in interactive development environments (IDEs). Modern IDEs aid in discoverability of language features and APIs by providing features such as context-aware code completion and quick fixes. Quick fixes provide a facility to quickly fix common errors by selecting a fix from a list of suggestions. In this chapter we propose to use quick fixes to present developers a list of candidate type or tag names for ambiguous concrete syntax fragments, allowing them to selectively fix problematic ambiguities and quickly discover possible fixes (illustrated in Figure 7.1).

We describe a general architecture for automatic generation of mixin grammars that combine meta languages with object languages, and give an algorithm for detecting possible fixes for ambiguities in the combined language. It allows meta-programmers to write untagged quotations and anti-quotations of concrete object syntax, prompting them with quick fix suggestions only in case of an ambiguity. Our interactive disambiguation approach is *fully language independent* and does not have to be adapted for a specific meta-programming language or its type system.

Outline We begin this chapter with an introduction of meta-programming with concrete object syntax. We then show how concrete object language

```
action-to-java-method:
  |[ action $[Id:name] {
       $[Statement*:s*]
     }
  ]| →
  |[ public void $[Id:name]() {
       $[Stm*:<statements-to-java> s*]
     }
  ]|
```

Figure 7.2 A rewrite rule that uses concrete object syntax notation to rewrite a WebDSL action to a Java method. *s∗* is a meta variable containing a list of statements. *name* refers to an identifier.

syntax can be embedded into meta languages using mixin grammars in Section 7.3. In Section 7.4 we describe how interactive disambiguation can be used to resolve ambiguities in concrete object syntax and give an algorithm to automatically detect possible fixes. We evaluate our approach in Section 7.5 and discuss applications and related work in Section 7.6. We conclude in Section 7.7.

7.2 META-PROGRAMMING WITH CONCRETE OBJECT SYNTAX

Meta-programming systems that support concrete object syntax use *quotations* of object-level code to match and construct code fragments in the object language. Concrete syntax quotations can be parsed at compile-time and translated to their abstract syntax equivalent [Visser, 2002]. Quotations can include *anti-quotations* that escape from the object language code in order to include variables or expressions from the meta language. As an example, Figure 7.2 shows a Stratego [Bravenboer et al., 2008] rewrite rule that uses quotations (indicated by `|[...]|`) and anti-quotations (indicated by `$[...]`) to rewrite a WebDSL [Groenewegen et al., 2010] action definition to a Java method.

Ambiguity As the meta language and object language are combined, *ambiguities* can arise in quotations and anti-quotations. A quotation or anti-quotation is ambiguous if it can be parsed in more than one way, leading to multiple possible abstract syntax representations. Ambiguities in quotations can also occur if the same quoting symbols are used for multiple non-terminals of the object language. Consider for example Figure 7.3. The quotation on the left is ambiguous, as it can represent a single class, a class declaration statement, or a compilation unit, each represented differently in the abstract syntax. The quotation in the middle makes it *explicit* that the intended non-terminal is a compilation unit. The quotation on the right is already unambiguous, because only complete Java compilation units can include a package declaration, and does not have to be explicitly disambiguated.

Similar to quotations, anti-quotations can be ambiguous if they can represent multiple possible non-terminals within the context of a quotation. For example, in a quotation

```
|[ $[q] class X {} ]|
```

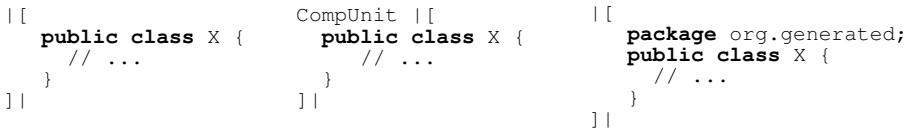


Figure 7.3 Quotations of a Java compilation unit. From left to right: an ambiguous quotation, a quotation that is disambiguated by tagging, and a quotation that is already unambiguous without tagging.

the escaped variable `q` could be a modifier of class X, a separate type declaration, etc.

Quotations and anti-quotations can either be explicitly disambiguated using tagged quoting symbols, as shown in Figure 7.3, or using type information from the meta language [Bravenboer et al., 2005; Vinju, 2005]. Both approaches use names based on the non-terminals in a syntax definition for the object language. Without loss of generality, we focus on a combination of tag-based disambiguation with interactive disambiguation in this chapter. Our implementation is based on Stratego, a largely untyped meta-language, but we describe how the approach could be used in combination with type-based disambiguation in Section 7.4.5.

7.3 CONCRETE SYNTAX EMBEDDING TECHNIQUES

Some meta-programming systems, such as Jak [Batory et al., 1998] and MetaAspectJ [Huang et al., 2008], have been specifically designed for a fixed object language. These systems use a carefully handcrafted grammar or parser for the combined meta and object language. Other systems are more flexible and can be configured for different object languages by combining the grammar for the meta and object languages, and generating a corresponding parser using a parser generator. Building flexible meta-programming systems using traditional parser generators is very difficult, because their grammars are restricted to LL or LR properties. This means that conflicts arise when the grammar of the meta language and the object language are combined [Bravenboer and Visser, 2004], and these must be resolved before the meta-and-object language parser can be constructed. A further impediment to language composition found in traditional parsers is the use of a separate scanner, requiring the use of a single lexical syntax definition for the combined language.

In previous work, Visser [2002] described a general architecture for introducing concrete syntax for any object language into any meta language, using modular syntax definition in SDF and SGLR [Bravenboer and Visser, 2004; Visser, 1997c] parsing. SGLR supports the full set of context-free grammars, which is closed under composition. This makes it possible to compose languages simply by combining grammar modules. The combination of SDF and SGLR also allows the combination of both lexical and context-free

```
module Stratego-Java
imports[1]
  Stratego
  Java
exports context-free syntax
%% Quotations
                "|[" ClassDec "]|" → Term {cons("ToMetaExpr")}
     "ClassDec" "|[" ClassDec "]|" → Term {cons("ToMetaExprTagged1")}
"Java:ClassDec" "|[" ClassDec "]|" → Term {cons("ToMetaExprTagged2")}
                "|[" BlockStm "]|" → Term {cons("ToMetaExpr")}
     "BlockStm" "|[" BlockStm "]|" → Term {cons("ToMetaExprTagged1")}
"Java:BlockStm" "|[" BlockStm "]|" → Term {cons("ToMetaExprTagged2")}
                "|[" CompUnit "]|" → Term {cons("ToMetaExpr")}
     "CompUnit" "|[" CompUnit "]|" → Term {cons("ToMetaExprTagged1")}
"Java:CompUnit" "|[" CompUnit "]|" → Term {cons("ToMetaExprTagged2")}

%% Anti-quotations
"$["                Term "]" → ClassDec {cons("ToMetaExpr")}
"$[" "ClassDec" ":" Term "]" → ClassDec {cons("ToMetaExprTagged")}
"$["                Term "]" → BlockStm {cons("ToMetaExpr")}
"$[" "BlockStm" ":" Term "]" → BlockStm {cons("ToMetaExprTagged")}
"$["                Term "]" → CompUnit {cons("ToMetaExpr")}
"$[" "CompUnit" ":" Term "]" → CompUnit {cons("ToMetaExprTagged")}
```

Figure 7.4 A mixin grammar for embedding object language Java into host language Stratego. For each "interesting" Java non-terminal, the mixin defines productions for quoting and anti-quoting. `ClassDec`, `BlockStm`, `CompUnit` are defined by the `Java` grammar.

syntax into one formalism, and supports the definition of concrete and abstract syntax together in the same production rules.

Grammar productions in SDF take the form $p_1 \ldots p_n \rightarrow s$ and specify that a sequence of strings matching (terminal or non-terminal) symbols p_1 to p_n matches the symbol s. Productions can be annotated with a constructor name n to identify them in the abstract syntax using the `{cons(`n`)}` annotation, where n becomes the name of the abstract syntax tree node.

7.3.1 *Mixin Grammars*

SDF facilitates language embedding by supporting *mixin grammars*: a mixin grammar extends an existing object language grammar with productions for quotation and anti-quotation. As such, the mixin grammar is where the gluing together of the object and meta languages takes place. Figure 7.4 shows an excerpt of a mixin grammar that embeds Java into the Stratego program transformation language.

Quotation productions have the form q_1 *osort* $q_2 \rightarrow$ *msort* and specify that a quotation of object-language non-terminal *osort*, surrounded by (sequences of) symbols q_1 and q_2, can be used in place of meta-language non-terminal *msort*. We sometimes refer to q_1 and q_2 collectively as the *quoting symbols*. In most of our examples, q_1 is `|[` and q_2 is `]|`, or a variation thereof. By extending

[1] This example uses plain imports to combine the meta and object languages. To avoid name clashes between non-terminals of the two grammars, actual mixin grammars use parametrized imports, so that all symbols are postfixed to make them uniquely named.

q_1 with a tag, we can declare to the parser which *osort* to expect inside the quotation.

As illustrated in Figure 7.4, for quotation productions where q_1 is untagged, we use the constructor label `ToMetaExpr`. For tagged quotations, we use `ToMetaExprTagged1`, and `ToMetaExprTagged2` for quotations with a language prefix in their tag. The two forms of tagged quotations can be used where untagged quotation would be ambiguous, or when there are multiple object languages that define the same non-terminal, such as `Id`.

Anti-quotation productions have the form q_1 *msort* $q_2 \rightarrow$ *osort* and specify that an anti-quotation of meta-language non-terminal *msort*, using quoting symbols q_1 and q_2, can be used in place of object-language non-terminal *osort*. We use the constructor label `FromMetaExpr` for untagged quotations, `FromMetaExprTagged` for tagged quotations. For anti-quotations we do not include a language prefix as they can only exist in the context of a guest language quotation.

Mixin grammars can be used to combine a single object language with a single meta language, or multiple object languages with a single meta language. In turn, object languages may include their own extensions and may embed other languages. Using nestable quotations and anti-quotations, meta and object language expressions can be arbitrarily nested.

7.3.2 *Assimilation of Concrete Object Syntax in Meta Languages*

When a meta program with concrete object syntax is parsed according to a mixin grammar, the parser produces a *hybrid parse tree* that combines fragments of meta-language and object-language code. The outer meta-language parse tree then contains subtrees with parse nodes belonging to the object language. In turn, these may contain meta-language subtrees, and so on, as quotations and anti-quotations may be nested. A hybrid parse tree cannot be directly used in either the meta nor object language front ends, as it does not correspond to either one language. The tree can be transformed to a regular parse tree for the meta language through *assimilation* [Bravenboer and Visser, 2004], a process that translates the object code subtrees in the hybrid parse tree into data structures of the meta language. Assimilation is essential to support concrete object syntax in a way that is independent of both the meta- and the object-language, in the sense that neither front end needs to be changed in order for the technique to work.

Consider an example of meta-programming using Java as both the meta language and the object language [Bravenboer et al., 2005]. The abstract syntax representation of the Eclipse JDT can be used to represent Java object programs. A rewrite rule-based assimilator can be used to translate hybrid parse trees into pure Java. The rewrite rules translate quoted concrete object syntax to API calls that construct and operate on the abstract syntax. As a basic example, the following rule translates a quoted `return` statement to a Java expression that constructs a `ReturnStatement` abstract syntax tree node:

```
Assimilate : |[ return; ]| → |[ _ast.newReturnStatement() ]|
```

Similar rules are used to translate other object language constructs to expressions in the meta language. For languages with a fixed, generic representation of object code, such as term rewriting languages, a more general solution is possible. The `meta-explode` tool for Stratego is an example of such a generic tool. It uses a uniform transformation to translate object code fragments to Stratego term building and matching operations [Bravenboer and Visser, 2004].

7.3.3 *Automatic Generation of Mixin Grammars*

Mixin grammars have usually been written by hand in the Stratego setting. The language engineer selects the subset of non-terminals from the object language that will be supported in quotations and anti-quotations. By selecting only a small set of important non-terminals, the engineer can reduce the likelihood of ambiguities in meta programs. With the tagged disambiguation approach, the engineer also decides which productions should receive disambiguation tags. It is not uncommon to see abbreviations for common tags, and the tag naming convention is up to the tastes of the designer of the mixin grammar. Unfortunately, for meta-programmers who have not yet mastered the use of a particular embedding, all these well-intended decisions make it difficult to become fully productive, as the productivity-improving shortcuts must be learned for each new embedding encountered.

In this chapter we take a generative approach to constructing mixin grammars, even though the disambiguator could be used with, and has been tested on, hand-written mixin grammars. The principal advantage of the generative approach is the ease of composition. Given an object language grammar written in SDF, we can quickly embed it into Stratego by running a grammar composition tool. The result provides a consistent and complete mixin grammar – all (anti-)quotations follow a single, predictable style and all necessary non-terminals are supported. In our experience, hand-written mixins usually omit some of the lesser-used non-terminals from the object language, as they require effort to develop and maintain as the object language grammar evolves. This is a non-issue when the mixin grammar can be derived fully automatically from the object language grammar.

Our mixin grammar generator automatically constructs a mixin grammar given a meta and one or more object language grammars, as illustrated in Figure 7.5. All grammars (meta, object, mixin) are fed into the parser generator, which produces the final parse table. This table is used by SGLR to parse the combined meta+object language. In case of ambiguities, the result will be a *parse forest*, i.e., a parse tree that compactly incorporates all the possible parse trees for an input, branching at the points of ambiguities. The forest is inspected by the disambiguator, and the meta-programmer is presented with suggestions to iteratively refine the meta program to prune out all sources of ambiguities. Between each iteration, the meta program is re-parsed.

Selecting non-terminals for quotations and anti-quotations When generating mixin grammars, it is important to provide a sufficiently large set of quota-

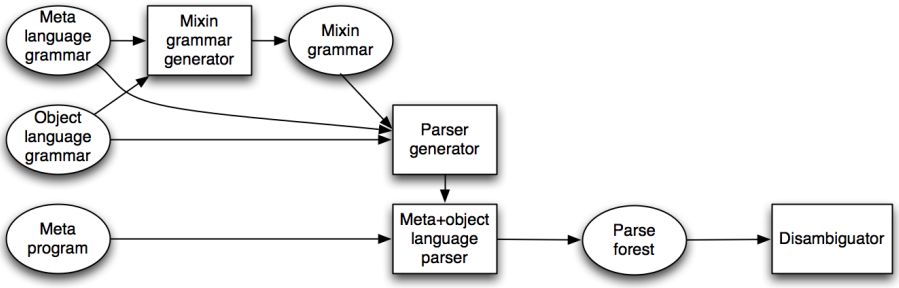


Figure 7.5 Architecture for generating a mixin grammar and a parser for the meta language with an embedding of the object language.

tions and anti-quotations so that each object language construct can be quoted and anti-quoted. However, as grammars include productions with alternatives and injections, care must be taken to provide a *minimal* set, to avoid redundant quoting productions. For instance, given an injection production `ExprName → Expr`, any quotation of an `ExprName` is also a quotation of an `Expr`, so a separate `ExprName` quotation would be superfluous.

The fundamentals of our approach for generating grammars are as follows. We generate quotation productions for all non-terminals that occur on the right-hand sides of context-free production rules with a constructor, which are the productions that contribute a new node to the abstract syntax tree. For example, we include `Expr`, occurring in the production

```
Expr "+" Expr → Expr {left, cons("Plus")}
```

but not `ElemVal`, which only occurs at the right-hand side of the injection productions

```
Expr → ElemVal
Anno → ElemVal
```

Conversely, we generate anti-quotation productions for non-terminals in the left-hand sides of context-free production rules with a constructor. For example, given

```
(Anno|ClassMod)* "enum" Id Interfaces? →
EnumHead {cons("EnumHead")}
```

we include anti-quotations for `Anno`, `ClassMod`, `Id`, and `Interfaces`. However, for an injection `Public → ClassMod`, we do not introduce a new anti-quotation for `Public`.

7.4 INTERACTIVE DISAMBIGUATION

In this section we describe how ambiguities in concrete object syntax can be interactively resolved by analyzing ambiguities and providing quick fix suggestions. We describe different classes of ambiguities and give an algorithm

for automatically determining disambiguation suggestions for a given parse forest and grammar.

7.4.1 *Classes of Ambiguities*

At the grammar level, there are a number of different classes of ambiguities. In this chapter we focus on ambiguities in quotations and anti-quotations. These ambiguities are inherent to the use of mixin grammars, as languages are woven together and fragments must be parsed with limited syntactic context. Disambiguation with tags or types can resolve these ambiguities. Other forms of ambiguities can be caused by the meta or object language, such as with the C language that notoriously overloads the $\star$ operator for multiplication and pointer dereference. Such ambiguities must be retained if they are part of the object language design, otherwise they should be resolved at the grammar level. Ambiguities can also arise by the combination of the two languages if the syntax between the meta and object language overlap. These cannot always be resolved by type-based disambiguation [Vinju, 2005], but can only be avoided by carefully selecting sensible quoting symbols in such a way that they do not overlap with the meta language and object language. Ideally, the symbols are chosen to be aesthetically pleasing characters or character combinations that never occur in either the object or meta language.

7.4.2 *Ambiguity in Quotations*

Quotations are ambiguous when they can be parsed in more than one way, either according to a single object language syntax or according to a combination of multiple object languages. To illustrate ambiguity suggestions for quotations, we revisit the class quotation from Figure 7.3:

```
|[
   public class X {
     // ...
   }
]|
```

Recall that this fragment could represent a single class, a class declaration statement, or a compilation unit. In the syntax, these alternatives are defined by the `ClassDec`, `BlockStm`, and `CompUnit` non-terminals. When faced with an ambiguous syntax fragment, a generalized parser such as SGLR produces a parse forest that branches at the point of the ambiguity, containing all possible subtrees for the ambiguous expression. Figure 7.6 illustrates a parse forest, with at the top a special "amb" tree node that has the three possible interpretations as its children. The gist of our technique is to analyze the different possible parse trees, and have the developer select which alternative they intended.

In the mixin grammar for the embedded Java language (shown in Figure 7.4), there are three untagged productions that produce the three interpretations of our example. The "tagged" productions of the figure parse the same object language non-terminal, but include distinguishing tags. These tags

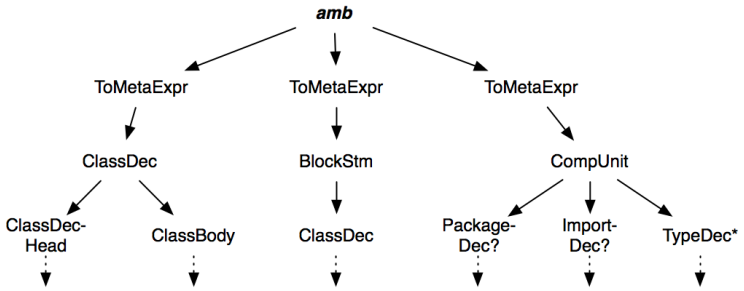


Figure 7.6 The parse forest for the quotation `|[ public class X {} ]|`.

```
CompUnit |[
  class $[x] {
    // ...
  }
]|
```

```
CompUnit |[
  $[x] class Y {
    // ...
  }
]|
```

Figure 7.7 Example of a local ambiguity (left) and a non-local ambiguity (right). In the first only the anti-quotation `$[x]` is ambiguous, in the other the entire contents of the quotation is ambiguous.

can be used to disambiguate the example: when one of the tags `ClassDec`, `BlockStm`, or `CompUnit` is added, there is only one possible interpretation of the quotation. By providing quick fix suggestions that automatically insert one of these three tags, meta-programmers can consider the three options and decide which is the interpretation they intended. In the event that the fragment could also be parsed using a different object language that happens to use the same tag names, the prefixed tags such as `Java:ClassDec` are proposed instead.

7.4.3 *Ambiguity in Anti-Quotations*

Anti-quotations can be disambiguated much like quotations. However, because they always occur in the context of a quotation, there is no need for language-prefixed quoting tags. For anti-quotations we also distinguish *local ambiguity*, where a single anti-quotation can be parsed in multiple ways, and *non-local ambiguity*, where a larger area of the quotation can be parsed in multiple ways. Non-local ambiguities arise as anti-quotations productions typically reduce to multiple possible non-terminals, whereas quotation productions typically reduce to only one, such as `Term` in Figure 7.4.

Local ambiguity Figure 7.7 (left) shows a local ambiguity. The anti-quotation `$[x]` may be interpreted as an identifier or as the signature of the quoted class. The remainder of the quotation is unambiguous, making it trivial to identify the cause of the ambiguity in the parse forest. Local ambiguities are the common case.

Non-local ambiguity Figure 7.7 (right) shows a non-local ambiguity. For this example, the entire body of the quotation can be interpreted in multiple ways: it can be either a class y with modifier x, or a package/import/type declaration x followed by a class y. For non-local ambiguities it is harder to identify the cause of the ambiguity, as the quotation expressions are no longer a direct subtree of the "amb" node as they are in Figure 7.6.

As the cause of non-local ambiguity cannot always be clearly identified, we provide disambiguation suggestions for *all* outermost untagged anti-quotations in ambiguous subtrees. In general, these subtrees are small and only include one anti-quotation, with the leftmost anti-quotation being the cause of the ambiguity. However, in the worst case multiple disambiguation suggestions are provided for anti-quotations that are not inherently ambiguous. A future refinement could be to speculatively parse anti-quotations with the suggested tags in place to filter spurious suggestions. In practice, however, non-local ambiguities are rare, especially those with multiple untagged anti-quotations.

Non-local ambiguities normally only occur for *anti*-quotations as they can match multiple different non-terminals. Quotations all match the same meta language non-terminal (`Term` in Figure 7.4), which means they do not introduce non-local ambiguities.

7.4.4 *Automatically Providing Disambiguation Suggestions*

In this subsection we describe an algorithm to automatically collect disambiguation suggestions, given a (mixin) grammar and a parse forest. The grammar contains the complete set of productions for all meta-expressions, i.e. productions for quotations or anti-quotations. These form the candidate disambiguation suggestions. The parse forest incorporates the possible parse trees and is used to select the right candidates. We made a prototype implementation of the algorithm[2] that integrates into Spoofax, providing a basis for interactive disambiguation of arbitrary SDF-based meta and object languages.

Figure 7.8 shows a pseudocode implementation of the disambiguation suggestions algorithm. At the top, the `CollectSuggestionsTop` function is the main entry point, which gets the parse forest and grammar as its input and returns a set of disambiguation suggestions as its output. For each outermost ambiguous subtree *amb*, it uses the `CollectSuggestions` function to find local disambiguation suggestions.

The `CollectSuggestions` function produces a set of disambiguation suggestions by inspecting each subtree of the *amb* tree node (line 7). For each branch, it searches for the outermost meta-expressions that are not yet completely tagged (line 8). For each meta-expression it determines the production *prod* that was used to parse it (line 9), and its left-hand and right-hand side non-terminals (line 10, 11). For SGLR parse trees, the production is encoded directly in the tree node, allowing it to be easily extracted. Only meta-expressions that are a direct child of *amb* (local ambiguities) and meta-

[2]Available from `http://strategoxt.org/Spoofax/InteractiveDisambiguation/`.

```
CollectSuggestionsTop(tree, grammar)
 1  ▷ Input:
 2      tree – the parse tree or forest
 3      grammar – the mixin grammar
 4  ▷ Output:
 5      results – a set of (treenode,production) tuples
 6  results ← {}
 7  foreach outermost subtree amb in tree where amb has the form amb(...)
 8      results ← results ∪ CollectSuggestions(amb, grammar)
 9  return results

CollectSuggestions(amb, grammar)
 1  ▷ Input:
 2      amb – an ambiguous subtree
 3      grammar – the mixin grammar
 4  ▷ Output:
 5      results – a set of (treenode,production) tuples
 6  results ← {}
 7  foreach child subtree branch in amb
 8      foreach outermost subtree expr in amb where IsTaggable(expr)
 9          prod ← the production for expr
10          lsort ← the non-terminal at left-hand side of prod
11          rsort ← the non-terminal at right-hand side of prod
12          if expr = branch ∨¬IsTagged(prod) then
13              results ← results ∪ { (expr,prod′)
14                                  | prod′ ∈ productions of grammar
15                                    ∧ IsTagged(prod′)
16                                    ∧ prod′ has the form (q1 lsort q2 → rsort)
17                                    ∧ prod′ and prod have the same constructor
18                                      prefix ToMeta or FromMeta }
19  return FilterAmbiguousSuggestions(results)

IsTaggable(t)
1  if t has a FromMetaExpr, ToMetaExpr, or ToMetaExprTagged1 constructor
2  then return true
3  else return false

IsTagged(p)
1  if p has a FromMetaExprTagged, ToMetaExprTagged1,
2     or ToMetaExprTagged2 constructor
3  then return true
4  else return false
```

Figure 7.8 Pseudo-code for collecting suggested quotation symbols.

```
FilterAmbiguousSuggestions(suggestions)
1  ▷ Input: suggestions – set of disambiguation suggestions
2  ▷ Output: results – set of non-conflicting disambiguation suggestions
3  return { (prod,expr) | (prod,expr) ∈ suggestions
4                       ∧ prod has the form (q1 lsort q2 → rsort)
5                       ∧¬∃ (expr′,prod′) ∈ suggestions:
6                       prod′ has the form (q1 lsort′ q2 → rsort′) }
```

Figure 7.9 Filtering ambiguous quotation suggestions.

expression subtrees that do not have any tag (non-local ambiguities) are considered for suggestions (line 12).[3]

For the selected meta-expressions, a set of possible disambiguation suggestions is collected (line 13). These suggestions take the form of tagged meta-expression productions (line 14) that contain the same left-hand and right-hand side non-terminals as the production *prod* (line 15). Of course, we only include quotation productions if the current expression is a quotation, and anti-quotation productions if it is an anti-quotation (line 16). After all corresponding suggestions are collected, the complete set is filtered using the `FilterAmbiguousSuggestions` function (line 18).

The `FilterAmbiguousSuggestions` function filters out any suggestions that are ambiguous with respect to each other. This is useful if two object languages both match a meta-expression *and* they use the same quotation tag `X`. In those cases, inserting the tag `X` would not resolve the ambiguity, and a tag with a language prefix of the form `Lang:X` should be used instead. The definition of `FilterAmbiguousSuggestions` is shown in Figure 7.9. For suggestions with quoting symbols q_1,q_2 (line 3, 4), it only returns those for which there is no other suggestion with the same quoting symbols (line 5, 6).

7.4.5 *Presentation of Suggestions*

In modern IDEs, files are parsed and analyzed with each key stroke, after a small delay passes. This behavior is essential for providing developers with rapid feedback, such as inline error markers and quick fix suggestions. Quick fixes are small program transformations that can be triggered by the developer in case of a code inconsistency or code smell. Quick fixes are unobtrusive: as developers write their program, errors or warnings are marked inline, but it is up to the developer to decide when to address the problems. By using a keyboard shortcut, or by using the mouse to click on an error marker, a popup menu is shown with suggested fixes (shown in Figure 7.1). For interactive disambiguation, quick fixes allow meta-programmers to write concrete

[3] A special case is the `ToMetaExprTagged1` constructor, used for tagged quotations without a language prefix. Suggestions are only provided for *local* ambiguities with this constructor.

syntax for expressions first, allowing the parser to decide whether or not it is ambiguous, proposing appropriate quick fixes when necessary.

The `CollectSuggestionsTop` function is executed each time the result of parsing the meta program is ambiguous. The quickfix menu is populated with the results, and any ambiguity can be addressed by adding the tag name or by inserting the type into the context of the quotation. In order to avoid spurious suggestions for multiple ambiguities, we only provides suggestions for the *outermost* expressions (`Collect-Suggestions`, line 8), allowing meta-programmers to incrementally fix any remaining ambiguities.

As each quotation and anti-quotation production has the form q_1 *lsort* q_2 $\rightarrow$ *rsort*, it is straightforward to extract the quoting symbols q_1,q_2 used for a tag-based disambiguation. To avoid cluttering the quick fix menu, we only show tags without a language prefix, unless only a prefixed tag is available. Actually inserting tags for a selected quick fix suggestion is a simple matter of inserting the quoting symbols into the meta program.

For type-based disambiguation, a transformation is required that inserts the type into the context of the quotation. One approach can be to insert a type cast: as observed in [Bravenboer et al., 2005], inserting a simple cast is usually sufficient to allow the type checker to disambiguate a meta-expression. For example, for an embedding of Java in Java itself, a cast `(CompUnit) |[ public class X {} ]|` is unambiguous. Another approach can be to replace or insert types of declarations in the context of an ambiguous quotation. For example, in Meta-AspectJ [Huang et al., 2008] types of local variables can be inferred using the **`infer`** keyword. If it cannot decide a type deterministically, it uses heuristic rules instead. For instance, a quotation **`infer`** `c = `[class X {}];` defaults to type `ClassDec` while it could also be a `MajCompilationUnit`. Using interactive disambiguation, this ambiguity can be brought to the attention of meta-programmers, so that they can make it explicit what type they intended to use in these cases, in order to avoid subtle, hard-to-detect problems.

7.5 EVALUATION

To evaluate and give further insights into the effectiveness of our approach we used the grammar generator to create two large embeddings. The first is an embedding of Java in Stratego, shown in examples throughout this chapter. We used this generated grammar to parse and disambiguate existing source files that were based on a handwritten mixin grammar that embedded Java in Stratego. The second grammar extends the Stratego-Java language with an additional object language, and is used to evaluate a language evolution scenario that occurs when new object languages or new object language features are introduced.

We used existing source files from the Dryad Java compiler (described in Chapter 2) to test our approach. For our experiment we focused on a subset of the sources that only uses Java concrete object syntax quotations. This includes the full type checker. The files use a total of 55 concrete syntax quo-

tations of a wide variety of different Java language constructs. Most are small quotations, but a few contain complete compilation units, used for compilation and for unit testing. The sources of the Dryad Compiler were written for a hand-written Stratego-Java mixin grammar. Our generated grammar uses the same quotation and anti-quotation symbols, but supports a wider selection of included non-terminals and uses consistent names for tags. After stripping existing tags, all sources could be (ambiguously) parsed using our generated grammar.

Disambiguation suggestions By following the interactive disambiguation suggestions, we could successfully disambiguate the untagged Dryad Compiler sources. We encountered no non-local ambiguities. While a code fragment similar to our example of non-local ambiguity in Section 7.4.3 was included, it was described using abstract syntax. By translating it to concrete syntax form and selecting the appropriate suggestions we could also express it unambiguously using concrete syntax.

After adding the disambiguation tags for the generated Stratego-Java grammar, we introduced the WebDSL [Groenewegen et al., 2010] domain-specific web programming language as an additional object language and re-parsed the files. WebDSL is a curly-bracket programming language and uses binary operators very similar to those in Java. This leads to a number of new ambiguities in the source files, as they were not written to disambiguate between Java and WebDSL. Since both the WebDSL and Java syntax definitions use a non-terminal named `Expr` for expressions, even tagged quotations such as `Expr |[ $[Expr:`x`] == $[Expr:`y`] ]|` would be a valid quotation for either language. The quick fixes provided language-prefixed suggestions for these cases and helped us through this transition process. Still, a future refinement could be to add an option to apply these operations in batches for this scenario. The comprehensiveness of the generated grammar paid off for this scenario, as the grammar includes disambiguation tags with and without language prefixes, even though the latter are only rarely used.

Generated vs hand-written mixin grammar Compared to the original sources written for the handwritten Stratego-Java mixin grammar, the revised sources contain many more disambiguation tags. This is because the generated grammar supports a much wider range of non-terminals in quotations and anti-quotations. For example, using the hand-written Stratego-Java grammar, a quotation `|[ $[Expr:`x`] = $[Expr:`y`] ]|` would be parsed only as an assignment expression, while it could also be an annotation element initializer or part of a variable declaration. Since the latter two are rarely quoted in meta-programs, the designers of the mixin grammar decided not to include quotations for those cases. In our generated grammar, such special cases are included, although we take special care not to include non-terminals from unrestricted injection productions. This means they need to be disambiguated using additional tags. Still, using a generated grammar avoids the need for the expertise and manual labor required for the development of a handwritten mixin grammar. For users of the grammar it avoids the idiosyncrasies that

are common with handwritten mixin grammars. The additional tags can be avoided by using a complementary disambiguation approach such as type-based disambiguation. In the future we would also like to investigate *runtime* disambiguation for meta programs that are not (fully) typed.

Implementation and performance Runtime efficiency of interactive disambiguation is determined by the efficiency of the interactive disambiguator itself and by the efficiency of the parser. In the disambiguator implementation we cache operations such as collecting productions from the grammar for efficiency, while in the algorithm we described in Section 7.4 we abstract from those optimizations. All in all, the number of computations required at disambiguation time are rather limited: the disambiguator is a depth-first traversal for each ambiguous branch, a set of hashtable lookups to find the corresponding productions – one per branch – and finally a filtering of the set of suggestions to eliminate conflicting suggestions suffices. Experience with the prototype tells us that the performance overhead of the suggestions algorithm almost negligible. The real performance cost of our approach comes from generalized parsing, which is bounded to cubic time complexity [Johnstone et al., 2004]. Experience from our evaluations on the Java and WebDSL embeddings using our prototype tells us that the time taken to parse and compute suggestions are within acceptable bounds for interactive use. A limitation of the current prototype is that is a stand-alone implementation that has to be manually triggered and does not yet integrate into the standard editors of meta-languages such as Stratego.

Summary Our quick fixes correctly identified disambiguation suggestions. They provide language-prefixed suggestions in case of ambiguities between multiple different object languages. Using a generated grammar saves the time and effort spent in creating the mixin grammar, and makes it very easy to extend an embedding with an additional object language. Still, while interactive disambiguation makes it much easier to add tags for disambiguation, the approach works best when used together with a complementary approach to avoid excessive tagging.

7.6 DISCUSSION AND RELATED WORK

There are different options for embedding object language code into a meta language, including string embedding or interpolation, the use of abstract syntax, concrete object language syntax, and the use of syntax macros in cases when the object and meta language are the same. Arguments used when evaluating one choice over another include the level of well-formedness guaranteed, the conciseness and readability of the notation, its suitability for code generation and pattern matching, the difficulty of implementation. Expressing code fragments in the syntax of the object language is generally held to be the most readable alternative, which places string interpolation, concrete syntax embeddings and certain syntax macro systems at the top when rating

for readability. Vinju [2005] gives an overview and shows examples of some of the different approaches.

String interpolation and template engines String template engines such as StringTemplate [Parr, 2004] and Xpand [Efftinge et al., 2008] remain a pragmatic and popular choice for all kinds of code generation. Some programming languages, such PHP and Ruby, come with string interpolation built in. String-based code generation is particularly common for constructing SQL queries, as found in database-centric applications. Unfortunately, as string interpolation is notoriously poor at guaranteeing syntactic well-formedness of the output, it is a preferred vector for injection attacks aimed at web-based services [Bravenboer et al., 2010]. Since string interpolation engines always regard their generated product as unstructured text, ambiguities are a non-issue for this approach.

Using syntax embedding techniques, the well-formedness short-comings of pure string interpolation can be rectified. StringBorg implements a general approach for syntax-safe embedding of domain-specific languages into general-purpose programming languages [Bravenboer et al., 2010]. Repleo [Arnoldus et al., 2007] is a syntax-safe template engine that generates strings based on a syntax embedding. Both rely on a parser to guarantee well-formedness of the embedded code. Similar to our approach, they do this by composing host and embedded language grammars using SDF and SGLR. A crucial difference is that they are ultimately only concerned with the lexical structure of the object program. This simplifies ambiguity handling. When faced with ambiguous branches in a parse forest, their solution is to consider each branch and if one of them successfully generates a well-formed string they ignore the remainder, under the assumption that they would unparse to the same string. This assumption holds when the quoting symbols cause no ambiguities between host and embedded language.

Wachsmuth [2009] and Heidenreich et al. [2009b] describe techniques for deriving syntax-safe template languages from the grammar of any object language. Wachsmuth's work is founded on natural semantics, and guarantees the syntactic correctness of all outputs generated from valid templates. The paper only focuses on the abstract syntax, and does not consider the ambiguity problems that arise with a parser implementation. Heidenrich et al. consider parser generation, but do not employ a generalized parser, which means they cannot handle ambiguities at parse-time. Their proposed solution is to help the meta-programmer rule out ambiguities by design (i.e., requiring some form of explicit tagging always), through interactive refactorings. This ensures that the combination of meta and object language is unambiguous.

Common to all string interpolation approaches is their focus on code generation. They may employ parsers, but the abstract representation is not used for anything but well-formedness checking. This makes the approaches hardly suitable for meta-programming where structural pattern matching on object code is required.

Concrete object syntax The technique of expressing object code fragments using concrete syntax tries to obtain the best of all worlds: both readable code, and a structured representation with well-formedness guarantees. There are now many meta-programming systems that support meta-programming with concrete object syntax [Bravenboer et al., 2005; Vinju, 2005]. When manipulation – and not only generation – of object code is desirable, removing all ambiguities of the quoted object becomes necessary, either by tag- or type-based disambiguation. Still, keeping the syntactic noise to a minimum is desirable, and therefore using a minimum of tags is often the goal.

In previous work, we described a general, language-independent approach for meta-programming with concrete object syntax [Visser, 2002], and applied it in several case-studies [Bravenboer and Visser, 2004] using tag-based disambiguation. The approach was further refined to type-based disambiguation [Bravenboer et al., 2005]. In this chapter, we introduce a complementary, language- and type system-independent approach for interactive disambiguation. Our technique reduces the need for tagging the (anti-)quotations by reducing the need for quotation noise; the programmer need only quote where absolutely necessary, and is interactively helped to introduce quotation symbols where required. Interactive disambiguation is a complementary approach to both tag-based and type-based disambiguation, however, in this chapter we focused on a tag-based approach. While we emphasize interactivity, it should be noted that the technique does not necessarily require an IDE. Quotation alternatives can also be displayed as part of the build process and used with generic text editors that may not interactively parse and analyze the meta language source code.

Tag- and type-based disambiguation Interactive disambiguation is independent of the type system of the host language and its interaction with symbols in the object language. It is highly resilient to changes in the meta-program as it is edited and works even when meta-programs are not type-correct. The approach mixes well with tag-based disambiguation: with interactive disambiguation, explicit disambiguation tags are required only where the parser cannot decide otherwise, and the meta-programmer is assisted to only introduce them where necessary.

Interactive disambiguation can also be combined with type-based disambiguation. It can assist in cases where type-based disambiguation is inadequate, as multiple type-based interpretations are type correct. These cases particularly arise when combining the technique with type inference, as seen with Meta-AspectJ [Huang et al., 2008], or when forgoing quoting symbols that distinguish between the meta and the object language, as observed by Vinju [2005]. Interactive disambiguation can also assist when programs are not yet type consistent, providing suggestions for inserting type declarations or type casts. As meta-programmers are more likely to be more familiar with the concrete syntax of a language than its abstract syntax, suggestions also aid the programmer in quickly discovering the types of non-terminals without consulting the grammar or its documentation.

Importance of good quoting symbols Carefully selecting sensible quoting symbols when mixing the object and meta languages is very important for preventing needless ambiguities. Ideally, the symbols are chosen to be aesthetically pleasing characters or character combinations that never occur in either the object or meta language. This way ambiguities because of overlap between the meta and object language are avoided.

Vinju [2005] described how concrete object syntax quotations and anti-quotations can be used *without* quoting symbols. As he indicated in his paper, it is ultimately a matter of taste whether or not this leads to better readability. He also showed that type-based disambiguation is often inadequate to resolve ambiguities resulting from unquoted embeddings. To resolve these, he used a set of heuristic filters. A problem with heuristic filters is that they are often hard to predict by meta-programmers, making meta programs that rely on them harder to understand and maintain. In this chapter we have made a case for using explicit, interactive disambiguation instead. We assist the meta-programmer in explicitly resolving ambiguities for those cases where the parser or type checker cannot decide which interpretation was intended.

We have not considered meta-variables in this chapter, e.g. assigning special (anti-)quotation meaning to certain identifiers, such as `e` for expressions and `stm` for statements. In general, meta-variables are hard to discover for the tools, and they are error-prone for the user. One common pitfall is for meta-programmers unfamiliar with a given embedding to accidentally hit a meta-variable inside their quoted code, without intending to, e.g. `|[ catch(Exception e) { ... } ]|`, where `e` was intended as an object language variable but is interpreted as a meta language variable.

Future work Stratego is largely untyped, ruling out type-based disambiguation for our present prototype. A typed variant of Stratego [Lämmel, 2003] might be a suitable testbed for experiments combining interactive disambiguation and type inference. One goal would be to get rid of heuristics, by letting the programmer interactively, and thus more predictably, resolve these statically.

On the dynamic side, we have had promising experimental results using *runtime disambiguation*, where the decision of the correct interpretation of a meta-expression is delayed until run-time, when the actual values of meta-level expressions are known. Based on a static analysis of the meta-program, it is possible to determine which quotations can safely be disambiguated at runtime.

While we have discounted meta variables in this chapter, we do not rule them out. They could be supported with our approach, and be made less error-prone by letting the IDE highlight meta-variables inside concrete syntax fragments.

It would be interesting to apply the approach of interactive disambiguation for applications other than meta-programming. One particular area of interest is that of pluggable language components, where different parties can create their own language extensions that can be combined by "end programmers."

Interactive disambiguation could be applied in this area to avoid syntactic conflicts between different plugins.

7.7 CONCLUSION

Modern IDEs significantly increase the productivity of programmers, by providing many different kinds of editor services specific to the syntax and semantics of a language. Language workbenches are IDEs that integrate meta-programming tools and provide a comprehensive, interactive environment for working with meta-programming languages and for developing program analyses, transformations, and new languages. Meta-programming languages can use concrete object syntax to clearly and concisely express object program fragments. Interactive disambiguation is a technique to help identify and interpret ambiguities in these fragments, and to semi-automatically disambiguate them by inserting disambiguation tag or type names. The technique relieves meta-programmers from the burden of learning these names for each embedded language by heart (or by rote), and allows them to be explicit rather than rely on fixed defaults or heuristics to select the intended interpretation.

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 the anonymous reviewers of SLE 2010 for providing useful feedback on an earlier version of this chapter.

Integrated Language Definition Testing: Enabling Test-Driven Language Development

8

ABSTRACT

The reliability of compilers, interpreters, and development environments for programming languages is essential for effective software development and maintenance. They are often tested only as an afterthought. Languages with a smaller scope, such as domain-specific languages, often remain untested. General-purpose testing techniques and test case generation methods fall short in providing a low-threshold solution for test-driven language development. In this chapter we introduce the notion of a *language-parametric testing language (LPTL)* that provides a reusable, generic basis for declaratively specifying language definition tests. We integrate the syntax, semantics, and editor services of a language under test into the LPTL for writing test inputs. This chapter describes the design of an LPTL and the tool support provided for it, shows use cases using examples, and describes our implementation in the form of the Spoofax testing language.

8.1 INTRODUCTION

Software languages provide linguistic abstractions for a domain of computation. Tool support provided by compilers, interpreters, and integrated development environments (IDEs), allows developers to reason at a certain level of abstraction, reducing the accidental complexity involved in software development (e.g., machine-specific calling conventions and explicit memory management). *Domain-specific* languages (DSLs) further increase expressivity by restricting the scope to a particular application domain. They increase developer productivity by providing domain-specific notation, analysis, verification, and optimization.

With their key role in software development, the correct implementation of languages is fundamental to the reliability of software developed with a language. Errors in compilers, interpreters, and IDEs for a language can lead to incorrect execution of correct programs, error messages about correct programs, or a lack of error messages for incorrect programs. Erroneous or incomplete language implementations can also hinder understanding and maintenance of software.

Testing is one of the most important tools for software quality control and inspires confidence in software [Beck, 2003]. Tests can be used as a basis

for an agile, iterative development process by applying test-driven development (TDD) [Beck, 2003], they unambiguously communicate requirements, and they avoid regressions that may occur when new features are introduced or as an application is refactored [Beizer, 2002; Myers, 2008].

Scripts for automated testing and general-purpose testing tools such as the xUnit family of frameworks [Hamill, 2004] have been successfully applied to implementations of general-purpose languages [Goodenough, 1980; Wichmann and Ciechanowicz, 1983] and DSLs [Gómez et al., 2001; Strembeck and Zdun, 2009]. With the successes and challenges of creating such test suites by hand, there has been considerable research into *automatic generation* of compiler test suites [Boujarwah and Saleh, 1997; Kossatchev and Posypkin, 2005]. These techniques provide an effective solution for thorough black-box testing of complete compilers, by using annotated grammars to generate input programs.

Despite extensive practical and research experience in testing and test generation for languages, rather less attention has been paid to supporting language engineers in writing tests, and to applying TDD with tools specific to the domain of language engineering. General-purpose testing techniques, as supported with xUnit and testing scripts, require significant investment in infrastructure to cover test cases related to syntax, static semantics, and editor services, specific for the tested language. They also use isolated test programs to test particular language aspects, requiring a certain amount of boilerplate code with each test program (e.g., import headers), and require users to manually code how to execute the test (parse/compile/run/etc.) and how to evaluate the result and compare it to an expectation. Tool support for writing test cases and specifying test conditions is lacking, particularly for negative test cases where errors are expected. Test generation techniques are an effective complementary technique for stress testing complete compiler implementations, but are less effective during the development of a new language definition.

In this chapter, we present a novel approach to language definition testing by introducing the notion of a *language-parametric testing language (LPTL)*. This language provides a reusable, generic basis for declaratively specifying language definition tests. It can be *instantiated* for a specific *language under test* through language embedding: we integrate the syntax, semantics, and editor services of a language under test into the LPTL for writing test inputs.

For the generic basis of the LPTL, we provide general constructs to configure test modules and to declaratively specify test conditions. Based on the observable behavior of languages implementations, we selected an open-ended set of test condition specification constructs. These form the heart of the testing language, and support writing tests for language syntax, static semantics, editor services, generated code, and dynamic semantics.

To support language engineers in writing and understanding tests, we show how full language-specific IDE support can be provided for writing test cases. The instantiated LPTL provides editor services such as syntax high-

lighting, syntax and semantic error marking, and content completion, based on the definition of the language under test.

Using an LPTL significantly reduces the threshold for language testing, which is important because such a threshold is often a reason for developers to forgo testing [Gamma and Beck, 1998]. First, by providing a reusable infrastructure for language test specifications that facilitates test execution, analysis, maintenance, and understanding. Second, by providing full language-specific IDE support for writing tests.

The contributions of this chapter are as follows.

- The design of a generic, declarative test specification language for language definition testing.
- A fully language-agnostic approach to language embedding that incorporates syntactic, semantic, and editor service aspects of a language under test.
- The implementation of such a testing language as the *Spoofax testing language*[1] and a description of its implementation architecture.

Outline We begin this chapter with background on language definitions. Next, we discuss the design of a language-parametric testing language from three angles: first from a purely linguistic perspective in Section 8.3, then from a tool support perspective in Section 8.4, and finally by illustrating use cases with examples in Section 8.5. Our implementation architecture is described in Section 8.6. We conclude with related work on language testing approaches and directions for future work.

8.2 BACKGROUND: LANGUAGE DEFINITIONS

The development of a compiler for a DSL for a domain comprises many tasks, ranging from construction of a parser to a semantic analyzer and code generator. In addition to a compiler, the construction of an integrated development environment (IDE) is essential, as developers increasingly rely on IDEs to be productive. Traditionally, a lot of effort was required for each of these tasks. Parsers, data structures for abstract syntax trees, traversals, transformations, and so on would be coded by hand for each language. The implementation of editor services expected from modern IDEs, such as syntax highlighting, an outline view, reference resolving for code navigation, content completion, and refactoring, added to this already heavy burden. This meant that a significant investment in time and effort was required for the development of a new language.

Language engineering tools Nowadays there are many tools that support the various aspects of language engineering, allowing language engineers to

[1] Distributed as part of Spoofax, available from `http://www.spoofax.org/`.

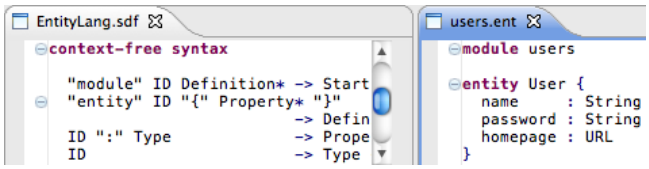


Figure 8.1 Language workbenches can combine language definition (left) and language use (right).

write high-level *language definitions* rather than handwrite every compiler, interpreter and IDE component. Particularly successful are parser generators, which can generate efficient parsers from declarative syntax definitions. For semantic aspects of languages, there are numerous meta-programming languages and frameworks. For the development of IDE support there are also various tools and frameworks that significantly decrease the implementation effort.

Language workbenches are a new breed of language development tools [Fowler, 2005a] that integrate tools for most aspects of language engineering into a single environment. Language workbenches make the development of new languages and their IDEs much more efficient, by *a)* providing full IDE support for language development tasks and *b)* integrating the development of the language compiler/interpreter and its IDE. Examples of language workbenches include MPS [Voelter and Solomatov, 2010], MontiCore [Krahn et al., 2008], Xtext [Efftinge and Voelter, 2006], and our own Spoofax (Chapter 4).

Language workbenches allow for an agile development model, as they allow developers to use an IDE and to "play" with the language while it is still under development. Figure 8.1 shows a screenshot illustrating how they can combine the development of a language with the use of generated editors for that language. Once a syntax definition has been developed (at the left), they can generate an editor with basic editor services such as syntax highlighting and syntax error marking. From there, language engineers can incrementally and iteratively implement new language components and editor services.

Examples and tests Many new languages start with sketches. Sketches of example programs or code snippets that solve a particular problem in the domain of the language. With language workbenches, it is common practice to maintain a "scratch pad" with some example program that focuses on new features that are under development. Language engineers can interact with it in various ways. For example, they can introduce type errors ("does the type checker catch this?"), control-click on an identifier ("does this hyperlink point to the right place?") or generate and run code for the example.

Example programs quickly grow unwieldy, and, as commonly seen with interactive systems [Myers, 2008], they are often thrown away once they show satisfactory results. This is a major problem as these test cases are a valuable investment that simply disappears after testing is completed. The problem results from the high threshold of setting up tests for languages and their IDE,

running tests in an automated fashion, ensuring that the observed behavior complies with the expectations, and so on. The effort does not compare to the ease of testing it interactively. Without better tool support, proper testing remains an afterthought, even in an integrated language engineering environment such as a language workbench.

8.3 TEST SPECIFICATION LANGUAGE DESIGN

In this section we describe the design of a language-parametric testing language and show how it can be used to test different aspects of language definitions. The design of the language is highly intertwined with the tool support that is provided for it and how users can interact with it. We discuss those aspects of the language in the next section.

The central goal set out for design of an LPTL is to provide a low-threshold test specification language that forms the basis for a reusable infrastructure for testing different languages. The design principles of this language are as follows:

P1 Provide a language-agnostic framework. The language should provide a generic, language-agnostic basis that caters for a wide spectrum of different types of tests.

P2 Maintain implementation independence. The language should emphasize black-box testing [Myers, 2008], allowing tests to be written early in the design process, and abstracting over implementation specifics.

P3 Support language-specific instantiation. It should be possible to instantiate the language for a specific *language under test*, thereby integrating the two languages and the tool support provided for the two.

P4 Facilitate series of tests with test fixtures. The language should support test fixtures to specify series of tests with common boilerplate code such as import headers.

In the remainder of this section we show how these principles can be realized, and show the design of the Spoofax testing language, our implementation of an LPTL.

A language-agnostic framework (P1) Key to providing a reusable, language agnostic framework is providing a generic language that can quote test fragments and can specify conditions to validate for those tests. We realize this using the following syntax to specify tests:

```
test description [[
  fragment
]] condition*
```

where *description* is a string that describes the current test case in human readable form and *fragment* is an embedded program or program fragment in

```
test Cannot assign an integer to a string [[
  module Example

  function f() {
    var s : String = 1;
  }
]] 1 error
```

Figure 8.2 A basic mobl test case.

the language under test. The *condition** elements specify the expectations of the test case, and control what test is performed for the input fragment.

Figure 8.2 shows an example test case where we test the mobl [Hemel and Visser, 2011] language, a domain-specific language for mobile applications. In this example we declare a local variable `s` of type `String` and assign an integer literal value to it. This is a *negative test case*: a value of type `String` would be expected here. The conditions clause of this test case indicates that exactly one error was *expected* here, which means that the test case *passes*.

To ensure the test specification syntax is language agnostic, it cannot have any specific elements for a particular language. The set of different possible tests that can be specified must be based on a generic interface for observable behavior commonly supported by languages. Furthermore, the quotation mechanism cannot be limited to only a single, fixed sequence of characters such as the double square brackets above, since those may also be in use by the language under test. In the Spoofax testing language we address this issue by supporting additional quotation markers such as `[[[`, `[[[[`, and variations with series of `""` quotes.

Implementation independence via black-box testing (P_2) Black-box tests [Myers, 2008] test the interface rather than the internal workings of a software artifact. They are independent of the internal workings of a tested artifact or unit and focus only on its observable behavior (output) given some input. The example of Figure 8.2 illustrates this principle, as it reveals nor assumes anything about the implementation of the language under test.

As inputs of a black-box language test we use *1)* the quoted fragment of code, *2)* the conditions clause, and *3)* selections of code within the quoted fragment. The first two were illustrated in the example of Figure 8.2. The test input fragment indicates the input to feed to the language implementation, and the conditions clause indicates what action to trigger and what check to perform. In our example, the `1 error` clause indicates that the implementation should perform semantic checking and that only one error is expected. Other clauses can specify other actions and checks such as syntactic checks, name resolution, refactoring, and execution. In some cases, they specify user input such as a name for a rename refactoring or command-line arguments for an execution test. We give an overview of these facilities at the end of this section.

For many forms of tests it is useful to specify some form of selection in the input fragment. For instance, consider Figure 8.3, which shows a content completion test. The double brackets inside the quotation indicate a selected

```
test basic completion [[
  module Example
  function get(argument : String) : String {
    return [[arg]];
  }
]] complete to "argument"
```

Figure 8.3 A test to verify that the selected identifier `arg` completes to `argument`.

part of the program where content completion would be applied. Selections can indicate identifiers or larger program fragments for tests of features such as reference resolving, content completion, and refactorings. Some tests use multiple selections, in particular for reference resolving, where both a name reference and its declaration can be selected.

Language-specific instantiation (P_3) Instantiation of the testing language for a specific language under test requires that the test suite specifies which language to use for its test cases. Optionally tests suites can also specify which syntactic start symbol they use, e.g. a module, statement, or expression. Based on this information, it becomes possible to evaluate the test cases by invoking or interpreting the language implementation. To fully realize language-specific instantiation, the IDE that supports the testing language can also be adapted to incorporate the syntax and semantics of the tested language, as we illustrate in the next section.

We organize suites into one or more modules (i.e., files), where each module has a series of test cases and its own configuration. For each module we use headers that indicate their name, what language to use, and what start symbol to use:

```
module test-assignments
language Mobl
start symbol Expression
```

Of these headers, only the `language` header is compulsory. Once the language under test is specified, the LPTL and the language under test are composed together, and quoted test fragments are no longer treated as mere strings but as structured part of test specifications.

Test fixtures (P_4) A common technique in testing frameworks such as the xUnit [Hamill, 2004] family of frameworks, is to use `setup()` and `tearDown()` methods[2] to create test fixtures. These methods respectively initialize and de-initialize common state for a series of tests. For language testing, the same principle can be applied. We use *setup blocks* to specify common elements for tests.

Consider again our basic example of Figure 8.2. Suppose we want to write multiple, similar tests for assignments in the same context. This would require writing the same boilerplate in for each test case: a module, function, and variable declaration. These commonalities can be factored out using setup

[2]Note that more recent implementations such as JUnit 4 often use annotations for this functionality.

```
language mobl

setup [[
  module Example

  function f() {
    var s : String = "";
    [[...]]
  }
]]

test Cannot assign an integer to a string [[
  s = 1;
]] 1 error

test Can assign a string to a string [[
  s = "a string";
]]
```

Figure 8.4 A testing module with a shared setup block.

blocks, as shown in Figure 8.4. Instead of writing the same boilerplate for every test case, it suffices to write it only once using the shared setup block. The contents of the setup block serves as a template for the test cases, where the `[[...]]` placeholder is filled in with the contents of each test block.[3] The placeholder is optional: if none is specified, we assume it occurs at the end of the setup block.

Setup blocks are essentially a purely syntactic, language-agnostic feature, but they are highly flexible. They can be used to factor out boilerplate code from individual tests, such as module and import declarations. They can also be used to declare types, functions and values used in test cases. Much like with the `setup()` and `tearDown()` methods of xUnit, they can also be used to perform tasks such as database initialization for test cases that execute tested programs.

Overview We conclude this section with an overview of our test specification syntax, shown in Figure 8.5. So far we already discussed test configuration, test cases, and tested fragments. The table also shows the possible condition clauses for syntactic, static semantic, and dynamic semantics tests, and the patterns that can be used with some condition clauses. We further illustrate those elements with a series of examples in Section 8.5.

8.4 TEST SPECIFICATION INTERACTION DESIGN

Tool support is an important factor for productivity with programming languages. For the domain of testing in particular, good tool support is important to lower the threshold of testing [Gamma and Beck, 1998]. In this section we show how tool support in the form of IDE components can be applied to lower

[3] Note that we overload the quotation brackets to specify anti-quotations for selections and for placeholders in setup blocks. This design ensures minimal syntactic interference with the language under test, as language engineers can pick which quotation markers to use (e.g., `[[`, `[[[`, and so on).

Configuration	
module *name*	Module name *name*.
language *language*	Use *language* as the language under test.
start symbol *symbol*	Use syntactic start symbol *symbol*.
Test cases	
test *description f c**	A test case where *f* must satisfy conditions *c**.
test *description e*	A white-box test case where freeform test condition *e* must be satisfied.
setup *description f*	A setup block for test cases in this module. Can use `[[...]]`[4] in *f* for placeholders.
Tested fragments (*f*)	
`[[(`*code*`|[[`*code*`]])* ]]`[4]	Partial code fragments in the language under test.
Test conditions (*c*)	
succeeds	Fragment succeeds parsing and has no semantic errors or warnings (default condition).
fails	Fragment has semantic errors or warnings.
parse *pattern*	Fragment parses according to *pattern*.
n **error** | *n* **errors**	Fragment has exactly *n* semantic error(s).
n **warning** | *n* **warnings**	Fragment has exactly *n* semantic warning(s).
/regex/	Fragment has an error or warning matching regular expression *regex*.
resolve(#*n*)?	Successfully resolves the identifier at the (n^{th}) selection.
resolve #*n* **to** #*m*	Resolves the identifier in the n^{th} selection to a declaration at the m^{th} selection.
complete(#*n*)? **to** *x*	Content completion proposals for the (n^{th}) selection include a name *x*.
refactor(#*n*)? *r* (**(***arg***)**)? *p*	Applies refactoring *r* with argument string *arg* according to pattern *p*.
build *builder*(**(***arg***)**)? *p*	Builds the fragment using builder *builder* with argument *arg* according to *p*.
run *runner*(**(***arg***)**)? *p*	Executes the fragment using runner *runner* with argument *arg* according to *p*.
e	Freeform expression *e*, a predicate specified in the language definition language, is satisfied.
Patterns in test conditions (*p*)	
	Empty pattern: same as **succeeds**.
succeeds	Operation (i.e., refactoring, builder, execution) is expected to be successful.
fails	Operation is expected to fail.
to *term*	The result should match a term pattern such as `PropAccess("a","b")`.
to *fragment*	The result should match a code fragment *fragment*.
to file *file*	The result should match the contents of file *file*.

Figure 8.5 Summary of the test specification syntax.

the threshold to language definition testing and to increase the productivity of language engineers when editing and running tests.

We propose a combination of four forms of tool support for language definition testing. For editing tests, we propose to aid language engineers by providing editor services for *1)* the generic test specification language, and *2)* editor services of the language under test in test fragments. For running tests, we propose a combination of *3)* live evaluation of test cases as they are edited, and *4)* a batch test runner for testing larger test suites. In the remainder of this section we show how these forms of tool support are realized in the Spoofax testing language and how they can be used. The underlying implementation aspects are discussed in Section 8.6.

8.4.1 *Editor Services for Test Specification*

Integrated development environments are a crucial factor in programmer productivity [Selby, 2007]. Modern IDEs incorporate many different kinds of *editor services*, assisting developers in code understanding and navigation, directing developers to inconsistent or incomplete areas of code, and even helping them with editing code by providing automatic indentation, bracket insertion, and content completion.

Most editor services provided in modern IDEs are *language specific*, and can be defined as part of the language definition. The challenge in providing effective IDE support for language definition testing is in providing language-specific support for both the testing language and for the embedded language under test.

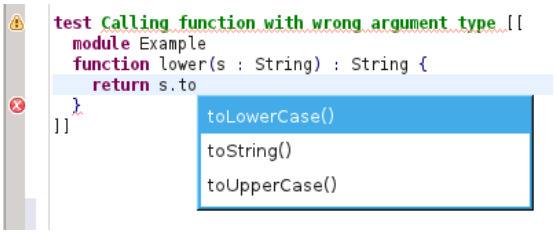
Editor services for the generic testing language Editor services for the generic testing host language are the meat and potatoes for making language engineers more productive with testing. Our implementation provides the full range of syntactic and semantic editor services for working with the testing language, ranging from syntax highlighting to error marking and content completion for all elements of Figure 8.5.

Editor services for language under test Rather than treat tested fragments as an opaque input string, we use editor services of the language under test to support them as first-class parts of a test specification. Our implementation provides services such as syntax highlighting, syntax error marking, semantic error marking, and content completion, as shown in the screenshots of Figure 8.6a and 8.6b. Note that error markers are only shown for failing test cases, not for negative test cases where errors are expected (Figure 8.6c).

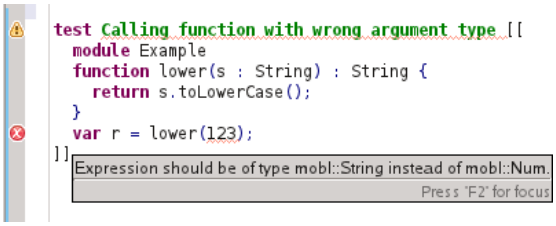
8.4.2 *Running Language Definition Tests*

Live evaluation of test cases Live evaluation of test cases as they are edited ensures that language engineers get the same rapid feedback and editing experience as they get with "throwaway" programs used to test language defi-

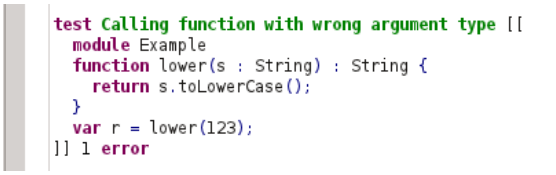
[4] Alternatively, `[[[...]]]` or `[[[[...]]]]` can be used.



(a) Content completion for the language under test.



(b) Online evaluation of tests and error markers.



(c) A passing test case specifying negative test condition.

Figure 8.6 IDE support for test specifications.

nitions. To achieve this effect, our implementation evaluates tests in the background and shows which tests fail through error and warning markers in the editor. With this feedback, developers can quickly determine the status of tests in a testing module. Since some operations may be long-running, we exclude test cases that depend on building or executing the test from background execution, and instead focus on tests of the syntax, static semantics, and transformations defined for a language.

Batch execution To support long-running test cases and larger test suites, we also provide a batch test runner as described in [Gamma and Beck, 1998]. Such a test runner is particularly important as a language project evolves and the number of tests grows substantially and tests are divided across multiple test modules. Figure 8.7 shows a screenshot of our graphical test runner. The test runner gives a quick overview of passing and failing tests in different modules and allows developers to navigate to tests in a language project. Tests can also be evaluated outside the IDE, for example as part of a continuous integration setup.

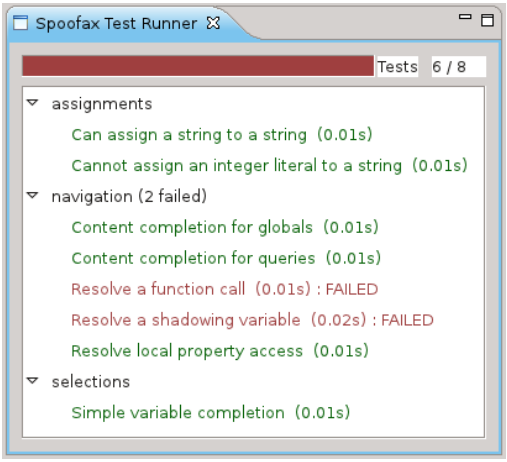


Figure 8.7 The batch test runner.

8.4.3 *Using Integrated Language Definition Testing*

Rather than designing a complete new language "on paper," before its implementation, it is good practice to incrementally introduce new features and abstractions through a process of evolutionary, iterative design [Fowler, 2005a; Visser, 2007]. The LPTL approach makes it possible to start a language design with examples that (eventually) form test cases.

Testing from the point of inception of a language requires that the tested language implementation artifact is in such a state that it can produce some form of output for a given input program. Language workbenches such as Spoofax can generate an executable – and thus testable – language plugin from only a (partial) syntax definition (as described in Chapter 4). Additional features, in the form of editor services and static and dynamic semantics, can then be iteratively and incrementally added. With an LPTL, each new feature can be tested at at any stage of the development process. This makes it possible to develop languages in a test-driven fashion, following the rhythm described in [Beck, 2003]:

1. Write a test case.
2. Watch it fail.
3. Implement the tested feature.
4. Watch all tests succeed.
5. Refactor when necessary and repeat.

Our approach facilitates this process for language engineering by providing a specialized language testing infrastructure that gives direct feedback at any stage in this development cycle.

```
Start ::= "module" QId Def*

Def ::= "entity" ID "{" EBD* "}"
      | Function
      | Stm

EBD ::= ID ":" Type
      | Function

Function ::= "function" ID
             "(" (FArg ("," FArg)*)? ")"
             ":" Type "{" Stm* "}"

Stm ::= "var" ID ":" Type "=" Exp ";"
      | "var" ID "=" Exp ";"
      | Exp "=" Exp ";"
      | Exp ";"
      | "{" Stm* "}"
      | "if" "(" Exp ")" Stm ("else" Stm)?
      | "foreach" "(" ID "in" Exp ")" Stm
      | "foreach" "(" ID ":" Type "in"
        Exp ")" Stm

Exp ::= STRING
      | NUMBER
      | "null"
      | "this"
      | Exp "." ID
      | ID "(" (NameExp ("," NameExp)*)? ")"

NameExp ::= ID "=" Exp | Exp

FArg ::= ID ":" Type

Type ::= ID | "Collection" "<" Type ">"

QId ::= ID | QId "::" ID
```

Figure 8.8 A subset of mobl's syntax, from [Hemel and Visser, 2011].

8.5 LANGUAGE DEFINITION TESTING BY EXAMPLE

In this section we show how different language aspects can be tested through examples using the mobl language.

Mobl Mobl is a statically typed language and compiles to a combination of HTML, Javascript, and CSS. Mobl integrates sub-languages for user interface design, data modeling and querying, scripting, and web services into a single language. In this chapter we focus on the data modeling language.

An excerpt of the syntax of mobl is shown in Figure 8.8. In mobl, most files starts with a `module` header, followed by a list of entity type definitions, functions, and possibly statements. An example of a mobl module that defines a single entity type is shown in Figure 8.9. Entities are persistent data types that are stored in a database and can be retrieved using mobl's querying API. We import the `tasks::datamodel` example module in tests throughout this section.

```
module tasks::datamodel

entity Task {
  name : String
  date : DateTime
}
```

Figure 8.9 A mobl definition of a `Task` data type.

```
language mobl
start symbol Stm

test Named parameters [[
  var e = Task(name="Buy milk");
]] parse succeeds

test 'true' is a reserved keyword [[
  var true = 1;
]] parse fails

test Test dangling else [[
  if (true)
    if (true) {}
    else {}
]] parse to
    IfNoElse(True, If(True, _, _))

test Nested property access [[
  v = a.b.c;
]] parse to
    Assign("v",
    FieldAccess(FieldAccess("a", "b"), "c"))
```

Figure 8.10 Syntactic tests.

8.5.1 *Syntax*

The syntax definition forms the heart of the definition of any textual language. It incorporates the concrete syntax (keywords etc.) and the abstract syntax (data structure for analysis and transformations) of a language, and is generally the first artifact developed with a new language definition. The syntax can be considered separately from the remainder of a language definition, and can be used to generate a parser and editor with basic syntactic services. It can also be tested separately from any semantic aspects of the language.

Syntax tests can be used to test newly added language constructs. They can include various non-trivial tests such as tests for operator precedence, reserved keywords, language embeddings, or complex lexical syntax such as the quotation construct of Figure 8.5.

We distinguish two forms of syntax tests. First, there are pure black-box tests, which test if a code fragment can be parsed yes or no. The first two examples of Figure 8.10 show positive and negative black-box tests. Next, we also support syntactic tests that use tree patterns to match against the abstract syntax produced by the parser for a given fragment. The third and fourth tests in the figure show examples of such tests.[5] These tests are not pure black-box

[5] We use prefix constructor terms to match against tree patterns, matching against the name of a tree node and its children. Wildcards are indicated with an underscore.

```
language mobl

setup [[
  module tasks
  import tasks::datamodel

  var todo = Task(name="Create task list");
]]

test Entity types have an all() built-in [[
  var all : Collection<Task> = Task.all();
]] succeeds

test Assigning a property to a Num [[
  var name : Num = todo.name;
]] 1 error /type/

test Local variable shadowing [[
  function f() {
    var a : A;
    {
      var a : A;
    }
  }
]] 1 error /already defined/
```

Figure 8.11 Tests for static semantic checks.

tests as they expose something about the implementation of the parser. They may not rely directly on the internals of the parser, but they still depend on the technology used. Many parser generators rely on restricted grammar classes [Kats et al., 2010], placing restrictions on the syntax definition, making it difficult to produce certain trees such as the left-recursive trees for field access in Figure 8.10. In Spoofax, these restrictions are not an issue since we use a generalized-LR parser.

8.5.2 *Static Semantic Checks*

Static semantic checks in languages play an important role in the reliability of programs written in that language. With DSLs such as mobl, these checks are often specific to the domain of the language, and not supported by the target platform. In mobl's case, a dynamic language is targeted that performs no static checks at all.

With tests we can have better confidence in the static checks defined for a language. Examples are shown in Figure 8.11. We use a setup block[6] in this figure to import the `tasks::datamodel` module, and to initialize a single `Task` for testing. The first test case is a positive test, checking that the built-in `all()` accessor returns a collection of tasks. The other tests are negative tests. For such test cases, it is generally wise to test for a specific error message. We use regular expressions to catch specific error messages that are expected, e.g. `/type/` to match messages with the substring `type`.

[6] Recall that the `[[...]]` placeholder notation is optional: by default, we assume that the placeholder is at the end. This default elegantly allows most mobl tests to avoid explicit placeholders.

```
language mobl

setup [[
  module navigation
  import tasks::datamodel

  var example = "Magellan";
]]

test Resolve a shadowing variable [[
  function getExample() : String {
    var [[example]] = "Columbus";
    return [[example]];
  }
]] resolve #2 to #1

test Resolve a function call [[
  function [[loop]](count : Num) {
    [[loop]](count + 1);
  }
]] resolve #2 to #1

test Content completion for globals [[
  var example2 = [[e]];
]] complete to "example"

test Content completion for queries [[
  var example2 = Task.[[a]];
]] complete to "all()"
```

Figure 8.12 Reference resolving and content completion tests.

8.5.3 *Navigation*

Modern IDEs provide editor services for navigation and code understanding, such as reference resolving and content completion. These services are a manifestation of the *name analysis* that is highly important for the dependability of a language implementation. Tests for reference resolving and content completion test not only the user experience in an editor, but also the underlying name analysis and any other analyses it may depend on.

Figure 8.12 shows examples of tests for reference resolving and content completion. Note how reference resolving tests can use multiple selected areas. Our first test case tests variable shadowing, while the second one tests reference resolving for function calls. For content completion we test completion for normal local variables, and for built-ins such as the `all()` accessor.

8.5.4 *Transformations and Refactorings*

Transformations can be used to create views and for compilation. To test transformations, we use the notion of a *builder*. Builders are transformations that can be triggered by the user, displaying a view or generating code (Section 4.4.4). Mobl implements several, for use by both end-programmers and meta-programmers. The first test case in Figure 8.13 shows an example of a test for the `desugar` builder, one of the builders used by the designers of mobl to inspect the desugared version of a module.

```
language mobl

setup [[
  module Example
  import tasks::datamodel
]]

test Desugaring adds a type to foreach [[
  foreach (t in Task.all()) {
    // ...
  }
]] build desugar to [[
  foreach (t : Task in Task.all()) {
    // ...
  }
]]

test Rename refactoring [[
  var x = 1;
  function x(x : Num) : Num {
    return [[x]];
  }
]] refactor rename("y") to [[
  var x = 1;
  function x(y : Num) : Num {
    return y;
  }
]]
```

Figure 8.13 Tests for transformations and refactorings.

Refactorings are transformations that rely on pre-conditions and post-conditions to perform behavior-preserving transformations [Fowler and Beck, 1999]. With tests, language engineers can gain more confidence about the transformation performed for a refactoring and its pre- and post-conditions. The second test case in Figure 8.13 is an example of a refactoring test. The example tests the `rename` refactoring with the input string `"y"` which determines the new name of the selected identifier. The test condition compares the output to the expectation, where behavior is only preserved if just the function parameter `x` is replaced and not the other `x`'s.

8.5.5 *Code Generation and Execution*

The ultimate goal of most language definitions is to generate or interpret code for execution. Sometimes, languages also generate artifacts for inspection by the user, such as a graphical view of all entities in a mobl application. Testing can be used to confirm that exactly the right output is generated for a particular input, but those tests are often rather fragile: one small change in a compiler can break a test case even if the program still compiles correctly. It is more practical to use an external oracle for those tests, such as a compiler or lint-type checker. Another strategy is to ensure that the program is executable and to simply run it: execution tests can indirectly serve as tests of generated code correctness. For execution tests we use the notion of a *runner*. Similar to a builder, runners are operations that execute code, through interpretation or by running a generated program.

```
language mobl

setup [[
  module runtimetests
  import tasks::datamodel

  foreach (n in range(0, 10)) {
    add(Task(name="Task "+n));
  }

  function getResult() : Num {
    return [[...]];
  }
]]

test Compile to HTML/JavaScript/CSS [[
  1 + 1
]] build generate-artifacts

test String API [[
  "string".indexOf("r")
]] run run-test("getResult") to "2"

test Counting query [[
  Task.all().count()
]] run run-test("getResult") to "10"
```

Figure 8.14 Code generation and execution tests.

Figure 8.14 shows tests for code generation and execution. We use a setup block to initialize the database by adding new `Task` instances that can be used in the tests. In the first test case we have a test that only triggers code generation, using mobl's `generate-artifacts` builder. In this case the builder is only required to succeed, we do not explicitly check its output. The other test cases use a runner `run-test`, which invokes the `getResult()` function and returns the result as a string for comparison.

8.5.6 *Testing for End-Programmers*

So far we have considered testing for meta-programmers. End-programmers that use a language are generally not interested in testing the syntax or static semantics of a language. They are, however, interested in the dynamic semantics; writing unit tests for programs written in the language. An LPTL can be used as a basis for maintaining and running such tests. End-programmers then get the same language-specific feedback and tooling for writing tests as meta-programmers, and can use the same testing language for testing multiple DSLs that may be employed in a project.

The LPTL as we designed it is aimed at meta-programmers, and provides a general platform for testing. For end programmers it can be specialized for one particular language (eliminating the `language` header) and for the purpose of execution tests (simplifying the `run` clause). Providing specialized instances of the test specification language is considered future work.

```
language Spoofax-Testing

test Testing a mobl test specification [[[
  module test-mobl
  language mobl
  test Testing mobl [[
    module erroneous
    // ...
  ]] 1 error
]]] succeeds
```

Figure 8.15 Testing the test specification language.

8.5.7 *Freeform Tests*

The test specification language is open-ended: if there are aspects of a language definition that need testing but are not covered by the fixed conditions in the table of Figure 8.5, *freeform* test expressions can be used. In the Spoofax testing language, we use the Stratego language [Bravenboer et al., 2008] to specify them, as it is also the language used to define semantic aspects of language definitions in Spoofax. Freeform expressions can directly interact with the language implementation to express white-box test cases. For example, they can test whether an internal function that retrieves all the ancestors in the inheritance chain of a class works, or they can test that `generate-artifacts` correctly writes a `.js` file to disk.

8.5.8 *Self Application*

An interesting capability of the testing language is that it can be applied to itself. In our implementation of the Spoofax testing language, it can be applied to any language designed in the language workbench, including instantiations of the testing language. Figure 8.15 shows an example. Note how we use the triple-bracket quotation form (i.e., `[[[ ... ]]]`) in this example, as the testing language itself uses the normal double brackets. For the outer test specification, any selections or setup placeholders should then also be specified using triple brackets. The inner test specification is free to use double brackets.

8.6 IMPLEMENTATION

In this section we describe our implementation of an LPTL and the infrastructure that makes its implementation possible. We implemented the Spoofax testing language as a language definition plugin for the Spoofax language workbench. Spoofax itself is, in turn, implemented as a collection of plugins for the extensible Eclipse IDE platform. Most Spoofax language definitions consist of a combination of a declarative SDF [Heering et al., 1989; Visser, 1997c] syntax definition and Stratego [Bravenboer et al., 2008] transformation rules for the semantic aspects of languages, but for this language we also wrote parts of the testing infrastructure in Java.

8.6.1 *Infrastructure*

The Spoofax language workbench provides an environment for developing and using language definitions. It provides a number of key features that are essential for the implementation of an LPTL.

A central language registry Spoofax is implemented as an extension of the IDE Meta-tooling Platform (IMP) [Charles et al., 2009] which provides the notions of languages and a language registry. The language registry is a component that maintains a list of all languages that exist in the environment. It also allows for runtime reflection over the services they provide and any meta-data that is available for each language, and can be used to instantiate editor services for them.

Dynamic loading of editor services Spoofax supports dynamic, headless loading of separate language and editor services of the language under test. This is required for instantiation of these services in the same program instance (Eclipse environment) but without opening an actual editor for them.

Functional interfaces for editor services Instantiated editor services have a functional interface. This decouples them from APIs that control an editor, and allows the LPTL to inspect editor service results and filter the list of syntactic and semantic error markers shown for negative test cases.

Support for a customized parsing stage Most Spoofax plugins use a generated parser from an SDF definition, but it is also possible to customize the parser used. This allows the LPTL to dynamically embed a language under test.[7]

These features are not trivially supported in language workbenches, but there are other workbenches that support a subset. For instance, where many language workbench implementations generate relatively opaque, autonomous Eclipse plugins, MPS [Voelter and Solomatov, 2010] is an example of a workbench with first-class languages and a language registry. Many workbenches support some level of dynamic loading of services, although their implementation may be tied to IDE interfaces that may make it hard to instantiate them in a headless fashion. Functional interfaces for editor services are rare, but could be implemented for workbenches that generate the service implementations. MontiCore [Krahn et al., 2008] is an example of a workbench that applies similar techniques with regard to combining host and embedded language parsers.

8.6.2 *Syntax and Parsing*

Language engineers can instantiate the testing language for any Spoofax language that is loaded in the Eclipse environment, either as an Eclipse plugin, or as a language project in source form. Once the developer specifies which

[7]Note that even though Spoofax supports generalized parsing and syntax embedding techniques, a different approach is required in this case as the embedding cannot be expressed as a context-free grammar, as we discuss in Section 8.6.2.

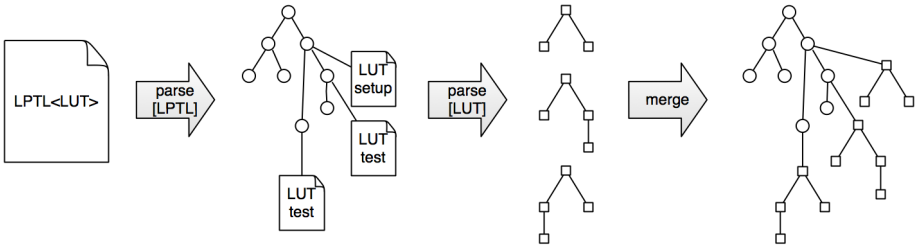


Figure 8.16 Parsing the LPTL with the embedded language under test (LUT).

language to test, the syntax of the testing language is instantly specialized by integrating the syntax of the language under test. This makes it possible to provide syntactic editor services such as syntax highlighting, and to parse the file to a single abstract syntax tree that is used in the implementation of semantic editor services and tests evaluation.

In previous work, we have shown how generalized parsers can be used to syntactically compose languages [Bravenboer and Visser, 2004]. Generalized parsers support the full class of context-free grammars, which is closed under embedding. That is, any two context-free grammars can be composed to form again a context-free grammar. This makes it possible to support modular syntax definitions and allow for language composition scenarios such as embedding and extension.

Unfortunately, the embedding as we have defined it for the LPTL is not context-free. First, because test fragments can only be parsed when their setup block (context) is taken into consideration. Second, because test fragments are allowed to contain syntax errors, such as spurious closing brackets. Even when considering syntax error recovery techniques, which use a local or global search space or token skipping techniques [Degano and Priami, 1995], the test fragments must be considered in isolation to ensure correct parsing of the test specification.

As we cannot compose the LPTL parser with that of the language under test at the level of the syntax definition, we have chosen for a more ad hoc approach, parsing the LPTL and the language under test in separate stages. Figure 8.16 illustrates these stages. First, we parse the LPTL using a skeletal test specification syntax, where every setup and test fragment is parsed as a lexical string. Second, we parse each setup and test fragment again using the parser of the language under test. For this, the setup fragment is instantiated for each test, and also parsed separately. As a third step, we merge the abstract syntax trees and token streams[8] of the skeletal LPTL and of the test cases. The merged tree and token stream are then used for editor services and to evaluate the tests in the module.

Our approach makes it possible to directly instantiate the language without generating a new parser for the instantiated LPTL. Our Java-based scannerless

[8]Note that Spoofax uses scannerless parsing, but still constructs a token stream *after* parsing for editor services, as described in Section 6.8.4.

generalized-LR (JSGLR) parser is relatively efficient, ensuring good runtime performance and allowing for interactive use of the Spoofax testing language. With support for error recovery techniques (described in Chapter 6), JSGLR also ensures that a valid abstract syntax tree is produced for providing editor services in case a test module is syntax incorrect (as seen in Figure 8.6a). There are still opportunities for performance optimizations; e.g. the three stages could be more tightly integrated and caching could be added for parsing the test fragments, but so far we have not found the need for this.

8.6.3 *Tool Support*

The language registry provided by Spoofax and IMP maintains a collection of all languages supported in the environment, and provides access to factory classes to instantiate language-specific editor services (e.g. a syntax highlighter, content completion service, or code generator). Using the language registry, and the dynamic editor service loading facilities of Spoofax, it is possible to access the parser, syntactic start symbols, and a list of editor services that can be instantiated for a language, given its name. We use the registry to instantiate these services for editor support in the language under test and for evaluating tests.

Editor service support in test fragments is provided by *delegation* to services of the language under test. An example is content completion support to help write test cases. The editor services for the testing language simply detect that they are invoked inside a test fragment, and then delegate the work to a service of the language under test. The only special cases are the syntax error marking and semantic error marking service. These produce lists of errors that must be filtered according to the test expectations (e.g., if an error is *expected*, the IDE should not add a red marker for it).

The parser and editor services are transparently loaded on demand once they are used for a test case. As a result, the editor for a test module is instantly specialized by simply specifying the name of the language under test. Further configuration is not required. Test cases are also automatically re-evaluated in the editor if a language definition is changed.

Test evaluation Tests are evaluated by instantiating the appropriate editor services for the language under test and applying them to the abstract syntax tree that corresponds to the test input fragment. For example, consider a reference resolving test such as that of Figure 8.12. To evaluate such a test, the language registry is used to instantiate services for semantic analysis and reference resolving. Like all editor services, the reference resolver has a functional interface, which essentially gets an analyzed program and an abstract syntax tree node of a reference as its input, and returns the declaration abstract syntax tree node. To test it, we give it the analyzed program, obtained from the analysis service, and the tree node that corresponds to the reference selected in the test fragment. The result is then compared to the expected result of the test case.

Execution tests often depend on some external executable that runs outside the IDE and that may even be deployed on another machine. Our implementation is not specific for a particular runtime system or compiler back end. Instead, language engineers can define a custom "runner" function that controls how to execute a program in the language. For a language such as mobl, a JavaScript engine such as Rhino or a browser emulator such as WebDriver can be used. At the time of writing, we have not yet completed that binding for mobl yet.

Test evaluation performance Editor services in Spoofax are cached with instantiation, and run in a background thread, ensuring low overhead and near-instant responsiveness for live test evaluation. Most editor services are very fast, but long-running tests such as builds or runs are better executed in a non-interactive fashion. We only run those through the batch test runner of Section 8.4.2 and display information markers in the editor if the test was changed after it last ran.

8.7 DISCUSSION AND RELATED WORK

Related work on testing of language implementations can be divided into a number of categories: testing with general-purpose tools, testing with homogeneous and heterogeneous language embeddings, and test case generation.

Testing with general-purpose tools Considerable experience exists in the use of general-purpose testing tools and scripts for tests of language definitions [Frens and Meneely, 2006; Goodenough, 1980; Gómez et al., 2001; Stodte, 2001; Malloy et al., 2002]. Sometimes, they take the form of a simple shell script that builds all files in a directory. In other cases they use JUnit or a related xUnit-family [Hamill, 2004] testing framework.

The use of these tools introduces a number of challenges when applied to language definitions. First, a major issue is that a significant investment in language-specific testing infrastructure must be made to support tests for different aspects of languages, ranging from syntax to semantics and editor services. We support a generic, declarative language for testing these aspects. A second issue is that to make sure tests are considered in isolation, each test case is generally put in a separate file. Using separate files for test cases introduces boilerplate code such as import headers. It also makes it harder to organize tests, requiring conventions for file and directory names. Using test files specified purely in the tested language also separates test conditions and expectations from the test program. With an additional investment in effort, some of these issues can be solved, but only on a per-language basis. We provide a language-generic solution.

Another limitation of these general testing tools is that they do not provide specialized IDE support for writing tests. Standard IDEs are only effective for writing valid programs and report spurious errors for negative test cases where errors are expected. Batch test runners such as JUnit also do not have

the capability to directly direct users to the failing line of code in a test input in case a test fails.

Testing with homogeneous embeddings Language embedding is a language composition technique where separate languages are integrated. An example is the embedding of a database querying language into a general-purpose host language. These embeddings can be *heterogeneous*, where an embedded language can be developed in a different language than the host language, or *homogeneous*, where the host language is used to define the embedded language [Hudak, 1998; Tratt, 2008]. Homogeneous embeddings of DSLs are sometimes also called internal DSLs [Fowler, 2011]. The embedding technique applied in this chapter is a heterogeneous embedding.

Homogeneously embedded languages must always target the same host language. This can be a weakness when execution on a different platform is desired (e.g., JavaScript in the case of mobile development with mobl [Hemel and Visser, 2011]). It can also be a strength in terms of tool support. As embedded languages all target the same platform, they can be tested in the same way. General-purpose testing frameworks can then be applied more effectively, since they can directly use the embedded language. In MPS [Voelter and Solomatov, 2010], which primarily targets Java, tests based on JUnit can be evaluated as they are typed, much like with our testing language. A restriction of the approach used in MPS is that it can only be used test the dynamic semantics of a language, and only allow for *positive* test cases. With MPS, a projectional editor is used that even restricts the tests so they can only follow the existing syntax of the language, ruling out sketches of new syntax and test-driven development of new syntactic constructs. Our test specification language is more flexible, supporting a much wider spectrum of test conditions (Figure 8.5), including negative test cases for syntax or static semantics and refactoring tests.

Testing with heterogeneous embeddings In previous work, we developed *parse-unit*, a grammar testing tool developed as part of the Stratego/XT program transformation system and tool [Bravenboer et al., 2008]. This tool formed a precursor to the present work. Parse-unit would embed quoted program fragments in a test module, much like in the Spoofax testing language, but only supported syntactic test cases. There was also no IDE support for parse-unit, and all quoted program fragments were treated as strings rather than forming a first-class part of the language.

Other testing tools that support embedding have similar limitations as parse-unit, supporting only syntactic tests, and lacking IDE support. A notable example is gUnit [gUnit, 2007], a testing language for ANTLR grammars. Facilities as those provided by gUnit have been lacking in current language workbenches and other interactive tools for building and debugging parsers such as ANTLRWorks [Bovet and Parr, 2008].

Test case generation techniques There is a long history of research on test case generation techniques for language implementations. An overview is given in surveys by Boujarwah and Saleh [1997], and Kossatchev and Posyp-

kin [2005]. These techniques use grammars to generate test programs. To control the theoretically infinite set of programs that can be generated using most grammars, they use annotations in grammars, external control mechanisms, or even imperative generators [Daniel et al., 2007], to constrain this set. In some cases, sophisticated, hand-tailored program generators are used, such as Csmith [Yang et al., 2011], a successful, 40,000-line C++ generator program for randomly generating C programs.

The set of test programs generated with these approaches can be used to stress-test compiler implementations. For example, they can be used to compare a compiler to its reference implementation, or to check for validity of generated code for the subset of programs that typecheck. As such, they provide an excellent complementary approach to our test specifications, possibly catching corner cases that a language engineer did not think of. However, as they only test complete compilation chains and rely on a test oracle such as a reference compiler, they are less effective for testing languages while they are still under development. In contrast, our approach can be used from the point of inception of a language and even in the design process. By applying test-driven development, test specifications can be used to guide the development process. Our test specifications also provide a more varied array of tests by providing an extensive, open-ended set of test condition specification constructs for observable behavior of language implementations.

Unit tests for domain-specific languages As a side-effect of providing a language for testing language implementations, our test specification language can also be used to test programs written in that language (see Section 8.5.6). This makes it particularly useful in the domain of testing DSL programs, where testing tools and frameworks are scarce. Traditionally, language engineers would have to build such tools and frameworks by hand, but recently Wu et al. [2008] provided a reusable framework for language testing. They require language engineers to extend their language with scripting constructs that generate JUnit test cases. The combined scripting and DSL language can then be used to specify tests. Their framework ensures that the mapping between the DSL test line numbers and the generated JUnit tests is maintained for reporting failing tests. They also provide a graphical batch test runner. While our approach does not provide the same flexibility and requires tests to be specified with a quotation marks and `language` and `run` clauses, it is interesting to note how our approach relieves language engineers from much of the heavy lifting required for implementing a DSL testing solution. We only require language engineers to specify a binding to an execution engine (Section 8.6.3), and we provide a generic test specification host language that is combined with the DSL to test.

8.8 CONCLUDING REMARKS

In this chapter we proposed an approach to language definition testing by introducing the notion of a language-parametric testing language. The LPTL provides a zero-threshold, domain-specific testing infrastructure based on a

declarative test specification language and extensive tool support for writing and executing tests. Our implementation in the form of the Spoofax testing language shows the practical feasibility of the approach.

Tests inspire confidence in language implementations, and can be used to guide an agile, test-driven language development process. Unfortunately, in current software language engineering practice, tests are still too often an afterthought. Especially DSLs often remain untested, as they are developed in a short timespan with limited resources. We believe that declarative language test suites should become standard components of language definitions, just as BNF-style grammars are. Supported by an LPTL, tests are concise, implementation-independent, and require little to no effort to setup.

Future work In this chapter we emphasized testing of observable behavior of languages, such as reported errors and name analysis as manifested by reference resolving in an IDE. Other analyses such as the type and flow analyses of Chapter 5 are not manifested that way, but it can be useful to write test cases for them. Right now, these aspects are either indirectly tested, or tested using the generic "builders" interface for custom transformations. Direct support for testing such language definition aspects could be a worthwhile addition. Alternatively, rather than seeking to support all possible compiler and IDE aspects in a testing language, perhaps a better test abstraction mechanism is needed to specify multiple tests that interact with a language definition in the same way. Similarly, an abstraction mechanism for setup blocks could be introduced for improved modularization of test suites, e.g. by allowing of setup blocks to be put in libraries, to support multiple arguments, and to support composition.

For the interaction design of the LPTL, our work on interactive disambiguation of Chapter 7 could be applied for handling ambiguities of quoted test programs. Test understandability can also be improved using further visual aids, for example to emphasize differences between test inputs and outputs for refactoring tests.

The declarative basis provided by the LPTL can be used to integrate generally applicable supportive techniques for testing, such as test case prioritization, coverage metrics, coverage visualization, mutation testing, and mutation-based analyses for untested code. In particular, the integration of such techniques specialized for the domain of parser and compiler testing [Boujarwah and Saleh, 1997; Kossatchev and Posypkin, 2005; Lämmel, 2001] is an important area of future work.

Acknowledgments This research was supported by NWO project 612.063.512, *TFA: Transformations for Abstractions* and the NIRICT LaQuSo Build Farm project. We would like to thank Martin Bravenboer for his work on the parse-unit project, which provided a basis for the Spoofax testing language; Danny Groenewegen for his inspiring tests scripts and test collection for WebDSL; Zef Hemel for his work on mobl that was used in examples in this chapter; and Maartje de Jonge, Sander Vermolen, and the anonymous referees for suggestions for this chapter.

Conclusion

9

In this dissertation we have studied techniques, methods, and tools for domain-specific language engineering. Our goal has been to introduce abstractions for high-level, declarative language definitions, and to make it possible to support language engineers in working with those abstractions in an integrated, interactive language engineering environment. To this end we have studied topics in three main research themes:

- Applying **domain-specific languages** for declarative specification of languages and IDEs;
- supporting **declarative syntax definition** for generating a parser-based, interactive development environment;
- and providing **interactive meta-tooling support**, exploring the application of modern IDE technology to DSL engineering.

In the remainder of this chapter we give a summary of the key contributions of this dissertation in relation to these themes, describe how we evaluated our approach, revisit the research questions of the introductory chapter, and give recommendations for future work.

9.1 SUMMARY OF CONTRIBUTIONS

Each of the chapters in this thesis lists distinct contributions. We summarize the core contributions below:

Domain-specific languages for declarative specification of languages and IDEs

- An architecture for black-box extensibility of language definitions (Chapter 2).
- Techniques for using aspect-oriented programming to facilitate language portability (Chapter 3).
- Abstractions for the combined specification of languages and IDEs (Chapter 4).
- A combination of attribute grammars and strategic programming to abstract over typical compiler construction idioms (Chapter 5).
- A language-parametric specification language for testing DSLs (Chapter 8).

Declarative syntax definition

- Syntax-level abstractions for editor services implemented using scannerless parsing and error recovery (Chapter 4 and 6).
- Syntax error recovery techniques for scannerless, generalized parsers (Chapter 6).
- Techniques for deriving functionality from declarative syntax, in particular editor services, recovery rules, and disambiguation suggestions (Chapters 4, 6, and 7).

Interactive meta-tooling support

- A comprehensive language workbench implementation based on DSLs for declarative specification of languages and IDEs (Chapter 4).
- An approach for diagnosing ambiguities in concrete object syntax quotations and interactively resolving them (Chapter 7).
- Interactive facilities for testing DSL definitions by embedding language syntax, semantics, and editor services (Chapter 8).

9.2 EVALUATION

For the approaches introduced in the core chapters of this dissertation, we described the rationale, design, implementation, and their applications. In Chapter 2, we evaluated our approach using the Java language as a realistic case study, comparing the implementation of well-known language extensions such as including traits, partial classes, and iterator generators against traditional open compiler approaches. We performed an extensive case study of porting the Stratego language in Chapter 3 and showed limited runtime overhead in the implementation. In Chapter 4, we showed how typical language and editor services can be implemented through a study of the NWL language, and described our significant experience with languages and courses that use Spoofax. We showed the effectiveness of the decorated attribute grammars paradigm of Chapter 5 with an extensive selection of typical compiler construction idioms from the attribute grammar literature, describing how to implement them based on a library of decorators. We also described a case study of a grammar analysis and transformation tool that applies these idioms. In Chapter 6, we performed extensive studies of performance overhead and recovery quality for different configurations of error recovery techniques. In Chapter 7, we described how to diagnose different classes of ambiguities, following [Vinju, 2005], and studied quality and performance of interactive disambiguation for existing meta programs using different object languages and scenarios. Finally, in Chapter 8 we described how to write tests for all semantic editor services and other observable behavior of language implementations of Chapter 4, applying the approach to mobl [Hemel and Visser, 2011] and the testing language itself.

9.3 RESEARCH QUESTIONS REVISITED

Research Question 1

> *How can modular language plugin definitions abstract over the implementation architecture of a particular programming language? Can such plugins use lower-level features provided by the target platform?*

Source-to-source program translators implement programming language extensions by translating the extended language to to the base language. They use the base language as a *linguistic abstraction* over the base language implementation. The base language implementation, in turn, provides an abstraction over some target platform, e.g. the Java Virtual Machine. Through abstraction, a target language may hide parts of the platform, thereby providing only a limited interface for the implementation of language extensions.

In Chapter 2 we showed how a compiler can be designed based on the principle of *compilation by normalization*, compiling a rich base language to a small core language. In this design, all primitives of the language are exposed in the core language but also in the rich base language. Consequently, the language provides a much richer interface for the implementation of language extensions. Using the Java language as a realistic case study, we showed how the design can be used for the implementation of a wide variety of language extensions, including traits, partial classes, and iterator generators.

In related work [Hemel et al., 2009], we described a variation on the compilation by normalization approach, extending the (Java) target with features such as partial classes and methods with the purpose to provide a better programming abstraction for source-to-source translators. Together with compilation by normalization, these techniques provide a rich linguistic programming interface for source-to-source translators that exposes platform primitives together with base language-level abstractions and additional abstractions to simplify common translation idioms.

Research Question 2

> *How can DSLs be efficiently ported to another platform, taking into consideration their reliance on platform-specific operations and characteristics? Is it possible to do so without changing existing DSL programs and libraries written in the language?*

Domain-specific languages provide abstractions for a certain domain of computation and allow developers to focus on the essential complexity of problems in that domain rather than the accidental complexity of its implementation. Consequently, they have the potential to abstract over a particular implementation platform and can potentially support multiple target platforms [Herndon and Berzins, 1988]. Targeting multiple platforms can be realized by creating multiple back ends for the DSL that generate appropriate code for each platform. In Chapter 3 we showed that aspect weaving can be used to address portability issues of existing programs and libraries written

in the DSL. In particular, we identified four classes of aspects that address portability concerns by invasively introducing glue code, code that facilitates migration, code that facilitates platform integration, and code for optimization. We described our experience in applying these techniques to retarget the Stratego language to Java platform, which formed one of the cornerstones of the implementation of the Spoofax language workbench.

Research Question 3

> *How can editor service specifications be integrated into syntactic and semantic specifications of DSLs, balancing reuse and separation of concerns? How can language analysis components be structured to expose an interface for use by semantic editor services?*

Modern editor services depend on a combination of syntactic and semantic properties of languages. In Chapter 4 we studied the dependencies between language syntax, semantics, and editor services, and introduced abstractions to declaratively describe editor services in terms of those dependencies.

A key abstraction for both syntactic and semantic editor services is abstraction over language syntax, requiring separation of concerns between syntax and editor service specification. We showed how the two can be separately specified and linked using *symbolic integration* to reuse the syntax definition, allowing services such as syntax highlighting and the outline view to be described in terms of syntactic categories and constructor names specified in the syntax definition. The abstraction can be realized by language-independent techniques for integrating editor services with the parser and error recovery strategy (Chapter 6).

Semantic editor services can be integrated with the static semantics of a language by decomposing the name analysis and the type analysis, allowing them to be reused separately. In essence, this decomposition can be realized by using the name analysis to resolve declarations, rather than directly resolve types. Based on a separate, reusable name analysis specification, semantic editor services such as reference resolving and content completion can be specified without having to re-implement the analysis or make invasive changes to it.

Using origin tracking [van Deursen et al., 1993], language analyses and transformations can abstract over maintaining position information. in pure rule-based transformation systems, Chapter 4 described how to implement a form of origin tracking for strategic term rewriting systems that follows on our work on source tracing in Chapter 2. While the technique has previously been applied for relaying errors and debugging transformations, we showed that it can also be applied for separation of concerns between analyses and transformations and editor services that heavily depend on accurate position information to relate semantic information to the editor.

Research Question 4

> *Is it possible to generalize over common attribute grammar abstraction mechanisms? What primitives are needed for this generalization? Given these primitives, is it possible to introduce new abstractions for common analyses of DSLs?*

Attribute grammar systems use attribute evaluators to determine an evaluation strategy for attribute equations, i.e. a traversal order over a tree that calculates the values of attributes. Dynamic attribute evaluators determine this order dynamically by evaluating attributes as their value is requested, traversing to adjacent nodes on demand. In Chapter 5 we identified primitives that can express basic traversal operations as those used in dynamic attribute evaluators, traversing *upward* to adjacent parent nodes, or *downward* to child nodes. By applying *strategic programming* [Visser et al., 1998; Lämmel et al., 2003], it is possible to abstract over these traversal primitives and express more complex traversals such as top-down or innermost traversals. By providing primitives for strategic tree traversal, programmatic abstractions can be introduced for common attribute propagation patterns such as those provided by attribute grammar extensions. In addition to strategic traversal operators, we provide primitives that *reflect over attribute equations* in order to express more sophisticated abstractions.

Together, these primitives are at the heart of *decorators*, which form abstractions over attribute equations and attribute propagation patterns. We developed Aster, an attribute grammar system based on decorators, and used it to implement a library of decorators to express common abstractions such as copy rules, collection attributes, and circular attribute. In Aster, attributes and decorators are *first-class* and can be used to define new attributes and decorators. By combining layers of decorator abstractions, we showed how new abstractions can be introduced for name analysis, type analysis, error messages, and data-flow analysis of DSLs.

Research Question 5

> *What techniques are needed to efficiently diagnose and recover from syntax errors with scannerless, generalized parsers? Is it possible to support error recovery without breaking the abstraction of pure and declarative syntax definition?*

A grammar is a finite set of production rules that describe which strings are part of a language. For strings that are part of a language, a parser for a grammar can derive a structured representation. Strings that are not are considered syntactically incorrect.

In Chapter 6 we showed how grammars can be enriched with additional *recovery production rules* to consider syntactically incorrect strings as valid parts of a language. These recovery rules can take the form of rules that can process strings of arbitrary characters (*water recovery rules*) and rules that can parse an empty string in place of an input symbol at the left-hand side of other productions (*insertion recovery rules*). In combination with normal parse production rules, these recovery rules form a *permissive grammar* that can derive a structured representation even for a syntactically incorrect input.

Pure and declarative syntax definitions describe language syntax independently from the semantics of a grammar and independent from imperative actions that determine the correctness or structure of an input string [Kats et al., 2010]. This makes them amenable for automated analysis and processing. We used the Aster system of Chapter 5 to create a tool to analyze SDF grammars and to *automatically derive* permissive grammars, ensuring that language engineers do not need to specify recovery strategies by hand.

To parse permissive grammars we used generalized parsing to our advantage, which can derive both the normal and recovery interpretation for strings. Using a combination of backtracking and layout-based region selection we showed that permissive grammars can be efficiently parsed and in many cases provide good or excellent recovery results following the quality criteria of Pennello and DeRemer [1978].

Research Question 6

> *Can ambiguities in concrete syntax quotations be automatically diagnosed in order to determine the possible syntactic disambiguations? Can an IDE for meta-programming provide unobtrusive, interactive feedback based on such a diagnosis?*

Generalized parsing makes it possible to embed arbitrary object languages in a meta language [Visser, 2002]. They gracefully cope with ambiguities by produce a parse forest with all possible interpretations for an ambiguous input. This parse forest can be used as the basis for a diagnosis of the ambiguity and the possible ways that it can be resolved. In Chapter 7 we described an algorithm that uses a declarative specification of a language composition as a basis to determine the possible syntactic interpretations and the corresponding explicit quotations that can resolve the ambiguity.

Modern IDEs can provide feedback to meta programmers by parsing meta programs in the background and marking ambiguities as errors. By providing *quick fix* proposals that can be applied at the discretion of the meta programmer, they can unobtrusively apply the possible resolutions that address the ambiguity.

Research Question 7

> *Is it possible to define a general abstraction for systematic testing of DSL definitions? How can IDEs facilitate the development of DSL tests?*

In Chapter 8 we defined a generic domain-specific meta-language for the specification of test cases of DSL implementations. Essential elements of such a language are abstractions to test language syntax (i.e., the parser), static semantics of languages (i.e., the compiler or semantic checker), dynamic semantics of languages (i.e., the chain of compilation and execution or interpretation), and editor services. Common to all these facilities is the notion of a test input written in the DSL being tested. To specify such test inputs, we showed how the concrete syntax of the DSL can be embedded into a test specification language, forming a *language-parametric testing language*. By embedding the

syntax, semantics, and editor services of the DSL being tested into the testing language, IDE support can help in writing, maintaining, and understanding test cases.

9.4 RECOMMENDATIONS FOR FUTURE WORK

This dissertation presented research on techniques, methods, and tool support for domain-specific language engineering, centered on the Spoofax language workbench. The introduction of better abstractions and tools is an open-ended problem and a topic of ongoing research. In the chapters of this thesis we raised many issues to be investigated further, including composability of language plugins, integration of other meta-languages into Spoofax, and test-driven language development. To conclude we revisit them below.

Composability of language plugins An important area of future work is in providing further support for first-class modular language plugins. Based on the modular SDF syntax formalism, SGLR parsing, and modular Stratego and editor service definitions, it is possible to decompose languages into separate, reusable components. As an example, WebDSL [Groenewegen et al., 2010] embeds the HQL language for queries. Still, to fully generalize these results and to minimize the effort required for creating new compositions, open challenges exist related to composition of syntax, semantics, and editor services. By minimizing the effort and risk of composition it can even become possible to realize user-level composition, where users of a language could just pick and match the language features they need, and that the environment would compose them.

For composable syntax, generalized parsers provide a general, language-independent approach. Generalized parsers support the full class of context-free grammars, which is closed under composition, unlike subsets of the context-free grammars such as LL or LR. A limitation of the approach is that it is not possible to statically guarantee that there are no ambiguities in the syntax composition. This is particularly apparent when two independent language extensions are composed that have the same surface syntax, e.g. a new $*$ operator that acts as a wildcard and another $*$ operator that acts as a regular expression–style repetition operator. A composition of these extensions is *inherently ambiguous* and cannot be statically avoided. Still, projectional editors such as MPS [Voelter and Solomatov, 2010] can avoid even these kinds of ambiguities by requiring users to select the desired extension from a menu or using keyboard shortcuts, which is subsequently stored in a structured fashion and not as text. A direction for textual editors may be to follow the approach of Chapter 7 and provide quick fixes for disambiguation. For normal (non-meta) programming, this requires a refinement of the algorithm to detect and select disambiguations, and a general representation of disambiguations, stored either in text or as meta-data.

For composable semantics, modular definition of language extensions provide a high degree of compositionality that depends on the type of analysis and transformation performed (global or local). For global transformations,

an ordering of application must be determined, which requires implementation-level knowledge. There is no obvious solution for this issue, but one direction may be to further increase the expressiveness of local transformations, as we have aimed for in Chapter 2 by mixing high-level language code and low-level core language code.

For composable editor services, challenges to composition relate to those of syntactic and semantic composition. Composition of semantic editor services such as view and refactorings is particularly non-trivial, since they depend on both forms of composition. Refactorings are particularly challenging with this in mind, since they must guarantee preservation of behavior [Fowler and Beck, 1999].

Integration of meta-languages In Chapter 4 we showed that the Stratego language can be used for concise specifications of analysis, transformations, and code generation. Still, many other meta-programming languages exist that each have their own merits and uses. We believe that Spoofax has the potential to become a common, open platform for hosting multiple meta-programming languages. Spoofax defines a lightweight, technology-agnostic interface between editor services and semantic analyses, making it a suitable testbed for experimentation with other meta-languages.

In Chapter 5 we proposed to pursue full integration of the Aster language into Spoofax. Aster is currently distributed as part of Spoofax, and can be used to define analyses for Spoofax language plugins, but additional experience is needed to identify patterns for specifying editor services and complete languages with Aster. We would particularly like to explore decorators that encapsulate logic for typical editor service components, incremental compilation concerns, and related aspects.

Test-driven language development Test-driven and example-based development of languages is a very promising approach for systematically constructing new DSLs. In our work on testing we have emphasized testing of observable behavior of languages, such as reported errors and name analysis as manifested by reference resolving in an IDE. Other analyses such as the type and flow analyses of Chapter 5 are not manifested that way, but it can be useful to write test cases for them. There are essentially two possible directions for future work in that area. The first is to provide further abstractions for white-box tests, to efficiently test the internals of a language definitions. The second is to expose these semantic aspects of a language in such a way that they are amenable to black box tests, by designing fixed interfaces and investigating idioms to decompose these semantic aspects in a way similar to how the name analysis is exposed in Chapter 4.

Bibliography

Adams, B., De Meuter, W., Tromp, H., and Hassan, A. E. (2009). Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD 2009)*, pages 243–254, New York, NY, USA. ACM. (Cited on page 67.)

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley. (Cited on pages 21, 38, 50, and 122.)

Arnoldus, J., Bijpost, J., and van den Brand, M. (2007). Repleo: a syntax-safe template engine. In *Proceedings of the 6th international conference on Generative programming and component engineering (GPCE 2007)*, pages 25–32, New York, NY, USA. ACM. (Cited on pages 180 and 196.)

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). abc: an extensible AspectJ compiler. In *Aspect-oriented software development (AOSD 2005)*, pages 87–98, New York, NY, USA. ACM. (Cited on page 44.)

Avgustinov, P., Ekman, T., and Tibble, J. (2008). Modularity first: a case for mixing AOP and attribute grammars. In *Proceedings of the 7th international conference on Aspect-oriented software development (AOSD 2008)*, pages 25–35, New York, NY, USA. ACM. (Cited on page 67.)

Baars, A., Swierstra, D., and Löh, A. (2003). UU AG system user manual. *Department of Computer Science, Utrecht University, September*. (Cited on page 109.)

Bachrach, J. and Playford, K. (2001). The Java syntactic extender (JSE). In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA 2001)*, volume 36 of *SIGPLAN Notices*, pages 31–42, New York, NY, USA. ACM. (Cited on page 43.)

Baker, H. G. (1995). CONS should not CONS its arguments, part II: Cheney on the M.T.A. *SIGPLAN Notices*, 30(9):17–20. (Cited on page 52.)

Batory, D., Lofaso, B., and Smaragdakis, Y. (1998). JTS: Tools for implementing domain-specific languages. In *Proceedings of the Fifth International Conference on Conference on Software Reuse (ISRE 1998)*, pages 143 –153. IEEE. (Cited on pages 180 and 183.)

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional. (Cited on pages 14, 15, 201, 202, and 212.)

Beizer, B. (2002). *Software testing techniques*. Dreamtech Press. (Cited on page 202.)

Bentley, J. (1986). Programming pearls: little languages. *Communications of the ACM*, 29:711–721. (Cited on page 1.)

Bird, R. S. (1984). Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250. (Cited on page 112.)

Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., and Pascual, V. (1989). Centaur: the system. *SIGPLAN Notices*, 24(2):14–24. (Cited on pages 6 and 72.)

Boujarwah, A. S. and Saleh, K. (1997). Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625. (Cited on pages 15, 202, 224, and 226.)

Bovet, J. and Parr, T. (2008). ANTLRWorks: an ANTLR grammar development environment. *Software–Practice & Experience*, 38:1305–1332. (Cited on page 224.)

Boyland, J. (1996). *Descriptional Composition of Compiler Components*. PhD thesis, EECS Department, University of California, Berkeley. (Cited on pages 12, 16, 110, 121, 122, 123, and 125.)

Boyland, J. and Graham, S. L. (1994). Composing tree attributions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1994)*, pages 375–388, New York, NY, USA. ACM. (Cited on page 111.)

Boyland, J. T. (2005). Remote attribute grammars. *Journal of the ACM*, 52(4):627–687. (Cited on pages 12, 16, 109, 110, and 121.)

Brabrand, C., Möller, A., and Schwartzbach, M. I. (2002). The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114. (Cited on page 180.)

Bracha, G. (2008). Room 101: Monkey patching. `http://gbracha.blogspot.com/2008/03/monkey-patching.html`. (Cited on page 69.)

van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., and Visser, J. (2001). The Asf+Sdf Meta-environment: A component-based language development environment. In Wilhelm, R., editor, *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer. (Cited on pages 7, 74, 103, and 175.)

van den Brand, M. G. J., de Jong, H. A., Klint, P., and Olivier, P. A. (2000). Efficient annotated terms. *Software – Practice & Experience*, 30(3):259–291. (Cited on pages 53, 58, 100, 101, 127, 137, and 166.)

van den Brand, M. G. J., Heering, J., Klint, P., and Olivier, P. A. (2002). Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368. (Cited on pages 5, 72, 74, 134, and 180.)

van den Brand, M. G. J., Scheerder, J., Vinju, J., and Visser, E. (2002). Disambiguation filters for scannerless generalized LR parsers. In Horspool, N., editor, *Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France. Springer-Verlag. (Cited on pages 33, 42, 135, and 147.)

Bravenboer, M., Dolstra, E., and Visser, E. (2010). Preventing injection attacks with syntax embeddings. *Science of Computer Programming*, 75:473–495. (Cited on pages 20, 135, 136, 164, and 196.)

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70. (Cited on pages 5, 9, 33, 40, 49, 53, 61, 65, 72, 75, 79, 86, 111, 116, 126, 134, 180, 182, 219, and 224.)

Bravenboer, M., Tanter, E., and Visser, E. (2006a). Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-LR parsing. In Cook, W. R., editor, *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, volume 41 of *SIGPLAN Notices*, pages 209–228, Portland, Oregon, USA. ACM. (Cited on pages 135, 138, and 145.)

Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. (2006b). Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178. (Cited on pages 87, 93, and 103.)

Bravenboer, M., Vermaas, R., Vinju, J., and Visser, E. (2005). Generalized type-based disambiguation of meta programs with concrete object syntax. In Glück, R. and Lowry, M., editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 157–172. Springer Berlin / Heidelberg. (Cited on pages 180, 183, 185, 193, and 197.)

Bravenboer, M. and Visser, E. (2004). Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In Vlissides, J. M. and Schmidt, D. C., editors, *Proceedings of the 19th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, volume 39 of *SIGPLAN Notices*, pages 365–383. ACM. (Cited on pages 9, 10, 30, 31, 32, 75, 107, 132, 135, 138, 180, 183, 185, 186, 197, and 221.)

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T. J. (2004). *Eclipse Modeling Framework*. Addison-Wesley. (Cited on page 74.)

Burke, M. G. and Fisher, G. A. (1987). A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9(2):164–197. (Cited on page 174.)

Chamberlin, D. D. and Boyce, R. F. (1974). SEQUEL: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD)*

workshop on data description, access and control (SIGFIDET 1974), pages 249–264, New York, NY, USA. ACM. (Cited on page 1.)

Charles, P. (1991). *A practical method for constructing efficient LALR(k) parsers with automatic error recovery*. PhD thesis, New York University. (Cited on pages 153 and 174.)

Charles, P., Fuhrer, R. M., Sutton, Jr., S. M., Duesterwald, E., and Vinju, J. (2009). Accelerating the creation of customized, language-specific IDEs in Eclipse. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2009)*, volume 44 of *SIGPLAN Notices*, pages 191–206. ACM. (Cited on pages 6, 72, 95, 98, and 220.)

Charles, P., Fuhrer, R. M., and Sutton, Jr., S. M. (2007). IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE 2007)*, pages 485–488. ACM. (Cited on pages 6, 72, 98, and 174.)

Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. (2000). MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, pages 130–145, New York, NY, USA. ACM. (Cited on page 24.)

Clifton, C., Millstein, T., Leavens, G. T., and Chambers, C. (2006). MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575. (Cited on pages 24 and 69.)

Cook, S., Jones, G., Kent, S., and Wills, A. C. (2007). *Domain-Specific Development with Visual Studio DSL Tools*. Addison Wesley. (Cited on page 73.)

Cordy, J. R., Halpern-Hamu, C. D., and Promislow, E. (1991). TXL: a rapid prototyping system for programming language dialects. *Computer Languages*, 16:97–107. (Cited on pages 5, 72, 105, and 180.)

Danaher, J. S., Angelina Lee, I. T., and Leiserson, C. E. (2006). Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171. (Cited on page 45.)

Daniel, B., Dig, D., Garcia, K., and Marinov, D. (2007). Automated testing of refactoring engines. In Crnkovic, I. and Bertolino, A., editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the Int. Symposium on Foundations of Software Engineering (ESEC/FSE 2007)*, pages 185–194. ACM. (Cited on page 225.)

de Jonge, M., Nilsson-Nyman, E., Kats, L. C. L., and Visser, E. (2009). Natural and flexible error recovery for generated parsers. In van den Brand, M.,

Gasevic, D., and Gray, J., editors, *Proceedings of the Second International Conference on Software Language Engineering (SLE 2009)*, volume 5969 of *Lecture Notes in Computer Science*, pages 204–223. Springer. (Cited on pages 17, 133, 154, and 169.)

Degano, P. and Priami, C. (1995). Comparison of syntactic error handling in LR parsers. *Software – Practice & Experience*, 25(6):657–679. (Cited on pages 132, 145, 147, 149, 173, and 221.)

Detlefs, D. and Agesen, O. (1999). Inlining of virtual methods. In Guerraoui, R., editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2009)*, volume 1628 of *Lecture Notes in Computer Science*, pages 258–278. Springer. (Cited on page 66.)

van Deursen, A. and Klint, P. (1998). Little languages: little maintenance? *Journal of Software Maintenance*, 10(2):75–92. (Cited on page 1.)

van Deursen, A., Klint, P., and Tip, F. (1993). Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545. (Cited on pages 43, 75, 90, 100, 104, and 230.)

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36. (Cited on pages 1 and 71.)

van Deursen, A. and Kuipers, T. (1999). Building documentation generators. In *IEEE International Conference on Software Maintenance (ICSM 1999)*, page 40. IEEE. (Cited on pages 136, 153, and 176.)

van Dijk, M. H. H. and Koorn, J. W. C. (1990). GSE, a generic syntax-directed editor. Technical Report CS-R9045, Centrum voor Wiskunde en Informatica (CWI). (Cited on page 7.)

DLTK (2007). Dynamic language toolkit (DLTK). `http://www.eclipse.org/dltk/`. (Cited on pages 6, 72, and 98.)

Dov, A. B. (2008). infomancers-collections. `http://code.google.com/p/infomancers-collections/`. (Cited on page 28.)

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388. (Cited on pages 21, 25, and 133.)

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102. (Cited on page 5.)

Efftinge, S. et al. (2008). openArchitectureWare User Guide. Version 4.3. Available from `http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/`. (Cited on pages 104 and 196.)

Efftinge, S. and Voelter, M. (2006). oAW xText: a framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*. (Cited on pages 7, 74, 103, 104, and 204.)

Ekman, T. and Hedin, G. (2004). Rewritable reference attributed grammars. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169. Springer. (Cited on page 129.)

Ekman, T. and Hedin, G. (2007). The JastAdd extensible Java compiler. In Gabriel, R. P., Bacon, D. F., Lopes, C. V., and Jr., G. L. S., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 1–18. ACM. (Cited on pages 20 and 43.)

Erdweg, S., Kats, L. C. L., Rendel, T., Kästner, C., Ostermann, K., and Visser, E. (2011a). Growing a language environment with editor libraries. In Denney, E. and Schultz, U. P., editors, *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE 2011)*. ACM. (Cited on pages 16, 101, and 107.)

Erdweg, S., Rendel, T., Kästner, C., and Ostermann, K. (2011b). SugarJ: Library-based syntactic language extensibility. In Fisher, K. S., editor, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, SIGPLAN Notices, Portland, Oregon, USA. ACM. (Cited on pages 16, 101, and 107.)

Farnum, C. (1992). Pattern-based tree attribution. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'92)*, pages 211–222. (Cited on page 111.)

Ford, B. (2002). Packrat parsing: simple, powerful, lazy, linear time. Functional pearl. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming (ICFP 2002)*, pages 36–47, New York, NY, USA. ACM. (Cited on pages 135 and 175.)

Fowler, M. (2005a). Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`. (Cited on pages 7, 72, 73, 80, 160, 204, and 212.)

Fowler, M. (2005b). PostIntelliJ. `http://martinfowler.com/bliki/PostIntelliJ.html`. (Cited on page 71.)

Fowler, M. (2009). A pedagogical framework for domain-specific languages. *IEEE Software*, 26:13–14. (Cited on page 72.)

Fowler, M. (2011). *Domain-Specific Languages*. Addison Wesley. (Cited on pages 3, 7, 72, and 224.)

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional. (Cited on pages 217 and 234.)

Frens, J. D. and Meneely, A. (2006). Fifteen compilers in fifteen days. In Baldwin, D., Tymann, P. T., Haller, S. M., and Russell, I., editors, *Proceedigns of the 39th Technical Symposium on Computer Science Education (SIGCSE 2006)*, pages 92–96. ACM. (Cited on page 223.)

Gamma, E. and Beck, K. (1998). Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50. (Cited on pages 203, 208, and 211.)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional. (Cited on page 128.)

Goldschmidt, T., Becker, S., and Uhl, A. (2008). Classification of concrete textual syntax mapping approaches. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA 2008)*, pages 169–184, Berlin, Heidelberg. Springer-Verlag. (Cited on page 103.)

Gómez, R., Augusto, J. C., and Galton, A. (2001). Testing an event specification language. In *Proceedings of the 13th International Conference on Software Engineering and Knowledge Engineering (SEKE'01)*, pages 341–345. (Cited on pages 202 and 223.)

Goodenough, J. B. (1980). The Ada compiler validation capability. In *Proceedings of the ACM-SIGPLAN symposium on Ada programming language*, pages 1–8. ACM. (Cited on pages 202 and 223.)

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification*. Prentice Hall PTR, Boston, Mass., third edition. (Cited on pages 33 and 36.)

Groenewegen, D. and Visser, E. (2011). Integration of data validation and user interface concerns in a DSL for web applications. *Software and Systems Modeling*, pages 1–18. (Cited on page 16.)

Groenewegen, D. M., Hemel, Z., and Visser, E. (2010). Separation of concerns and linguistic integration in WebDSL. *IEEE Software*, 27(5):31–37. (Cited on pages 60, 66, 82, 87, 102, 107, 182, 194, and 233.)

Groenewegen, D. M. and Visser, E. (2008). Declarative access control for WebDSL: Combining language integration and separation of concerns. In Schwabe, D., Curbera, F., and Dantzig, P., editors, *Proceedings of the Eighth International Conference on Web Engineering (ICWE 2008)*, pages 175–188. IEEE. (Cited on page 16.)

Grune, D. and Jacobs, C. J. H. (2008). *Parsing techniques: a practical guide*. Springer-Verlag New York Inc. (Cited on page 5.)

gUnit (2007). gUnit - grammar unit testing. http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing. (Cited on page 224.)

Hamill, P. (2004). *Unit Test Frameworks, chapter. Chapter 3: The xUnit Family of Unit Test Frameworks*. O'Reilly. (Cited on pages 15, 202, 207, and 223.)

Hardwick, J. C. and Sipelstein, J. (1996). Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University. (Cited on page 44.)

Hedin, G. (2000). Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317. (Cited on pages 110, 119, 120, and 129.)

Hedin, G. and Magnusson, E. (2003). JastAdd – an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58. (Cited on pages 5, 72, 105, 113, 127, and 128.)

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75. (Cited on pages 5, 9, 75, 78, 82, 133, 135, and 219.)

Heering, J. and Klint, P. (2000). Semantics of programming languages: A tool-oriented approach. *SIGPLAN Notices*, 35(3):39–48. (Cited on page 72.)

Heidenreich, F., Johannes, J., Karol, S., Seifert, M., and Wende, C. (2009a). Derivation and refinement of textual syntax for models. In Paige, R. F., Hartman, A., and Rensink, A., editors, *Proceedings of Model Driven Architecture - Foundations and Applications, 5th European Conference (ECMDA-FA 2009)*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer. (Cited on pages 7, 74, 103, and 104.)

Heidenreich, F., Johannes, J., Seifert, M., Wende, C., and Marcel, B. (2009b). Generating safe template languages. In *Proceedings of the eighth international conference on Generative programming and component engineering (GPCE 2009)*, pages 99–108, New York, NY, USA. ACM. (Cited on page 196.)

Hemel, Z., Groenewegen, D. M., Kats, L. C. L., and Visser, E. (2011). Static consistency checking of web applications with WebDSL. *Journal of Symbolic Computation*, 46(2):150–182. (Cited on pages 4, 16, and 102.)

Hemel, Z., Kats, L. C. L., Groenewegen, D. M., and Visser, E. (2009). Code generation by model transformation. A case study in transformation modularity. *Software and Systems Modeling*, 9(3):375–402. (Cited on pages 16, 96, 102, and 229.)

Hemel, Z., Kats, L. C. L., and Visser, E. (2008). Code generation by model transformation. A case study in transformation modularity. In Gray, J., Pierantonio, A., and Vallecillo, A., editors, *Theory and Practice of Model Transformations. First International Conference on Model Transformation (ICMT 2008)*,

volume 5063 of *Lecture Notes in Computer Science*, pages 183–198, Heidelberg. Springer. (Cited on page 16.)

Hemel, Z. and Visser, E. (2009). PIL: A platform independent language for retargetable DSLs. In van den Brand, M. and Gray, J., editors, *Proceedings of the Second International Conference on Software Language Engineering (SLE 2009)*, volume 5969 of *Lecture Notes in Computer Science*, pages 224–243. Springer. (Cited on pages 66, 68, and 101.)

Hemel, Z. and Visser, E. (2011). Declaratively programming the mobile web with mobl. In Fisher, K. S., editor, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, SIGPLAN Notices, Portland, Oregon, USA. ACM. (Cited on pages 101, 106, 206, 213, 224, and 228.)

Herndon, Jr., R. M. and Berzins, V. A. (1988). The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14:803–809. (Cited on page 229.)

Hirzel, M. and Grimm, R. (2007). Jeannie: granting Java Native Interface developers their wishes. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA 2007)*, volume 42 of *SIGPLAN Notices*, pages 19–38, New York, NY, USA. ACM. (Cited on page 44.)

Huang, S. S., Zook, D., and Smaragdakis, Y. (2008). Domain-specific languages and program generation with Meta-AspectJ. *ACM Transactions on Software Engineering and Methodology*, 18:6:1–6:32. (Cited on pages 180, 183, 193, and 197.)

Hudak, P. (1998). Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse (ICSR 1998)*, pages 134–142, Washington, DC, USA. IEEE. (Cited on pages 3, 106, 129, and 224.)

Hugunin, J. (2006). Dynamic languages on .NET - IronPython and beyond: IronPython 1.0 released today! `http://blogs.msdn.com/hugunin/archive/2006/09/05/741605.aspx`. (Cited on page 48.)

Jalili, F. (1983). A general linear time evaluator for attribute grammars. *SIGPLAN Notices*, 18(9):35–44. (Cited on page 127.)

Johnson, S. C. (1975). YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories. (Cited on page 5.)

Johnstone, A., Scott, E., and Economopoulos, G. (2004). Generalised parsing: Some costs. In Duesterwald, E., editor, *Proceedings of the 13th International Conference on Compiler Construction (CC 2004)*, pages 89–103. (Cited on pages 149 and 195.)

Jones, S. P., Ramsey, N., and Reig, F. (1999). C−−: A portable assembly language that supports garbage collection. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP 1999)*, volume 1702 of *Lecture Notes in Computer Science*, pages 1–28. Springer. (Cited on page 68.)

Jouault, F., Bézivin, J., and Kurtev, I. (2006). TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative and Component Engineering (GPCE 2006)*, pages 249–254. ACM. (Cited on pages 103 and 104.)

Kalleberg, K. T. and Visser, E. (2006). Combining aspect-oriented and strategic programming. In Cirstea, H. and Marti-Oliet, N., editors, *Proceedings of the 6th International Workshop on Rule-Based Programming (RULE 2005)*, volume 147 of *Electronic Notes in Theoretical Computer Science*, pages 5–30. Elsevier Science Publishers. (Cited on pages 55 and 68.)

Kalleberg, K. T. and Visser, E. (2007a). Fusing a transformation language with an open compiler. In Sloane, A. and Johnstone, A., editors, *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 21–36. Elsevier. (Cited on pages 49, 59, 67, and 100.)

Kalleberg, K. T. and Visser, E. (2007b). Spoofax: An interactive development environment for program transformation with Stratego/XT. In Sloane, A. and Johnstone, A., editors, *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 47–50. Elsevier. (Cited on pages 75 and 105.)

Kam, J. B. and Ullman, J. D. (1977). Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317. (Cited on page 38.)

Kastens, U. and Waite, W. M. (1994). Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627. (Cited on pages 5, 109, and 113.)

Kats, L. C. L., Bravenboer, M., and Visser, E. (2008a). Mixing source and bytecode. A case for compilation by normalization. In Kiczales, G., editor, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*, volume 43 of *SIGPLAN Notices*, pages 91–108, New York, NY, USA. ACM. (Cited on page 17.)

Kats, L. C. L., de Jonge, M., Nilsson-Nyman, E., and Visser, E. (2009a). Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In Leavens, G. T., editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, volume 44 of *SIGPLAN Notices*, pages 445–464, New York, NY, USA. ACM. (Cited on pages 17 and 133.)

Kats, L. C. L., Kalleberg, K. T., and Visser, E. (2009b). Domain-specific languages for composable editor plugins. In *Proceedings of the Ninth International Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 149–163. Elsevier. (Cited on page 75.)

Kats, L. C. L., Kalleberg, K. T., and Visser, E. (2011a). Interactive disambiguation of meta programs with concrete object syntax. In van den Brand, M., Malloy, B., and Staab, S., editors, *Proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *Lecture Notes in Computer Science*, pages 327–336. Springer. (Cited on page 17.)

Kats, L. C. L., Sloane, A. M., and Visser, E. (2008b). Decorated attribute grammars: Attribute evaluation meets strategic programming. Technical Report TUD-SERG-2008-038a (extended version), Software Engineering Research Group, Delft University of Technology. (Cited on page 127.)

Kats, L. C. L., Sloane, A. M., and Visser, E. (2009c). Decorated attribute grammars: Attribute evaluation meets strategic programming. In de Moor, O. and Schwartzbach, M. I., editors, *Proceedings of the 18th International Conference on Compiler Construction (CC 2009)*, volume 5501 of *Lecture Notes in Computer Science*, pages 142–157. Springer. (Cited on pages 17 and 111.)

Kats, L. C. L., Vermaas, R., and Visser, E. (2011b). Integrated language definition testing: Enabling test-driven language development. In Fisher, K. S., editor, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, SIGPLAN Notices. ACM. (Cited on page 17.)

Kats, L. C. L. and Visser, E. (2010a). Encapsulating software platform logic by aspect-oriented programming: A case study in using aspects for language portability. In Marinescu, C. and Vinju, J., editors, *Proceedings of the Tenth International Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*, pages 147–156. IEEE. (Cited on page 17.)

Kats, L. C. L. and Visser, E. (2010b). The Spoofax language workbench: rules for declarative specification of languages and IDEs. In Cook, W. R., Clarke, S., and Rinard, M. C., editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, pages 444–463. ACM. (Cited on pages 16 and 17.)

Kats, L. C. L., Visser, E., and Wachsmuth, G. (2010). Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA 2010)*, volume 45 of *SIGPLAN Notices*, pages 918–932, New York, NY, USA. ACM. (Cited on pages 105, 215, and 232.)

Khwaja, A. A. and Urban, J. E. (1993). Syntax-directed editing environments: issues and features. In *Proceedings of the 1993 ACM/SIGAPP symposium on*

Applied computing: states of the art and practice (SAC 1993), pages 230–237, New York, NY, USA. ACM. (Cited on page 7.)

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355. (Cited on pages 55, 56, 69, and 133.)

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 1997)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer. (Cited on pages 11, 21, 44, 49, 128, and 133.)

Klint, P. (1993). A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201. (Cited on pages 7, 14, 74, 103, 105, and 175.)

Klint, P., van der Storm, T., and Vinju, J. (2009). Rascal: a domain specific language for source code analysis and manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM 2009)*, pages 168–177. (Cited on pages 5, 72, 105, and 180.)

Klusener, S. and Lämmel, R. (2003). Deriving tolerant grammars from a base-line grammar. In *International Conference on Software Maintenance (ICSM 2003)*, pages 179–189. IEEE. (Cited on page 176.)

Knuth, D. E. (1968). Semantics of context-free languages. *Theory of Computing Systems*, 2:127–145. 10.1007/BF01692511. (Cited on pages 109 and 111.)

Koorn, J. W. C. (1993). Connecting semantic tools to a syntax-directed user-interface. *Proceedings of the Conference on Computing Science in the Netherlands (CSN 1993)*, pages 217–228. (Cited on pages 74 and 103.)

Kossatchev, A. and Posypkin, M. (2005). Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19. (Cited on pages 15, 202, 224, and 226.)

Krahn, H., Rumpe, B., and Völkel, S. (2007). Efficient editor generation for compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, technical report TR-38, pages 218–228. University of Jyväskylä. (Cited on page 175.)

Krahn, H., Rumpe, B., and Völkel, S. (2008). Monticore: Modular development of textual domain specific languages. In Paige, R. F. and Meyer, B., editors, *Objects, Components, Models and Patterns, TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer. (Cited on pages 7, 74, 103, 104, 135, 175, 204, and 220.)

Kuhn, T. and Thomann, O. (2006). Eclipse corner: Abstract syntax tree. `http://eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`. (Cited on page 169.)

Kuiper, M. F. and Saraiva, J. (1998). Lrc - a generator for incremental language-oriented tools. In *Proceedings of the 7th International Conference on Compiler Construction (CC 1998)*, pages 298–301, London, UK. Springer-Verlag. (Cited on pages 6 and 72.)

Lämmel, R. (2001). Grammar testing. In Hußmann, H., editor, *Proceedings of the Fourth International Conference on Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 201–216. Springer. (Cited on page 226.)

Lämmel, R. (2003). Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 54(1):1–64. (Cited on pages 128 and 198.)

Lämmel, R., Visser, E., and Visser, J. (2003). Strategic programming meets adaptive programming. In *Proceedings of Aspect-Oriented Software Development (AOSD 2003)*, pages 168–177, Boston, USA. ACM. (Cited on pages 110, 114, 115, and 231.)

Lavie, A. and Tomita, M. (1993). GLR* – an efficient noise skipping parsing algorithm for context free grammars. In *Proceedings of the Third International Workshop on Parsing Technologies*, pages 123–134. (Cited on page 174.)

Lee, P. (1989). *Realistic compiler generation*. MIT Press, Cambridge, MA, USA. (Cited on page 72.)

Lévy, J.-P. (1971). *Automatic Correction of Syntax Errors in Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, USA. (Cited on page 174.)

Liang, S. (1999). *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 44.)

Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition. (Cited on pages 37 and 45.)

Logozzo, F. and Fähndrich, M. (2008). On the relative completeness of bytecode analysis versus source code analysis. In Hendren, L., editor, *Proceedings of the 17th International Conference on Compiler Construction (CC 2008)*, volume 4959 of *Lecture Notes in Computer Science*, pages 192–212. Springer. (Cited on page 42.)

Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., and Schröder-Preikschat, W. (2006). A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*

(EuroSys 2006), volume 40 of *ACM SIGOPS Operating Systems Review*, pages 191–204, New York, NY, USA. ACM. (Cited on page 67.)

Magnusson, E., Ekman, T., and Hedin, G. (2007). Extending attribute grammars with collection attributes – evaluation and applications. *Proceedings of the Int. Working Conference on Source Code Analysis and Manipulation*, pages 69–80. (Cited on pages 110, 121, and 129.)

Magnusson, E. and Hedin, G. (2007). Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming*, 68(1):21–37. (Cited on pages 12, 16, 110, 122, 123, 125, and 129.)

Malloy, B. A., Power, J. F., and Waldron, J. T. (2002). Applying software engineering techniques to parser design: the development of a C# parser. In *Proceedings on the research conference of the South African institute of computer scientists and information technologists on Enablement through technology (SAICSIT 2002)*. South African Institute for Computer Scientists and Information Technologists. (Cited on page 223.)

Mauney, J. and Fischer, C. N. (1988). Determining the extent of lookahead in syntactic error repair. *ACM Transactions on Programming Languages and Systems*, 10(3):456–469. (Cited on page 174.)

Melton, J. and Eisenberg, A. (2000). *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*. Morgan Kaufmann. (Cited on pages 20 and 44.)

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):344. (Cited on pages 1, 3, 48, and 71.)

Meyer, J. and Downing, T. (1997). *Java Virtual Machine*. O'Reilly & Associates, Inc., Sebastopol, CA, USA. (Cited on page 33.)

Miecznikowski, J. and Hendren, L. (2001). Decompiling Java using staged encapsulation. *Proceedings of the Workshop on Decompilation Techniques, appeared in Proceedings of the Working Conference on Reverse Engineering (WCRE 2001)*, pages 368–374. (Cited on page 28.)

Moonen, L. (2001). Generating robust parsers using island grammars. In *Proceedings of the Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE. (Cited on pages 136, 137, 153, and 176.)

Moonen, L. (2002). Lightweight impact analysis using island grammars. In *Proceedings of the 10th IEEE International Workshop of Program Comprehension*, pages 219–228. IEEE. (Cited on pages 136 and 153.)

de Moor, O., Backhouse, K., and Swierstra, S. D. (2000). First-class attribute grammars. *Informatica*, 24(3):329–341. (Cited on page 128.)

de Moor, O., Peyton Jones, S. L., and Van Wyk, E. (1999). Aspect-oriented compilers. In Czarnecki, K. and Eisenecker, U. W., editors, *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GPCE 1999)*, volume 1799 of *Lecture Notes in Computer Science*, pages 121–133. Springer. (Cited on page 67.)

Murer, S., Omohundro, S., Stoutamire, D., and Szyperski, C. (1996). Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15. (Cited on page 27.)

Myers, G. J. (2008). *The art of software testing, 2nd edition*. Wiley-India. (Cited on pages 202, 204, 205, and 206.)

Nilsson-Nyman, E., Ekman, T., and Hedin, G. (2009). Practical scope recovery using bridge parsing. In Gasevic, D., Lämmel, R., and Wyk, E. V., editors, *Proceedings of the First International Conference on Software Language Engineering (SLE 2008)*, volume 5452 of *Lecture Notes in Computer Science*, pages 95–113. (Cited on pages 133, 153, and 154.)

Nilsson-Nyman, E., Ekman, T., Hedin, G., and Magnusson, E. (2008). Declarative intraprocedural flow analysis of Java source code. In Johnstone, A. and Vinju, J., editors, *Proceedings of the 8th International Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238 of *Electronic Notes in Theoretical Computer Science*, pages 155–171. Springer. (Cited on pages 12, 16, 120, 122, and 123.)

Nutter, C. (2008). Promise and peril for alternative Ruby impls. `http://blog.headius.com/2008/04/promise-and-peril-for-alternative-ruby.html`. (Cited on page 48.)

Nystrom, N., Clarkson, M. R., and Myers, A. C. (2003). Polyglot: An extensible compiler framework for Java. *Proceedings of the 12th International Conference on Compiler Construction (CC 2003)*, 2622:138–152. (Cited on pages 5, 20, 43, and 72.)

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala*. Artima Press. (Cited on pages 26, 51, and 65.)

OSGi (2009). *OSGi Service Platform, Core Specification, Release 4, Version 4.2*. OSGi Alliance. (Cited on pages 6 and 79.)

Paakki, J. (1995). Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255. (Cited on pages 12 and 109.)

Pai, A. B. and Kieburtz, R. B. (1980). Global context recovery: A new strategy for syntactic error recovery by table-drive parsers. *ACM Transactions on Programming Languages and Systems*, 2(1):18–41. (Cited on pages 155 and 174.)

Parr, T. and Fisher, K. (2011). LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 425–436, New York, NY, USA. ACM. (Cited on pages 5 and 135.)

Parr, T. J. (2004). Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web (WWW 2004)*, pages 224–233. ACM. (Cited on page 196.)

Pedroni, S. and Rappin, N. (2002). *Jython Essentials*. O'Reilly Media, Inc. (Cited on page 45.)

Pennello, T. J. and DeRemer, F. (1978). A forward move algorithm for LR error recovery. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL 1978)*, pages 241–254, New York, NY, USA. ACM. (Cited on pages 166, 169, and 232.)

Peyton Jones, S. L. and Santos, A. L. M. (1998). A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47. (Cited on pages 23 and 44.)

Pfeiffer, M. and Pichler, J. (2008). A comparison of tool support for textual domain-specific languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 1–7. (Cited on page 103.)

Pleban, U. F. (1984). Formal semantics and compiler generation. In Morgenbrod, H. and Sammer, W., editors, *Programmierumgebungen und Compiler*, pages 145–161. Teubner-Verlag. (Cited on page 72.)

Rekers, J. (1992). *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam. (Cited on page 13.)

Rekers, J. and Koorn, J. W. C. (1991). Substring parsing for arbitrary context-free grammars. *SIGPLAN Notices*, 26(5):59–66. (Cited on page 174.)

Renggli, L., Gîrba, T., and Nierstrasz, O. (2010). Embedding languages without breaking tools. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *Lecture Notes in Computer Science*, pages 380–404. Springer-Verlag. (Cited on pages 106 and 107.)

Reps, T. W. and Teitelbaum, T. (1989). *The synthesizer generator: a system for constructing language-based editors*. Springer-Verlag New York, Inc., New York, NY, USA. (Cited on pages 6 and 72.)

Reynolds, A., Fiuczynski, M. E., and Grimm, R. (2008). On the feasibility of an AOSD approach to linux kernel extensions. In *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software (ACP4IS 2008)*, pages 8:1–8:7, New York, NY, USA. ACM. (Cited on page 67.)

van Rossum, G. (2000). *Python Reference Manual*. iUniverse. (Cited on page 128.)

Salomon, D. J. and Cormack, G. V. (1989). Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178. (Cited on pages 5, 13, and 134.)

Salomon, D. J. and Cormack, G. V. (1995). The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical report, TR 95/06, Dept. of Comp. Sci., University of Manitoba, Winnipeg, Canada. (Cited on pages 5, 13, and 134.)

Saunders, S., Fields, D. K., and Belayev, E. (2006). *IntelliJ IDEA in Action*. Manning. (Cited on page 71.)

ScalaNet (2008). Scala on .NET: quirks. `http://www.scala-lang.org/node/169`. (Cited on page 68.)

Scheidgen, M. (2010). Textual modelling embedded into graphical modelling. In *Model Driven Architecture–Foundations and Applications*, pages 153–168. Springer. (Cited on pages 74, 103, and 104.)

Schinz, M. and Odersky, M. (2001). Tail call elimination on the Java Virtual Machine. In *Proceedings of the Workshop on Multi-Language Infrastructure and Interoperability (BABEL 2001)*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 158–171. Elsevier. (Cited on page 52.)

Selby, R. W., editor (2007). *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*. Wiley-Computer Society Press. (Cited on page 210.)

Seymour, K. and Dongarra, J. (2001). Automatic translation of Fortran to JVM bytecode. In *Proceedings of the joint ACM-ISCOPE conference on Java Grande (JGI 2001)*, pages 126–133, New York, NY, USA. ACM. (Cited on pages 44 and 45.)

Shani, U. (1983). Should program editors not abandon text oriented commands? *SIGPLAN Notices*, 18:35–41. (Cited on page 7.)

Sheard, T. and Jones, S. P. (2002). Template meta-programming for Haskell. *SIGPLAN Notices*, 37:60–75. (Cited on page 106.)

Simonyi, C. (1995). The death of computer languages, the birth of Intentional Programming. Tech. report, MS Research. (Cited on page 73.)

Singh, N., Gibbs, C., and Coady, Y. (2007). C-CLR: a tool for navigating highly configurable system software. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software (ACP4IS 2007)*. ACM. (Cited on page 67.)

Sloane, A. M., Kats, L. C. L., and Visser, E. (2009). A pure object-oriented embedding of attribute grammars. In Ekman, T. and Vinju, J., editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers. (Cited on page 129.)

Smaragdakis, Y. and Batory, D. (2002). Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255. (Cited on page 43.)

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99. (Cited on page 1.)

Spinellis, D. and Guruprasad, V. (1997). Lightweight languages as software engineering tools. In *Proceedings of the Conference on Domain-Specific Languages, October 15-17, 1997, Santa Barbara, California, USA*, pages 67–76. USENIX. (Cited on page 1.)

Spoofax (2011). The Spoofax project. `http://www.spoofax.org/`. (Cited on pages 133 and 164.)

Stahl, T., Voelter, M., and Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons. (Cited on pages 47 and 48.)

Steel, Jr., T. B. (1961). A first version of UNCOL. In *IRE-AIEE-ACM '61 (Western): Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 371–378, New York, NY, USA. ACM. (Cited on page 68.)

Steele, G. (1999). Growing a language. *Higher-Order and Symbolic Compututation*, 12(3):221–236. (Cited on pages 12, 19, and 110.)

Stodte, M. (2001). Jacks: Java compatibility testing, the open source way. IBM developerWorks. (Cited on page 223.)

Strembeck, M. and Zdun, U. (2009). An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292. (Cited on page 202.)

Sun Microsystems (2004). The annotation processing tool (apt). `http://java.sun.com/j2se/1.5.0/docs/guide/apt`. (Cited on pages 20 and 45.)

Swierstra, S. D. and Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming, Second International School–Tutorial Text*, pages 184–207, London, UK. Springer-Verlag. (Cited on page 175.)

Synytskyy, N., Cordy, J. R., and Dean, T. R. (2003). Robust multilingual parsing using island grammars. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research (CASCON 2003)*, pages 266–278. IBM Press. (Cited on page 176.)

Tatsubori, M., Chiba, S., Itano, K., and Killijian, M.-O. (1999). OpenJava: A class-based macro system for Java. In Cazzola, W., Stroud, R. J., and Tisato, F., editors, *Proceedings of the First OOPSLA Workshop on Reflection and Software Engineering (OORaSE'99)*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer. (Cited on page 43.)

Tolmach, A. (2001). An external representation for the GHC core language. `http://haskell.org/ghc/docs/papers/core.ps.gz`. (Cited on page 44.)

Tomita, M. (1988). Efficient parsing for natural language: A fast algorithm for practical systems. *Computational Linguistics*, 14(2). (Cited on pages 5, 13, 132, 135, and 150.)

Tratt, L. (2008). Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems*, 30:31:1–31:40. (Cited on pages 106 and 224.)

Valkering, R. (2007). Syntax error handling in scannerless generalized LR parsers. Master's thesis, University of Amsterdam. (Cited on page 174.)

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999). Soot - a Java bytecode optimization framework. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research (CASCON 1999)*, pages 13–23. IBM Press. (Cited on pages 34 and 44.)

Van Wyk, E., Bodin, D., Gao, J., and Krishnan, L. (2010). Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54. (Cited on pages 123 and 128.)

Van Wyk, E., de Moor, O., Backhouse, K., and Kwiatkowski, P. (2002). Forwarding in attribute grammars for modular language design. In Horspool, R. N., editor, *Proceedings of the 11th International Conference on Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes on Computer Science*, pages 128–142, London, UK. Springer-Verlag. (Cited on pages 43 and 128.)

Van Wyk, E., Krishnan, L., Bodin, D., and Johnson, E. (2006). Adding domain-specific and general purpose language features to Java with the Java language extender. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006)*, pages 728–729. ACM. (Cited on page 113.)

Van Wyk, E., Krishnan, L., Schwerdfeger, A., and Bodin, D. (2007). Attribute grammar-based language extensions for Java. In Ernst, E., editor, *Proceedings of the European Object Oriented Programming (ECOOP 2007)*, volume 4609 of *Lecture Notes on Computer Science*, pages 575–599. Springer Verslag. (Cited on pages 20, 43, and 123.)

Vermolen, S. D., Wachsmuth, G., and Visser, E. (2011). Generating database migration for evolving web applications. In Denney, E. and Schultz, U. P., editors, *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE 2011)*. ACM. (Cited on pages 16 and 101.)

Viera, M., Swierstra, S. D., and Swierstra, W. (2009). Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. *SIGPLAN Notices*, 44:245–256. (Cited on page 128.)

Vinju, J. J. (2005). Type-driven automatic quotation of concrete object code in meta programs. In Guelfi, N. and Savidis, A., editors, *Proceedings of the Second International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2005)*, volume 3943 of *Lecture Notes in Computer Science*, pages 97–112. Springer. (Cited on pages 180, 183, 188, 196, 197, 198, and 228.)

Visser, E. (1997a). A case study in optimizing parsing schemata by disambiguation filters. In *International Workshop on Parsing Technologies (IWPT 1997)*, pages 210–224, Boston, USA. Massachusetts Institute of Technology. (Cited on page 135.)

Visser, E. (1997b). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam. (Cited on pages 13, 58, 132, and 135.)

Visser, E. (1997c). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam. (Cited on pages 5, 9, 53, 75, 78, 82, 133, 135, 175, 180, 183, and 219.)

Visser, E. (2002). Meta-programming with concrete object syntax. In Batory, D. S., Consel, C., and Taha, W., editors, *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer. (Cited on pages 14, 40, 96, 105, 113, 180, 182, 183, 197, and 232.)

Visser, E. (2007). WebDSL: A case study in domain-specific language engineering. In Lämmel, R., Visser, J., and Saraiva, J., editors, *International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Heidelberg. Springer. (Cited on pages 16, 80, and 212.)

Visser, E., Benaissa, Z.-E.-A., and Tolmach, A. P. (1998). Building program optimizers with rewriting strategies. In Felleisen, M., Hudak, P., and Queinnec, C., editors, *Functional programming*, pages 13–26. ACM. (Cited on pages 54, 86, 103, 110, 114, 115, and 231.)

Voelter, M. and Solomatov, K. (2010). Language modularization and composition with projectional language workbenches illustrated with MPS. In van den Brand, M., Malloy, B., and Staab, S., editors, *Proceedings of the Second International Conference on Software Language Engineering (SLE 2010)*, volume

6395 of *Lecture Notes in Computer Science*, pages 32–46. Springer. (Cited on pages 7, 73, 107, 204, 220, 224, and 233.)

Wachsmuth, G. (2009). A formal way from text to code templates. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009)*, pages 109–123, Berlin, Heidelberg. Springer-Verlag. (Cited on page 196.)

Waddington, D. and Yao, B. (2007). High-fidelity C/C++ code transformation. *Science of Computer Programming*, 68(2):64–78. (Cited on page 135.)

Waite, W. M. and Goss, G. (1984). *Compiler construction*. Springer, New York. (Cited on page 92.)

WALA (2006). The WAtson Libraries for Analysis. `http://wala.sourceforge.net/`. (Cited on page 72.)

Ward, M. P. (1994). Language-oriented programming. *Software — Concepts and Tools*, 15(4):147–161. (Cited on page 73.)

Warmer, J. and Kleppe, A. (2006). Building a flexible software factory using partial domain specific models. In Gray, J., Tolvanen, J.-P., and Sprinkle, J., editors, *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM 2006)*, volume TR-37 of *Computer Science and Information System Reports*, pages 15–22, Finland. University of Jyväskylä. (Cited on page 24.)

Warth, A., Stanojević, M., and Millstein, T. (2006). Statically scoped object adaptation with expanders. In *Proceedings of the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006)*, volume 41 of *ACM SIGPLAN Notices*, pages 37–56, New York, NY, USA. ACM. (Cited on page 69.)

Waters, R. C. (1982). Program editors should not abandon text oriented commands. *SIGPLAN Notices*, 17:39–46. (Cited on page 7.)

Wichmann, B. A. and Ciechanowicz, Z. (1983). *Pascal compiler validation*. John Wiley & Sons, Inc. New York, NY, USA. (Cited on page 202.)

Wu, H., Gray, J., and Mernik, M. (2008). Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience*, 38(10):1073–1103. (Cited on page 225.)

XDoclet (2000). XDoclet: attribute-oriented programming. `http://xdoclet.sourceforge.net/`. (Cited on page 20.)

Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI 2011)*, pages 283–294, New York, NY, USA. ACM. (Cited on page 225.)

Samenvatting

BOUWSTENEN VOOR TAALONTWIKKELOMGEVINGEN

– Lennart C. L. Kats –

Programmeertalen hebben een belangrijke rol in de informatica. Ze maken het mogelijk om complexe softwaresystemen te beschrijven en ontwikkelen op een gestructureerde manier. Oorspronkelijk richtten programmeertalen zich vooral op het aansturen van hardware, zoals allocatie, deallocatie en manipulatie van geheugen en het bedienen van randapparatuur. Moderne talen maken het mogelijk om software op een hoger niveau te beschrijven, door te abstraheren over deze primitieve operaties. Ze bieden *abstracties* voor programmeerconcepten als berekeningen, functies en objecten. Door de introductie van nieuwe abstracties maken moderne programmeertalen het mogelijk om software van aanzienlijk grotere schaal, complexiteit en flexibiliteit te maken.

Domein-specifieke talen zijn programmeertalen gericht op een bepaalde toepassingsdomein, zoals verzekeringen of bedrijfsadministratie, of een technische domein, zoals databases of gebruikersinterfaces. Deze talen bieden abstracties specifiek voor het beoogde domein en maken het mogelijk een bijpassende *notatie* te gebruiken. Doordat de talen expliciet en op precieze wijze concepten in het domein kunnen beschrijven is het ook mogelijk om een gespecialiseerde *ontwikkelomgeving* te maken. Die kan helpen bij het schrijven van programma's en het opsporen van fouten bij het gebruik van een domeinspecifieke taal. De combinatie van domein-specifieke abstracties, notatie en ontwikkelomgeving maken het mogelijk om op efficiëntere wijze software te ontwikkelen en te onderhouden.

Er komt veel kijken bij de ontwikkeling van een nieuwe domein-specifieke taal: het ontwerpen en implementeren van een bijpassende notatie, het toewijzen van betekenis (semantiek) aan deze notatie zodat programma's geschreven in de taal uitgevoerd kunnen worden. Daarnaast dient een ontwikkelomgeving voor de taal ontworpen te worden. Om de voordelen van domeinspecifieke talen optimaal te benutten is het belangrijk dat deze aspecten van het ontwikkelen van talen efficiënt kunnen worden uitgevoerd.

Dit proefschrift richt zich op onderzoek naar tools, technieken en methodieken om op efficiënte wijze nieuwe domein-specifieke talen te ontwikkelen. Het uitgangspunt daarbij is het gebruik van en het ontwikkelen van technieken voor een *taalontwikkelomgeving* (*language workbench*), een ontwikkelomgeving waarin diverse middelen voor het ontwerpen en implementeren van nieuwe talen worden verenigd. Daarbij onderscheiden we drie onderzoeksthema's die hieronder worden beschreven.

Talen voor declaratieve taalspecificatie. Net als bij de ontwikkeling van andere softwaresystemen kan het gebruik van domein-specifieke talen helpen bij het implementeren van nieuwe talen. Domein-specifieke talen kunnen worden toegepast bij taken als het definiëren van de notatie van een nieuwe taal, de betekenis van deze notatie en de ontwikkelomgeving. Dit proefschrift introduceert *Spoofax*, een taalontwikkelomgeving waarin deze drie aspecten samen beschreven kunnen worden in een *taaldefinitie*. Een taaldefinitie in Spoofax bestaat uit een samenspel tussen de talen SDF voor notatie, Stratego voor semantiek en de editor descriptor taal ESV voor ontwikkelomgevingen.

Belangrijke eigenschappen van taaldefinities zijn *abstractie*, *modulariteit*, *uitbreidbaarheid* en *portabiliteit*. Een overzicht van Spoofax en een beschrijving van de opbouw van taaldefinities en *abstracties* voor ontwikkelomgevingen wordt gegeven in hoofdstuk 4. Door taaldefinities op modulaire wijze op te bouwen, wordt het makkelijker om deze uit te breiden met nieuwe componenten. Hoofdstuk 2 beschrijft een nieuwe aanpak voor de *modulariteit* en *uitbreidbaarheid* van talen, waarbij het mogelijk is met een minimale kennis van een specifieke taaldefinitie toch uitbreidingen op de taal te schrijven. Hoofdstuk 3 onderzoekt technieken om de *portabiliteit* van taaldefinities te verzekeren, i.e. het mogelijk maken om een taal ook op een ander platform te gebruiken. Hoofdstuk 5 richt zich op verdere *abstracties* voor taaldefinities en introduceert een nieuwe domein-specifieke taal waarin taalontwerpers gebruik kunnen maken van een uitbreidbare bibliotheek met abstracties voor taalontwikkeling.

Declaratieve specificatie van syntaxis. De *syntaxis* (of notatie) van een taal is een set regels die bepaalt welke zinnen en welke woorden deel uitmaken van de taal. Die regels kunnen beschreven worden in de vorm van een *grammatica*. Er zijn veel talen die gebruikt kunnen worden om grammatica's te beschrijven, maar vaak leggen die beperkingen op aan de regels en aan het hergebruiken van bestaande regels. De taal SDF heeft als bijzondere eigenschap dat het *de volledige klasse van context-vrije grammatica's* ondersteund, waardoor taalontwerpers veel vrijheid hebben in het opstellen van regels en het hergebruiken van bestaande regels. Daarmee is het een van de onderscheidende componenten van Spoofax (hoofdstuk 4).

Moderne ontwikkelomgevingen maken gebruik van grammatica's om te controleren of een programma aan de regels voldoet en om eigenschappen van het programma te bepalen om de programmeur beter te ondersteunen. Daarbij is het belangrijk dat de ontwikkelomgeving ook overweg kan met programma's met fouten of onvolledigheden erin, die vaak voorkomen tijdens het bewerken van een programma. Als dit niet het geval is, kan de ontwikkelomgeving programmeurs geen ondersteuning bieden in die gevallen. Het bieden van deze ondersteuning is niet eenvoudig, aangezien grammatica's enkel regels bevatten voor programma's die wél correct en volledig zijn volgens de syntaxis. Hoofdstuk 6 beschrijft hoe het mogelijk is om automatisch nieuwe regels af te leiden voor "foute" programma's en die regels op efficiënte wijze toe te passen.

Interactieve ondersteuning in een taalontwikkelomgeving. Moderne ontwikkelomgevingen bieden op interactieve wijze uitgebreide ondersteuning aan programmeurs, bijvoorbeeld door fouten in programma's te markeren of door suggesties te doen tijdens het typen. Dit proefschrift introduceert nieuwe technieken om op diezelfde wijze taalontwerpers te ondersteunen bij het ontwikkelen van nieuwe talen.

Een werkwijze die bij het ontwikkelen van talen wordt toegepast is het beschrijven van concepten van de taal door middel van citaties van fragmenten van de taal. Een probleem met korte citaties is vaak dat de betekenis niet duidelijk is bij een gebrek aan context: een voorbeeld uit de natuurlijke taal is het woord "vertrek", waarbij het kan gaan om weggaan of een (woon)ruimte. Hoofdstuk 7 laat zien hoe het mogelijk is om met een taalontwikkelomgeving automatisch een diagnose te stellen van dit soort gevallen en taalontwerpers in de gelegenheid te stellen om expliciet de betekenis aan te geven.

Het systematisch testen van software is een belangrijk principe voor de betrouwbaarheid van softwaresystemen. Hoofdstuk 8 introduceert een algemene aanpak om taaldefinities te testen. Het uitgangspunt hierbij is een domein-specifieke taal waarin tests beschreven kunnen worden voor notatie, semantiek en ontwikkelomgevingen van talen. Ook hier vormen citaties van taalfragmenten de basis: elke test bestaat uit een taalfragment en een specificatie van bepaalde eigenschappen waaraan het moet voldoen. Door middel van interactieve, taal-specifieke ondersteuning bij het schrijven van tests en de mogelijkheid tests automatisch te draaien kan dan op efficiënte wijze worden gecontroleerd of een taaldefinitie aan de verwachtingen voldoet.

Curriculum Vitae

Lennart C. L. Kats

23 december 1982
Geboren te Amsterdam

1995–2001
VWO diploma
Goois Lyceum in Bussum
Nature & Technology and Nature & Health profiles

2001–2007
M.Sc. in Computer Science
Utrecht University
Department of Information and Computing Science
Cum laude ('met lof')

2007–2011
Ph.D. in Computer Science
Delft University of Technology
Department of Software Technology

2011–present
Postdoctoral Research
Delft University of Technology
Department of Software Technology

Titles in the IPA Dissertation Series Since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science,

Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of*

Empirical Studies about the UML. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engi-

neering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image*

Analysis. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*.

Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23