Exercises in Free Syntax

Syntax Definition, Parsing, and Assimilation of Language Conglomerates

Exercities in vrije syntax

Syntax definitie, ontleding, en assimilatie van taalagglomeraten
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. J.C. Stoof, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op maandag 21 januari 2008 des ochtends te 10.30 uur

door

Martin Bravenboer

geboren op 11 februari 1979, te Epe

Promotor: Prof. dr. S.D. Swierstra
Co-promotor: Dr. E. Visser

This research was financially supported by the Netherlands Organisation for Scientific Research (NWO) project 638.001.201, TraCE: Transparent Configuration Environments.

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

ISBN 978-90-393-4740-9

Preface

This dissertation is all about the fascinating topic of syntax. Such a statement will probably be frowned upon by many computer scientists and programmers. Syntax, in particular textual syntax, seems to be a rather unpopular topic these days. Grammars and parsing are not particularly active research topics, resulting in a lack of innovation in one of the most foundational topics of computer science. In fact, textual syntax is so unpopular that people try to avoid it altogether and attempt to move to development methods without a textual syntax. However, instead of abandoning textual syntax for its problems, I suggest we solve those problems and advance the state of the art in grammars and parsing.

About six years ago I started working with the infrastructure for parsing programming languages in the program transformation system Stratego/XT. Stratego/XT employs the grammar formalism SDF and the scannerless generalized LR parser SGLR, which is the target of the SDF parser generator. The combination of SDF and SGLR is very easy to use. Indeed, I have been using the SGLR parser for years without knowing at all how it works! The essential advantages of scannerless parsing over parsing with a separate scanner are *(1)* lexical disambiguation by context, *(2)* full description of the syntax of a language in a uniform grammar formalism, and *(3)* expressive lexical syntax. These advantages are easy to understand. They were originally motivated by application to minor issues in parsing existing programming languages, such as the lexical ambiguity between subrange types versus floating point literals in Pascal. Although these examples clearly illustrate the advantage of scannerless parsers, the problems with implementing parsers for these languages are not serious enough to make a strong case for scannerless parsing.

The first time Eelco Visser presented his then latest ideas on the use of concrete object syntax [Visser 2002], where the syntax of an object language is embedded in a metalanguage, I was rather skeptical about combining languages in this way. Slowly, however, I realized the full potential of scannerless generalized parsing for combining languages. This thesis is largely about explaining this potential, contributing technical bits and pieces here and there. Indeed, I consider my work to be an exploration in what we could achieve once we are liberated from the limitations of conventional parsing techniques. This explains the title, which is a reference to the beautiful song *Exercises in Free Love* by Freddie Mercury[1] (various puns intended). The Dutch title is a pun as well, referring to the almost military operations I usually set up to finish our papers before a conference deadline.

There is still a lot of work left to do. Reflecting on my work, I feel that one of the main contributions is that I provide a strong motivation for rethinking

[1] Exercises in Free Love is a song performed without lyrics as a predecessor to *Ensueño* (Dream), from the album Barcelona by Montserrat Caballé and Freddie Mercury.

the way we work with syntax and languages, and renewing research into fully automatic parser generation. I wish I discovered earlier that this is what I wanted to attack (and for example had spent less time implementing a Java typechecker), but well, I suppose a thesis project is never finished.

Finally, I would like to emphasize that this area ended up being the subject of my thesis *despite* Eelco Visser's supervision. Since Eelco designed the current revision of SDF and integrated scannerless parsing and generalized LR parsing, this thesis almost looks like a natural continuation of his work. Surprisingly, Eelco continuously encouraged me to move beyond parsing and syntax, but despite his best efforts my work moved closer and closer to parsing issues.

ACKNOWLEDGEMENTS

Supervisors

First of all I thank Eelco Visser, my *'co-promotor'*, for being my supervisor. He was not only my supervisor, but also a mentor and a friend. Almost everything I know about research, I learned from him. Our collaboration was absolutely brilliant. We always worked together on all my publications. The balance between supervising me and contributing to the work we did was perfect. I appreciate how Eelco always gives his students the opportunity to discover things on their own and express their own ideas, as opposed to just telling everything he knows immediately.

I do not remember any formal meeting. We would just end up talking about work every time we wanted to. This must have been difficult sometimes, due to the constant context switches a supervisor has to go through. I am impressed by the capability of some researchers to come up with brilliant remarks just after such a context switch.

Eelco Visser also taught me a lot about publishing strategies. I learned how to target a topic to a general audience, which is most important if you are in a certain niche and your motivation will not be immediately obvious to other researchers. I feel well-equipped to start working as an independent researcher, which is entirely due to Eelco's teaching. Also, Eelco's excellent network gave me the opportunity to meet many interesting researchers, learn how to review papers, and even resulted in finding my future wife. How can one possibly have more impact?

Although I will be moving on to a different country and different universities, I hope we will continue to work together now and then.

I also thank Doaitse Swierstra, my *'promotor'*. Although we didn't meet frequently, I was always struck how well aware he was of our work and my situation. His advice was much appreciated. Also, I have learned that I should probably never buy a cottage for holidays.

Peers

I thank the members of the reading committee Mark van den Brand, Dick Grüne, Johan Jeuring, Kees Koster, and Oege de Moor for reviewing my thesis.

Many anonymous reviewers of conferences provided useful suggestions to improve my work. Also, several people voluntarily provided useful feedback on one or more of my papers: Eelco Dolstra, Jeff Gray, Shan Shan Huang, Merijn de Jonge, Karl Trygve Kalleberg, Emmanuel Onzon, and Martijn Vermaat.

All the work of this thesis has been done in collaboration with other people. I thank the co-authors of my publications for their contributions: Eric Bouwers, Arthur van Dam, Eelco Dolstra, René de Groot, Karl Trygve Kalleberg, Koen Muilwijk, Karina Olmos, Éric Tanter, Rob Vermaas, Jurgen Vinju, and Eelco Visser.

Friends

Arthur van Dam was always happy to help me with LaTeX and thesis design issues. There must be some contribution from him at almost every single page. He also suggested the Dutch translation *exercities* for *exercises*, which is so much more appropriate than the original *oefeningen*! I will miss speed skating together and even more will I miss our road bicycle racing adventures across the country. How am I supposed to deal with the wind without you? I am sorry I beat you at the only serious hill we climbed together (he still claims he just let me win). I should make sure you get your sweet revenge.

Eelco Dolstra provided me the source of his excellent thesis, which saved me a lot of time during the final preparations of my thesis. He was also a very pleasant officemate. I miss his disquisitions and his exclamations of delight when he has found more remarkable *trivia* at Wikipedia. Eelco also carefully reviewed the Dutch abstract of this thesis.

My friend and ex-colleague Rob Vermaas was always annoying and funny in his own special way. TraCE, a grammar for Shell, and Unicode support are only a few of the things he would happily bother me with almost every week. I will miss the excellent pies prepared occasionally by Lizi Vermaas at the occasion of an accepted paper!

This thesis would not be complete without mentioning the IRC channel whose name must not be mentioned. One of the members, Armijn Hemel, deserves special mentioning for his unique approach towards friendship, but I have no clue what to say about him here. Let me just slap him.

Logistics

I pulled quite a few all-nighters (also known as a *bravo*) and I do not think I could have done this without listening to the music of Blank & Jones, the Pet Shop Boys, Moby, and Freddie Mercury. I highly recommend everyone straightforward, gay dance music to get through a long night. I also recommend tea, not coffee. Yunnan green tea works for me.

My loyal computer Logistico could not deal with the finalization of my thesis and the prospect of being abandoned. She died just after submitting my thesis to the reading committee. I owe her every single byte I produced for this thesis.

Family

I thank my parents, my sister, and my brother-in-law for their continuous support. Finally, there is Shan Shan, my lovely fiancée. I do not know what to say here about this miracle. Anything I would say about you would be insufficient. I am looking forward to spending my life with you.

Martin Bravenboer
December 6, 2007
Delft

Contents

Introduction

1

THESIS

The aggregate of modular syntax definition, generalized scannerless parsing techniques, and parse table composition is the technique for principled combination of multiple textual software languages into a single, composite language.

1.1 LANGUAGE COMPOSITION IN PRACTICE

In modern software development the use of multiple software languages to constitute a single application is ubiquitous. For example, a web application could use *(1)* Java for the main application code *(2)* SQL to query a database, *(3)* HTML to present results in a web browser, *(4)* CSS to style those web pages, *(5)* Javascript to make web pages more interactive, *(6)* XSLT to transform an XML data format to HTML, *(7)* an XML vocabulary to respond to web-service requests, *(8)* XPath to obtain input from web-service requests, *(9)* regular expressions to validate user input, *(10)* an XML configuration format for the application server, *(11)* LDAP search filters for user authentication, *(12)* Unix shell commands to invoke external programs, and *(13)* even the application framework and libraries used by the program constitute a language, though expressed in the syntax and structure of Java. Clearly, the principle of choosing the most suitable software language for the work at hand has deeply penetrated modern software development, resulting in a wide range of *domain-specific languages*. Many of these languages interact closely. For example, the Java web application might compose an LDAP or SQL query based on some user input, pass parameters to an XSLT processor, pass a regular expression to a regular expression library, append a temporary filename to a Unix shell command, and generate some XML fragment. The other way around, the XML configuration file might refer to Java classes and methods to instruct the application server and application frameworks how to respond to incoming requests.

> *However, despite the omnipresent use of combinations of languages, the principles and techniques for using languages together are ad-hoc, unfriendly to programmers, and result in a poor level of integration.*

String Embedding The most recognizable example of a lack of integration is the standard practice of combining a host language (e.g. Java) and some guest languages (e.g. SQL or regular expressions) by writing the guest language programs in string literals of the host program. In this approach, the host language compiler has no understanding whatsoever of the languages that are

```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

Figure 1.1 Vulnerable composition of an SQL query, enabling an injection attack

```
SELECT id FROM users
WHERE name = '...'
AND password = '' OR 'x' = 'x'
```

Figure 1.2 Query constructed by the program of Figure 1.1 if the user enters the password `' OR 'x' = 'x`. The where clause is now a tautology.

used in the string literals. This results in a complete lack of static checking of the syntax and semantics of the embedded guest programs, whereas static checking of the host program is often considered to be most valuable. Also, this approach often results in the need for complex escaping of characters with a special meaning in the host language, in particular if combined with related escaping rules of the guest language. Furthermore, using large guest fragments is often awkward if the host language does not support multi-line string literals or *here documents*. Finally, but most importantly, the application might become vulnerable to *injection attacks*, one of the largest classes of security problems, where specially crafted user input results in an unexpected guest program that exposes private information or attacks the functioning of the application.

To prevent injection attacks in the case of dynamically generated sentences, such as SQL queries or shell invocations, the main challenge is to ensure that composing the query with user input is done in a grammatically well-formed way. For example, if the programmer expects the user of the application to provide the content of a literal string in an SQL query, then the user of the application should not be able to craft a query where the user input actually contains other fragments of the guest language, perhaps to construct a tautology (see Figures 1.1 and 1.2). Similar guarantees that composition of guest fragments is done in a grammatically well-formed way are actually already implemented in some metaprogramming systems, but no experiments have been done to bring these techniques to general-purpose programming.

To check the syntax and semantics of the guest language fragments statically, the compiler of the host language needs to be aware of the presence of the guest languages that are used embedded in the program. A guest language plugin to the host compiler could provide the information on how to parse, type-check, and compile the program fragments written in the guest language. To prevent injection attacks, the guest language plugin can provide information to the host compiler on how to construct guest language

```
JTextArea text = new JTextArea(20,40);
JPanel panel = new JPanel(new BorderLayout(12,12));
panel.setBorder(BorderFactory.createEmptyBorder(15, 15, 15, 15));
panel.add(BorderLayout.NORTH, new JLabel("Please enter your message"));
panel.add(BorderLayout.CENTER, new JScrollPane(text));
JPanel south = new JPanel(new BorderLayout(12,12));
JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
buttons.add(new JButton("Ok"));
buttons.add(new JButton("Cancel"));
south.add(BorderLayout.EAST, buttons);
panel.add(BorderLayout.SOUTH, south);
```

Figure 1.3 Sequence of statements for composing a user interface with Swing API methods. The hierarchical structure of the user interface components is not clear.

sentences by composing literal guest language fragments with user input, for example by escaping special characters in the input and checking the lexical structure of the input for characters that are not allowed.

API Syntax A different example of poor integration is the insufficient domain-specific syntax provided by frameworks and libraries. Indeed, most programming languages are designed to allow programmers to implement *semantic* domain abstractions but do not support *syntactic* domain abstraction. Hence, the programmer using the library is forced to use the *generic* lexical syntax and structure of the host language, usually consisting of method invocations, expression that can be nested, and sequences of statements.

> *"An API is like declaring a vocabulary, a domain-specific language adds a grammar which allows to write coherent sentences"* — Martin Fowler

Unfortunately, the generic syntax of a general-purpose language is often not suitable for writing coherent sentences over the vocabulary of an API (see Figure 1.3). To provide a more convenient syntax, the interface to libraries and frameworks is sometimes designed in a way that makes the general-purpose host language look like a domain-specific language (see Figure 1.4). This often involves runtime metaprogramming, heavily exercising the static or dynamic features of the type system of the host language, and frequently results in unusual signatures for functions and methods to provide context to the continuation of the host program. In many cases, methods return a reference to their object (this) only to allow the programmer to write a sequence of method calls. Thus, this approach does not separate the design of the syntactic abstraction from its implementation.

While this approach can lead to remarkable results in programming languages with a generic or liberal syntax, such as Ruby, Haskell, and Lisp, there are always fundamental limitations to the syntax that can be provided to the programmer using the library. Not only are the lexical syntax and the structure of the language limited by the host language, but also the host compiler or interpreter has no understanding of the 'language' defined by the library or framework interface. Hence, it cannot easily provide the programmer with

```
public void testBuysWhenPriceEqualsThreshold() {
  mainframe.expects(once())
    .method("buy").with(eq(QUANTITY))
    .will(returnValue(TICKET));

  auditing.expects(once())
    .method("bought").with(same(TICKET));

  agent.onPriceChange(THRESHOLD);
}
```

Figure 1.4 Testcase implemented using jMock [Freeman & Pryce 2006], where the general-purpose language Java is crafted to look like a domain-specific language. For this test, the mock objects mainframe and auditing are instructed to expect a method invocation

domain-specific error reports about problems with the syntax or semantics of programs. Even languages that are designed to support syntactic domain abstractions, for example using syntax macros, do not allow syntactic extensions without restrictions (Section 2.6.4). Finally, there is usually no separate, formal definition of the syntax of the domain-specific language.

In some cases, very common design patterns and application libraries are lifted to the host language level. For example, LINQ and Cω [Meijer & Schulte 2003a, Meijer & Schulte 2003b] lift the area of data-access to the language C# itself, and the Xtatic language [Xtatic], a follow-up of XDuce [Hosoya & Pierce 2000], extends C# with XML specific support. However, the growth of a language with more domain-specific constructs is limited by the general application area of the language. Therefore, these is a need for the introduction of *concrete syntax* for domain abstractions as a *plugin* to the host language, but without restrictions on the embedding and assimilation of the syntax of the domain-specific language in the host program. The plugin defines *(1)* the concrete syntax for using a specific library, *(2)* in which way the syntax is embedded in the host language, *(3)* how the concrete syntax translates into host language code using the library (a program transformation called *assimilation*), and *(4)* could optionally include typing rules for checking the combination of the host and the guest language.

Effort for Specific Combinations In some cases, specific combinations of languages are crafted carefully because that combination is particularly useful, thus making it acceptable to put more effort in providing a well-engineered combination of these languages to the programmer. We use the term *language conglomerate* for programming languages that are in fact mixtures of various sublanguages. For example, AspectJ has been carefully designed to extend Java with syntax for aspects, advice, and pointcuts; the SQL-92 standard [ISO 1992] defines an embedding of SQL in host languages such as C; and in research the Meta-AspectJ (MAJ) [Zook et al. 2004] code generation tool has integrated the syntax of a particular object language, i.e. AspectJ, in a very user-friendly way in the host language Java.

```
class Foo {
  pointcut foo() : call(void *0.Ef());
}
$ ajc Foo.java
Foo.java:2 [error] Invalid float literal number
pointcut foo() : call(void *0.Ef());
                            ^
```

```
class Foo {
  pointcut foo() : call(boolean *.if*());
}
$ ajc Foo.java
Foo.java:2 [error] Syntax error on token "if", invalid allowable
                   token in pointcut or type pattern
```

Figure 1.5 Tokenization bugs in the official AspectJ compiler

However, these languages conglomerates are not a *principled* combination of their sublanguages. That is, the complete syntax of language conglomerates is usually not formally defined, the grammar is not based on separate syntax definitions for the sublanguages, and the applied parsing techniques are ad-hoc, to work around issues in parsing language conglomerates. The use of ad-hoc parsing methods results in compilers that have surprising bugs. Also, the limitations of the parsing techniques influence the design of the languages, which leads to syntax unfriendly to the programmer, e.g. unnecessarily reserving keywords globally. Due to the lack of principled techniques for combining languages, there is a substantial effort for engineering a specific combination of sublanguages into a language conglomerate. Worse, this effort can often not be reused for developing a related conglomerate with a slightly different configuration of sublanguages.

Hence, crafting specific conglomerates does not scale to the full vision of guest languages as plugins to the host compiler. Third parties (i.e. not the developer of the host compiler) should be able to offer plugins and the end-programmer should be able to compose such plugins arbitrarily without doing any difficult metaprogramming or recompilation of the host compiler[1]. In particular, for applications in code generation and query embedding, it is not an option to spend considerable effort to create an implementation for each element of the cross-product of host and guest languages {Java, C#, PHP, Perl, ...} × {SQL, JDOQL, HQL, EJBQL, OQL, LDAP, XML, HTML, XPath, XQuery, Shell, Java, C#, PHP, Perl, ...}. Moreover, it is not acceptable to require implementation effort for every specific *subset* of extensions to the host language. Therefore, while specific combinations are interesting for discovering the requirements for embedded languages, there is a need for guest languages as true plugins to the host language, rather than engineering specific combinations of languages again and again.

[1]Though we want to compose guest languages arbitrarily, some combinations of guest languages inherently do not work together, comparable to incompatible combinations of libraries, e.g. the Swing and SWT user interface toolkits.

1.2 OUTLINE

In this dissertation, we work towards a set of techniques for introducing guest languages in a host language without restrictions on the lexical and context-free syntax of the languages involved. In general, we discuss extensively how the syntax of language conglomerates can formally be defined and parsed. We work towards a *principled* and *generic* solution to language extension by studying the applicability of modular syntax definition, scannerless parsing, generalized parsing algorithms, and program transformations. Where necessary, we extend and improve the existing work in these areas. This dissertation studies a series of compelling state of the art applications of generalized parsing techniques and program transformation, namely

- parsing and disambiguation of syntax embeddings for adding domain-specific syntactic abstractions to a general-purpose language [Bravenboer & Visser 2004 (Chapter 2), Bravenboer et al. 2006],
- concrete object syntax for metaprogramming [Batory et al. 1998, Visser 2002, Bravenboer et al. 2005 (Chapter 3)],
- replacing the fragile and vulnerable practice of embedding languages in string literals [Bravenboer et al. 2007 (Chapter 4)],
- and formal definition of the syntax of language conglomerates, e.g. AspectJ [Bravenboer et al. 2006 (Chapter 5)].

Concerning parsing of language conglomerates, these studies have resulted in considerable more insight in the applications of scannerless parsing and the generalized LR parsing algorithm. Also, the need for programmer-friendly syntax has resulted in novel disambiguation methods for embedded guest languages, relieving end-programmers from the need to indicate the syntactic category of guest language fragments.

The applications we have studied require syntax embeddings to act as true plugins, which can be selected and combined by the end-programmer. The need for a more dynamic configuration of the combination of host and guest languages has resulted in the development of parse table components. Finally, since all this work depends heavily on syntax definitions, solid grammar engineering practices are most important. We improve these practices by introducing a method for recovering and comparing precedence rules of operators from grammars.

Chapters

In Chapter 2 we present the MetaBorg method for introducing concrete syntax for object-oriented libraries and frameworks. This method consists of the embedding of a domain-specific syntax in the general-purpose programming language and an unrestricted program transformation to translate the extended language to the basic host language. The program transformation, called an assimilation, replaces the domain-specific syntax with uses of the

library, which still provides the semantic domain abstraction. This chapter discusses in detail why modular syntax definition and scannerless parsing is important in this application area.

In Chapter 3 we describe an approach to resolving ambiguities that arise when a guest language is embedded in a host language. In particular for metaprogramming, where small program fragments of an object language are quoted, ambiguities are ubiquitous. The approach presented in this chapter makes the existing work on metaprogramming with concrete object syntax [Visser 2002] more programmer-friendly without requiring additional implementation effort for embedding a specific object language. In this chapter, we propose to use a generic extension of the type system of the host language to disambiguate quoted code fragments, thus allowing a more lightweight syntax. As such, it is a generalization of the same programmer-friendly concrete object syntax approach of Meta-AspectJ [Zook et al. 2004].

In Chapter 4 we present the StringBorg method for preventing injection attacks by embedding the syntax of guest languages in a host language. The embedded syntax is translated into invocations of a generated library that guarantees that the sentences are constructed in a grammatically well-formed way. The contribution of this *application* is that it prevents injection attacks *by construction*, rather than *detecting* injection attacks at runtime. To make the syntax of the embedded guest languages as programmer-friendly as possible, we introduce a lightweight disambiguation technique for concrete object syntax. This approach does not depend on the type system of the host language, hence it can be applied to dynamically typed languages, such as PHP. StringBorg is the culmination of genericity: guest language plugins can not only be combined arbitrarily, but can also be used for any host language. For example, a plugin for SQL can be used for PHP as well as Java.

In Chapter 5 we extensively discuss the issues with the parsers of the two main compilers for AspectJ. In particular, we discuss the use of lexical states to tokenize language conglomerates. To improve on the situation that the full syntax of AspectJ is ill-defined, we present a formal definition of the complete syntax of AspectJ. From this formal syntax definition a scannerless generalized LR parser can be generated. To handle the various AspectJ keyword policies, we introduce *grammar mixins*, which are in general useful if the same sublanguage can appear in multiple contexts of a conglomerate.

In Chapter 6 we present an algorithm for composing separately compiled parse table components in a very efficient way just before parsing source files written using a combination of languages corresponding to these components. The parse table composition algorithm allows the syntax of guest language to be deployed as truly separate plugins, which can be combined with the host language and other guest languages efficiently. This solves the problem of having to generate a compound parser from scratch for every particular combination of language extensions. The generation of parse tables for language combinations is expensive, which is a particular problem when the composition configuration (i.e. the set of language extensions) is not fixed.

In Chapter 7 we take a side-path into grammar engineering. All our applications and techniques depend heavily on having high-quality grammars available for the guest and host languages. These grammars have to be defined in the same modular syntax definition formalism, targeting a scannerless generalized LR parser. To make the vision of this dissertation reality, we need solid practices for developing, reverse engineering, and maintaining grammars. In this chapter we solve the particular problem of recovering precedence rules of operators from legacy YACC or SDF grammars and comparing the rules of different grammars to each other. This work enabled the development of a high quality PHP grammar, which we have used for our case studies.

1.3 ORIGIN OF CHAPTERS

Except for our latest work (Chapter 6), all chapters are directly based on peer-reviewed publications at programming languages and software engineering conferences and workshops. Since all chapters are directly based on papers that have been published independently, they can also be read independent of each other. While all papers have distinct core contributions, there is some redundancy in the introduction of background material, motivation, and examples. This redundancy has not been eliminated to make it possible to read the extended and revised papers independently.

- Chapter 2 is a slightly updated version of the OOPSLA 2004 paper *Concrete Syntax for Objects – Domain-Specific Language Embedding and Assimilation without Restrictions.* [Bravenboer & Visser 2004]
- Chapter 3 is an updated and extended version of the GPCE 2005 paper *Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax* [Bravenboer et al. 2005].
- Chapter 4 is an extended version of the GPCE 2007 paper *Preventing Injection Attacks with Syntax Embedding – A Host and Guest Language Independent Approach* [Bravenboer et al. 2007].
- Chapter 5 is a revision of the OOPSLA 2006 paper *Declarative, Formal, and Extensible Syntax Definition for AspectJ – A Case for Scannerless Generalized LR Parsing* [Bravenboer et al. 2006].
- Chapter 6, our latest work, is an unpublished paper *Parse Table Composition – Separate Compilation and Binary Extensibility of Grammars*
- Chapter 7 is an extended version of the LDTA 2007 paper *Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking* [Bouwers et al. 2007].

Concrete Syntax for Objects

2

ABSTRACT

Application programmer's interfaces give access to domain knowledge encapsulated in class libraries without providing the appropriate notation for expressing domain composition. Since object-oriented languages are designed for extensibility and reuse, the language constructs are often sufficient for expressing domain abstractions at the semantic level. However, they do not provide the right abstractions at the syntactic level. In this chapter we describe MetaBorg, a method for providing *concrete syntax* for domain abstractions to application programmers. The method consists of *embedding* domain-specific languages in a general purpose host language and *assimilating* the embedded domain code into the surrounding host code. Instead of extending the implementation of the host language, the assimilation phase implements domain abstractions in terms of existing APIs leaving the host language undisturbed. Indeed, MetaBorg can be considered a method for promoting APIs to the language level. The method is supported by proven and available technology, i.e. the syntax definition formalism SDF and the program transformation language and toolset Stratego/XT. We illustrate the method with applications in three domains: code generation, XML generation, and user interface construction.

2.1 INTRODUCTION

Class libraries encapsulate knowledge about the domain for which the library is written. The application programmer's interface to a library is the means for programmers to access that knowledge. However, the generic language of method invocation provided by object-oriented languages does often not provide the right notation for expressing domain-specific composition. General purpose languages, particularly object-oriented languages, are designed for extensibility and reuse. That is, language concepts such as objects, interfaces, inheritance, and polymorphism support the construction of class hierarchies with reusable implementations that can easily be extended with variants. Thus, OO languages provide the flexibility to develop and evolve APIs according to growing insight into a domain.

Although these facilities are often sufficient for expressing domain abstractions at the semantic level, they do not provide the right abstractions at the syntactic level. This is obvious when considering the domain of arithmetic or logical operations. Most modern languages provide infix operators using the well-known notation from mathematics. Programmers complain when they have to program in a language where arithmetic operations are made

available in the same syntax as other procedures. Consider writing `e1 + e2` as `add(e1, e2)` or even `x := e1; x.add(e2)`. However, when programming in other domains such as code generation, document processing, or graphical user interface construction, programmers are forced to express their designs using the generic notation of method invocation rather than a more appropriate domain notation. Thus programmers have to write code such as

```
JPanel panel = new JPanel(new BorderLayout(12,12));
panel.setBorder(BorderFactory.createEmptyBorder(15,15,15,15));
```

in order to construct a user interface, rather than using a more compositional syntax reflecting the nice hierarchical structure of user interface components in the Swing library. Building in syntactic support for such domains in a general purpose language is not feasible, however, because of the different speeds at which languages and domain abstractions develop. A language should strive for stability, while libraries can be more volatile.

In this chapter we describe MetaBorg[1], a method for providing *concrete syntax* for domain abstractions to application programmers. The method consists of *embedding* domain-specific languages in a general purpose host language and *assimilating* the embedded domain code into the surrounding host code. Instead of extending the implementation of the host language, the assimilation phase implements domain abstractions in terms of existing APIs leaving the host language undisturbed. Indeed, MetaBorg can be considered a method for promoting APIs to the language level [Mernik et al. 2005].

For example, to improve the construction of user-interfaces in Java, we have designed a little *Swing user interface Language* (Swul) that makes the compositional structure of the Swing components visible in application programs. Using our method we have embedded this language in Java, such that it is directly available to application programmers. They can now write *within their Java programs* expressions such as

```
JPanel panel = panel of border layout {
  north = label "Welcome"

  center = scrollpane of
    input : textarea {
      rows    = 20
      columns = 40
    }

  south = panel of border layout {
    east = button for ExitAction
  }
};
```

in order to implement a user interface consisting of a panel with border layout, containing a label, a text area, and another panel with a button. Such a program is *assimilated* into the surrounding Java code by translating it to the sequence of Swing method calls that one would write by hand.

[1] MetaBorg provides generic technology for allowing a host language (collective) to incorporate and assimilate external domains (cultures) in order to strengthen itself. The ease of implementing embeddings makes resistance futile.

Our work stands in a long line of approaches to add syntactic extensibility to programming languages [Leavenworth 1966, Weise & Crew 1993, Cardelli et al. 1994, Shalit 1996, Batory et al. 1998, Brabrand & Schwartzbach 2002, Begel & Graham 2004]. Although our work has many commonalities with other approaches, it is distinguished by its generality, i.e. the lack of restrictions on either the syntax or the semantics of embedding and assimilation. In addition, implementation of embeddings is high-level and concise; definition and embedding of Swul required only 100 lines of syntax definition and 170 lines of assimilation rules. Our method has the following characteristics:

Syntactic Embedded code fragments are checked syntactically at compile-time. This is in contrast with approaches to compose program fragments using string literals.

No restrictions on syntax definition Our maxim is that it should be possible to design a notation that is fitting for the domain without placing artificial restrictions on the syntax to be used. This means that both lexical and context-free syntax should be definable. Furthermore, all aspects of the embedding, including quotation symbols, if any, should be adaptable. Only the full class of context-free grammars allows such natural syntax definition. This is in contrast to languages with user-definable operators [Hudak 1996], overloading of (a fixed set of) operators, syntax macros [Leavenworth 1966], or grammar formalisms supporting only a subset, such as LL or LALR, of the context-free grammars [Cardelli et al. 1994, Batory et al. 1998]. The only proviso we make is that host and embedded language have a *context-free* syntax.

Not restricted to a single host language The method is not specific to a particular host language [Batory et al. 1998], but can be used to embed any language in any host language.

Interaction with host language Embedded code fragments should be able to refer to artifacts in the host program and vice versa. This is in contrast to approaches based on a separate domain-specific language from which code is generated [Smaragdakis & Batory 2000, Mernik et al. 2005].

Combination of extensions It should be possible to combine multiple domain notations, in contrast to hard-wired language extensions.

No restrictions on assimilation The translation of embedded fragments to the host language should not be limited to a simple homomorphism or other fixed translation order [Brabrand & Schwartzbach 2002], but should allow use of context-sensitive information, global analysis, and multi-stage transformations.

As a consequence of these characteristics we do *not* require that language extensions are implemented *within* programs in the host language [Cardelli et al. 1994], since such approaches lead to restrictions in many of the areas

mentioned above. We also do *not* expect language design and implementation skills from the average application programmer. Instead we opt for a separation of roles between the metaprogrammer defining the language embedding and assimilation, and the application programmer using a domain notation. However, our techniques are sufficiently high-level that a knowledgeable programmer can use them to create new embeddings. That is, the method is based on SDF2 [Visser 1997b, van den Brand et al. 2002], a syntax definition formalism used to define embeddings, and Stratego/XT [Visser 2004, de Jonge et al. 2001] a language and toolset for program transformation used to implement assimilation. These tools are mature and freely available from [SDF Website] and [Stratego Website], respectively. The applications of MetaBorg developed in this chapter are available at `http://www.metaborg.org`.

Organization In Section 2.2 we examine the practice of object-oriented programming in three domains: code generation, document generation, and graphical user interface construction. For each of these domains we show how the readability of programs improves dramatically by employing domain-specific concrete syntax. In particular, we show how to generate Java programs and XML documents in Java. Furthermore, we describe the domain-specific *Swing user interface Language* (Swul), which provides a nice compositional language for user interface composition in Java. In Section 2.3 we explain how concrete syntax embeddings and the corresponding assimilations are realized using the MetaBorg method and illustrate this by the implementation of embeddings for the three applications from Section 2.2. In Section 2.4 we give an overview of the syntax definition formalism SDF2. In Section 2.6 we discuss the relation with competing approaches such as user-definable operators, syntax macros, application generators, and domain-specific languages. We discuss future work in Section 2.7, and conclude in Section 2.8.

2.2 CONCRETE SYNTAX FOR OBJECTS

In this section we examine three application domains that suffer from the misalignment between language notation and the domain: code generation, XML document generation, and graphical user interface construction. For each of these domains we discuss the methods that are used for programming in these domains using an object-oriented language and we show how our concrete syntax method dramatically improves the readability and writability of applications in these domains. For all examples in this chapter we use Java as the host language, but the techniques are equally well applicable to other languages.

2.2.1 *Code Generation*

A *code generator* automates the production of boilerplate code by translating a compact high-level specification of a problem into full blown code. Typical applications include the generation of data types for the representation of *abstract syntax trees*, the generation of *XML data binding* [Bourret 2007] code for convert-

ing XML to a specific data type, and the generation of *object-relational binding* code for connecting an object-oriented program to a database system. Numerous tools are available for these purposes; LLBLGen [Bouma] is an object-relational binding generator; ApiGen [de Jong & Olivier 2004] and JTB [Tao et al.] are abstract syntax tree generators, JAXB [JAXB Website] and Castor [Castor Website] are XML data binding tools, to name but a few.

The implementation of a code generator requires an internal representation of program code and an interface for accessing this representation in order to compose and transform code fragments. Ideally, generators use a structured representation of programs, i.e. a data structure to represent abstract syntax trees. Such a representation makes it easy to compose, analyze, and transform fragments. For example, the XML data binding tool JAXB uses a full abstract syntax tree in its code generator. However, in practice, many generators are string-based, meaning that code is generated by directly printing strings to a file, or by representing fragments as strings and composing those. For example and ironically, the abstract syntax tree generators ApiGen [de Jong & Olivier 2004] and JTB [Tao et al.] are text-based code generators. Castor [Castor Website] is an example of a hybrid approach which uses a combination of an abstract syntax tree for the global structure and text for method bodies.

Neither implementations using a data structure, nor using string literals are satisfactory. The advantage of using string literals or text templates is that one can use *concrete syntax*, i.e. the fragments are readable as program code, and it is trivial to implement in a general purpose language. The approach is illustrated by the Castor example in Figure 2.1, which builds up a string representation of a program fragment. However, the disadvantages far outweigh the advantages. Escaping to the host language in order to insert a fragment of code computed elsewhere is cumbersome. The syntax of the generated code is not checked. No further manipulation of the code is possible. And runtime overhead is incurred for parsing, analysis, compilation or interpretation if code is to be executed.

The advantage of using a data structure is that the generated code is structured and is amenable to further processing. Type correctness of the generator often entails syntactic correctness of the generated program. It is also easy to implement in a general purpose object-oriented language. The approach requires the creation of a class hierarchy which can be substantial for a real language. Code generators such as ApiGen and JTB can be used for that purpose. However, composition of code fragments is done via calls to the code API, leading to very verbose metaprograms. It is usually hard to understand the structure of the generated code when inspecting such metaprograms.

The MetaBorg method combines the best of both worlds. Code fragments are written using the concrete syntax of the programming language, but the implementation is based on an API for code representation. This is illustrated in Figure 2.2 with a fragment of a Java program generating a Java program, corresponding to the example in Figure 2.1. The generator uses ATerms [van den Brand et al. 2000] for the representation of generated code. Instead of using constructors from the ATerm class hierarchy to create a code fragment,

```
String x = "propertyChangeListeners";
jsc.add("if (");
jsc.append(x);
jsc.append(" == null) return;");
jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, f, v1, v2);");
jsc.add("");
jsc.add("for (int i = 0; i < ");
jsc.append(x);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(x);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Figure 2.1 Code generation in Castor.

```
ATerm x = id |[ propertyChangeListeners ]|;

ATerm stm = bstm |[ {
    if(x == null) return;
    PropertyChangeEvent event = new PropertyChangeEvent(this, f, v1, v1);
    for(int c=0; c < x.size(); c++) {
      ((PropertyChangeListener) x.elementAt(c)).propertyChange(event);
    }
  }
]|;
```

Figure 2.2 Code generation with concrete syntax.

it is written as a regular piece of Java code. The code fragments are distinguished from the surrounding code by the delimiters |[and]|. The delimiters are not fixed in MetaBorg and can even be left out when appropriate. The `bstm` and `id` tags are used to indicate that the fragments are of syntactic category *statement* and *identifier* respectively. A fragment does not need to be closed, but can incorporate code fragments generated elsewhere. For example, in Figure 2.2 the identifier assigned to variable *x* is used in the fragment assigned to the variable *stm*.

Metaprograms with embedded object-program fragments are translated to pure Java programs in which code construction is expressed directly in terms of calls to the code representation API. This tool also guarantees that the code fragments are syntactically correct. The type system of the host language and a properly defined underlying API will then guarantee that *compositions* are syntactically correct as well.

Note that MetaBorg is not restricted to Java in Java. The same method can be used to embed other languages in Java (e.g. to generate C# code), or to use a different host language (e.g. to generate Java code with a C# program). The realization of these embeddings will be discussed in Section 2.3.

```
out.startDocument();
out.startElement("", "html", "html", noAttrs);
out.startElement("", "body", "body", noAttrs);
out.startElement("", "p", "p", noAttrs);
out.characters(text.toCharArray(), 0, text.length());
out.endElement("", "p", "p");
out.endElement("", "body", "body");
out.endElement("", "html", "html");
out.endDocument();
```

Figure 2.3 Composition of an XML document in Cocoon.

```
out.write doc %>
    <html>
      <body>
        <p><% text :: cdata %></p>
      </body>
    </html>
  <%;
```

Figure 2.4 Composition of an XML document with concrete syntax with underlying SAX ContentHandler invocations.

2.2.2 *XML Document Generation*

XML is used on a large scale for the exchange of data between programs. This requires programs to read and write XML documents and to convert internal data to XML and back. Applications that generate XML documents by filling in templates suffer from the lack of support for XML syntax in general purpose programming languages. The problems are similar to that in code generation. Text-based solutions cannot guarantee that the produced text is well-formed XML. Typical examples are server-side scripting languages such as JSP and ASP .NET, which support the embedding of a programming language in XML or HTML. Further manipulation of the document after generation is impossible. A typical example of post generation transformation is the addition of statistics to a generated web page. Many web page generators just put this information outside the HTML tags, which is of course invalid, but is accepted by web browsers if the page does not claim to be well-formed XML.

Other solutions are based on data structures. In libraries such as W3C DOM, JDOM, and XOM, documents are represented by instances of classes corresponding to the generic structure of XML, e.g. `Document`, `Element`, and `Attribute`. Thus document construction is achieved with the constructors of these classes. Alternatively, construction can be achieved with events emitted to a SAX `ContentHandler`, which can be an XML serializer or a DOM constructor, as illustrated in Figure 2.3 with a code fragment from Cocoon [Cocoon Website]. By using these APIs the code is guaranteed to produce well-formed XML as long as the library does its job properly. However, these solutions result in sequences of method invocations that are hard to read.

The MetaBorg solution is the same as in the case of code generation. There

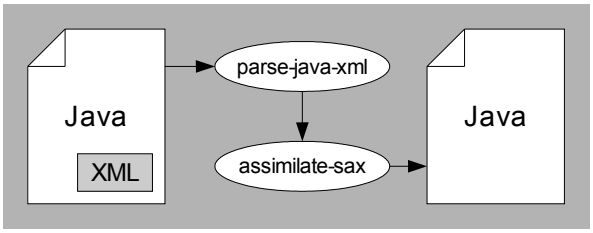


Figure 2.5 A program with embedded concrete syntax (e.g. for XML) is translated to a pure Java program.

is a good notation for this domain, i.e. XML itself. With the MetaBorg method we make this notation available to the application programmer, while keeping the properties of APIs for structured representation of documents. Figure 2.4 shows a statement in a Java program with an embedded XML document. The symbols `%>` and `<%` are used to delimit the embedded XML from the surrounding code. Within the document is a reference to a document fragment defined earlier in the program, using the `<%` and `%>` antiquotation delimiters. The assimilation process (Figure 2.5) transforms a Java program with embedded XML to a pure Java program with calls to a SAX `ContentHandler`.

The big difference between the embedding of XML and the embedding of a programming language is that the lexical structure of XML is completely different from that of, say, Java. This makes the parsing problem more complicated.

2.2.3 *Graphical User Interface Construction*

In the cases of code and XML generation a domain notation is readily available to improve the readability of application programs. However, there are many other domains with APIs representing a 'language' in the sense of a coherent set of concepts and composition facilities, but without a concrete notation. Programming in these domains can also be improved by employing domain-specific notation, which should then first be designed. Consider for example the construction of graphical user interfaces in Java.

Graphical user interfaces can be generated with a visual tool or can be written by hand. A handwritten user interface typically instantiates GUI components such as buttons and textfields and arranges these components in panels by using fixed positioning or a layout manager that allows the user interface to adapt itself to changes in the size of the window. Despite catalogs of design patterns for developing graphical user interfaces, this code is still one of the most unwieldy parts of a program (Figure 2.6). A handwritten user interface takes quite a few lines of code and the resulting code is difficult to understand and maintain.

There are a few proposals for languages specifically geared towards user interfaces, e.g. Mozilla's XUL, W3C's XForms and Microsoft's XAML, but these

```
JTextArea text = new JTextArea(20,40);
JPanel panel = new JPanel(new BorderLayout(12,12));
panel.setBorder(BorderFactory.createEmptyBorder(15, 15, 15, 15));
panel.add(BorderLayout.NORTH, new JLabel("Please enter your message"));
panel.add(BorderLayout.CENTER, new JScrollPane(text));
JPanel south = new JPanel(new BorderLayout(12,12));
JPanel buttons = new JPanel(new GridLayout(1, 2, 12, 12));
buttons.add(new JButton("Ok"));
buttons.add(new JButton("Cancel"));
south.add(BorderLayout.EAST, buttons);
panel.add(BorderLayout.SOUTH, south);
```

Figure 2.6 Sequence of statements for composing a user interface with Swing API methods

```
JPanel panel = panel of border layout {
  hgap = 12
  vgap = 12
  north = label "Please enter your message"
  center = scrollpane of textarea {
    rows    = 20
    columns = 40
  }
  south = panel of border layout {
    east = panel of grid layout {
      hgap = 12
      vgap = 12
      row = {
        button "Ok"
        button "Cancel"
      }
    }
  }
};
```

Figure 2.7 SWUL implementation for composing the same user interface as Figure 2.6

solutions do not integrate well with the host language. They abstract from a GUI toolkit, which is not always an advantage, and restrict the way the GUI can interact with the host language. XUL clones for Java, such as SwiX^{ml} and Luxor, use reflection and a convention for the location of methods (or `Action` fields) to invoke code in the host language. Although this works reasonably well, it limits the way the host language is able to interact with the GUI code. For example, a model-view-controller design, where the GUI components are directly updated from their (Swing) models, is not possible with these toolkits. Furthermore, the XML formatted GUI specifications are interpreted, not checked statically for internal consistency or correct interaction with code in the host language.

Using the MetaBorg method we have developed a solution that provides domain-specific notation for the construction of user interfaces and integrates well with the rest of the program written in the host language. Expressions in the *Swing User Interface Language* (SWUL) correspond to components of a Swing

user interface. The language encourages the compositional construction of complex components, in contrast to the assembly language spaghetti style used with direct calls to Swing. Thus, subcomponents are subexpressions, and do not have to be added afterwards. Attributes of components are named, avoiding the need to continuously look up the order of method parameters. The declaration of the layout style is combined with the instantiation of the subcomponents. Embedded in Java, the Swul language can be used directly to program user interfaces. Swul expressions can refer to elements such as variables and methods from other parts of the program. For example, host language names and expressions can be used to set the value of a layout attribute or to pass an event handler.

The example in Figure 2.7 illustrates the use of Swul, creating in a single Java assignment (be it multiline) the same user interface implemented directly using Swing methods in Figure 2.6. Although the Swul solution uses more lines, the difference in number of non-space characters (507 vs 243) indicates that the Swul solution is significantly more concise.

2.2.4 *Other Applications*

We have discussed three application domains in which concrete domain notation improves programs. The same approach can be applied to many other domains, including embedded query languages such as XPath and SQL. Unfortunately, queries in these languages are usually embedded in string literals. They are often even composed at runtime by concatenating strings. In this case the SQL statements and XPath queries cannot be checked syntactically at compile time. This might result in runtime errors or, even worse, security problems. If a value from the host language is embedded in an SQL query by string concatenation, then there is no guarantee that the string actually is of the syntactic category expected at this point in the SQL statement. This results in a security issue that is rather easy to exploit by entering SQL constructs where plain strings are expected. To prevent this, the embedded string has to be escaped. A missing escape will immediately pose a security threat. This is a general problem of runtime composition of code fragments by string concatenation. XML applications that use string concatenation might be a security risk as well. In such applications proper escaping is required at many different places in the code. This task can be automated by applying our language embedding tools. We discuss the StringBorg method for preventing injection attacks using syntax embedding in Chapter 4.

Another example of a little language that is often embedded in string literals is the language of regular expressions used for pattern matching of character sequences. A regular expression is encoded in a string literal which is interpreted or compiled at runtime to a matcher. The clarity of a regular expression is reduced by requiring special characters to be escaped. C# reduces the number of characters that have to be escaped by adding verbatim strings to the language, where only double quotes have to be escaped. Just as in all string literal based embeddings the regular expression is not checked

syntactically at compile-time.

The difference with the other examples is that the libraries operating on these languages usually only accept textual input. This makes a structured representation of the query more annoying than advantageous. Yet, the MetaBorg method can still be used by assimilating the embedded code fragments to the corresponding string operations. This will make the embedding of meta values safer and easier, and the embedding will guarantee the syntactic correctness of the embedded expressions. To this end the MetaBorg tools support the representation of embedded fragments in a full *parse tree*, which can be yielded to a string.

2.3 REALIZING CONCRETE SYNTAX

Introducing domain-specific notation in a host language requires (1) an *embedding* of the domain-specific language in the host language and (2) *assimilation* of the embedded domain fragments into the surrounding host code. In this section we describe the method of embedding and assimilation from the point of view of the metaprogrammer creating the embedding. We illustrate the method with several examples. In the next sections the technology behind the method will be discussed.

2.3.1 *Embedding and Assimilation*

The architecture of the MetaBorg method is illustrated by the diagram in Figure 2.8. The embedding of domain notation in a host language requires a syntax definition for the host language, a syntax definition for the embedded language, and a syntax definition for the combination of the two languages, embedding the latter in the former. A parser then uses this combined syntax definition to parse programs in the extended language. Next, an assimilator applies a set of rules to assimilate the embedded domain fragments, reducing the program to the pure host language.

The diagram also illustrates the roles of programmer, metaprogrammer, and MetaBorg tooling. An application programmer using the domain extension uses the combination of parser and assimilator for the extension as a single tool, which could even be integrated with the compiler. Thus, programming in the extended language is no different than programming in the host language, except for the additional expressivity that is available. The metaprogrammer implementing a domain extension needs to provide the syntax definitions for host and domain language, the syntax for the embedding and the assimilation rules. Generally, the syntax definition for the host language can be reused from a library of syntax definitions. The syntax definition of the domain language can be reused as well, if the domain language is an existing language that is already used on its own. If the host API is a shared target between several domain notations, then a set of generic assimilation rules can be used. This is typically the case for metaprogramming domains where a standard API for abstract syntax trees is used. Finally, syntax definitions and

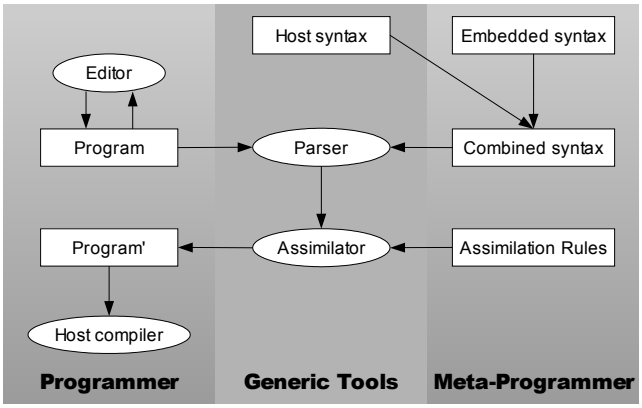


Figure 2.8 Architecture of embedding and assimilation framework.

```
module Java-Tuple
imports Generic-Java
exports
  context-free syntax
    "(" Expr "," Expr ")" -> Expr {cons("NewTuple")}
    "(" Type "," Type ")" -> Type {cons("TupleType")}
```

Figure 2.9 Syntax definition of Java with tuples.

```
module Java-Tuple-Assimilate
imports Generic-Java
rules
  AssimilateTuple :
    expr |[ (e1, e2) ]| -> expr |[ Tuple.construct(e1, e2) ]|

  AssimilateTuple :
    type |[ (t1, t2) ]| -> type |[ Tuple<t1, t2> ]|
```

Figure 2.10 Rewrite rules for assimilation of tuples in Java

assimilation rules are processed by the generic tools provided in the Stratego/XT toolset [Stratego Website, Visser 2004]; MetaBorg can be considered as a particular pattern of usage of these tools.

To make these concepts more concrete we will illustrate the method with several examples. Before diving into the interesting aspects, we look at a very simple example, i.e. the extension of Java 1.5 with concrete syntax for tuples.

Tuples are instances of a class `Tuple<`*A*`, `*B*`>`, which is parameterized with the type of the first and second item of the tuple. We define an extension of Java providing the notation `(x,y)` for the construction and declaration of tuples. Thus, `(Integer, String)` is a type expression and corresponds to the type `Tuple<Integer, String>`. The expression `(1, "foo")` is an instance of `Tuple<Integer, String>`, where the first item in the tuple is the integer 1 and

the second is the String "foo".

To realize this extension all we need is the syntax definition in Figure 2.9 and the assimilation rules in Figure 2.10. The syntax definition in Figure 2.9 defines the language Java-Tuple, which extends Java with syntax for tuple expressions and tuple types. The extension of the Java language is achieved by a simple import of the syntax definition of Java. Thus, it is not necessary to create a copy of that syntax definition and add the rules for tuples to it. The Java-Tuple syntax module is written in SDF, which is the syntax definition formalism employed by MetaBorg. A syntax definition mainly consists of *productions* of the form s_1 ... s_n -> s_0 {cons("c")}, declaring that a phrase of type s_0 can be formed by concatenating phrases of types s_1 ... s_n. The *constructor* c is a name for the production that is used in the construction of abstract syntax trees. The features of SDF that enable concise modular syntax definition will be explained in Section 2.4.

Assimilation of the language extension is achieved by translating the new language constructs to the base language. For example, the Java-Tuple declaration

```
(Integer, String) t = (1, "Hello world!");
```

is translated into the Java declaration

```
Tuple<Integer, String> t = Tuple.construct(1, "Hello world!");
```

This translation is achieved by the *rewrite rules* in Figure 2.10. The assimilation into the host language is implemented in the Stratego program transformation language. Rewrite rules play a central role in Stratego. A labeled rewrite rule L: p_1 -> p_2 rewrites a program fragment matching the pattern p_1 to the program fragment p_2, where the *metavariables* in p_2 are instantiated with the corresponding fragments found when matching p_1. In the first rewrite rule of Figure 2.10, *e1* and *e2* are metavariables denoting Java expressions. In the second, *t1* and *t2* are metavariables denoting Java types. Thus, the first rule replaces a tuple expression (*e1*, *e2*) with a call to the construct method of the Tuple class, passing the expressions *e1* and *e2* [2].

Note that the Stratego rewrite rules use concrete syntax as well. That is, the rules are transformations on Java programs and thus use the syntax of Java to indicate code fragments. However, the rules are *implemented* as transformations on abstract syntax trees. This is achieved by the same method as applied to Java programs [Visser 2002].

2.3.2 *Java with Swul*

Next we examine a serious example, i.e. the embedding of Swul, the *Swing user interface Language*, into Java. Plain Java code using the Swing API is a kind of assembly language in which intermediate results are bound to variables to

[2] We use the static factory method construct rather than the constructor of the Tuple class to work around the lack of inference of type arguments for constructors in Java 5 and 6. See also http://gafter.blogspot.com/2007/07/constructor-type-inference.html

```
module Java-Swul
imports Java-Prefixed Swul-Prefixed
exports
  context-free syntax
    SwulComponent -> JavaExpr {cons("ToExpr")}
    SwulLayout    -> JavaExpr {cons("ToExpr")}

    JavaExpr -> SwulBorder    {cons("FromExpr")}
    JavaExpr -> SwulComponent {cons("FromExpr")}
```

Figure 2.11 Syntax definition of Java with Swul; concrete syntax for user interface construction with Swing

be used elsewhere. This is illustrated in Figure 2.7. We have custom designed Swul in order to provide a compositional syntax for the construction of user interfaces with Swing in Java. Swul is just a more attractive notation for the same code. The assimilation of embedded Swul produces the sequence of statements that one would normally write by hand.

Embedding

Swul is defined as a separate language with its own syntax definition in SDF module `Swul`, introducing productions such as

```
"panel" "of" Layout                   -> Component {cons("Panel")}
"border" "layout" "{" LayoutProp* "}" -> Layout {cons("BorderLayout")}
```

to define syntax for Swing concepts such as panels and layout schemes.

The embedding of the language in Java is achieved by creating a new SDF module `Java-Swul` (Figure 2.11) that imports the syntax of Java and the syntax of Swul. The actual imported modules are prefixed wrappers of the real syntax definition, a feature of SDF further explained below.

Combining the two syntax definitions by importing them in a new module does not actually achieve the embedding of the syntax of Swul into Java. The languages are strictly separated from each other since the productions of Java do not refer to nonterminals of Swul and vice versa. The embedding of Swul in Java is achieved by *adding* productions to the combined syntax definition that allow Swul expressions to be used as Java expressions and vice versa. These productions just connect the languages at the desired location. Note that the embedding of the language is completely user-definable with an ordinary SDF module.

The kind of connection between the languages is indicated by specific constructor names. Productions that allow Swul constructs to be used as Java expressions use the constructor `ToExpr`. Similarly, productions that allow Java expressions to be used in Swul use the constructor `FromExpr`. Thus, in the embedding defined above, the `ToExpr` productions declare that Swul *component* and *layout* expressions can be used as Java expressions. The `FromExpr` productions declare that Java expressions can be used as Swul border or component expressions. Note that no special quotation symbols are needed to inject Swul expressions into Java expressions or vice versa.

```
module Java-Prefixed
imports Java
  [ ImportDec            => JavaImportDec
    CompilationUnit      => JavaCompilationUnit
    TypeDec              => JavaTypeDec
    ...
    Expr                 => JavaExpr ]
```

Figure 2.12 Syntax definition that prefixes all Java nonterminals with the name of the language.

Another point to note about the embedding is the import of the modules `Java-Prefixed` and `Swul-Prefixed` instead of `Java` and `Swul`. The `-Prefixed` SDF modules are (generated) SDF modules that import the actual syntax and rename all nonterminals in this imported definition by prefixing them with the name of the language. An example of such a module is shown in Figure 2.12. Note that these renamings do not require an actual copy of the definitions, but rather an import with a set of renamings applied.

These renamings are necessary to keep the two languages strictly separated, except for the explicitly defined connections. For example, both languages might define an `Id` or `Expr` nonterminal. If these syntax definitions are just imported directly, then there will be just one `Id` or `Expr` nonterminal. Productions using these nonterminals will then refer to the productions for these nonterminals from both languages. This embedding is not explicit and should therefore be prevented. By making the nonterminal names unique for both languages, undesired embedding is avoided[3].

Assimilation

Assimilation is again achieved via rewrite rules implemented in Stratego. The rewrite rules transform Swul expressions to Java expressions. These rules express the knowledge of the Swing API captured in the language by translating each Swul construct to the appropriate sequence of Swing method calls. Figure 2.13 illustrates this for some of the Swul constructs. Note again that although these examples are all written in concrete syntax, the actual representation that is being transformed is a structured abstract syntax tree. The Swul and Java code fragments are all syntactically checked when compiling the generator.

Most of the rewrite rules have a `where` clause. The Stratego construct <*s*> *t* => *p* applies the rewriting *s* to the expression *t* and matches the result of applying *s* to *t* to the pattern *p*. In the examples the patterns are simple variables. In this case, <*s*> *t* => *x* is comparable to an ordinary assignment of *s* applied to *t* to the variable *x*. The rewrite rules use metavariables in the code fragments. That is, `ps*`, `c`, `x`, `e` and `bstm*` are Stratego variables, used directly in the Java code fragment. These variables are bound to abstract syntax trees

[3]In Chapter 5 we improve the method of renaming nonterminals by introducing grammar mixins.

```
Swulc-Component :
  swul |[ scrollpane of c ]| -> expr |[ new JScrollPane(e) ]|
  where <Swulc-Component> c => e

Swulc-Component :
  swul |[ textarea {ps*} ]| ->
  expr |[ {|JTextArea x = new JTextArea(); bstm* |x|} ]|
  where new => x
      ; <map(Swulc-SetProp(|x))> ps* => bstm*

Swulc-AddComponent(|x) :
  swul |[ north = c ]| -> bstm |[ x.add(BorderLayout.NORTH, e); ]|
  where <Swulc-Component> c => e

Swulc-Layout :
  swul |[ grid layout {ps*} ]| -> expr |[ new GridLayout(i,j) ]|
  where <nr-of-rows> ps* => i
      ; <nr-of-columns> ps* => j

Swulc-SetProp(|x) :
  swul |[ border = b ]| -> bstm |[ x.setBorder(e); ]|
  where <Swulc-Border> b => e

Swulc-Component :
  swul |[ x: c ]| -> expr |[ {| t x = e; | x |} ]|
  where <java-type-of> c => t
      ; <Swulc-Component> c => e
```

Figure 2.13 Some of the rewrite rules for assimilation of GUI abstractions using Swing API

and are not matched literally as Java variables. The Stratego primitive `new` (used in the `where` clauses) generates a new unique identifier.

Most rewrite rules in Figure 2.13 are straightforward translations from Swul constructs to corresponding Swing API invocations. For example, the rewrite rule for a `scrollpane` directly translates into a constructor call of the `JScrollPane` class. Some constructs in Swul provide even more abstraction than an alternative syntax. For example, the rewrite rule for the `grid layout` calculates the number of rows and columns. The rewrite rule for the Swul construct `x : c`, the last rule in the figure, shows another abstraction; if possible, it determines the type of the component and declares a variable `x` of this type with the initial value `c`. This inline declaration is very useful since Swul does not cover the full Swing API, but just the most common constructs. If unusual things have to be configured for a component, then this can be done later without ruining the compositional definition of the user interface.

The transformation makes use of a convenience extension of Java with expression blocks, which are removed by a separate transformation. In fact the compositional nature of Swul (and other extensions) is based on this extension. Constructing Java objects is composable in Java itself as long as all required operations can be performed by invoking a single method or constructor. As soon as further method calls are required there is a problem in composing objects without binding them to intermediate variables. The ex-

```
context-free syntax
  Stm -> JavaStm {cons("ToStm")}

  JavaId "." "write"       "%>" Content  "<%" ";" -> Stm {cons("Write")}
  JavaId "." "write" "doc" "%>" Document "<%" ";" -> Stm {cons("Write")}

  "<%" JavaStm  "%>" -> Content              {cons("FromStm")}
  "<%" JavaExpr "%>" -> DoubleQuotedPart     {cons("FromExpr")}
  "<%" JavaExpr "::" "cdata" "%>" -> Content {cons("TextFromExpr")}
```

Figure 2.14 Syntax definition of Java with XML

tension of Java with expression blocks solves this problem introducing two expression: `{| bstm* | e |}` and `{| e | bstm* |}`. An expression block is thus a list of block-level statements followed by an expression or the other way around. The expression is the value of the expression block. In the first case the statements are lifted to the block-level statements *before* the context expression. In the second case they will be lifted to block-level statements *after* the context expression. The syntax of this extension is defined by two SDF production rules and the constructs are translated into ordinary Java by a small Stratego program[4].

The expression block extension makes the definition of assimilation, and code generators in general, much simpler. For example, the declaration of a `textarea{ps*}` is translated to a declaration of a new variable `x` of type `JTextArea`, which is instantiated to a new `JTextArea`. The instantiation of the properties of the textarea is achieved with additional statements, which are executed after creating the textarea. The context in which the `textarea` was placed receives as value the variable `x`. Thus a linear sequence of statements building the components of the user interface is realized.

2.3.3 *Java with XML*

Our next example is the embedding of XML in Java. The application of this embedding was illustrated in Figure 2.4.

Embedding

Some of the production rules of the syntax embedding are shown in Figure 2.14. These production rules are somewhat different from usual embeddings, since the embedded XML syntax is translated to the SAX API. XML construction in SAX is not done using expressions and objects, but by invoking methods of the `ContentHandler` interface. Typically, the methods of a `ContentHandler` are invoked to report parsing events of an XML parser as callbacks to an ap-

[4]Kats [Kats 2007] observed that this transformation is not always possible to implement by trivially lifting the statement to the first surrounding statement level, for example if the expression block is used in a loop condition. The work of Kats proposes a new kind of open compiler based on mixing Java source and bytecode, which makes expression blocks trivial to implement as a language extension by leveraging the greater flexibility of Java bytecode.

plication. However, any code can use the `ContentHandler` interface to report the content of an XML document.

The embedding extends Java with syntax for writing XML content to a `ContentHandler`. The `ContentHandler` instance is specified by a Java identifier. The `ToStm` and `Write` constructors are used to represent the switch from Java to XML. The `Write` constructor has two arguments: a Java identifier of the `ContentHandler` and the XML content to write to it.

In addition to the embedding of XML in Java, we also define production rules for escaping from XML back to Java. First, the escape from XML content to Java is represented by the `FromStm` constructor. This escape consists of a Java statement, which might be surprising, since an escape to the host language usually is an *expression*. However, the SAX `ContentHandler` produces XML content by statements, not by expressions. Second, the embedding defines an escape from XML attribute values (`DoubleQuotedPart`) to Java. Third, the embedding defines a more specific anti-quotation for character data, using the constructor `TextFromExpr`. The result of the Java expression must be a String, which is emitted to the `ContentHandler`.

Assimilation

Most of the Stratego rewrite rules for the assimilation of the embedded XML fragments into Java are shown in Figure 2.15. For a single XML construct the resulting code fragments are now larger than in the previous examples. Hence, the XML syntax of this embedding is a major abstraction from the API interface provided by SAX. In this assimilation, the left-hand side of the rewrite rules are not written in concrete XML syntax, but rather in abstract syntax. The left-hand side patterns are so small that using concrete syntax would only be confusing. This illustrates that programs using concrete syntax can fall-back to the assimilated notation in the host language (in this case abstract syntax for XML) when this is more appropriate.

Figure 2.15 shows the rewrite rules for XML documents, elements, text and character data escapes to Java. The translation of `Document`[5] and `Text`[32] is straightforward. The translation of an `Element`[13] is somewhat more involved, since the list of attributes has to be assimilated into an instance of the `AttributesImpl` class. Character data escapes to Java are translated using expression blocks[26], as discussed before. This translation to an expression block is necessary because we need the value (bound to *y*) at multiple locations; for calculating the length of the string and for translating it into a character array that is passed to the `ContentHandler`. Note that the rules use several metavariables: *x*, *y*, *bstm1**, *bstm2**, and *e**. Also, the translation of an XML element uses a special construct *inside* a string literal: `"~n"`. The `~` construct denotes an escape to the host language, which is Stratego in this case. Anti-quotation and metavariables inside such lexical constructs are a unique feature of SDF and SGLR. This is possible by preserving the structure of lexemes, which will be discussed in more detail in Section 2.4.

```
1  explode-write :
2    ToStm(Write(Id(x), c)) -> bstm |[ { bstm* } ]|
3    where <content-to-stm(|x)> c => bstm*
4
5  content-to-stm(|x) :
6    Document(c) -> bstm* |[
7       x.startDocument();
8       bstm1*
9       x.endDocument();
10   ]|
11   where <content-to-stms(|x)> c => bstm1*
12
13 content-to-stm(|x) :
14   Element(Name(None(), n), atts, kids) -> bstm* |[
15      org.xml.sax.helpers.AttributesImpl y
16        = new org.xml.sax.helpers.AttributesImpl();
17      bstm1*
18      x.startElement("", "~n", "~n", y);
19      bstm2*
20      x.endElement("", "~n", "~n");
21   ]|
22   where <map(content-to-stms(|x))> kids => bstm2*
23       ; new => y
24       ; <map(attr-to-stm(|y))> atts => bstm1*
25
26 content-to-stm(|x) :
27   TextFromExpr(e) -> bstm |[
28     x.characters({| String y = e; | y.toCharArray() |},0,y.length());
29   ]|
30   where new => y
31
32 content-to-stm(|x) :
33   Text(s) -> bstm* |[ x.characters(new char[]{e*}, 0, i); ]|
34   where <explode-string> s => cs
35       ; <length; int-to-string> cs => i
36       ; <map(escape-char)> cs => e*
```

Figure 2.15 Rewrite rules for assimilation of XML to SAX ContentHandler invocations.

2.3.4 *Java with Java*

Lastly, we return to our first example in Section 2.2: Java embedded in Java. This combined language is called JavaJava. JavaJava is a language for Java metaprogramming, i.e. Java programs that generate or manipulate Java programs. The embedding and assimilation of JavaJava are defined with the MetaBorg method. The implementation involves the definition of an embedding of Java in Java, and an assimilation of the embedded Java fragments into the host language.

Embedding

A selected number of productions from the embedding of Java in Java are shown in Figure 2.16. The complete embedding is much larger, since the Java syntax definition contains many nonterminals. For each nonterminal that is to be quoted or anti-quoted in JavaJava, productions have to be defined in the

```
context-free syntax
         "⟦" Expr "⟧" -> JavaExpr        {cons("ToExpr")}
  "expr" "⟦" Expr "⟧" -> JavaExpr        {cons("ToExpr")}

  "type"  "⟦" Type      "⟧" -> JavaExpr  {cons("ToExpr")}
  "bstm*" "⟦" BlockStm* "⟧" -> JavaExpr  {cons("ToExpr")}

  "~"  JavaExpr ->  Expr                 {cons("FromExpr")}
  "~*" JavaExpr -> {Expr ","}*           {cons("FromExpr")}
  "~*" JavaExpr -> {VarInit ","}*        {cons("FromExpr")}
  "~*" JavaExpr -> {FormalParam ","}*    {cons("FromExpr")}
  "~*" JavaExpr -> ClassBodyDec*         {cons("FromExpr")}

variables
  "e"    [0-9]*     -> Expr             {prefer}
  "t"    [0-9]*     -> Type             {prefer}
  "e"    [0-9]* "*" -> {Expr ","}*      {prefer}
  "e"    [0-9]* "*" -> {VarInit ","}*   {prefer}
  "bstm" [0-9]* "*" -> BlockStm*        {prefer}
  [ij]   [0-9]*     -> DeciLiteral      {prefer}
  [xyz]  [0-9]*     -> Id               {prefer}
```

Figure 2.16 Syntax definition of Java with Java

combined syntax definition. The JavaJava syntax definition applies some of the more advanced features in SDF for combining the syntax of the embedded language and the host language.

The first two productions show that the declaration of nonterminals of concrete syntax fragments is optional. To this end, there is a production rule for embedding Java expressions using just the quotation symbols ⟦ and ⟧ and there is a production with an explicit declaration `expr` of the nonterminal. The embedding of `BlockStm*` shows that lists of nonterminals, in this case `BlockStm`, can be embedded as well. The tags `expr`, `type`, and `bstm*` are necessary for disambiguation. In Chapter 3 we will discuss a method for avoiding these tags by employing a disambiguating type-checker. In Chapter 4 we use an alternative, more lightweight method of runtime disambiguation.

The escapes to the meta level use the ~ symbol in this embedding. That is, meta Java expressions can be included by using only a ~ before the meta expression. The `~*` escape can be used to escape to a Java expression that produces a list of a certain nonterminal, even if they are separated by tokens. The `~*` escape is defined by an SDF production rule that produces a list nonterminal (the right-hand side of the production rules). For example, the escape for `{Expr ","}*` is used in method invocations, which take a number of expressions separated by commas. In the argument list of a method in concrete syntax it is possible to escape to the meta level using `foo(~*args)`, but even `foo("bar",~arg,~*args)` and `foo(~*args1,~*args2)` are allowed. All these escape are defined by a single production rule in the embedding of Java in Java. No additional SDF productions are required, since productions that produce list symbols are desugared to a set of more basic productions that are sufficient to handle all these combinations.

JavaJava also provides metavariables, which is an even more compact method for embedding variables from the meta language in embedded code fragments. The SDF syntax definition formalism has been designed for application in metaprogramming and therefore it has the built-in notion of metavariables. Metavariables are defined in a `variables` section. This section consists of SDF productions that define a special meaning for certain identifiers. For example, the first production in the `variables` section of Figure 2.16 defines that the identifier `e` is a metavariable denoting a Java expression. Some other metavariables in JavaJava are `e*` for lists of expressions separated by commas and `x`, `y`, and `z` for identifiers.

Assimilation

The MetaBorg examples of Java with Swul and Java with XML apply assimilations that are specific to the host language *and* the embedded language. The assimilation cannot be reused for different embedded languages since the assimilator rewrites specific constructs of the embedded language to the host language. Such a specific assimilator can incorporate domain-specific knowledge, which adds a level of abstraction to the syntactic embedding. For example, the Swul assimilator calculates the number of rows and columns of a GridLayout.

However, for some applications it is possible to implement a *generic* assimilation of an arbitrary embedded language to an API in the host language. Hence, the assimilation can be reused for the embedding of *any* language in the host language. For instance, generic assimilation is possible for assimilating embedded languages in metaprogramming. Metaprogramming frameworks often follow a certain standard procedure for representing a subject program in a class hierarchy. Examples of such frameworks are JJForester [Kuipers & Visser 2001], ApiGen [de Jong & Olivier 2004], Java Tree Builder [Tao et al.] with JavaCC [JavaCC Website], and SableCC [Gagnon & Hendren 1998]. For a specific metaprogramming framework the mapping from a language definition to a class structure is fixed or at least reproducible. This fixed translation scheme can be used for the implementation of a generic assimilator. Similarly, a generic assimilator is also possible if the target API itself is generic. This is the case if the embedded language is to be represented using a generic class hierarchy for trees, such as ATerms [van den Brand et al. 2000] or XML.

The generic assimilation from an embedded language to the construction of a generic tree representation can be implemented particularly well in a language that supports generic programming. To illustrate the implementation of a generic assimilation we have implemented an assimilation from the embedded Java code in the JavaJava language to the ATerm library. A generic assimilation requires generic programming. Stratego allows generic term construction and deconstruction using the `#` operator. A term `foo(bar(), fred())` can be deconstructed into the constructor of the term, `foo`, and a list of children to which the construct is applied, `bar()` and `fred()`. The pattern

```
AssimilateAppl(rec) :
  fun#([t*]){} -> expr |[ _factory.makeAppl(e, e*) ]|
  where <AssimilateAFun> (fun, <length> t*) => e
      ; <map(rec)> t* => e*

AssimilateAFun :
  (fun, arity) -> expr |[ _factory.makeAFun("~fun", i, false) ]|
  where <int-to-string> arity => i

AssimilateMetaVar :
  meta-var(x) -> expr |[ x ]|

AssimilateInt :
  i -> expr |[ _factory.makeInt(j) ]|
  where <int-to-string> i => j

AssimilateString :
  s -> expr |[ _factory.makeAppl(_factory.makeAFun("~s", 0, true)) ]|
  where <is-string> s
```

Figure 2.17 Generic assimilation of abstract syntax trees in Java with ATerms

`fun#([t*])` binds the constructor name to the variable `fun` and the list of children to the variable `t*`.

The generic assimilation of ATerms to Java is implemented by the Stratego rewrite rules shown in Figure 2.17. The rewrite rules `AssimilateAppl` and `AssimilateAFun` handle ATerm constructor applications and use generic term deconstruction. The right-hand sides of the rewrite rules are Java expressions. For the construction of ATerms in the resulting Java code, methods defined in the `ATermFactory` interface of the ATerm library are invoked. The rules `AssimilateInt` and `AssimilateString` handle the ATerm integer and string constructs. The `AssimilateMetaVar` rule rewrites metavariables to real Java variables.

This generic assimilation can be applied to all embedded languages in the host language Java. Hence, an embedding of a different subject language only needs to define the combined syntax definition.

API specific assimilations for the *same* JavaJava language can be implemented as well. For example, an assimilator could target the class hierarchy for Java abstract syntax trees in Eclipse.

2.4 SYNTAX DEFINITION

The embedding of languages poses some challenges on syntax definition and parser technique. MetaBorg employs the syntax definition formalism SDF [Visser 1997b] and SGLR [Visser 1997a, van den Brand et al. 2002], a Scannerless Generalized LR parser.

In the previous section we described several MetaBorg embeddings. So far, we have not revealed how these combined languages are actually parsed and what the features of the syntax definition formalism SDF are. Full insight in why SDF is the appropriate syntax definition formalism for defining lan-

guage embeddings requires more detailed knowledge of SDF. In this section, the SDF syntax definition formalism is introduced and we show how it is applied to achieve concrete syntax for objects. Several features of the syntax definition formalism allow the concise definition of the syntax of embeddings. Two features of this parsing technology are essential for defining the syntax of language embeddings: the *GLR* algorithm [Tomita 1985, Rekers 1992] and *scannerless* parsing [Salomon & Cormack 1989]. These features are combined in SGLR [Visser 1997a, van den Brand et al. 2002].

2.4.1 *SDF Overview*

The SDF syntax definition formalism supports concise and natural expression of the syntax of context-free languages. SDF integrates lexical and context-free syntax in a single formalism. The complete syntax of a language is thus defined in a single definition. SDF supports the entire class of context-free grammars. SDF does therefore not restrict the grammars to a subclass of the context-free grammars, such as LL or LALR. SDF syntax definitions can be split into modules, and SDF modules can be reused in different syntax definitions. Disambiguation of grammars is not done by grammar hacking, but by applying special purpose disambiguation facilities in SDF, such as priorities, reject productions, and follow restrictions. To illustrate the key features we explain a syntax definition of expressions. For more documentation and examples we refer to [SDF Website, Visser 1997b].

Context-Free and Lexical Syntax

Syntax is defined in syntax sections, which are either *context-free* or *lexical*. The difference between these two kinds of syntax sections is that in lexical syntax no layout is allowed between symbols. In context-free syntax sections layout is allowed between the symbols of a production. The term *context-free syntax* should not be confused with the expressiveness of the production rules. The expressiveness of the lexical and context-free syntax sections is not different: lexical syntax is not restricted to a regular grammar. In fact, lexical and context-free syntax are even translated into a single core syntax definition formalism [Visser 1997b].

Lexical syntax sections are used to define constructs such as identifiers, layout, comments, and literals. Layout is a special nonterminal in SDF named `LAYOUT`. A symbol for this nonterminal is inserted between the symbols of productions in a context-free syntax section to allow layout between the symbols there. The following lexical syntax section defines layout (`LAYOUT`), identifiers (`Id`), and integer constants (`IntConst`).

```
lexical syntax
  [A-Za-z][A-Za-z0-9]* -> Id
  [0-9]+               -> IntConst
  [\r\n\t\ ]           -> LAYOUT
```

These lexical sorts can be used in a context-free syntax section to define the more complex constructs of a language. The context-free syntax of the example language consists of variables, integer literals, arithmetic operations,

method invocations and conditional expressions. Note that the definition of expressions is concise, without the hurdles of introducing additional nonterminals to handle priority and associativity of operators.

```
context-free syntax
  Id          -> Exp {cons("Var")}
  IntConst    -> Exp {cons("Int")}

  Exp "+" Exp -> Exp {cons("Plus")}
  Exp "-" Exp -> Exp {cons("Min")}
  Exp "*" Exp -> Exp {cons("Mul")}
  Exp "/" Exp -> Exp {cons("Div")}

  Exp "." Id  "(" {Exp ","}* ")" -> Exp {cons("Call")}
  "if" Exp "then" Exp "else" Exp -> Exp {cons("If")}
```

The productions are annotated with a constructor name (`cons("...")`). These constructor names are used to construct an abstract syntax tree from the parse tree that results from parsing an input text. We use the ATerm format [van den Brand et al. 2000] for the representation of parse and abstract syntax trees. The ATerm format is somewhat comparable to XML, but is more structured. It supports lists, tuples, integers, string literals, and of course constructor applications. For example, the input `4 + 1.get(5)` is represented by the ATerm:

```
Plus(Int("4"), Call(Var("1"), "get", [Int("5")]))
```

Disambiguation

SDF supports the full class of context-free grammars, including ambiguous grammars. The implementation of SGLR is able to produce a parse forest of all possible parse trees, but obviously we would like to define what parse trees are preferred over others. For the purpose of disambiguation SDF allows separate disambiguation filters instead of hacking the syntax definition itself into a non-ambiguous form. Separate priority definitions, follow restrictions, reject, avoid, and prefer filters can be used to disambiguate a syntax definition [van den Brand et al. 2002].

The syntax definition above is highly ambiguous for several reasons. First of all there is an *associativity* problem for the binary operators. The input `1 + 2 + 3` can be parsed as either

```
Plus(..., Plus(...,...))   or   Plus(Plus(..., ...), ...)
```

Since there is more than one possible interpretation, parsing the input will result in a parse forest. The ambiguous phrases are represented by `amb` nodes in the parse tree. Production rules in SDF can be annotated with `right`, `left`, `assoc` or `non-assoc` to define the associativity of an operator. In this case the `+` operator is left associative, therefore the new production rule is:

```
Exp "+" Exp -> Exp {left, cons("Plus")}
```

Another problem is the *priority* of operators. The input `1 + 2 * 3` can be parsed as

```
Plus(..., Mul(..., ...))   or   Mul(Plus(..., ...), ...)
```

A related problem is the input `4 + x.get()`. Although it might seem apparent that the method `get` must be invoked on the variable `x`, this is not the only possible parse. The input can be parsed as

```
Plus(..., Call(..., ..., ...))   or   Call(Plus(..., ...), ..., ...)
```

These ambiguities can be solved by defining priorities of production rules. Priorities in SDF are defined *relatively* and not by defining priority levels. The priority of `*` is usually higher then the priority of `+` and the priority of the method call is higher than all operators in our example language. Since such priorities are not properties of a single production, there is a separate section in an SDF module for defining priorities. Production rules in priority definitions can also be grouped, which means that their priority is equal. In this case the associativity of such a group should be defined. If `-` and `+` are in the same priority group, then the associativity of the group determines how for example the input `1 - 2 + 3` is parsed. The resulting priority definition is:

```
context-free priorities
    Exp "." Id  "(" {Exp ","}* ")" -> Exp
  > {left:
      Exp "/" Exp -> Exp
      Exp "*" Exp -> Exp
    }
  > {left:
      Exp "+" Exp -> Exp
      Exp "-" Exp -> Exp
    }
```

These priorities solve all ambiguity problems in the example. There is still one problem left, but it is not an ambiguity problem in this example. The syntax definition is too liberal: identifiers are not recognized in a greedy way, which means that the input `if 1 then aelse b` is valid. Also, the syntax definition does not forbid `if 1 thena else b`, since the restriction that keywords cannot be immediately followed by an identifier character has not been expressed. The syntax definition must exclude these two input texts by recognizing identifiers and integer literals in a greedy way and by disallowing a keyword to be followed by an identifier character. The longest match policy is implicit in most parser generators, especially in those with a separate scanner. In SDF a longest match restriction can be defined in the syntax definition itself by using a *follow restriction*. A follow restriction `A -/- CC` forbids that the nonterminal `A` is followed by a character from the character class `CC`.

```
lexical restrictions
  Id       -/- [A-Za-z0-9]
  IntConst -/- [0-9]
  "if" "then" "else" -/- [A-Za-z0-9]
```

2.4.2 *The Importance of Modularity*

Modularity is essential for the definition of the syntax of language embeddings. To make the embedding of the syntax of a language A in a host language B concise and maintainable, it must be possible to develop the syntax

definitions of language A and B independently from each other. Modularity in the definition of language embeddings becomes even more important if more than one language needs to be embedded. For example, XML and SQL, XPath and XML, Java and XML. Defining these embeddings in a single syntax definition is unacceptable from the point of view of clarity, maintenance, and reusability.

Most syntax definition techniques that are used in practice are limited to some subset of the context-free grammars, since they target a parser generator that applies a certain restricted parsing algorithm. This is illustrated by the fact that most syntax definition languages are coupled with a parser generator implementation. Depending on the parsing algorithm that the generated parser is using, the syntax definition is restricted to a certain subclass of the context-free grammars, such as LALR for YACC, CUP and SableCC, or LL for ANTLR and JavaCC. Restricting syntax definitions to some proper subclass of context-free grammars is to a certain extent acceptable for generating parsers from monolithic syntax definitions of single programming languages, but it is not for combining programming languages, for example in the embedding of languages in host languages.

An interesting formal result is that there is no proper subclass of the context-free grammars that is closed under union. As a consequence, grammar languages that are restricted to some proper subset of the context-free languages cannot be modular since the combination of two, for example LR, syntax definitions is not guaranteed to be in this subset. Therefore, SDF supports the full class of context-free grammars.

Supporting the full class of context-free grammars in a syntax definition formalism introduces some challenges to the applied parsing technology. Parsers and parser generators that allow the full class of context-free grammars apply Generalized LR (GLR) [Tomita 1985, Rekers 1992] or Earley [Earley 1970] parsing. The GLR parsing algorithm maintains multiple LR parsing states in parallel. At phrases where the parser cannot choose from multiple production rules to apply, it forks the current parser. The forked parsers meet again if a token in the input is to be parsed with the same nonterminal.

The modularity features of SDF are comprehensive. Since using modular syntax definition in SDF introduces not a single awkwardness, defining syntax definitions in a modular way is even encouraged. Renamings make the combination *and* isolation of syntax definitions very flexible as explained in Section 2.3.2. SDF is modular to the core.

2.4.3 *The Importance of Scannerless Parsing*

Most parsers apply a separate scanner for lexical analysis. The purpose of this lexical analysis phase is to break up the input into tokens, such as identifiers, numbers, layout, and specific keywords (e.g. `if`, `switch`, `try`, `catch`). That is, the lexical analysis phase decides what kind of token a lexeme is. A separate lexical analysis phase allows the parser to operate on the list of lexemes, ignoring their actual contents. The lexical syntax of a language is

usually specified by regular expressions. The scanner applies finite automata to recognize the tokens specified by these regular expressions.

Context of Tokens Scanners that do not interact with the parser cannot consider the context of a sequence of characters in the decision which token a sequence represents. This is essentially the reason for having reserved keywords, which cannot be used as identifiers in the language. The separate lexical analyzer is also the reason for not allowing nested block comments (`/* /* */ */`) and for disallowing `a -- b` to be parsed as `a - (-b)`. Thus, the lexical analyzer decides what token a list of characters represents, without considering the context of the tokens in the input stream.

This may be reasonable for parsing inputs written in a single language, in particular since this language can be designed with these restrictions in mind. Yet, it is more problematic when the input consists of a combination of languages. By not considering the context of a sequence of characters in the input stream, it is impossible to assign it different tokens, since this ultimately depends on the context of the lexeme. As regards embedded languages, the scanner cannot assign different token types to the same lexeme in embedded code and in host code. This is a serious problem if the lexical syntax of the embedded language is entirely different from the lexical syntax of the host language. It is not a problem if the embedded language is the same language as the host language (for example Java embedded in Java), since the lexical syntax is in this case not different in the embedded code fragments. However, embedding XML in Java is more challenging, since the lexical syntax of XML is completely different from Java. In this case the lexical analyzer must consider the context of a sequence of characters before deciding what kind of token it is.

Preserve Structure of Lexemes A scanner usually passes lexemes as an unstructured atomic value to the parser. The internal structure of the tokens is thus not preserved. This is a pity since tokens such as floating point and String literals can have quite a complex structure. The string literal `"Hello\r\n world!"` will typically result in the token `"\"Hello\\r\\n world!\""`. The semantic tools that operates on parse (or abstract syntax) trees must analyze the structure of the tokens again to determine its meaning. However, preservation of the structure of a lexeme is essential if it must be possible to escape to the metalanguage *inside* a token. We have already seen such escapes in our examples, namely the escape in XML attribute values and Java string literals (see Section 2.3.3). Parsing with a separate scanner does not support such structured lexemes, since lexemes are passed as a plain sequence of characters to the parser. In *scannerless parsing* the structure of lexemes can be preserved. For example, in our Java tools the string literal above is represented as:

```
String([
  Chars("Hello"), NamedEscape(114), NamedEscape(110), Chars(" world!")
])
```

The preservation of lexical structure is mostly a matter of convenience if parsing inputs of single programming languages. However, for the embedding

of a language in a host language, the preservation of lexical structure is extremely useful, if not a requirement. For example, the following XML attribute contains an escape to the meta level inside an XML attribute:

```
<a href="http://www.<% s %>.org">Stratego/XT</a>
```

In scannerless parsing the structure of the attribute value can be preserved. In the embedding of XML in Java, the attribute is represented as:

```
Attribute(
  QName(None, "href")
, DoubleQuoted([
    Literal("http://www.")
  , Literal(FromExpr(ExprName(Id("s"))))
  , Literal(".org")
  ])
)
```

Note that preserving the structure of lexical syntax is related to the context of tokens. The content of a string literal is in a sense an embedded language. Scanning string literals could result in separate tokens of the lexical syntax of this language. This requires the scanner to know about the context of a sequence of characters. After all, the lexical syntax of this 'embedded' string literal language is completely different from the lexical syntax of the host language.

Solutions Of course these problems can all be solved in hand-written scanners. Such a scanner can interact with the parser in arbitrary ways. However, writing efficient parsers and scanners by hand that can handle ambiguous input streams is very difficult. Writing scanners and parsers by hand is in general not an option if languages need to be combined.

The solution is to discard the separate lexical analysis phase completely. The parser directly operates on the characters of the input. In syntax definitions for scannerless parsers the lexical syntax and context-free syntax can be specified in a single syntax definition formalism. Hence, the full syntax of a language is specified in a single definition. All information regarding the disambiguation can be in the same syntax definition and there are no implicit disambiguation rules based on the parser technology that is applied. Since the parser operates on individual characters the context-free analysis of the parser can be used to determine the types of individual characters. Thus, the context of the character is taken into account. If a certain token is not expected at a certain location in the input, then it will not be considered. Parsing without a separate scanner is called scannerless, a term that was coined in [Salomon & Cormack 1989]. This solution combined with the GLR algorithm is known as Scannerless Generalized LR parsing [Visser 1997a]. The parser generator for SDF (pgen) generates a parse table for a Scannerless Generalized LR parser (sglr). The implementation is available at [ASF+SDF MetaEnv, SDF Website].

2.5 PREVIOUS WORK

SDF [Heering et al. 1989] was originally designed for use as a general syntax definition formalism. However, through its implementation it was closely tied to the algebraic specification formalism ASF+SDF [van Deursen et al. 1996], which is supported by the ASF+SDF Meta-Environment [van den Brand et al. 2001]. Redesign and reimplementation as SDF2 [Visser 1997b, van den Brand et al. 2002] has made the language available for use outside the Meta-Environment. SDF2 is also distributed as part of the Stratego/XT bundle of program transformation tools [Visser 2004, de Jonge et al. 2001]. Syntax definition in SDF2 is limited to *context-free grammars*. This is a limitation for languages with context-sensitive syntax such as that of Haskell (offside rule). However, in the setting of embedded concrete syntax, in which small fragments are used and not all context is always available, any parsing technique will have a hard time. The combination of SDF with ASF was the first to use SDF for the definition of embedded syntax.

Stratego [Visser et al. 1998, Visser 2004, Stratego Website] is a language for the implementation of program transformation based on rewrite rules under control of programmable rewriting strategies. Rewrite rules transform first-order terms. In [Visser 2002] an extension of Stratego with concrete syntax for the program fragments manipulated by its rewrite rules is introduced. The paper presents a single *generic* assimilation transformation, mapping term representations of abstract syntax trees into Stratego terms, similar to the assimilation of Java programs to ATerms in Section 2.3.4. The paper also gives an outline of a general architecture for extension of an arbitrary host language with concrete syntax.

In [Fischer & Visser 2004] the approach is extended to Prolog as a metalanguage, mainly to provide concrete syntax for the schemas of the AutoBayes program synthesis engine. Since Prolog is also a term-based language the assimilation is defined in a similar generic way.

The contributions of this chapter are the extension of the approach to an object-oriented host language, the targeting of specific APIs by the assimilation transformation, and the specific extensions of Java for code generation, XML generation, and user interface construction. The design of SWUL and its embedding in Java is a nice by-product of this project.

2.6 RELATED WORK

There have been many other approaches to make programming languages syntactically extensible. We give an overview of such approaches and discuss their relation to MetaBorg.

2.6.1 *Extensible Syntax*

The extensible syntax [Cardelli et al. 1994] applied in the implementation of $F_{<:}$ [Cardelli 1993] was the first step into incremental syntax definition by

extending and restricting a small core language. The definition of syntax extensions and the assimilation into the host language are in the source code of the host language. Syntax extensions can be local (`syntax ... in ... end`) or defined at top-level. The syntax can be defined incrementally by updating, extending, or adding production rules to an existing grammar. The syntax extensions include a rewriting to the host language by constructor applications or action definitions that rewrite the syntax directly to the target language.

The implementation of extensible syntax is based on LL parsing, which makes the syntax definition cumbersome and not modular, since the class of LL grammars is not closed under union and concatenation. It would be interesting to develop a comparable inline extensible syntax mechanism based on the more powerful parsing technique of scannerless generalized LR parsing, which was not mature enough when extensible syntax was developed. The latest work in this direction is the extensible generalized LR parser Dypgen [Onzon 2007] for Objective Caml. Dypgen supports extensible syntax, but the lexical syntax of the base language cannot be extended. Also, modifying the syntax triggers a reconstruction of the full parse table. Chapter 6 addresses this issue using *parse table composition*. Integration of parse table composition in a runtime extensible parser is future work.

Although the implementation of user-definable syntax in this work is very flexible (despite the LL parsing), we do not believe that there is a need for defining *ad-hoc* syntax extensions so powerful. Rather, in practice syntax extensions are carefully designed in a separate role. Developing appropriate domain-specific syntax extensions of a host language is thus a separate role and does not necessarily need to be performed in the host language. The implementation can be provided in separate tools. The additional advantage of these separate tools is that the metaprogrammer can choose the most appropriate programming language for implementing an assimilation. The MetaBorg method aids the implementation of these tools by providing powerful and generic tools for parsing and assimilation into the host language.

2.6.2 *Harmonia's Blender*

Blender [Begel & Graham 2004], developed in the Harmonia project, targets parsing inputs that cannot be designed to be non-ambiguous. Begel and Graham [Begel & Graham 2004] thoroughly analyze the problems in handling ambiguous inputs, especially embedded languages. Blender generates from Flex- and Bison-like definitions a lexical analyzer, parse tables and class definitions for representing abstract syntax trees in C++.

Blender applies GLR parsing with an incremental lexical analyzer that is forked together with the LR parsers in the GLR algorithm. The scanner is thus not completely separated from the parser, but parsing is not scannerless. If multiple lexical types can be assigned to a token, then the scanner reports all possible interpretations to the parser. For each interpretation of a token a parser and scanner is forked. This approach was also used by the implementation of SDF before the introduction of scannerless parsing [Heer-

ing et al. 1989]. The approach was abandoned because of huge numbers of forks and the increased complexity of the algorithm when trying to reduce the number of forks. In addition, the use of context-free grammars instead of regular grammars greatly increases expressivity, allowing nested comments and anti-quotation within lexical syntax, for example.

GLR parsing where the scanner is forked together with LR parsers should solve most issues with using a separate scanner. We have not been able to investigate whether this is really a performance improvement over Scannerless Generalized LR parsing, since no benchmarks are provided and Blender is not available for download. The main argument for not using scannerless parsing in the Harmonia project is to facilitate interactive environments using incremental parsing. In our view parse trees of scannerless parsing can be persisted as well and be reused to reparse just the edited parts. Begel and Graham suggest this as well and remark that the size of the parse trees might be a problem. However, the efficient ATerm format [van den Brand et al. 2000] has already proven to be successful in reducing the size of parse trees through maximal sharing.

Syntax definitions of Blender are modular, thus languages can be developed and maintained independently. Unfortunately, the syntax definition formalism used in Blender is much less concise than SDF. The lexical analyzer applies longest match and order-based matching. Although this is appropriate for most programming languages, we believe that implicit disambiguation by the parser should be avoided. Order-based selection is especially problematic if the lexical syntax is defined in several modules, as is typically the case when defining the syntax of an embedded language. The built-in preference for the longest match and tokens defined first, requires insight in the algorithms applied by the lexical analyzer generator. This makes debugging of the syntax definition more complex and requires studying traces of the parser. The declarative disambiguation methods of SDF allow the complete definition of the syntax of a language, independent from the tools applied to the definition.

2.6.3 *Jakarta Tool Suite (JTS)*

The Jakarta Tool Suite (JTS) [Batory et al. 1998] is a set of tools for extending Java with domain specific constructs. The main tools of JTS are Jak, which allows metaprogramming for Java in Java using the Java concrete syntax, and Bali, a frontend of JavaCC [JavaCC Website]. JTS targets the implementation of component-based generators, called GenVoca generators.

The Jak language of JTS supports the generation of Java programs by using the concrete syntax of Java in Java. Code fragments are embedded in tree constructors (for example `exp{...}exp`). From these fragments it is possible to escape to the meta language by an escape such as `$exp(...)`. The Jak language is comparable to our JavaJava language, but since our embedding is entirely user-definable and easy to extend, JavaJava supports a wider range of quotations (tree constructors) and anti-quotations (escapes). In Java-

JAVA the explicit declaration of nonterminals in tree constructors and escapes is not required if the nonterminal of code fragments can be determined by the parser. This makes code fragments much more concise. We would like to remove even more nonterminal declarations by disambiguating the code fragments in an extended type checker for Java. This future work is discussed later. In addition to anti-quotations, JAVAJAVA also supports *metavariables* which is a very compact embedding of variables from the meta language in the generated code fragments. Both solutions guarantee the syntactic correctness of the code fragments and construct an abstract syntax tree representation of the generated code.

Jak also features a matching facility. We have not implemented this yet in JAVAJAVA, because there is no easy mapping of match statements to existing APIs or language constructs in Java. The Tom pattern matching compiler [Moreau et al. 2003] could be used to implement support for patterns in concrete syntax.

JTS Bali is a tool for generating tools for extensions of Java. Bali is a front-end of JavaCC. Bali generates abstract syntax tree definitions and a parser by passing a grammar to JavaCC. Relying on JavaCC parsing technology introduces a lot of problems. Syntax definitions are much less concise compared to SDF and even to other available parser tools. JavaCC is an LL(k) parser generator, which implies that grammars are not composable and have to be manipulated to solve ambiguities. This parsing technology is appropriate for parsing languages that are designed not to be ambiguous, but in a toolkit for defining language extensions it is a major issue, as has been discussed in Section 2.4.

Bali composes grammars by combining the lexical and grammar rules. This merging is not guaranteed to succeed since LL(k) grammars are not composable. Bali uses heuristics to combine lexical rules such that the best results are produced in the common cases. Lexical scanners prefer the longest match and select rules that apply by order. Therefore, keyword rules are put before the more general rules, such as for identifiers. This implies that keywords for the embedded language may cancel use of those keywords as identifiers in the host language. The syntax definition and parsing techniques of SDF are much easier to use and can be applied for arbitrary embeddings. Because of the parser technology which Bali relies on, Bali will not be able to handle syntactic extensions that have an entirely different lexical syntax. SDF has no problem such embeddings, as discussed in Section 2.4.

2.6.4 *Syntax Macros*

Syntax macros [Leavenworth 1966] define syntactic abstractions over code fragments. Syntax macros usually operate on abstract syntax tree representations of programs. A syntax macro accepts abstract syntax tree arguments and generates a new abstract syntax tree that replaces the syntax macro invocation. To ensure the production of syntactically correct programs, the resulting

abstract syntax tree of a syntax macro invocation must be of the same type as the macro invocation.

In all implementations of syntax macros, a macro invocation must start with a unique macro delimiter, which is usually an identifier consisting of letters. This identifier indicates the syntax macro that is invoked. Such a macro delimiter is required since the syntax of macro invocations is defined in the input file itself. Some syntax macro implementations allow overloading of these identifiers to refer to different syntax macro definitions using the same identifier. The fixed syntax for a macro invocation is in many cases acceptable. The fixed syntax for invocations can even result in attractive, data-like, programs (called a Very Domain-Specific Language, VDSL, in [Brabrand & Schwartzbach 2002]).

However, the fixed invocation syntax of syntax macros is a limited extension of the syntax of the host language. The embedding of a domain-specific language in MetaBorg can be an arbitrary context-free language. The expressivity of our extensions is illustrated by the embedding of Java, XML, a tuple syntax, and SWUL. MetaBorg also allows an embedded language to use an entirely different lexical syntax, which is not the case for syntax macros, where lexical analysis of the macros is performed by the scanner of the base language. In particular, this feature of MetaBorg is illustrated by the embedding of XML in Java.

The main difference between syntax macro systems is the expressiveness of argument definitions. The most limited systems only allow a fixed number of arguments, usually with a syntax comparable to procedure calls. The most expressive systems are *MS*^{2} [Weise & Crew 1993], which allows regular languages, and *Dylan* [Shalit 1996] and *Metamorphic Syntax Macros* [Brabrand & Schwartzbach 2002], which allow context-free languages. The nonterminals appearing in actual arguments of macros in Dylan, are not represented as abstract syntax trees, but as a list of tokens. In [Brabrand & Schwartzbach 2002] the user-defined nonterminals are represented as an abstract syntax tree. These user-defined nonterminals are however required to have a fixed associated nonterminal in the host language. The MetaBorg method releases this restriction by completely separating the assimilation into the host language from the syntax definition of language extensions.

Implementations of syntax macros also differ in the expressivity of the rewriting to the base language in the syntax macro definitions, called assimilation in MetaBorg. Usually a syntax macro definition consists of the definition of the invocation syntax, which is some language over terminals and nonterminals of the host language. The arguments of the syntax macro can be used in the definition of code to be produced by a macro invocation. In [Leavenworth 1966] conditional syntax macros were already proposed, which increases the expressivity of the rewriting language. In the programmable syntax macros of *MS*^{2} [Weise & Crew 1993] the macro language is a small extension of the host programming language. This extension is targeted at rewriting syntactic extensions to the base language. *C++ templates* are well-known to be Turing complete, but this expressivity is not based on rewriting trees, but

on constant folding.

The Java Syntactic Extender (JSE) [Bachrach & Playford 2001] macro system is mainly inspired by Dylan. The expressiveness of the syntax that can be introduced is limited. As in most macro systems, a macro identifier is required in invocations and the parser used by JSE is not extensible. Rather, a source file is parsed by a fixed parser to a 'skeleton syntax tree' (SST). The SST is a lexical representation of a Java source file, but it is somewhat more structured than a plain sequence of tokens. JSE macros are implemented in Java and operate on the SST, i.e. lexical syntax. Hence, the source code fragments that are manipulated by JSE are not syntactically structured and JSE does not guarantee that the fragments are syntactically correct. Processing of the input arguments is done by pattern matching, or by arbitrary procedural code. Therefore, JSE supports any syntax extensions that can be represented in the SST, but it is not possible to introduce syntax that does not conform to the lexical syntax of Java. To improve error reporting, JSE tries to maintain the original source code location in the generated code. The source location will be lost if arbitrary manipulations of the SST are performed.

For a more extensive survey of the properties of various systems supporting syntax macros, see [Brabrand & Schwartzbach 2002].

2.6.5 *Lexical Macros*

Lexical macro languages, such as CPP [CPP Manual] and M4 [GNU M4 Website], replace characters or tokens in an input file by a sequence of tokens defined in the macro definition. Macro definitions consist of an identifier of the macro, a fixed number of arguments, and a sequence of characters and references to the arguments of the macro. Lexical macros provide abbreviations at the lexical level of a language. Lexical macro languages do not have any knowledge of the structure of the input file and the context of a macro invocation. This may be considered an advantage, since these macro languages can thus be used for input files of arbitrary languages. On the other hand this is an disadvantage since the macro language cannot use language specific knowledge to make the substitution more reliable. Since the lexical syntax must identify parts of the input that form a macro invocation, the invocation syntax is not liberal enough to allow the extension of languages with domain-specific notations. Using lexical macros is error-prone since the result of the macros can interfere with the context of the invocation and is thus parsed entirely differently than might be expected. Lexical macros are therefore usually loaded with parentheses to enforce correct parsing.

2.6.6 *Metafront*

Metafront [Brabrand et al. 2003] was designed as a more general solution to the parsing issues experienced in metamorphic syntax macros [Brabrand & Schwartzbach 2002]. Metafront applies a novel parsing algorithm: *specificity parsing*. Metafront extends the formalism of context-free grammars with a

separate set of regular terminal languages. The lexical syntax of a language is thus still defined separately from the context-free syntax, but there is not a separate scanner and parser tool. However, Metafront is not a scannerless parser in the sense of not tokenizing the input stream at all. It uses a separate scanner for tokenizing the input file. If at a certain point in the input stream multiple options out of the set of terminal languages are available, then the most specific one is chosen. Specificity parsing thus has a built-in preference for the longest token. Specificity parsing in Metafront does not return to a choicepoint in the input stream; it immediately commits the choice for a certain token. Left recursion is not allowed in syntax definitions to ensure termination. Metafront adds a form of lookahead called *attractors* to solve ambiguities if there is not a most specific token. An attractor specifies after how many successful tokens for a certain production this alternative must be chosen. Attractors are also used to reject certain alternatives.

Because of the attractor disambiguation construct, syntax definitions in Metafront are less concise then syntax definitions in SDF2. Left recursion has to be removed, which is unfortunate. However, Metafront syntax definitions are indeed modular, unlike other parser generators that do not use Earley or GLR. The main goal of Metafront is not to sacrifice performance when there is a need for extensible syntax definitions. Since the current implementation is a prototype that interprets language definitions, it is difficult to determine whether specificity parsing will in practice have significantly better performance than Scannerless Generalized LR parsing.

Metafront also features a transformation language. Metafront transformations are guaranteed to terminate and will transform syntactically correct input to syntactically correct output. It is unclear to us why termination guarantees are important in this context. Guaranteeing termination sacrifices the expressivity and abstraction facilities of the transformation language and we prefer a more expressive language over termination guarantees. In the Stratego/XT project we have developed many complex transformations, in particular program optimizations. Testing transformations has in practice proven to be sufficient to see whether a transformation terminates or not. It would be interesting to separate Metafront in a parsing and transformation component and experiment with the performance of specificity parsing.

2.7 FUTURE WORK

2.7.1 *Application Domains*

There are many opportunities for applying MetaBorg to create domain-specific extensions of Java and other host languages. We summarize some of our ideas to give more insight into the scope of concrete syntax for objects.

In *linguistic reflection* a program can generate new programs at runtime and execute these generated programs as part of its own execution [Kirby et al. 1998]. Linguistic reflective programs are program generators and as such the generator is concerned with the syntactic and semantic correctness of the

generated programs. In [Kirby et al. 1998] programs are generated by constructing a textual representation in strings, which is unclear and error-prone. The authors suggest as further work to generate code using an abstract syntax and place the challenge to have the generator code look as the generated language would normally be written. Our solution for embedding languages using MetaBorg makes this possible by embedding a concrete syntax for the abstract syntax of a programming language.

Linguistic reflection tools targeting a specific platform might even target standardized abstract syntax trees such as the CodeDOM of .NET. These abstract syntax trees can then be accepted directly by a compiler or interpreter, which makes an expensive and error-prone intermediate textual form unnecessary. It might even be possible to directly compile embedded fragments to Java bytecode or the .NET Intermediate Language[5], for example by defining a concrete syntax for the objects used by an existing byte code construction library, such as the Byte Code Engineering Library (BCEL) [BCEL Website].

JJTraveler [Visser 2001] is a visitor combinator framework for Java. Visitor combinators can be used to compose a tree traversal rewriting functionality from basic visitor combinators. The set of basic combinators of JJTraveler is inspired by the strategy primitives of Stratego, a strategic program transformation language where rewriting rules are applied according to user-definable rewriting strategies. However, JJTraveler visitor construction is somewhat verbose compared to the concise notation for strategies in Stratego. In essence JJTraveler provides classes for all strategy operators of Stratego. The embedding of concrete syntax for these strategy operators would be an interesting application of MetaBorg.

Doug Lea's *java.util.concurrent* library introduces useful abstractions for concurrent programming. This library provides utility classes for concurrent programming developed in [Lea 2000]. Other programming languages already have these abstractions over the low-level facilities for concurrent programming as built-in constructs. Domain-specific notations for these abstractions can be developed with MetaBorg as an extension of the Java language.

XQuery is a query language developed by the W3C. It enjoys wide support by researchers and the industry. JSR 225 aims at standardizing an API for evaluating XQuery operations. XQueries are however even less attractive for inclusion in String literals than XPath and SQL statements since the queries tend to be larger and span multiple lines. Obviously, this language needs to be embedded in Java to make static syntactic verification possible and embedding of values from the host language easier and more secure.

2.7.2 *Open Compilers*

Open compilers can improve the integration of MetaBorg based embedded languages in the compilation process. Assimilators can be arbitrarily complex and in some cases they will need to have more detailed semantic information

[5] Indeed, the new open compiler approach of Kats [Kats 2007] is based on embedding bytecode syntax in Java to facilitate code generation and the implementation of language extensions.

on the input program than a structured representation of the input program in an abstract syntax tree provides. We already experience some problems where assimilators needed to know the type of expressions.

Improved integration of the assimilators in the compiler will provide the required information to the assimilator and will improve the analysis that can be performed on programs with a reasonable amount of work. Application of assimilators in different phases of an open compiler (e.g. parsing, semantic analysis, and translation to intermediate code) will improve error reporting since the aspects of assimilation into the host language can be implemented separately for each phase by extending components of the open compiler.

A useful experiment of extending the components of an open compiler is delayed disambiguation in the input programs. In many cases the embedding of a language requires disambiguation because the fragments of the embedded language can be parsed as more than one nonterminal. In the embedding of Java in Java this disambiguation is performed by optionally prefixing the fragments with `exp` and `bstm` to indicate the nonterminal of the fragment. In the embedding of XML in Java the nonterminal of anti-quotations has to be defined since the anti-quotation might refer to character data as well as content. When this explicit disambiguation is required is not always clear and excessive use of disambiguations makes the code less concise.

An interesting solution to this problem is to solve the ambiguities by means of interaction with a type-checker. From a fragment such as `Expr e = expr ⟦ 1 + 2 ⟧` it is easy to see that there is unnecessary duplication of type declarations. If the type-checker of the host language can be extended, then during type-checking the appropriate alternative from a set of ambiguous abstract syntax trees can be chosen. By preserving the ambiguities until the type checking phase they can be handled at this point. This requires the parser to be able to report all possible parses, which SGLR supports. In Chapter 3 we discuss the method of a disambiguating type-checker in detail.

2.8 CONCLUSIONS

Programming language design these days often is about selecting candidates for tasteful inclusion in the core language. Design patterns and application libraries are among the candidates for assimilation into the host language. Extending existing programming languages with new features that used to be provided by libraries has several advantages. The core language can provide an attractive syntax for these features, and tools such as type-checkers, refactoring tools, IDEs and debuggers will (need to) have built-in knowledge of the extensions.

Cω (first called Xen [Meijer & Schulte 2003b, Meijer & Schulte 2003a]) lifts the area of data-access to the language level. To this end, it makes the type system of the host language more general and extends the language of 'object literals' with an alternative XML syntax. Cω provides languages features for iterating, collecting, filtering, and applying functions over data structures. The Xtatic language [Xtatic], a follow-up of XDuce [Hosoya & Pierce 2000], extends

an existing programming language with XML specific support as well, but uses a translation into the host language, which is C#.

However, the growth of a language with more domain-specific constructs is limited by the general application area of the language. Extensions of the language can potentially be applied in *all* code written in this language. Therefore very domain-specific notation will never be accepted to the core language itself. The kitchen sink is only interesting for kitchen related applications. Hence, language extensions tend to abstract over control-flow rather than over data. A domain-specific concrete syntax for objects will usually not be lifted to the host language, since objects are much more application specific than routines.

Syntax macros take domain-specific extension to the other extreme by allowing the programmer to extend the host language within the program itself. This is a very lightweight extension of the language, but is restricted by a host of technical problems. Our MetaBorg method is somewhere in between designing a new language and ad-hoc extensions with syntax macros. Since extensions need to be defined carefully by a domain expert, domain-specific notations are introduced in a separate role. To conclude, MetaBorg is more flexible than Cω in that it does not target a specific domain and MetaBorg allows arbitrary extensions, as opposed to syntax macros. We expect that extensions will be implemented in MetaBorg or similar toolkits before they are assimilated into the host language for good. For those extensions that will never make it there, MetaBorg provides the way to still make them available to application programmers.

ACKNOWLEDGMENTS

Many people have contributed to the development of Stratego/XT. Merijn de Jonge developed the pretty-printing tools of Stratego/XT. Niels Janssen, Rob Vermaas and Jonne van Wijngaarden contributed to the development of XML support. SDF is maintained and further developed at the Centrum voor Wiskunde en Informatica (CWI) and the Eindhoven University of Technology (TU/e) by Mark van den Brand, Giorgios Robert Economopoulos, and Jurgen Vinju. We thank Eelco Dolstra, Jeff Gray, Merijn de Jonge, Martijn Vermaat and the anonymous reviewers of OOPSLA 2004 for providing useful feedback on an earlier version of this chapter. Finally, we thank the users of Stratego/XT and the MetaBorg method for their inspiration.

Generalized Type-Based Disambiguation of Concrete Object Syntax

3

ABSTRACT

In metaprogramming with concrete object syntax, object-level programs are composed from fragments written in concrete syntax. The use of small program fragments in such quotations and the use of meta-level expressions within these fragments (anti-quotation) often leads to ambiguities. This problem is usually solved through explicit disambiguation, resulting in considerable syntactic overhead. A few systems manage to reduce this overhead by using type information during parsing. Since this is hard to achieve with traditional parsing technology, these systems provide specific combinations of meta and object languages, and their implementations are difficult to reuse.

In this chapter, we generalize these approaches and present a *language independent* method for introducing concrete object syntax without requiring explicit disambiguation. The method uses scannerless generalized LR parsing to parse meta programs with embedded object-level fragments, which produces a forest of all possible parses. This forest is reduced to a tree by a disambiguating type-checker for the metalanguage. To validate our method we have developed embeddings of several object languages in Java, including AspectJ and Java itself.

3.1 INTRODUCTION

Meta-level programs analyze, transform, and generate *object-level* programs. It is commonly agreed that such program manipulations are best carried out on a structured representation of the object program in order to achieve compositionality of transformations and to guarantee well-formedness of the resulting program. Furthermore, structured representations support type safety and hygiene more easily. However, the notation for structured representations is usually verbose and rather different from the notations of the language under consideration, rendering it impractical as a syntax for object programs. Using the concrete syntax of the object language as a notation for this structured representation provides the best of both worlds. The metaprogram can be written using the concise, well-known syntax of the object language, while the underlying representation is still structured.

Syntactically checked concrete object syntax is now available in many metaprogramming systems. Syntax macro systems such as <bigwig> [Brabrand & Schwartzbach 2002], code generators such as Jak (JTS/AHEAD) [Batory et al. 1998] and Meta-AspectJ (MAJ) [Zook et al. 2004], and program transformation

systems such as ASF+SDF [van Deursen et al. 1996], DMS [Baxter et al. 2004], Stratego/XT [Visser 2002] and TXL [Cordy et al. 1991] all provide concrete object syntax. Some of these systems are designed for a specific object language, others are configurable for different object languages. In [Visser 2002] we presented a general architecture for introducing concrete syntax for any object language in any metalanguage. 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 [Visser 1997b, van den Brand et al. 2002].

A remaining problem of concrete object syntax is that the syntax of the combined meta and object languages is usually highly ambiguous if the object language is embedded using a single pair of quotation and anti-quotation symbols. Most systems solve this by using a different quotation and anti-quotation symbol for each nonterminal of the object language, leading to considerable syntactic clutter and requiring the metaprogrammer to be intimately familiar with the syntactic structure of the object language. Because of the irregularity of the embedding, the set of syntactic categories that can be quoted and unquoted is usually limited. Moreover, in a language with manifest typing that already requires programmers to declare the types of all variables, the disambiguation of quotations feels redundant. For example, consider the following fragment written in Jak (part of the JTS/AHEAD Tool Suite [Batory et al. 1998]):

```
Stmt stm2 = stm{ if($exp(e1)) { $stm(stm1); }; }stm;
```

Here a statement *stm2* is constructed from an expression *e1* and a statement *stm1*. The syntactic categories of the quotation `stm{...}stm` and the antiquotations `$exp(`*e1*`)` and `$stm(`*stm1*`)` within it are explicitly indicated using the identifiers `stm` and `exp`.

Meta-AspectJ (MAJ) [Zook et al. 2004], an extension of Java for the generation of AspectJ programs, reduces the need for different quotation and anti-quotation symbols by means of a context-sensitive parser, taking variable declarations into account during parsing. For example, in MAJ the Jak fragment above can be written as follows:

```
Stm stm2 = `[ if(#e1) { #stm1 } ]
```

The syntactic categories of the fragment and the variables are inferred from the explicit declaration of their types in the program. Thus, MAJ requires from the programmer less knowledge of the embedding and the syntactic details of the object language. However, the implementation of MAJ is specific to the embedding of AspectJ in Java, and is not easily reusable for embeddings of other languages, due to a number of limitations. First, the scanner for meta and object language is the same, which precludes embedding of languages with a different lexical syntax. Second, it is not possible to extend the metalanguage with concrete object syntax for *multiple* languages, since the implementations of context-sensitive parsing do not compose. Finally, the implementation of parsing and type-checking is tangled, which leads to complex

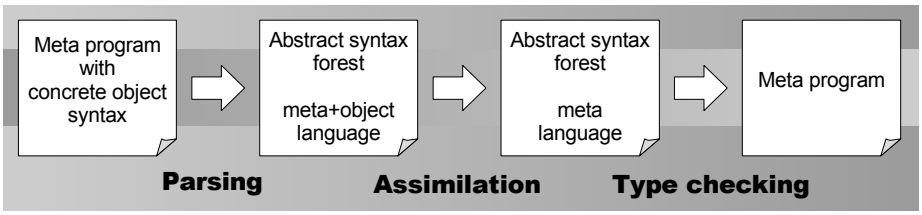


Figure 3.1 Architecture of generalized type-based disambiguation.

and hard to maintain code that has limitations that might surprise users. For example, MAJ cannot always handle overloaded methods that are invoked with quoted arguments.

In this chapter, we describe an extension of our general architecture for concrete object syntax with type-based disambiguation that allows embeddings with minimal syntactic overhead. The main characteristic of our approach is that ambiguities are preserved by the parser and are solved in a separate phase by an extension of a type-checker that operates on an abstract syntax forest. This separation of phases is illustrated in Figure 3.1. As a result, language embedding and assimilation (expansion of embedded object code to the metalanguage) can remain compositional. Therefore, it is easy to add new object languages and to combine object language embeddings. Since ambiguities are solved *after* assimilation, the implementation of disambiguation for a metalanguage is *object language independent*. However, we require that the representation of object programs in the metalanguage is typed and that distinct syntactic categories have a different type in this representation (see Section 3.4.4 and 3.5.2). Disambiguation is achieved by a natural and orthogonal extension of the type system. By separating the issue of disambiguation from the type-checker, we can handle ambiguities in complex typing situations for free, hence reducing the number of exceptions and heuristics. Also, the approach is *not restricted to a single metalanguage*. Disambiguation is implemented as an extension of the type checker of the metalanguage, so the method is restricted to statically typed metalanguages, and not applicable to untyped languages. Furthermore, the method is particularly suitable (and desirable) for languages that use manifest typing (e.g. C, Java, C#). We have no experience with metalanguages using type inference.

We proceed as follows. In the next section we recapitulate the embedding and assimilation of an object language in a metalanguage. In Section 3.3 we examine the ambiguities caused by such embeddings and previous solutions used for them. In Section 3.4 we present a generalized type-based disambiguation method for concrete object syntax. In Section 3.5 we describe our experience with the method in a generic disambiguation implementation for Java as a metalanguage with embeddings of AspectJ and Java itself. In Section 3.6 we discuss previous, related, and future work.

```
37 module JavaJava
38 imports Java-15-Prefixed Java-15
39 exports
40   context-free syntax
41     "⟦" Expr "⟧"      -> MetaExpr {cons("ToMetaExpr")}
42     "#[" MetaExpr "]" -> Expr     {cons("FromMetaExpr")}
```

Figure 3.2 Syntax definition for simple embedding of Java expressions in Java

3.2 METAPROGRAMMING WITH CONCRETE OBJECT SYNTAX

In this section, we recapitulate the general method for adding support for concrete object syntax to a metalanguage, which was presented in [Visser 2002, Bravenboer & Visser 2004 (Chapter 2)]. Introduction of concrete object syntax in a metalanguage requires (1) embedding the syntax of the object language in the meta language and (2) assimilation of the embedded object code fragments to the metalanguage, expressed in terms of the underlying structured representation. The generality of the approach is based on syntax definition in the modular syntax definition formalism SDF for defining the embedding and the transformation language Stratego for the assimilation. We illustrate the approach with the introduction of concrete syntax for Java in Java.

3.2.1 *Embedding*

The embedding of an object language in a metalanguage requires the combination of syntax definitions for both languages. From this combined syntax definition a parser is generated, which is used to parse metaprograms that use concrete object syntax. Thus, the embedding of Java in Java is achieved by the module in Figure 3.2. The module imports [38] the Java syntax twice; once as the metalanguage (`Java-15-prefixed`) and once as the object language (`Java-15`). To avoid confusion between the two languages (or language roles in this case), the module `Java-15-prefixed` prefixes the nonterminals of the metalanguage with '`Meta`' by renaming them in its import declaration of `Java-15`.

Next, to actually integrate the meta and object language, the combination of these syntax definitions is extended with productions that determine the possible transitions from the metalanguage to the object language (quotation) and vice versa (anti-quotation). A *quotation* quotes a fragment of an object-level program and embeds it in a meta-level program. The first production [41] in Figure 3.2 defines that an object-level `Expr` between ⟦ and ⟧ can be used as a meta-level `MetaExpr`. The `cons` annotation in the production declares the constructor to be used in the abstract syntax tree. The following Java statement illustrates the quotation of a Java method call:

```
Expression x = ⟦ resultSet.getInt(4) ⟧
```

The meaning of this statement is Java code for the construction of the abstract syntax tree corresponding to the quoted fragment.

An *anti-quotation* is an escape from a quotation to the meta-level, to splice in pieces of object code computed elsewhere. The second production[42] in Figure 3.2 declares that a `MetaExpr` between `#[` and `]` can be used as an object-level `Expr`. For example, in the following quotation the method argument is an expression *foreignkey* that is determined from some domain specification:

```
Expression x = ⟦ resultSet.getInt(#[ foreignkey ]) ⟧;
```

3.2.2 *Assimilation*

Assimilation transforms a program with embedded object code to a pure meta-level program by translating the embedded fragments to code in the metalanguage that constructs the underlying abstract syntax tree representation. For example, in our Java in Java embedding we use the Eclipse JDT Core DOM [JDT Website] for representing the object programs. Hence, the Java constructs must be translated to invocations of the methods in this API. The following Stratego rewrite rules illustrate the assimilation for some Java language constructs. The first rule translates a return statement, the second rule a method invocation. The Stratego rewrite rules use concrete object syntax as well.

```
Assimilate(rec) :
  ⟦ return; ⟧ -> ⟦ _ast.newReturnStatement() ⟧

Assimilate(rec) :
  ⟦ e.y(e*) ⟧ ->
  ⟦ {| MethodInvocation x = _ast.newMethodInvocation();
       x.setName(~e:<AssimilateId(rec)> y);
       x.setExpression(~e:<rec> e);
       bstm*
     | x |}
  ⟧
  where <newname> "inv" => x
      ; <AssimilateArgs(rec | x)> e* => bstm*
```

In the assimilation rules we use a small extension `{|stmt*|expr|}` of Java, called an eblock, that allows the inclusion of statements in expressions. The value of an eblock is the expression. In the assimilation rules, the *italic* identifiers (e.g. *e*, *y*, and *e**) indicate meta-level variables, a convention we use in all the code examples. `~e:` denotes an anti-quotation where the result is a Java expression. `<s> p` applies the rewriting *s* to the pattern *p*. `s => p` matches the result of *s* to *p*. `newname` creates a fresh, unique, name, which guarantees hygiene in the assimilation. `AssimilateArgs` is a helper strategy that assimilates a list of expressions to arguments of the method invocation.

As an example, consider the result of assimilating the last example above, which illustrates the advantage of concrete syntax.

```
MethodInvocation inv = _ast.newMethodInvocation();
inv.setName(_ast.newSimpleName("getInt"));
inv.setExpression(_ast.newSimpleName("resultSet"));
List<Expression> args = inv.arguments();
args.add(foreignkey);
Expression x = inv;
```

```
1  "⟦" CompilationUnit "⟧" -> MetaExpr {cons("ToMetaExpr")}
2  "⟦" TypeDec         "⟧" -> MetaExpr {cons("ToMetaExpr")}
3  "⟦" BlockStm        "⟧" -> MetaExpr {cons("ToMetaExpr")}
4  "⟦" BlockStm*       "⟧" -> MetaExpr {cons("ToMetaExpr")}

5  "#[" MetaExpr "]"-> ID   {cons("FromMetaExpr")}
6  "#[" MetaExpr "]"-> Expr {cons("FromMetaExpr")}
```

Figure 3.3 Syntax definition for embedding of Java in Java

In the examples of this chapter, the assimilation is embedding specific, since the mapping of the object language to an existing API is inherently embedding specific. However, if there is a fixed correspondence between the syntax definition and the API, then the assimilation can be generic. This is typically the case if the API is generated from the syntax definition using an API generator such as ApiGen [van den Brand et al. 2005].

3.3 AMBIGUITY IN CONCRETE OBJECT SYNTAX

In this section we discuss how ambiguities can arise when using concrete object syntax. Also, we discuss how these ambiguities are handled in related work.

3.3.1 *Causes of Ambiguity*

Lexical State If a separate lexical analysis phase is used to parse a metaprogram, then ambiguities will arise if the lexical syntax of the object language is different from the metalanguage. The set of tokens of both languages cannot just be combined, since the tokens of both languages are only allowed in certain contexts of the source file. For example, `pointcut` is a keyword in embedded AspectJ, but should not be in the surrounding Java code.

Quotation Ambiguous quotations can occur if the same quotation symbols are used for different nonterminals of the object language. If the object code fragment in the quotation can be parsed with both nonterminals, then the quotation itself is ambiguous as well. For example, consider the SDF productions [1] and [2] in Figure 3.3 that define a quotation for a compilation unit and a type declaration. With these two quotation rules, the fragment ⟦ `class Foo { }` ⟧ is ambiguous, since the quoted Java fragment can be parsed as a compilation unit as well as a type declaration. Note that not all quotations are ambiguous: if the object code includes a package declaration or imports, then it cannot be parsed as a type declaration. A similar ambiguity issue occurs if the embedding allows quotation of lists of nonterminals as well as single nonterminals. For example, consider the SDF productions [3] and [4] in Figure 3.3 for quoting block statements. A quotation containing a single statement is now ambiguous, since it can be parsed using both production rules.

Anti-Quotation Similar ambiguity problems occur when using the same anti-quotation symbols for different nonterminals of the object language. For example, consider the anti-quotations [5] and [6] in Figure 3.3 for identifiers and expressions. The anti-quotation in `⟦ #[a] + 3 ⟧` is ambiguous, since `#[a]` can represent an identifier as well as a complete expression.

3.3.2 *Solutions*

Lexical State Most systems use a separate scanner. The consequence is that the lexical analysis must consider lexical states and will often assume fixed quotation symbols to determine the current state. Alternatively, the scanner can interact with the parser to support a more general determination of the lexical state. Some other systems just take the union of the lexical syntax, hence forbidding reserved keywords of the object language in the metalanguage. MAJ also reserves several keywords to work around lexical ambiguities (e.g. `pointcut` is a meta keyword) and some of these keywords are not even part of the object language (e.g. `VarDec` and `args`). ASF+SDF and Stratego both use *scannerless parsing* for parsing meta programs. Lexical ambiguities are not an issue in scannerless parsing, since they inherently only occur if a separate scanner is used.

Explicit Typing Ambiguous quotations and anti-quotations can be solved by requiring explicit disambiguation by using different quotation symbols. For example, JTS uses different quotations for the class example: `prg{...}prg` for compilation units and `cls{...}cls` for class declarations. Stratego uses the same solution, but the disambiguated versions of the quotations are optional: if there is no ambiguity, then the general quotation symbols can be used. For example, `⟦ package foo; class Foo {} ⟧` is not ambiguous (it is a compilation unit), but a plain class declaration requires an explicit disambiguation, e.g. `compilation-unit ⟦ class Foo {} ⟧`.

JTS solves ambiguities between quotations of a single nonterminal and a list of nonterminals in two different ways. First, there are specific quotations for lists, for example `xlst{...}xlst` for the arguments of a method call. Second, some nonterminals only have a single quotation instead of two, where this single quotation always represents a list. In Stratego, list quotations are explicitly disambiguated, e.g. `bstm* ⟦ x = 5; ⟧`.

Context-sensitive Parsing MAJ uses context-sensitive parsing to solve ambiguous quotations and anti-quotations, by using type information at parse-time to infer the type of the quotation or anti-quotation to be parsed. Concerning list quotations, if `infer` is used, MAJ uses a single element if possible and an array if it must be a list. If the type of the variable is declared, then this type is considered. For example, the quotation in the statement `Stmt[] stms = ‘[ x = 4; ];` will be parsed to the construction of an array instead of a single statement. Hence, explicit disambiguation of the quotation itself is not necessary. Unfortunately, MAJ does not implement full support for the type system of Java and uses common interfaces for conceptually different

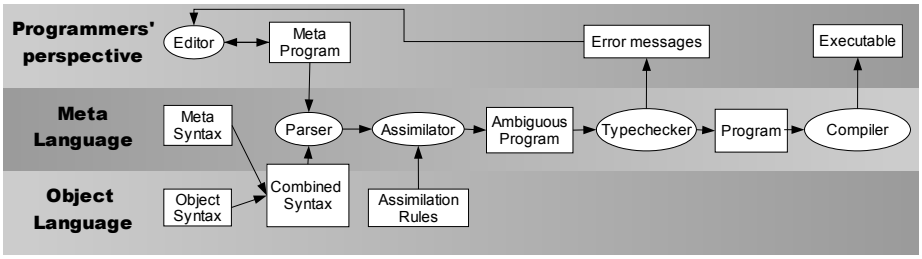


Figure 3.4 Architecture of embedding and assimilation framework with type-based disambiguation.

AST classes to work around issues in the quotation inference. Section 3.5.2 discusses these problems in more detail.

Grammar Specialization ASF+SDF is a system with first order types. It translates this type system to a context-free grammar, thus parsers can be generated that accept only type correct metaprograms. As a result, neither quoting of object fragments, nor anti-quoting of metavariables, nor explicit typing is necessary in ASF+SDF. However, the type system is limited to first order types only. Remaining ambiguities are currently solved by using heuristic disambiguation filters, such as injection count. In [Vinju 2005] a type-based solution for these ambiguities is presented, where grammar generation is no longer necessary.

3.4 GENERALIZED TYPE-BASED DISAMBIGUATION

In summary, quotation and anti-quotation can be used to introduce concrete syntax for object-level program fragments, but need some form of disambiguation. Explicit disambiguation methods introduce syntactic clutter that obscures metaprograms. Reduction of this syntactic clutter can be achieved by using type information for disambiguation. While MAJ does a great job at achieving this for the specific embedding of AspectJ in Java, its implementation is hard to generalize to other object languages and to the combination of multiple object languages, because of the poor compositionality of its context-sensitive parsing algorithm.

In this section, we introduce an alternative approach that generalizes easily to arbitrary object languages. Indeed it is generic in the embedded object language and can easily be transposed to other metalanguages, considering the restrictions on the type system, as mentioned in the introduction. We illustrate the method with the embedding of Java in Java, but stress that the architecture and implementation is object language independent. The basic idea of the approach is to perform *type-based disambiguation of an abstract syntax* forest *after assimilation*. The architecture of our method is illustrated in Figure 3.4. In the rest of this section we describe the elements of the pipeline.

3.4.1 *Syntax Definition and Parsing*

The first stage of the pipeline consists of parsing the metaprogram with a parser generated from the combined syntax definition. This phase preserves all the ambiguities in the metaprogram, by employing generalized LR parsing. The result is a parse *forest*, that is, a compact representation of all possible parses of the program. At points where multiple parses are possible the forest contains *ambiguity nodes* consisting of a set of all alternative parse trees, or in fact forests, since ambiguities can be nested. As a technical note, we actually consider *abstract syntax forests*, that is parse forests with irrelevant information such as whitespace, comments, and literals removed. For example, the Java assignment statement `dec = ⟦ class Foo {} ⟧;` is parsed to the following abstract syntax forest in term notation (where we have elided some details of the structure of class declarations having to do with modifiers and such):

```
Assign(
  ExprName(Id("dec"))
, 1> ToMetaExpr(CompilationUnit(... ClassDec(... Id("Foo") ...) ...))
  2> ToMetaExpr(ClassDec(... Id("Foo") ...))
  3> ToMetaExpr([ ClassDec(... Id("Foo") ...) ])
  4> ToMetaExpr(ClassDecStm(ClassDec(... Id("Foo") ...)))
  5> ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...)) ])
)
```

In this forest it is clear that the right-hand side of the assignment is ambiguous and has five alternative parses. We use the notation `1>...n>` to indicate the alternatives of an ambiguity node. The five alternatives are a compilation unit containing a class declaration, a class declaration on its own, a singleton list of a class body declaration declaring a member class (see Section 3.3.1 for a discussion of ambiguities caused by lists), a block statement of a class declaration local to a method, and a singleton list of the same block statement. The `ToMetaExpr` constructor represents a transition from the metalanguage to the object language (see Figure 3.3). To make the running example a bit more concise, we leave out the alternatives of the block statements (4 and 5). The first three alternatives are sufficient to illustrate all the issues.

3.4.2 *Assimilation*

The second stage in the pipeline is assimilation, i.e., the translation of the embedded language fragments to their implementation in the metalanguage as described in Section 3.2.2. The only difference is that assimilation now transforms a forest instead of a tree. If the assimilation rules are compositional (i.e. the transformed fragments are small) there is no interference between regular assimilation rules and ambiguities, that is, ambiguities are preserved during assimilation. Thus, after assimilation, the abstract syntax forest only contains metalanguage constructs and ambiguity nodes. For example, the following code fragment shows the intermediate result after assimilation of the example above to the Eclipse JDT Core DOM (again some details have been elided).

```
1> {| CompilationUnit cu0 = _ast.newCompilationUnit();
      List<AbstractTypeDeclaration> args1 = cu0.types();
      args1.add(
        {| TypeDeclaration class0 = _ast.newTypeDeclaration();
           class0.setName(_ast.newSimpleName("Foo"));
           ...
         | class0 |});
      ...
    | cu0 |}
2> {| TypeDeclaration class1 = _ast.newTypeDeclaration();
      class1.setName(_ast.newSimpleName("Foo"));
      ...
    | class1 |}
3> {| List<BodyDeclaration> decs0 = new ArrayList<BodyDeclaration>();
      decs0.add(
        {| TypeDeclaration class2 = _ast.newTypeDeclaration();
           class2.setName(_ast.newSimpleName("Foo"));
           ...
         | class2 |});
      ...
    | decs0 |}
```

3.4.3 *Type-Based Disambiguation*

In the final stage of processing the metaprogram, ambiguities are resolved. The disambiguation operates on an abstract syntax forest of the metalanguage without any traces of the object language. Thus, the disambiguation phase does not have to be aware of quotations and anti-quotations, or of their contents. The disambiguation is implemented as an extension of a type-checker for the metalanguage that analyses the abstract syntax forest and eliminates the alternatives that are not type correct. The algorithm for disambiguation is sketched in Figure 3.5. From within the type-checker the disambiguate function is invoked for every node *node* in the abstract syntax forest after typing it.

The disambiguate function distinguishes three cases, which we discuss in reverse order. If the node *node* is not ambiguous it is just returned. If one of the sub-nodes of *node* is ambiguous, its alternatives are lifted to the current node by lift-ambiguity. Its definition states that if n is equal to some ambiguity node within a context `c[.]`, the context is distributed over the ambiguity (We give an example of distribution of an assignment shortly). Finally, if the node *node* is directly ambiguous or after lifting the ambiguities from its sub-nodes, the resolve function is used to resolve the ambiguity.

The resolve function takes an ambiguous node and removes from it all alternatives that are not type correct. This may result in an empty set of alternatives (*#node'* = 0), which indicates a type error, a singleton set (*#node'* = 1), which indicates that the ambiguity is solved, or a set with more than one alternative (*#node'* > 1). In the latter case, if the ambiguity involves a statement or declaration no more context information can be used to select the intended alternative, hence it is reported as an ambiguity error. Otherwise, in the case of an expression, the ambiguity is allowed to be lifted into its parent level, where it may be resolved due to context information.

function disambiguate(*node*) =

```
1  if node is ambiguous then
2    return resolve(node)
3  else if node has ambiguous child then
4    return resolve(lift-ambiguity(node))
5  else
6    return node
```

function resolve(*node*) =

```
1  node' := remove from node all alternatives which are not type correct
2  if #node' = 0 then
3    report type error
4  else if #node' = 1 then
5    return node'
6  else if #node' > 1 then
7    if node' contains a meta statement or declaration then
8      report ambiguity error
9  else
10   return node'
```

function lift-ambiguity(*node*)

```
1  if node = c[ 1> node_1  2> node_2  ...  j> node_j ] then
2    return  1> c[node_1]  2> c[node_2]  ...  j> c[node_j]
```

Figure 3.5 Algorithm for type-based resolution of ambiguities

To illustrate the lifting and elimination of ambiguities, consider the ambiguity between the compilation unit, type declaration and list of body declarations in the assimilated example. If this ambiguous expression occurs in an assignment, i.e.

```
dec = 1> CompilationUnit 2> TypeDeclaration 3> List<BodyDeclaration>
```

then the ambiguity will be lifted out of the assignment (for brevity, the actual expression has been replaced by its type). This will result in a new ambiguity node with three alternatives for this assignment, i.e.

```
1> dec = CompilationUnit 2> dec = TypeDeclaration 3> dec = List<BodyDeclaration>
```

Depending on the type of the variable `dec`, two of these assignments will most likely be eliminated. For example, if the variable has type *TypeDeclaration*, then the *CompilationUnit* and *List<BodyDeclaration>* assignments will be eliminated, since these assignments cannot be typed. Note that this mechanism requires variables to be declared with a reasonably specific type. That is, if the variable `dec` has type `Object` then all the assignments can be typed and an ambiguity error will be reported.

Similarly, ambiguities are lifted out of method invocations: For example,

```
f(1> CompilationUnit 2> TypeDeclaration 3> List<BodyDeclaration>)
```

is lifted to

```
1> f(CompilationUnit) 2> f(TypeDeclaration) 3> f(List<BodyDeclaration>)
```

If `f` is just defined for one of these types, then just one of the invocations can be typed. Thus, the other invocations will be eliminated. On the other hand, if `f` is overloaded or defined for a supertype of two or more of the types, then the ambiguity will be preserved. It might be eliminated later, if the result types of `f` are different. If this is not the case, then an ambiguity will be reported, similar to an ambiguous method invocation in plain Java.

3.4.4 *Explicit Disambiguation*

For cases that are inherently ambiguous or just unclear, explicit disambiguation can be used. Most systems introduce special symbols for this purpose, but due to our integration in the type-checker one may use *casting* to inform the type-checker that something should a have certain type. The implementation of the explicit disambiguation comes for free, since incorrect casts cannot be type-checked. Thus, these alternatives will be eliminated. For example, in our running example a cast to a compilation unit `(CompilationUnit) ⟦ public class Foo {} ⟧` will cause the alternatives to be eliminated. In this way, any object language construct can be disambiguated, not only the ones that the developer of the embedding happens to support.

However, there is a situation where not even casting will help. Our method requires that the underlying structured representation of the object language is typed and that distinct syntactic categories in the object language have a different type in this representation. For example, if the structured representation is a universal data format such as XML or ATerms, then our method will not be able to disambiguate the concrete object syntax, since the different syntactic categories are not represented by different types in the metalanguage. Fortunately, a sufficiently typed representation is preferable anyway, since it would otherwise be possible to construct invalid abstract syntax trees. Note that similar problems occur in a dynamically typed languages. As mentioned in the introduction, our method is most suitable for statically typed languages.

3.5 EXPERIENCE

To exercise the general applicability of our method to the embedding of different object languages, we implemented two large embeddings. Small fragments of the first application have already been presented in several examples: the embedding of Java in Java using assimilation to the Eclipse JDT Core DOM. We call this embedding JavaJava. The second application embeds AspectJ in Java and mimics the object language specific implementation of MAJ. Although AspectJ is a superset of Java, these two applications are quite different, since the embedding of AspectJ assimilates to the MAJ abstract syntax tree. The applications substantiate our claims, but also give some interesting

insights in the limitations and the relation to object language specific implementations.

3.5.1 *JavaJava*

The implementation of JavaJava consists of a syntax definition (small fragment presented in Figure 3.3) and a set of assimilation rules that translate Java 5.0 abstract syntax tree constructs to the Eclipse JDT Core DOM [JDT Website] (examples have been shown in Section 3.2.2).

The mapping from the syntax definition to the Eclipse DOM is natural, since both are based on the Java Language Specification. Furthermore, the DOM is well-designed and uses distinct classes to represent distinct syntactic categories. Because of this, our type-based disambiguation works quite well for JavaJava.

An interesting issue is the type of containers used in the DOM. The DOM uses unparameterized standard Java collections, as opposed to arrays, or type specific containers. So although the DOM itself can *represent* parameterized types in an object program, the DOM implementation itself does not *use* parameterized types. Our disambiguating type-checker would benefit from parameterized collections, by harvesting the additional type information about the elements of a collection (e.g. `List<Expression>`). Fortunately, parameterized types and unparameterized types can be freely mixed, i.e. we can still use parameterized types in metaprograms. However, we prefer a more precisely typed DOM, such that unchecked conversions or explicit casts can be avoided. Note that this shows that a sufficiently typed representation is important for our method.

3.5.2 *Meta-AspectJ*

We developed the embedding of AspectJ in Java to compare our generalized and staged disambiguation solution to a specific implementation, namely MAJ. For this, we also had to study the behaviour of MAJ in more detail. Our syntactic embedding is based on a modular AspectJ and Java syntax definition in SDF and exactly mimics the syntax of MAJ. The syntactic embedding was very easy to implement using SDF: basically we just have to combine the existing syntax definitions in a new module. The syntax definition also supports the explicit disambiguations of MAJ, but these are not really necessary, since casts can be used in our embedding method. For the underlying structured representation we use the MAJ AST.

We learned that our generalized implementation of disambiguation in a separate type-checker has the advantage that it is much easier to implement support for more advanced Java constructs. For example, our implementation fully supports disambiguation of quotations in method invocations by performing complete method overload resolution, which MAJ does not. So, given the following overloaded method declarations:

```
Stmt     foo(CompilationUnit cu) { ... }
JavaExpr foo(ClassDec dec)       { ... }
```

our implementation can disambiguate invocations of the `foo` method that take quoted AspectJ code as an argument:

```
Stmt stmt     = foo(‘[ class MyClass {} ]);
JavaExpr expr = foo(‘[ class MyClass {} ]);
```

On the other hand, MAJ as an object language specific implementation provides some additional, object language specific, functionality that is not available in our generalized implementation. For instance, MAJ supports the conversion of Java (meta-level) values to AspectJ (object-level) expressions. For example, an array can be used as a variable initializer without converting it to the object-level by hand. Unfortunately, this conversion cannot be handled in a generic way, since it is not applicable to all object languages. However, for the specific embedding of Java in Java this could be added to the type-checker (see future work).

We have not implemented the `infer` feature of MAJ, which supports inferring the type of a local variable declaration. Hence, the types of all variables should be declared in our implementation. The `infer` feature itself is not hard to implement, but we would have to introduce heuristics to disambiguate ambiguous expressions, since no type is declared for the variable. MAJ applies such heuristics, for example by choosing a `ClassDec` if the type of the variable is `infer`, even if a `MajCompilationUnit` would also be possible. To work around incorrect choices, similar abstract syntax tree classes implement a common interface. For example, `ClassDec` and `MajCompilationUnit` implement the common interface `CompUnit`. This is a nice example of the problem mentioned in Section 3.4.4: distinct syntactic categories share a common interface. Thus, the declaration `CompUnit c = ‘[class Foo {}];` will result in an ambiguity error in our approach.

3.6 DISCUSSION

3.6.1 *Previous Work*

We use the modular syntax definition formalism SDF [Visser 1997b], with integrated lexical and context-free syntax and declarative syntactic disambiguation constructs. It is implemented using scannerless generalized LR parsing [Visser 1997b, van den Brand et al. 2002]. SDF is developed in the context of the ASF+SDF Meta-Environment [van Deursen et al. 1996], but is used in several other projects such as ELAN [van den Brand & Ringeissen 2000]. Our Stratego/XT [Visser 2004] program transformation system uses SDF for parsing, in particular of metaprograms with concrete object syntax [Visser 2002]. Stratego is not statically type-checked. Therefore, it employs quoting with explicit typing, where necessary.

The ASF+SDF Meta-Environment is a metaprogramming system based on term rewriting. It uses grammar specialization to resolve ambiguities caused by object language fragments. To let the type system of ASF+SDF deal with parametric polymorphism, in [Vinju 2005] a separate disambiguating type-

checker replaces the grammar generation scheme. The solution of [Vinju 2005] instantiates the framework described in this chapter for ASF+SDF.

Section 3.2 describes previous work on hosting arbitrary object languages in any host language [Bravenboer & Visser 2004 (Chapter 2)], which generalizes the approach taken for Stratego [Visser 2002] to any general purpose programming language. In the examples of [Bravenboer & Visser 2004 (Chapter 2)], Java was used as the host language and we also embedded Java as the object language in Java. However, explicit disambiguation was required and an untyped underlying representation was used. The contribution of generalized type-based disambiguation is the introduction of a disambiguating type-checker to remove the need for explicit typing. Moreover, the implementation is generic in the embedded object language. Thus, we obtain a similar notation as found in ASF+SDF, but can handle more than simple first order type systems, and use no disambiguation heuristics.

3.6.2 *Related Work*

The subject of embedding the syntax of object languages into host languages has a long history. The following discussion is meant to position our work more precisely.

Early work on syntactic embeddings revolves around the concept of syntax macros [Leavenworth 1966]. They allow a user to dynamically extend a general purpose programming language with syntactic abstractions. These abstractions are defined in programs themselves. Implementations of this idea have been limited to certain subclasses of grammars, like LL(1) and LALR, as an argument of a fixed macro invocation syntax. Thus, these approaches can not be transferred to our setting of hosting arbitrary object languages.

The work of Aasa [Aasa et al. 1988] in ML is strongly related to our setting. By merging the parsing and type-checking phases for ML, and using a generalized parsing algorithm, this system can cope with arbitrary context-free object languages. It uses a fixed set of quotation and anti-quotation symbols that allow explicit typing to let the user disambiguate in case the type system cannot decide. As opposed to this solution, our approach completely disentangles parsing from type-checking, and allows user defined quotation and anti-quotation symbols.

DMS [Baxter et al. 2004] and TXL [Cordy et al. 1991] are specialized metaprogramming environments similar to ASF+SDF and Stratego/XT. In DMS the user can define AST patterns using concrete syntax, which are quoted and guarded by explicit type declarations. TXL has an intuitive syntax with keywords that limit the scope of object code fragments, instead of quoting symbols that surround every code fragment. Each code fragment, and each first occurrence of a meta variable is explicitly annotated by a type in TXL.

The Jakarta Tool Suite (JTS) [Batory et al. 1998] and the Java Syntax Extender (JSE) [Bachrach & Playford 2001] are Java based solutions for metaprogramming and extensible syntax. Our framework, consisting of scannerless generalized LR parsing and type-based disambiguation, is more general than

the parsing techniques used by these systems. JTS uses explicit quotation and explicit typing, which can be avoided with our framework. Maya [Baker & Hsieh 2002] uses extensible LALR for providing extensible syntax. Multi-dispatch is used to allow multiple implementations of new syntax, where the alternatives have access to the types of the arguments. Unfortunately, a separate scanner and LALR limit the syntactic flexibility. MAJ [Zook et al. 2004] obtains type-based disambiguation for the embedding AspectJ in Java, using context-sensitive parsing. We contribute by disentangling the parser from the type-checker, resulting in an architecture that can handle any context-free object language. Our architecture stays closer to the original Java type system, in order to limit unexpected behavior.

Camlp4 [de Rauglaudre 2003] is a preprocessor for OCaml for the implementation of syntax extensions using an extensible top down recursive descent parser. New language constructs are translated to OCaml code by syntax expanders that are associated to the syntax extensions. Camlp4 provides quotations and anti-quotations to allow the generation of OCaml code using concrete syntax. The contents of quotations is passed as a string to a quotation expander, which can then process the string in arbitrary ways. A default quotation expander can be defined, but all other quotations have to be typed explicitly. The same holds for syntactically ambiguous anti-quotations. As opposed to Maya, the syntax and quotation expanders cannot use type information to decide what code to produce.

The method of disambiguation we use is an instance of a more general language design pattern called "disambiguation filters" [Klint & Visser 1994]. Although there are lightweight methods for filtering ambiguities that are very close to the syntactic level [van den Brand et al. 2002], disambiguation filters can generally not be expressed using context-free parsing. For example, any parser for the C language will use an extra symbol table to disambiguate C programs. Either more computational power is merged in parsers, or separate disambiguation filters are implemented on sets of parse forests [van den Brand et al. 2003]. We prefer the latter approach, because it untangles parsing from abstract syntax tree processing.

The problem of disambiguating embedded object languages is different from disambiguation issues in type-checkers, such as resolution of overloaded methods and operators. First, disambiguation in type-checkers can be done locally, based on the types of the arguments of the expression. Hence, lifting of the ambiguity, an essential part of our algorithm, is not used in such type-checkers. Second, in our approach large fragments of the program can be ambiguous and are represented in completely different ways. For typical ambiguities in programming languages, such as overloaded operators, the alternatives can conveniently be expressed in a single tree.

3.6.3 *Future Work*

We are considering to widen the scope of the framework in two directions. First, we would like to experiment with languages that have other kinds of

type systems, such as languages with type inferencing and languages with union types. Second, the assimilation of an embedded domain-specific language (beyond object languages) often requires more complex transformations of the metaprogram and the object fragments. Applying type-based disambiguation after assimilation is a problem in this case. Extending the type-checker of the host language with object language specific functionality is one of the options to investigate for this purpose. Also, the disambiguating type-checker might be interesting for other applications than disambiguation of embedded concrete object syntax. For example, it could be used for the reclassification of ambiguous names, or even to perform overloading resolution.

Object Language Specific Conversions

Many programming languages define conversions between types that can be applied in assignments, function calls, etc. For example, Java defines several narrowing, widening, boxing and unboxing conversions. In a certain way, quotations can be used to perform additional, object language specific conversions. For example, an identifier can be converted to an expression by just quoting it. Similarly, a type declaration can be converted to a compilation unit.

```
String s = ...;
Expression e = |[ #[x] ]|;
TypeDeclaration classdec = ...;
CompilationUnit cu = |[ #[classdec] ]|
```

To the user of the metalanguage it might be confusing that no actual object language syntax is involved in these assignments, bot the direct assignments `e = s` or `cu = classdec` are not type correct. Furthermore, it is natural to be able to use a list of identifiers as a list of expressions. In our generic framework, we do not support such object language specific features. Unfortunately, implementing such conversion by hand in the metaprogram takes quite some code and makes to code generator less clear.

These object language specific features could be supported in our framework by extending the type-checker of the metalanguage to support such conversions. Thus, the object language specific conversions are added to the built-in conversions of the metalanguage. The conversions could be derived automatically from a syntax definition of the object language, thus making this feature object language independent. Every nonterminal that can produce another nonterminal without any additional syntax is a candidate for an object language specific conversion.

3.7 CONCLUSION

We have extended an existing generic architecture for implementing concrete object syntax. The application of a disambiguating type-checker, that is separate from a generalized parser, is key for providing single quotation and anti-quotation operators without explicit typing. This approach differs from other approaches due to this separation of concerns, which results in object

language independence. It can still handle complex configurations such as Java embedded in Java.

We have validated our design by means of two different realistic embeddings of object languages into Java, and comparing the results to existing systems for meta programming. The instances of our framework consist of meta programming languages that use manifest typing (i.e. Java), combined with object languages that have a well-typed meta representation (e.g., Eclipse JDT Core DOM).

We explicitly do not provide heuristics to automate or infer types, such that the architecture's behavior remains fully declarative and is guaranteed to be compatible with the type system of the metaprogramming language.

ACKNOWLEDGEMENTS

At Utrecht University this research was supported by the NWO/JACQUARD project 638.001.201 TraCE: Transparent Configuration Environments, and at CWI by the Senter ICT-doorbraak project CALCE. We would like to thank Karl Trygve Kalleberg, Eelco Dolstra and the anonymous reviewers of GPCE 2005 for providing detailed feedback.

Preventing Injection Attacks with Syntax Embeddings

4

ABSTRACT

Software written in one language often needs to construct sentences in another language, such as SQL queries, XML output, or shell command invocations. This is almost always done using *unhygienic string manipulation*, the concatenation of constants and client-supplied strings. A client can then supply specially crafted input that causes the constructed sentence to be interpreted in an unintended way, leading to an *injection attack*. We describe a more natural style of programming that yields code that is impervious to injections *by construction*. Our approach embeds the grammars of the *guest languages* (e.g. SQL) into that of the *host language* (e.g. Java) and automatically generates code that maps the embedded language to constructs in the host language that reconstruct the embedded sentences, adding escaping functions where appropriate. This approach is generic, meaning that it can be applied with relative ease to any combination of host and guest languages.

4.1 INTRODUCTION

In this chapter we propose using *syntax embedding* to prevent injection vulnerabilities in a language-independent way. Injections form a very common class of security vulnerabilities [Halfond et al. 2006]. Software written in one language often needs to construct sentences in another language, such as SQL, XQuery, or XPath queries, XML output, or shell command invocations. This is almost always done using *unhygienic string manipulation*, whereby constant and client-supplied strings are concatenated to form the sentence. Consider for example the following piece of server-side Java code that authenticates a remote HTTP user against a database, where getParam() returns a string supplied by the user, for instance through a form field:

```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

On testing, this code may appear to work correctly, but it is vulnerable to a widely known security flaw. For instance, if the user specifies as the password the string `' OR 'x' = 'x`, then the constructed SQL query will be

```
SELECT id FROM users WHERE name = '...' AND password = '' OR 'x' = 'x'
```

The condition in the WHERE-clause is now a tautology. Hence, the password check will always succeed and the user will be granted access, which is an example of an injection attack. The essence of the attack [Su & Wassermann 2006] is that the programmer intended the variable password to serve as an SQL string literal, but a specially crafted value like the one above causes it to be parsed as something else.

Injection attacks are one of the largest classes of security problems, possibly surpassing even buffer overflows[1]. SQL-constructing code is more likely to be vulnerable than not [Halfond & Orso 2005]. But injection attacks occur in many contexts other than SQL query construction in Java, as Figure 4.1 shows. SQL injections happen in any *host language* that dynamically computes SQL queries, such as PHP. Other *guest languages* are equally vulnerabl. For example, programs that build XPath or LDAP queries have the same problem. Likewise, many CGI scripts call the Unix shell with user-supplied data in an unhygienic way that allows arbitrary commands to be executed on the server. They also often send unhygienically constructed HTML to the client, enabling *cross-site scripting* (XSS) attacks that cause inappropriate content to be presented to a user, or malicious JavaScript code to be executed.

Injections can be prevented by *escaping* external input. For SQL string literals, this means that occurrences of the '-character must be doubled. Thus, the query construction above would become ... + escapeSQLStr(password) + ..., where escapeSQLStr performs the expected escaping. However, it is easy to forget to do so, and neither the compiler nor the runtime system can flag the omission of escape calls. There have been a number of proposals to detect unhygienically constructed sentences at runtime (e.g. [Su & Wassermann 2006, Halfond & Orso 2005, Huang et al. 2004]) or using static analysis (e.g. [Livshits & Lam 2005, Xie & Aiken 2006]).

A better solution, from a security perspective, is to use an *API* to build the sentence. Such an API can ensure that injections are impossible *by construction* (e.g. [McClure & Krüger 2005]). For instance, the query above could be expressed using some imaginary SQL-constructing API: SQL query = new Select(..., new Eq(new Var("password"), new Str(password))). The constructor for string literals Str then takes care of escaping. Furthermore, the type system ensures well-formedness of the sentence. But this style of programming is unattractive. It is inconvenient because it creates a cognitive gap between the programmer and the syntax provided by the guest language, which is after all a domain-specific language (DSL) designed to make certain kinds of tasks easier to express. Also, such APIs may not be available and may differ for each language. Finally, documentation and examples are expressed in terms of the concrete syntax of the DSL, not an API.

The approach that we propose in this chapter is to combine the security of using an API with the conceptual ease of string manipulation. We do this by *embedding* the syntax of the guest languages into the syntax of the host

[1] A scan of 168 SecurityFocus vulnerability reports updated in the period April 10–14, 2006 revealed at least 53 injection vulnerabilities: 24 SQL injections, 28 HTML injections, and 1 shell injection. By contrast, there were at least 30 buffer overflows and other memory-related problems.

```
$username = $_GET['username'];
$q = "SELECT * FROM users WHERE username = '" . $username . "'";
executeSQL($q);
```

SQL in PHP: SQL injection

```
String e = "/users[@name='" + name + "' and " +
                  "@password='" + password + "']";
factory.newXPath().evaluate(e, doc);
```

XPath in Java: XPath injection

```
$command = "svn cat \"file name\" -r" . $rev;
system($command);
```

Shell calls in PHP: command injection

```
String topic = getParam("topic");
String query = "SELECT body FROM comments WHERE topic = '" + topic + "'";
ResultSet results = executeQuery(query);
foreach (String body : results)
  println("<tr><td>" + body + "</td></tr>");
```

XML and SQL in Java: XSS vulnerability

Figure 4.1 Examples of unhygienic sentence construction

language, a technique pioneered by metaprogramming [Batory et al. 1998, Zook et al. 2004, Visser 2002, Bravenboer et al. 2005 (Chapter 3)]. For instance, the SQL-in-Java example above becomes

```
SQL q = <| SELECT id FROM users
           WHERE name = ${userName} AND password = ${password} |>;
if (executeQuery(q.toString()).size() == 0) ...
```

That is, the syntax of SQL is embedded directly into the syntax of Java expressions, using the *quotation* `<|...|>` to construct SQL code. Likewise, the *antiquotation* `${...}` embeds Java expressions into SQL to allow composition of SQL code. A preprocessor called an *assimilator* translates code written in this combined language into plain Java code that calls an API *generated* from the grammar of the guest language.

Of course, the idea of embedding a language is not new. For instance, the SQL-92 standard [ISO 1992] already defines an embedding of SQL in host languages such as C. However, these solutions have always been specific to a combination of guest and host languages, requiring considerable work to support other combinations. Examples of other combinations are SQL in a different host language, XPath, SQL, LDAP, and Shell together in the same host language, or XPath together with a scripting language in Java. The core

```
$username = $_GET['username'];
$q = <| SELECT * FROM users WHERE username = ${$username} |>;
executeSQL($q->toString());
```

SQL in PHP

```
XPath e = {- /users[@name=${name} and @password=${password}] -};
factory.newXPath().evaluate(e.toString(), doc);
```

XPath in Java

```
$command = <| svn cat "file name" -r${$rev} |>;
system($command->toString());
```

Shell calls in PHP

```
String topic = getParam("topic");
SQL query = <| SELECT body FROM comments WHERE topic = ${topic} |>;
ResultSet results = executeQuery(query.toString());
foreach (String body : results)
  println(<tr><td>${body}</td></tr>.toString());
```

XML and SQL in Java

Figure 4.2 Examples of hygienic sentence construction

contribution of this chapter is that we show that modular, scannerless parsing formalisms allow such embeddings to be created *generically*. That is, by specifying the grammar of a guest language, we can embed this language in all supported host languages; and by specifying the grammar of a new host language along with an API generator, the new host immediately allows embedding of all guest languages.

Figure 4.2 shows examples of hygienic, secure code corresponding to the unhygienic, insecure examples in Figure 4.1. Since we have grammars for guest languages such as SQL, XPath, and shell, and host languages such as Java and PHP, any combination of embeddings becomes immediately available: e.g. SQL in Java, SQL in PHP, XPath in Java, shell in PHP, LDAP in PHP, XML *and* SQL in Java, SQL *and* LDAP in PHP, and so on.

Contributions The contributions of this chapter are as follows.

- We describe a comprehensive solution to injection attacks that prevents them *by construction*. This style of programming is also more convenient than both string manipulation and high-level APIs. Preventing injection attacks by construction is a fundamentally more secure approach than *detecting* injections at runtime, as the latter is still vulnerable to denial-of-service attacks based on second-order injections.

- The approach is generic in that it can easily be adapted to new host and guest languages. Like a resourceable and retargetable compiler architecture, it takes effort $\Theta(N + M)$ rather than $\Theta(N \times M)$ to support N guest languages in M host languages. This is in contrast to previous work on injections, which addresses specific language combinations (e.g. [McClure & Krüger 2005, Cook & Rai 2005, Gould et al. 2004b, Gould et al. 2004a, Halfond & Orso 2005, Halfond et al. 2006]).

- This genericity is accomplished through a novel application of language embedding [Bravenboer & Visser 2004 (Chapter 2)], which is in turn enabled by modular, scannerless parsing. Genericity is further reached by automatically generating the underlying APIs from the context-free grammars of the guest languages. Finally, the assimilator that translates guest language fragments into API calls is fully generic and can be applied to every host language and arbitrary combinations of guest languages. As a result, *no metaprogramming* is required for adding a new guest language to the system.

- The well-formedness of constructed guest language sentences can be ensured at runtime and the well-formedness of the guest fragments is ensured statically.

- Antiquotations in metaprogramming can easily lead to ambiguities. For instance, in an SQL quotation `<| SELECT id FROM names ${where} |>`, the antiquote `${where}` can have several syntactic sorts (such as a `WHERE`- or `ORDER BY`-clause). Usually, the programmer is required to explicitly disambiguate such antiquotations. We present a novel technique for dealing with ambiguity that relieves the programmer from this burden. This technique makes the previous work in metaprogramming on embedding languages usable for programmers.

We have implemented this approach in a prototype called *StringBorg* (after the *MetaBorg* method [Bravenboer & Visser 2004 (Chapter 2)]), which is available as open source software at `http://www.stringborg.org`. StringBorg is implemented using Stratego/XT [Visser 2004], which provides the *Stratego* language for implementing program transformations, and a collection of tools (*XT*) for the development of transformation systems, based on the modular syntax definition formalism *SDF* [Visser 1997b].

4.2 APPROACH

In the previous section we saw how injection attacks are made possible by composition of sentences using string concatenation. In this section we introduce the design and implementation of the StringBorg approach.

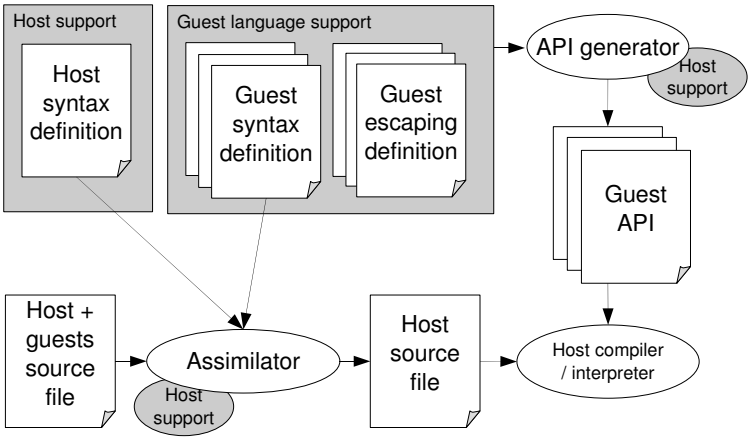


Figure 4.3 Overview of the StringBorg approach

4.2.1 *Overview*

The core problem underlying injection attacks is that a sentence (e.g. an SQL query) is parsed after its construction to a structure that does not correspond to the *intended* grammatical structure [Su & Wassermann 2006]. Unfortunately, in code using unhygienic string concatenation, the intended structure is implicit. Hence, the structure of the resulting query cannot easily be compared to the intended structure. The solution is to make the grammatical structure explicit so that any resulting sentence can be guaranteed to conform to it. StringBorg accomplishes this by parsing the guest language fragments as a pre-processing step and generating code to construct the sentences in a structured way. This ensures well-formedness of the resulting sentence and automatic escaping of antiquoted strings. StringBorg constructs sentences structurally according to the productions of the context-free grammar of the guest language.

Figure 4.3 provides an overview of StringBorg. The syntax of the guest language (e.g. SQL, Shell) is *embedded* in the syntax of the host language (e.g. Java, PHP). The programmer uses this combined syntax for writing programs. The *assimilator* uses the combined syntax definition to parse source files and transforms the embedded guest code to invocations of an API that manages the composition, escaping, and serialization of the guest code sentences. The API is produced fully automatically using an *API generator* that given a grammar of a guest language and an optional escaping definition produces an API in a specified host language. The generated API prevents injection attacks by *always* checking lexical values against the syntax for the lexical category as defined in the syntax definition. The generated API also automatically applies escaping rules to strings that are spliced into guest code using an antiquotation.

```
module SQL
exports
  context-free syntax [1]
    "SELECT" Id* "FROM" Id Where? -> Query
    "WHERE" Expr -> Where
    Expr "=" Expr -> Expr {left}
    String -> Expr
    Id -> Expr
  lexical syntax [2]
    [A-Za-z]+ -> Id
    [A-Za-z0-9\ \"\-\;] -> Char
    "'" ("''" | Char)* "'" -> String
```

Figure 4.4 Syntax definition for subset of SQL

Language Independence StringBorg combines the convenience of string concatenation and concrete syntax with the safety of an API. The main contribution of StringBorg is its genericity in the guest and host languages. That is, it is not restricted to a specific combination of a guest and host language. The language independence is not just methodological, rather the *implementation* of *(1)* support for a guest language is *independent* of the supported host languages, *(2)* the generator is easily retargetable to different host languages, and *(3)* the assimilator is generic in the guest language and almost generic in the host language. The grey boxes in Figure 4.3 indicate syntax definitions necessary to add support for a new host or guest language. The grey ellipses indicate code templates that need to be written to add support for a new host language to the assimilator and generator. These tools are written in the Stratego program transformation language [Visser 2004]. We will review the language independence of StringBorg in more detail in Section 4.2.5.

4.2.2 *Syntax Embedding and Parsing*

Throughout this chapter, we will use a small subset of SQL to illustrate StringBorg, but we stress that neither the approach nor the implementation are specific to SQL. StringBorg uses the modular syntax definition formalism SDF [Visser 1997b] to define the syntax of host and guest languages. Figure 4.4 shows the SDF grammar for our subset of SQL[2]. The SDF module defines the context-free syntax (at point [1]) of simple queries and expressions consisting of string literals, identifiers and equality comparisons. Furthermore, it defines the lexical syntax [2] of string literals. A syntax definition mainly consists of *productions* of the form $s_1 \ldots s_n$ -> s_0, declaring that a phrase of the syntactic category s_0 can be formed by concatenating phrases of categories s_1 ... s_n. Note that SDF definitions are similar to EBNF with the distinction that *(1)* productions are written left-to-right with the 'generating' nonterminal on the right-hand side, and *(2)* both lexical *and* context-free syntax are defined in the same formalism.

[2]The prototype provides a complete syntax definition for SQL.

```
module SQL-Java
imports Java SQL[3]
exports
  context-free syntax
    "<|" Query "|>" -> Expr[[Java]] {quote("SQL")} [4]
    "<|" Expr  "|>" -> Expr[[Java]] {quote("SQL")} [5]
    "${" Expr[[Java]] "}" -> Expr   {antiquote} [6]
    "${" Expr[[Java]] "}" -> String {antiquote} [7]
```

Figure 4.5 Syntax embedding of SQL in Java

```
String topic = getParam("topic");
SQL e = <| topic = ${topic} |>;
SQL q = <| SELECT body FROM comments WHERE ${e} |>
```

Figure 4.6 Composing SQL queries in Java

The use of concrete syntax for a guest language requires the embedding of the guest language in the syntax of the host language. Figure 4.5 illustrates how this is achieved in SDF. First, the syntax definition for SQL-Java imports the syntax definitions of Java and SQL [3]. Next, it adds new productions for *quotations*, i.e. for using the guest language syntax in the host language, and *antiquotations*, i.e. to escape from the guest language to the host language. In this case, the module defines quotations for SQL queries [4] and expressions [5] and antiquotations for SQL expressions [6] and strings [7]. Note that Expr is the nonterminal for SQL expressions imported from the module SQL, while Expr[[Java]] is the nonterminal for Java expressions imported from Java.

Figure 4.6 shows how quotations and antiquotation are used to compose guest sentences at runtime. The first quotation of an SQL expression uses an antiquotation to splice a Java String into the expression. At this point, the Java String will be escaped using the escaping rules for SQL. The SQL expression itself is spliced into the final SQL query. In this way, guest sentences can be composed at runtime not just by inserting string values, but also by using antiquotations for arbitrary syntactic categories of the guest language.

Parsing

Parsing source files that use a combination of a host language and various guest languages is a challenge for many parsing techniques. In our approach, the parser is generated fully automatically from the combined syntax definition, in this case the module in Figure 4.5. The problem with most mainstream parsing techniques is that grammars cannot easily be composed. Grammars for LR or LL parser generators cannot be composed in general, since LR and LL grammars are not closed under composition [Hopcroft et al. 2006]. Furthermore, lexical analyzers do not compose, since the lexical analysis of combinations of languages requires the recognition of a different lexical syntax for different locations of a source file, i.e. the lexical syntax is context-sensitive. For example, a guest language often has different keywords, operators, and

literals than the host language. Lexical analyzers are usually generated from a definition of a set of tokens, which cannot be unified into a single set of tokens for analyzing a combination of languages.

StringBorg is based on the SDF syntax definition formalism, which is implemented using *scannerless generalized LR parsing*. The parser is *scannerless*, which means that no separate lexical analyzer is used. (The lexical and context-free sections of SDF modules are desugared into a single context-free grammar.) In this way, the differences in lexical syntax between the host and guest languages, such as different keywords, operators, and literals, become a non-issue. The parser is based on the *generalized LR* (GLR) algorithm, which supports the full class of context-free grammars, which *is* closed under composition. Since SDF has a feature rich module system, we can *embed* languages in other languages in a very natural way. For an extensive discussion of the issues involved in parsing syntactic language embeddings we refer to [Bravenboer & Visser 2004 (Chapter 2), Bravenboer et al. 2006 (Chapter 5)].

Ambiguities

In applications of syntactic embeddings, ambiguities are an ubiquitous problem. For example, the antiquotation `${topic}` in Figure 4.6 can be interpreted as an SQL expression as well as an SQL string, because the same antiquotation syntax is used for both syntactic categories of the guest language, i.e. there is no way to syntactically distinguish the two antiquotations. In many applications, these ambiguities need to be resolved, either by requiring the programmer to tag the quotations and antiquotations [Batory et al. 1998, Visser 2002] with their syntactic category (e.g., `$str{topic}`), or by using a disambiguating type-checker to select the intended derivation [Zook et al. 2004, Bravenboer et al. 2005 (Chapter 3)]. Both solutions are rather unappealing for StringBorg. Tagging requires the programmer to be intimately familiar with the grammar of the guest language, which seriously affects the usability. A specially crafted disambiguating type-checker would require substantial work for every host language and would only be possible for statically typed languages.

Fortunately, for well-formedness of guest sentences in StringBorg, there is no need to know the exact syntactic category of all values. It is sufficient to verify that the value of an antiquotation actually syntactically 'fits' in the quotation into which it is spliced. StringBorg employs generalized LR parsing to *preserve* the ambiguities by letting the generalized LR parser produce all alternative derivations (i.e. a parse forest). In this way, the programmer does not have to tag quotations and antiquotations and also the approach is easy to implement for arbitrary host languages. In the next section we discuss how the ambiguities are efficiently represented at runtime by objects that can have *multiple* types, corresponding to the possible syntactic categories.

4.2.3 *API Generation*

StringBorg generates for a specific guest language an API that covers all aspects of constructing guest language strings. The API that is generated for a

```
1  public final class L {
2        private String string, Set symbols;
3        private L(String string, Set symbols) {...}
4
5        for each context-free production p : s_1 ... s_n -> s_0 :
6              public static L p(L arg_1, ..., L arg_n) {
7                for each arg_i where s_i is a lexical category:
8                      if arg_i.symbols = {string} then
9                          arg_i := escape_s_i(arg_i)
10                     if !match(dfa(s_i), arg_i) then
11                       throw exception
12
13               if ∀_{1≤i≤n}: s_i ∈ arg_i.symbols then
14                   syms := {s_0}
15                 else
16                   syms := ∅
17                pp := unparse arg_1, ..., arg_n
18               return new L(pp, syms)
19             }
20
21       for each lexical category s_i:
22             public static L literal_s_i(String s) {
23               return new L(s, {s_i})
24             }
25
26             private static L escape_s_i(String s) {
27               pp := escape s according to the escaping definition
28               return new L(pp, {s_i})
29             }
30
31       public static L ambiguity(L arg_1, ..., L arg_m) {
32          pp := ⋃_{i=1}^{m} {arg_i.string | arg_i.symbols ≠ ∅}
33          syms := ⋃_{i=1}^{m} arg_i.symbols
34          return if |pp| = 1 then new L(pp, syms) else new L("", ∅)
35       }
36
37       public static L lift(arg) {
38         if arg is of type L then
39           return arg
40         else
41           return new L(arg, {string})
42       }
43 }
```

Figure 4.7 API generation for Java-like languages

guest language is responsible for preventing injection attacks. That is, even without using the syntax embedding and assimilator, the use of the API guarantees that injection attacks cannot occur. The generated APIs have no runtime dependencies, support ambiguities, and provide support for unparsing, escaping, and checking of lexical values. The generator of the StringBorg prototype comes with back-ends for PHP and Java. Figure 4.7 outlines in pseudocode the API that is generated for Java-like languages from the syntax definition of a guest language, for example from the SQL syntax definition of Figure 4.4. The API generation for other languages such as PHP follows a substantially similar structure and is straightforward to implement. In the pseudocode *italic for each* loops are evaluated at generation-time to generate code for each production or symbol of a grammar.

For a guest language L, a class L is generated, whose instances (objects) represent sentences of L. Each object has two private fields [2]: its string representation and a set of syntactic symbols. The class has no public constructors [3]; instances of L can only be created via static factory methods. For each context-free production of the grammar, there is a corresponding static factory method [6] that constructs an L object. The formal parameters of this method correspond to the list of symbols $s_1 \ldots s_n$ in the left-hand side of the SDF production. For example, for the production `Expr "=" Expr -> Expr` the method `public static SQL` *`newEquality`*`(SQL arg1, SQL arg2)` is generated. *newEquality* is a symbolic name we use in the examples. The actual names of the factory methods are cryptographic hashes of the productions. Literals used in the left-hand side of the production, such as "=", are not passed to the factory methods.

The factory methods first apply automatic escaping to lexical values [9] (Section 4.2.3) and check if the resulting strings match the syntax of the lexical categories [10], using deterministic finite-state automata (DFA). For each lexical category, an automaton is generated from the regular grammar for this category in the syntax definition. The automata are constructed using the BRICS Automaton package [Møller 2005]. For example, this check will result in an error if a string with a newline is spliced into an SQL query, since SQL strings do not allow newlines and SQL does not provide escaping for newlines. The escaping rules are configurable in StringBorg, for example the MySQL dialect uses different escaping rules that do support newlines.

Next, the factory methods check if all the arguments of the methods have the required symbol in their set of symbols [14]. This set of symbols represents the possible syntactic categories of the L instance. For example, the factory method for an SQL `Query` checks that the last argument has the symbol `Where?`. If one of the arguments does not contain the required symbol, then the resulting L instance will have an empty set of symbols, which means that it is invalid. Note that this can only happen in antiquotations, since the syntactic correctness of a literal guest code fragment implies that all arguments will have the appropriate syntactic category in their set of symbols. If this would not be the case, then a parse error would have occurred. The construction of an L instance with an empty set of symbols does not raise an exception because of the requirement to support ambiguities (see Section 4.2.3). For the

same reason, L instances have a set of symbols, instead of just a single one.

Finally, the factory methods reconstruct the sentences of the guest language based on their arguments [17]. The generator analyzes the syntax definition of the guest language to generate the implementation of unparsing in the factory methods. The unparsers insert minimal whitespace between the symbols. Note that the actual construction of the string is hidden in the API and can be optimized by lazy unparsing to create only a single string, after the required interpretation has been determined.

Literals and Escaping

Strings that are used to construct guest sentences can originate literally from guest code templates or can be spliced in using an antiquotation. These two cases have to be handled differently, because literal strings are already escaped, whereas spliced strings are not. For literal strings, the generated API contains methods [22] to construct an L instance of symbol s for each lexical category s. Antiquoted strings are first lifted [37] to an L instance with a symbol that indicates that this is an unescaped string. Such an L instance is later used as an argument of a factory method, where it will be escaped according to the escaping rules of the lexical category. The escaping rules that need to be applied depend on the lexical category, so the strings are not immediately escaped in the lift method. In both cases, the L instances are checked by the factory methods using an DFA for the lexical category, whether they are literal or antiquoted strings. Lexical L instances are never used directly, but are only used to construct other L instances. If they are used directly, then an exception will be thrown, because this error could be a vulnerability.

The implementation of escaping [26] cannot be derived from the syntax definition, which defines the *syntax* of the escapes, but not the corresponding characters. Hand-written code for escaping strings is not an option, since preferably the escaping should be host language independent: if escaping functions have to be implemented for every particular combination of a guest and host language, then more effort than $\Theta(N + M)$ is required to support N guest languages in M host languages. To make the implementation generic, the API generator accepts an escaping definition, written in a small domain-specific language. Figure 4.8 shows two examples of escaping definitions. For SQL string literals single quotes have to be escaped using a single quote. For example, for the string `' OR 1=1` the `escapeString` method produces the safe String `''' OR 1=1'`. For XML attribute values within double quotes, several characters have to be replaced with an entity reference. The conversion for XML attribute values does not define a prefix and suffix because this embedding uses antiquotation *inside* double-quoted attribute values (string interpolation). The escape rules are optional in the configuration file, so plain conversions from host strings to lexical values can be defined as well.

Ambiguities

The API supports ambiguities in quotations and antiquotations by unifying the alternative representations to a single L instance. This is possible in String-

```
conversion string -> String {
  prefix "\'";
  suffix "\'";
  escape {
    [\'] -> "\'\'";
  }
}

conversion string -> DoubleQuotedString {
  escape {
    [\<] -> "&lt;";
    [\&] -> "&amp;";
    [\"] -> "&quot;";
  }
}
```

Figure 4.8 SQL string and XML attribute value escaping definitions

Borg because it does not matter syntactically which alternative is intended (i.e. they represent the same string). The generated method ambiguity [31] takes an arbitrary number of L arguments and composes them into a single L instance. The symbol set of the resulting L instance is the union of all the symbols of the alternatives [33]. The string of the new L instance can be the string of an arbitrary *well-formed* L instance (they are all the same), but to make this precise, we still test the cardinality [34] of the set of strings [32]. Note that some alternatives may have no symbols at all, which means that they are not well-formed. The filtering that needs to be performed in ambiguity is the reason for not throwing an exception earlier: to enable filtering of the alternatives, it is necessary to temporarily allow L instances that do not have any valid syntactic category. The toString method throws an exception when applied to such invalid L instances to guarantee that they are not used to produce a guest sentence.

Retargetable API Generation

The generator has been designed to be retargetable to different host languages by separating the implementation in a generic front-end, which produces an abstract representation of an API, and a host language specific back-end. For each host language, code templates need to be provided for generating automata, escaping, and pretty-printing. For the Java and PHP back-ends the templates amount to 420 and 400 lines of code, respectively.

To make the implementation of a new back-end as lightweight as possible, the generator back-ends produces *parse trees* of the host API, which can be unparsed to a source API without the need for a pretty-printer for the host language. In this way, only a syntax definition of the host language is required.

Programmer Protection

In injection attacks, the user of the system is the person the systems needs be protected against. Yet, if an API is present, but programmers still use strings to 'quickly' compose a sentence, then a potential enemy is the laziness of the programmer. In the case of the generative Java APIs, the constructor of the

L class is private, ensuring that it is not possible to create valid *L* instances from raw Java strings without the appropriate checks. Also, the *L* class is final to disallow subclassing. This level of safety is not available in all languages. Another issue is that access to the string-based API for evaluating guest sentences is usually still available. The detection of classical string-based use of such a library does not require complex static analysis. Arguments to the library interface should get the string value of the guest program directly from the API, for example `executeQuery(q.toString())`, where `q` is an instance of *L*.

4.2.4 *Assimilation*

The embedding of guest language syntax in the host language syntax enables parsing of sources using the combined syntax. The next step is to transform the quoted fragments to calls to the APIs for the guest languages. This transformation is called *assimilation*, as it assimilates the guest language into the host language [Bravenboer & Visser 2004 (Chapter 2)]. As an example of this transformation, Figure 4.9 shows a simple SQL quotation in Java and PHP, a sketch of the parse tree, and the result of assimilation. The sketch of the parse tree in Figure 4.9 presents the productions (see Figure 4.4) in italic using symbolic names. The argument of `quote` is an SQL fragment. The arguments of `antiquotes` are pieces of literal Java code. The example leaves out many details of the real parse tree format, which is a complete description of how the productions of the syntax definition are applied to produce the original source program, including literals, layout and comments. The result of assimilation is a one-to-one mapping from the parse tree to invocations of factory methods in the generated API. It illustrates an ambiguity and the lifting of antiquoted strings to SQL.

The assimilator operates on the full parse tree of the source program. Thus, the assimilator is layout preserving and like the API generator does not require a pretty-printer for the host language. The assimilator is fully *generic in the guest language*, i.e. there is no guest language specific code at all. This is possible because all the information about the guest language is already in the parse tree and the mapping from the syntax definition to the API is fixed, since the API is generated. This makes it easy to map the quoted guest code to the factory methods, which correspond directly to production applications.

In the actual APIs generated by StringBorg, the names of the factory methods are cryptographic hashes of the productions instead of symbolic names such as `newStringExpr`. This ensures the uniqueness of method names. Figure 4.5 shows that the productions for quotations are annotated with the name of the guest language. The assimilator uses this information to invoke methods of the appropriate factory, i.e. SQL in this example. Hence, by design the assimilator can deal with *combinations* of guest languages in a single source file.

Similar to the API generator, the assimilator is split in a front-end and a host language specific back-end. The front-end assimilates the embedded guest code to an *abstract* language that describes the factory method invocations, ambiguities, quotations, and antiquotations in a way that makes it trivial for

```
SQL e = <| topic = ${topic} |>;
```

⇒ *(parsing)*

```
LocalVarDecStm(
  ...
, VarDec(
    Id("e")
  , quote(
      Equality(
        IdExpr(Id("topic"))
      , amb([
          StringExpr(antiquote(ExprName("topic")))
        , antiquote(ExprName("topic"))
        ])
      ))))
```

⇒ *(assimilation)*

```
SQL e = SQL.newEquality(
          SQL.newIdExpr(SQL.newId(SQL.literalId("topic")))
        , SQL.ambiguity(
            SQL.newStringExpr(
              SQL.lift(topic)
            )
          , SQL.lift(topic)
          )
        )
```

```
$e = <| topic = ${$topic} |>;
```

⇒ *(parsing)*

```
Assign(
  Var("e")
, quote(
    Equality(
      IdExpr(Id("topic"))
    , amb([
        StringExpr(
          antiquote(Var("topic"))
        )
      , antiquote(Var("topic"))
      ])
    )))
```

⇒ *(assimilation)*

```
$e = SQL::newEquality(
       SQL::newIdExpr(SQL::newId(SQL::literalId("topic")))
     , SQL::ambiguity(
         SQL::newStringExpr(
           SQL::lift($topic)
         )
       , SQL::lift($topic)
       )
     )
```

Figure 4.9 Assimilation of SQL in Java and SQL in PHP

the back-end to generate host language specific code. Hence, the assimilator is easy to retarget (for Java only 48 lines of code, for PHP 44).

4.2.5 *Summary of Language Independence*

The genericity of our approach is important for making the method viable for practical use. Our goal is to make it possible to have a market of guest language embeddings that can be used in any host language *and* can be combined by users *without any metaprogramming experience*, just like programmers can already combine arbitrary libraries in a single program. StringBorg is even more generic: the implementation of a guest language is available to all supported host languages. To summarize the effort required to implement support for new guest and host languages:

- For adding a new *guest language*, no metaprogramming is required. No pretty-printer for the guest language is necessary. The assimilator is not modified to deal with a new guest language. To add support for a new guest language, one must only define its syntax, configure the escaping of literals, and define the quotation and antiquotations, all in a host language independent way.
- For adding a new *host language*, a syntax definition for the language is required. For the API generator and assimilator, only simple code templates need to be provided to their back-ends. No pretty-printer for the host language is necessary.
- For a *combination of guest languages* in a host language, no additional work is required. From the generic embeddings of the guest languages a parser can be generated fully automatically. The assimilator is designed to handle multiple embedded guest languages and the API generator does not need to be applied to combinations of guest languages: it is applied to their individual syntax definitions.

4.3 DISCUSSION

We have implemented StringBorg back-ends for the host languages Java and PHP and experimented with several (combinations of) guest languages: SQL, XPath, Shell, and XML. Our method guarantees *by construction* that injection attacks cannot occur, assuming, of course, that the API generator is correct. For evaluation, the *usability* of our method is more important: how difficult is it to rewrite an existing application to use StringBorg? We extracted use-cases of the patterns in which SQL queries and HTML responses are typically constructed from a number of web applications available from gotocode.com. Both sentences that are constructed all at once (i.e. in a single string-concatenating expression) and sentences that are constructed dynamically are fully supported. Thanks to the user-friendly support for ambiguities in StringBorg, the programmer does not need to learn disambiguation tags for quotations and antiquotations.

4.3.1 *Static versus dynamic type-checking*

The StringBorg-generated API we have presented checks *dynamically* if guest sentences are composed correctly. However, we have also implemented a Java back-end that performs these checks *statically* by using the Java type system. In this back-end every syntactic category of the guest language is represented by its own class and these are used as the return and parameter types of factory methods. Static type-checking has two major disadvantages for usability: *(1)* the programmer has to know all these syntactic categories and their mapping to types of the host language and *(2)* no ambiguities are allowed, which makes the syntax embedding more difficult to use. Obviously, the advantage is that static checking provides more static guarantees, but it is important to observe that this is not a *security* advantage. That is, both the statically and dynamically typed back-ends guarantee *statically* that an injection attack cannot occur. The dynamic or static type-checking only checks for *programming* errors, not for problems with input provided by the user. The generated APIs will never throw an 'injection attack exception'; the exceptions that can occur are either related to illegal characters in the input (e.g. the newline in SQL) or programming errors. The last category of exceptions does not depend on particular inputs, but only on execution paths, which are easier to detect using testing.

4.3.2 *Prevented classes of injection attacks*

A wide range of injection attack techniques are in use. Halfond et al. have proposed a classification of SQL injection attacks [Halfond et al. 2006]. For example, attacks can be classified by injection *mechanism* or the *intent* of the attack. We now discuss how our method deals with the classes of injection attacks identified by Halfond et al.

Injection through user input is the mechanism of using specially crafted user input to construct a query that has a different parse tree than originally intended. StringBorg prevents these attacks by checking the syntax of lexical values and automatic escaping of *all* strings.

Injection through cookies differs from injection through user input by exploiting input from cookies, which are sometimes naively assumed to be controlled by a web application. StringBorg checks and escapes *all* strings, irrespective of their origin, thus disabling this injection mechanism as well.

Injection through server variables employs yet another origin of strings, such as HTTP headers. Similar to attacks through cookies, these attacks are prevented since StringBorg escapes *all* strings.

Second-order injection attacks indirectly perform the attack by first introducing a malicious input in the system (e.g. a database), which is used later as the input of an affected query. Again, these attacks are prevented since StringBorg checks and escapes *all* strings, whether they originate directly from the user or not.

Tautology based attacks use an injection mechanism to craft a query where the condition always evaluates to true. StringBorg prevents the *mechanisms* of injection attacks from being applied, which implies that crafting tautologies is impossible.

Union query attacks are related to tautologies, but allow access to different tables than the ones originally involved in the query. Similar to tautology attacks, StringBorg prevents the mechanisms that are used.

Piggy-backed queries are malicious queries added to be executed in addition to the original query, for example by terminating an SQL query statement using a semicolon and adding a malicious one. Again, StringBorg prevents the mechanisms that are used.

Illegal query attacks are used to trigger syntax, type or logical errors. This often results in an error report that reveals information about possible exploits. StringBorg only throws an exception if an input string contains invalid characters that could not be escaped. StringBorg disables the construction of syntactically invalid queries.

Thanks to the prevention of injections, methods for triggering type and logical errors are disabled as well. The only exception is an embedding that allows conversion of input strings to table and column names (which is not the case in our embeddings). It is advisable to disallow this conversion and only allow literal table and column names. In general, allowing users to input identifiers can introduce a plenitude of options for manipulating the intended semantics of the constructed guest sentence (see also semantic injection attacks).

Inference attacks are related to illegal query attacks. They can be applied if a site is protected not to show error messages. By observing the success or failure of queries, the setup of the database can indirectly still be examined. The prevention of inference attacks does not differ from illegal query attacks.

Stored procedure attacks are a class of all known attacks applied to stored procedures. If stored procedures compose queries based on user input, then the same method for structured construction should be applied.

Alternate encoding attacks avoid detection and prevention of an attack by concealing the actual query in a different syntax or character encoding, which tricks the detection and prevention techniques into interpreting the query in a different way than the actual processor of the guest language does. In all known embeddings, StringBorg prevents encoding attacks since the encoding itself is escaped and lexical strings are checked syntactically.

However, due to the genericity of our method, it is not guaranteed that encoding attacks are prevented for *all* guest languages. For example, Java features Unicode escapes that can be used for *any* input character,

not just in string literals. If Java were used as a guest language, then Java's Unicode escapes can be used to terminate a string literal and inject code. This is currently not caught by our lexical checking, since the DFA does not unescape the Unicode escape. The fundamental reason for this problem is that the current set of definitions does not fully specify the language. The unicode escapes are basically a different surface language. The syntactic meaning of such unicode escapes is not formally defined in the syntax definition. This can be solved in several ways. *(1)* The escape sequence can be escaped. We do this in all of our embeddings, but this makes the escaping rules important for *security*, which was not the case until now. *(2)* Unescaping rules could be defined next to escape rules and applied before escaping and checking strings. *(3)* The syntax definition of the guest language can be restricted not to support Unicode escape sequences at all. *(4)* The syntax definition formalism could be extended to support lexical escape sequences.

The use of unexpected character encodings (not escape sequences) is another mechanism to hide an attack. For example, PostgreSQL was recently affected by an injection problem with multibyte character encodings [3]. This issue is host language specific and depends on the way strings are handled by the string data types that are used. This is beyond the scope of prevention techniques that check the *syntax* of queries.

Semantic injection attacks are a theoretical class of attacks that go beyond the current *syntactic* injection attacks by crafting a guest sentence that syntactically has the intended structure, but semantically has an unintended meaning. This theoretic class of attacks has not been mentioned in existing classifications due to the restricted expressiveness of the query languages that are studied. An example of such an attack could be the unintended capturing of a variable name, which is a well-known issue in metaprogramming. This attack vector becomes relevant for embeddings of languages that feature variable bindings, for example an embedding of JavaScript in Java.

StringBorg does not protect against semantic injection attacks, since the protection is based on syntax definitions. Similar to prevention of illegal query attacks, the embedding of a guest language can be restricted to only allow the conversion of strings to literals and not to identifiers. This guarantees that variable capture attacks are not possible. In general, allowing users to input identifiers can introduce a plenitude of options for manipulating the intended semantics of the constructed guest sentence. Moreover, it is usually unnatural to allow users to specify identifiers, since identifiers and variable bindings are not something the user is aware of, i.e. it is not the kind of variability in a query to leave to the user to specify.

The current formal definition of the essence of command injection attacks [Su & Wassermann 2006] is restricted to syntactic injection attacks,

[3] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-2314`

which illustrates that the scope of injection attacks is still unclear. The lack of a precise definition of an injection attack makes it impossible to formally prove that injection attacks cannot occur at all. In practice, many factors are involved that can defeat theoretic soundness proofs.

4.3.3 *External queries*

Some applications use queries that are not composed and executed directly in source code, but rather stored in files or written to a database and executed later. The safety of our method is based on the fact that injection attacks are impossible by construction. To make this work, all queries executed by a system have to be constructed, in one way or another, in a structured way. This applies to external queries as well. If the external queries are constructed by the program itself, then our method can be used: there is no need to directly execute a query after constructing it. If other tools are involved in the construction of the queries (potentially used by attackers), then the same structured way of query composition has to be used in these tools.

If the external query is complete (i.e. is not composed further), then it can just be executed as is. If the query needs to be composed further, then it needs to be converted to an L instance, the representation used by the API. The query could be parsed at runtime, but this has some runtime overhead, and complicates the portability of the method to other host languages, since a parser written in this host language has to be available for every guest language. Fortunately, parsing is not necessary, since StringBorg APIs are basically wrapper classes for producing strings. Hence, it is sufficient to have a typed representation of the external query string. To support importing guest code fragments from strings, an API could provide *unsafe* methods, which convert strings to the representation used by the API. For example, a conditional expression of a query could be constructed, unparsed to a string, stored in a database, and loaded later as a conditional expression, without parsing it.

```
SQL c = <| name = $str{s} |>;
String s = c.toString();
SQL c1 = SQL.unsafeExpr(s);
SQL q = <| SELECT * FROM users WHERE ${c1}; |>;
```

Clearly, the unsafe methods could be abused by a programmer and should only be used for strings for which it is absolutely guaranteed that they have been constructed in a safe way. Currently, the StringBorg API generator does not generate unsafe methods.

4.3.4 *Generalized parsing techniques*

Syntactic limitations StringBorg relies heavily on modular syntax definition and parser generation, implemented by SDF and scannerless generalized LR parsing. This requires the syntax of the host as well as the guest language to be expressible in a context-free grammar. Unfortunately, some languages do not have such a context-free grammar. For example, SDF does not support

languages with an indentation rule (Haskell, Python). A potential solution to the problem of indentation rules is to parse these programs using an ambiguous grammar or add basic features for context-sensitive languages to the parser. The StringBorg assimilator already supports ambiguous syntax definition for the host language.

Error reporting For generalized parsing techniques to be accepted in production compilers, the quality of error messages is most important. The current error reporting of scannerless generalized LR parsing is rather Spartan; the parser only gives the line and column numbers where parsing fails. Research on error reporting of scannerless and generalized LR parsers is necessary to make generalized parsing techniques applicable in production environments.

Efficient parser composition In the StringBorg prototype, a parser needs to be generated for every combination of host and guest languages. Parser generation is too expensive to do as part of the compilation of the program that uses this combination of languages, so parsers are currently generated separately. This is not difficult to do, yet it has some impact on the 'plugin experience' of StringBorg. To improve this, we expect to present parse table plugins in future work, which enables efficient composition of scannerless generalized LR parsers. Chapter 6 on *parse table composition* is an important step in this direction.

4.3.5 *Disambiguation design space*

In applications of syntactic embeddings, ambiguities are an ubiquitous problem. For example, the antiquotation in `SELECT * FROM users ${e}` could refer to an SQL where, group-by, having, or order-by clause. The underlying problem of this ambiguity is that the same antiquotation syntax (in this case `${...}`) is used for several syntactic sorts of the guest language. Similarly, a quotation can represent multiple syntactic sorts if the same quotation syntax is used for all of them. For example, an SQL quotation `<| Name |>` could be intended as a 'select item' (e.g. `SELECT Name FROM`), but it could also be a 'row constructor' (e.g. `WHERE Name = 'Foo'`).

The most common solution for resolving ambiguities is to disambiguate explicitly by using different quotation and antiquotation symbols for distinct syntactic sorts (e.g. [Batory et al. 1998, Visser 2002]). Usually, this has to be done by the programmer by explicitly tagging the quotations and antiquotations with the intended sort of their content. This is rather unappealing, since it requires the programmer to be very familiar with the structure of the grammar of the guest language and the quotations used for its syntactic sorts. Another solution is to restrict the number of quotations and antiquotations to avoid ambiguities. For example, if the quotations and antiquotations of SQL are restricted to statements, conditional expressions, and literals, then ambiguities do not arise, because these sorts syntactically exclude each other.

In this paper, we have used runtime disambiguation of ambiguities. However, for the Java back-end with static type-checking this is not possible, since the exact type of a sentence needs to be known at compile-time. In this alternative implementation, we have used quotations and antiquotations without tags for the most common language constructs. But, explicit tagging is used for the less common constructs, or for constructs that are inherently ambiguous, such as the antiquotation of the optional where clause of a query expression. In the static back-end, this solution has been chosen mostly for pragmatic reasons, since more user-friendly solutions to the problem of ambiguities are already available, which we will discuss next. In the following example of the generation of a query with an optional order-by clause, more explicit disambiguation is necessary than usual:

```
Option<OrderByClause> e; Stm stm;
if (...)
  e = ORDER BY? <| ORDER BY User |>;
else
  e = ORDER BY? <| |>;
stm = <| SELECT * FROM users ORDER BY? ${e}; |>;
```

The first and the second quotations need to be explicitly disambiguated using the syntax ORDER BY? to indicate that the content of the quotation is an optional order-by clause. Clearly, the second quotation is ambiguous because all optional clauses can be produced by the empty string. The first quotation distinguishes the *optional* order-by clause from a plain, non-optional order-by clause. The antiquotation, which is ambiguous with all other optional clauses of a query, is disambiguated using the same ORDER BY? syntax.

Type-based disambiguation

The type system of the host language can be used to disambiguate the embedded code fragments. In the order-by example, the type of the variable `e` already indicates that the type of the right-hand side of both assignments should be optional order-by clauses. Similarly, the type of `e` indicates that the antiquotation `${e}` in the quoted query refers to an optional order-by. This method of type-based disambiguation is supported by Meta-AspectJ [Zook et al. 2004], a language for generating AspectJ programs using quotes and antiquotes, and has later been generalized [Bravenboer et al. 2005 (Chapter 3)] by employing generalized LR parsing. The main idea of the generalized approach is to preserve the ambiguities by letting the generalized LR parser produce all alternatives (i.e. a parse forest), followed by a disambiguating type-checker that operates on a *forest* of host programs. This generalized approach is guest language independent, but obviously it is not host language independent, since it requires a specially crafted host type-checker. The disambiguating type-checker we have developed for Java can immediately be applied to the Java applications of StringBorg, but using this approach has great influence on the required infrastructure per host language and is only applicable to statically typed languages.

4.4 RELATED WORK

Injection attacks have attracted a great deal of attention in recent years, and consequently there has been a substantial amount of research in developing techniques to counter them. Our approach differs from the work discussed below in either or both of two ways:

- It is generic over a large number of host and guest languages, rather than being tied to a specific combination such as SQL in Java.
- It prevents injections *by construction* rather than detecting them in existing code.

We emphasize that the present work does not obviate the need for static or dynamic analysis techniques, as they enable *existing* programs written in a traditional style to be secured. The present approach, on the other hand, provides a fundamentally safer way to develop *new* programs that need to construct guest language sentences.

4.4.1 *Explicit escaping and filtering*

The standard response to injection attacks is to tell developers to either diligently escape all user-supplied strings, or to filter out malicious inputs. Filtering can be done by rejecting known bad inputs, an approach that is unlikely to capture all bad inputs (see e.g. [Maor & Shulman 2004]); or by accepting only those inputs that match a very specific "good" pattern, e.g., that contain only certain safe characters. The latter approach has the disadvantage that it may unduly restrict users, e.g., by not allowing user names with apostrophes such as O'Brien). Both escaping and filtering suffers from the fundamental flaw that they require developers to never forget to insert the appropriate code. As with buffer overflows, relying on programmers to "get it right" every time is a recipe for disaster.

4.4.2 *APIs*

SQL DOM [McClure & Krüger 2005] makes SQL safe by hiding the SQL query construction behind an API that ensures that string literals are properly escaped by construction. However, SQL DOM goes beyond the API that we generate from the SQL grammar: it is not merely a "static" API to build SQL abstract syntax trees, but rather is *generated* from a specific database schema. Thus, it can statically ensure that all queries are well-typed with respect to that database schema. Clearly, this is a valuable property. It is important to note, however, that whether a query is ill-typed is in most cases not determined by user input. If a query produced by some code path is well-typed with respect to its schema for some input, then it is likely to be well-typed for all inputs. This is not the case for the hygiene of string concatenation: a concatenation that produces correct results for some inputs may very well fail for others, namely those that contain unescaped characters.

A somewhat similar approach is Safe Query Objects [Cook & Rai 2005], which allows queries to be defined in plain Java expressions, which are compiled using OpenJava into the necessary JDO calls. This can be viewed as embedding a convenient syntax for queries, namely Java expressions, into a host language, which happens to be Java also; the assimilation is the translation into JDO calls. Like SQL DOM, HaskellDB [Leijen & Meijer 1999], provides type safety with respect to the database schema. This API can also ensure proper escaping. These approaches have the downside of introducing a cognitive distance from the SQL language, and are specific to a particular host language and a domain of guest languages (namely query languages).

4.4.3 *LINQ*

Syntactic hygiene is an important aspect of Haskell Server Pages [Meijer & van Velzen 2001], Cω [Bierman et al. 2005] and its successor LINQ. All three provide XML literals, enabling XML output generation in a sound way. The fact that the latter two provide XML literals and an SQL-like query syntax to languages such as Visual Basic illustrates the desire to have embedded syntax for output and query generation. Similar to Safe Query Objects in OpenJava, LINQ allows host expressions to be converted implicitly to an expression tree that can be processed in arbitrary ways. LINQ is not extensible, however, in that it is not possible to plug in the syntax of other guest languages.

4.4.4 *Static analysis techniques*

JDBC Checker [Gould et al. 2004b, Gould et al. 2004a] statically checks that SQL queries built through string concatenation in Java are type-correct. It does so by building a model of the ways in which the query can be built through data-flow analysis, and then comparing that against the database schema. While this work did not address injection attacks, it should be possible to extend this approach to either discover those sites where escape functions should be called, or modifying the code to add those calls automatically. A tool that uses static analysis to find various kinds of injections is described in [Huang et al. 2004]. Xie and Aiken [Xie & Aiken 2006] developed an interprocedural static analysis algorithm for PHP and apply it to SQL injections.

Livshits and Lam [Livshits & Lam 2005] describe a general approach that allows unsafe code to be identified through a specification of code patterns for the sources of "tainted" data, consuming functions such as executeQuery ("sinks") which must not be reached by tainted data, and propagators of tainted data (e.g., string concatenation functions). However, the fact that tainted data can flow from a source to a sink is only a security problem if the data is not validated, so user inspection may be necessary to determine whether an injection is in fact possible.

All static analysis approaches require a substantial effort to apply them to a different host language, due to, e.g., the complexity in implementing the precise data flow semantics of the language.

AMNESIA [Halfond & Orso 2005] statically builds an automaton corresponding with the ways in which query strings can be constructed. Nodes in the automaton are terminals in the language, and special nodes representing external user input. At runtime, each full SQL query is matched against the automaton. In the case of an injection, the query will almost certainly not be accepted by the automaton as additional terminals are present that do not occur in the automaton. In general, any approach that attempts to check for injections in string concatenating code cannot be both sound and complete due to the undecidability of string analysis [Christensen et al. 2003], but the scenarios under which AMNESIA reports a false negative are unlikely to occur in real code.

Any approach involving static analysis takes considerable effort to port to another host language. For instance, JDBC Checker and AMNESIA use the Java String Analysis library [Christensen et al. 2003] to track string concatenations. Implementing such a library for a different language would be a non-trivial undertaking, much more difficult than writing an SDF grammar and API generator rules for the language.

WASP [Halfond et al. 2006] prevents SQL injections by keeping track for each character in a string whether it is "trusted" (e.g., originates from a constant in the source). If SQL tokens containing untrusted characters contain unsafe characters (such as a '), the query is rejected. While WASP is very effective at preventing injections, it is not trivial to port the taint-tracking implementation to other host languages.

SqlCheck prevents injection attacks by wrapping user input in special markers, e.g., (|*s*|). (A similar approach is described in [Buehrer et al. 2005].) The grammar of the guest language is then augmented by accepting the markers around certain symbols in the grammar, e.g, so that it accepts (|'*s*'|)' in SQL wherever string literals are accepted. An injection attack would then fail to parse since in, e.g., ... WHERE password = (|" OR 'x' = 'x'|) there is no production that allows an arbitrary condition inside the markers. The markers are assumed to be special strings that the client cannot produce. If the client can do so, an injection is still possible.

The weakness in SqlCheck is its assumption that the client will not be able to produce the magic marker symbols. The paper argues that by choosing the markers as sufficiently long random strings, the chance of a malicious client guessing the markers can be minimised. However, it is tenuous to assume that markers will not be leaked: for instance, web applications have an unfortunate tendency to "echo" SQL queries to the user if an error occurs. Thus, it may be quite easy for the user to trick the web application into revealing its markers.

A more fundamental problem of runtime approaches such as AMNESIA, WASP and SqlCheck is that, at runtime, they can only flag errors and prevent them from escalating into a full security compromise. But since there is no way to dynamically *recover* from the error condition, a denial-of-service attack is still possible. Consider for example the XSS-attack in Figure 4.1, where a

string containing XML injections is inserted in a database and subsequently presented to each user. In this case, if the XML injection is first detected during page generation, no user will be able to view the page anymore, receiving a server error instead. So in the case of higher-order attacks, it is not enough to detect injections: they must not be possible at all. In the hygienic approach, the malicious string will be escaped according to the XML syntax and will trigger neither an error nor a security problem.

4.4.6 *SQL-specific techniques*

The SQL-92 standard [ISO 1992] defines embeddings for various host languages, but the implementation of these embeddings is highly specific to each host language. Its equivalent of antiquotations is syntactically heavy, requiring "shared variables" between the host and guest to be declared explicitly. Queries cannot be constructed dynamically (at least not hygienically), which may explain why this approach is not widely used.

Prepared statements allow an SQL query to be constructed safely. A prepared statement contains *placeholders* (or *dynamic parameter specifications*) that are replaced hygienically by the SQL query processor; e.g., `SELECT * FROM users WHERE name = ?` has a single placeholder. Placeholders are an inconvenient antiquotation mechanism, since the programmer must ensure that the arguments in the SQL processor call match up with the placeholders, which may be tricky in dynamically generated queries with a variable number of placeholders. In addition, programmers frequently abuse prepared statements and compute the prepared statement unhygienically, rather than passing a constant.

The use of *stored procedures* prevents injection attacks, provided that the stored procedure is called in a safe way [Anley 2002a]. Unfortunately, as with prepared statements, stored procedures are sometimes called in an unhygienic way, negating the approach [Anley 2002b].

4.4.7 *MetaBorg*

The method presented in this chapter is an extended application of our previous work on concrete object syntax, or *MetaBorg* [Bravenboer & Visser 2004 (Chapter 2)], which makes the use of libraries more convenient by providing an embedded domain-specific syntax for using them. For MetaBorg, we motivated the use of the scannerless generalized LR algorithm for parsing embedded domain-specific languages and the Stratego program transformation language for assimilation of the embedded code to the host language. Compared to this earlier work, the StringBorg method presented in this chapter is more generic, thanks to the independence of the support for guest and host languages. Also, the usability of embeddings has been improved considerably by supporting ambiguities, which makes the method applicable for use by programmers without metaprogramming experience. For StringBorg, the assimilation does not translate to an existing API, but to the StringBorg

API generated by the system itself. This makes the assimilation generic for all embedded guest languages. In that sense, this chapter describes "identity assimilations": the embedded guest language and its underlying API are simply used to reconstruct the embedded guest sentences as host language strings, but in such a way that well-formedness is guaranteed. The existence of this identity mapping is what makes the system generic and easy to extend with new guest languages. The StringBorg API is generated automatically from the grammar of the guest language and covers all aspects of generating strings for the guest language.

4.5 CONCLUSION

As noted in the introduction, injection attacks are one of the largest classes of security problems, possibly surpassing even buffer overflows. Modern programming languages already defend against the latter; the work presented here protects against the former. The main advantage over previous approaches is that it makes injections impossible by construction, and that it is generic — it is not necessary to produce APIs and assimilators for each element of the cross-product of host and guest languages {Java, C#, PHP, Perl, ...} × {SQL, JDOQL, HQL, EJBQL, OQL, XML, HTML, XPath, XQuery, Shell, ...}, but only to perform a relatively small amount of work for each host and guest language.

Since different languages are good for different things, it is important to help programmers plug them together. In the case of dynamically generated sentences such as SQL queries or shell invocations, the main challenge is to ensure that this plugging happens in a grammatically well-formed way. Indeed, given the interest in technologies such as LINQ, it appears clear that there is a great deal of enthusiasm for embedding languages such as SQL and XML. However, such embeddings are generally done in a rather *ad hoc* way. It would be a good thing if the languages of the future supported modularity "out of the box". Modular syntax definition makes it possible to accomplish this goal in an efficient and principled way.

ACKNOWLEDGMENTS

This research was supported by NWO/JACQUARD project 638.001.201, *TraCE: Transparent Configuration Environments*. The PHP grammar used by our StringBorg prototype was developed by Eric Bouwers, sponsored by the *Google Summer of Code 2006*. We thank Mark van den Brand, Giorgios Robert Economopoulos, Jurgen Vinju, and the rest of the SDF/SGLR team at CWI for their work on SDF. We thank the anonymous reviewers of GPCE 2007 for providing useful feedback.

Declarative, Formal, and Extensible Syntax Definition for AspectJ

5

ABSTRACT

Aspect-Oriented Programming (AOP) is attracting attention from both research and industry, as illustrated by the ever-growing popularity of AspectJ, the *de facto* standard AOP extension of Java. From a compiler construction perspective, AspectJ is interesting as it is a typical example of a *compositional language*, i.e. a language composed of a number of separate languages with different syntactic styles: in addition to plain Java, AspectJ includes a language for defining pointcuts and one for defining advices. Language composition represents a non-trivial challenge for conventional parsing techniques. First, combining several languages with different lexical syntax leads to considerable complexity in the lexical states to be processed. Second, as new language features for AOP are being explored, many research proposals are concerned with *further extending* the AspectJ language, resulting in a need for an extensible syntax definition.

This chapter shows how *scannerless parsing* elegantly addresses the issues encountered by conventional techniques when parsing AspectJ. We present the design of a modular, extensible, and formal definition of the lexical and context-free aspects of the AspectJ syntax in the Syntax Definition Formalism SDF, which is implemented by a scannerless, generalized LR parser (SGLR). We introduce *grammar mixins* as a novel application of SDF's modularity features, which allows the declarative definition of different keyword policies and combination of extensions. We illustrate the modular extensibility of our definition with syntax extensions taken from current research on aspect languages. Finally, benchmarks show the reasonable performance of scannerless generalized LR parsing for this grammar.

5.1 INTRODUCTION

"A language that is used will be changed" to paraphrase Lehman's first law of software evolution [Lehman 1980]. Lehman's laws of software evolution apply to programming languages as they apply to other software systems. While the rate of change is high in the early years of a language, even standardized languages are subject to change. The Java language alone provides good examples of a variety of language evolution scenarios. Language designers do not get it right the first time around (e.g. enumerations and generics). Programming patterns emerge that are so common that they can be supported directly by the language (annotations, `foreach`, crosscutting concerns). The environ-

ment in which the language is used changes and poses new requirements (e.g. JSP for programming dynamic webpages). Finally, modern languages tend to become conglomerates of languages with different styles (e.g. LINQ, E4X, and the embedding of XML in XJ).

The *risks* of software evolution, such as reduced maintainability, understandability, and extensibility, apply to language evolution as well. While experiments are conducted with the implementation, the actual language definition diverges from the documented specification, and it becomes harder to understand what the language is. With the growing complexity of a language, further improvements and extensions become harder and harder to make. These risks especially apply to language conglomerates, where interactions between language components with different styles become very complex.

AspectJ [Kiczales et al. 2001], the *de facto* standard aspect-oriented programming language, provides a good case in point. While the official `ajc` compiler for AspectJ extends the mainstream Eclipse compiler for Java and has a large user base, the aspect-oriented paradigm is still actively being researched; there are many proposals for further improvements and extensions (e.g. [Masuhara & Kawauchi 2003, Sakurai et al. 2004, Bodden & Stolz, Tanter et al. 2006, Allan et al. 2005, Ongkingco et al. 2006, Harbulot & Gurd 2006]). The AspectBench Compiler `abc` [Avgustinov et al. 2005] provides an alternative implementation that is geared to experimentation with and development of new aspect-oriented language features.

AspectJ adds support to Java for modularization of crosscutting concerns, which are specified as separate *aspects*. Aspects contain *advice* to modify the program flow at certain points, called *join points*. AspectJ extends Java with a sub-language for expressing *pointcuts*, i.e. predicates over the execution of a program that determine when the aspect should apply, and *advices*, i.e. method bodies implementing the action that the aspect should undertake. The pointcut language has a syntax that is quite different from the base language. This complicates the parsing of AspectJ, since its lexical syntax is *context-sensitive*. This is a problem for scanners, which are oblivious to context. The parsers of the `ajc` and `abc` compilers choose different solutions for these problems. The `abc` parser uses a stateful scanner [Hendren et al. 2004], while the `ajc` compiler uses a handwritten parser for parsing pointcut expressions. For both parsers the result is an *operational*, rather than declarative, implementation of the AspectJ syntax, in particular the lexical syntax, for which the correctness and completeness are hard to verify, and that is difficult to modify and extend.

In this chapter, we present a declarative, formal, and extensible syntax definition of AspectJ. The syntax definition is *formal and declarative* in the sense that *all* aspects of the language are defined by means of grammar rules. The syntax definition is *modular and extensible* in the sense that the definition consists of a series of modules that define the syntax of the 'component' languages separately. AspectJ is defined as an extension to a syntax definition of Java 5, which can and has been further extended with experimental aspect features.

We proceed as follows. First we give brief introductions to concepts of

parsing (Section 5.2) and aspect-oriented programming (Section 5.3). To explain the contribution of our approach we examine in Section 5.4 the issues that must be addressed in a parser for AspectJ and discuss how the parser implementations of `ajc` (Section 5.5) and `abc` (Section 5.6), two state-of-the-art compilers for AspectJ, solve these issues. In Section 5.8 we present the design of a syntax definition for AspectJ that defines its lexical as well as context-free syntax, overcoming these issues. Our AspectJ syntax definition is based on the syntax definition formalism SDF2 [Visser 1997b] and its implementation with scannerless generalized LR parsing (SGLR) [Visser 1997a, van den Brand et al. 2002]. The combination of scannerless [Salomon & Cormack 1989, Salomon & Cormack 1995] and generalized LR [Tomita 1985] parsing supports the full class of context-free grammars and integrates the scanner and parser. Due to these foundations, the definition elegantly deals with the extension and embedding of the Java language, the problems of context-sensitive lexical syntax, and the different keyword policies of `ajc` and `abc`. For the latter we introduce *grammar mixins*, a novel application of SDF's modularity features.

In Section 5.9 we examine the extensibility of `ajc` and `abc` and we show how grammar mixins can be used to create and combine extensions of the declarative syntax definition. In Section 5.10 we discuss the performance of our implementation. While LALR parsing with a separate scanner is guaranteed to be linear in the length of the input, the theoretical complexity of generalized LR parsing depends on the grammar [Rekers 1992]. However, obtaining a LALR grammar is often a non-trivial task and context-sensitive lexical syntax further complicates matters. The benchmark compares the performance of the parser generated from our syntax definition to `ajc`, `abc`, and ANTLR. We conclude with a discussion of previous, related, and future work. In particular, we analyze why SGLR is not yet in widespread use, and discuss research issues to be addressed to change this.

The contributions of this chapter are:

- An in-depth analysis of the intricacies of parsing AspectJ and how this is achieved in mainstream compilers, compromising extensibility;
- A declarative and formal definition of the context-free *and* lexical syntax of AspectJ;
- A modular formalization of keyword policies as applied by the `ajc` and `abc` AspectJ compilers;
- An account of the application of scannerless parsing to elegantly deal with context-sensitive lexical syntax;
- A demonstration of the extensibility of our AspectJ syntax definition;
- A mixin-like mechanism for combining syntactic extensions and instantiating sub-languages for use in different contexts;
- A case study showing the applicability of scannerless generalized LR parsing to complex programming languages.

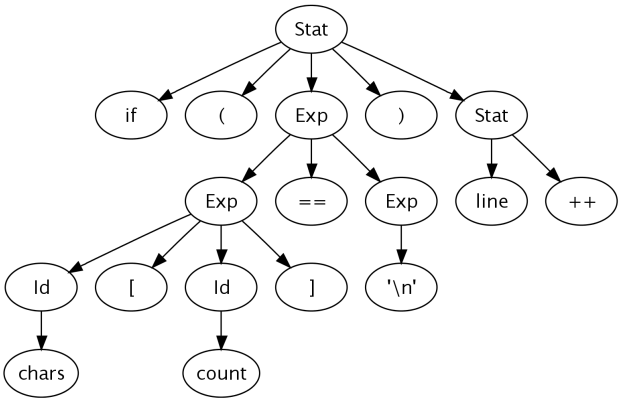


Figure 5.1 Simplified parse tree for a Java statement

AVAILABILITY The AspectJ syntax definition and parser are available as part of AspectJ-front, which is open source (LGPL) and available at `http://aspectj.syntax-definition.org`.

5.2 SCANNING AND PARSING

In this section we review the basic concepts of the conventional parser architecture using a separate scanner for tokenization and compare it to *scannerless* parsing, in which the parser reads characters directly. A parser transforms a list of characters (the program text) into a structured representation (a parse tree). For example, Figure 5.1 shows a (simplified) parse tree for the Java statement `if (chars[count] == '\n') line++;`. Parse trees are a better representation for language processing tools such as compilers than plain text strings.

5.2.1 *Tokenization or Scanning*

Conventional parsers divide the work between the proper parser, which recognizes the tree structure in a text, and a *tokenizer* or *scanner*, which divides the list of characters that make up the program text into a list of *tokens*. For example, the Java statement

```
if (chars[count] == '\n') line++;
```

is divided into tokens as follows

```
if ␣ ( chars [ count ] ␣ == ␣ '\n' ) ␣ line ++ ;
```

Figure 5.2 illustrates the collaboration between scanner and parser. The parser building a parse tree requests tokens from the scanner, which reads characters from the input string.

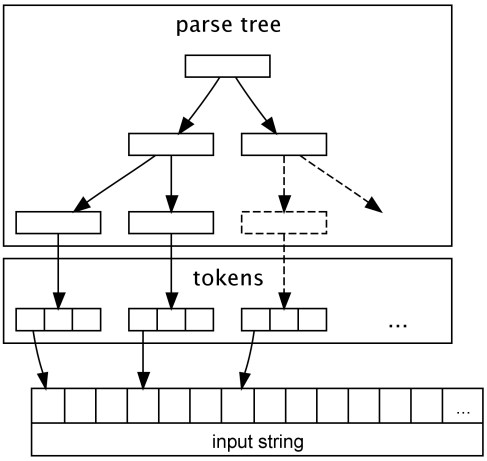


Figure 5.2 A scanner-based parser uses a scanner to partition the input string into tokens, which become the leafs of parse trees.

The reason for the division is the use of different techniques for the implementation of tokenization and parsing. Tokens can be recognized using a deterministic finite automaton (DFA), while parsers for recursive structures need a pushdown automaton, i.e. a stack. Furthermore, tokenization reduces the number of items the parser has to consider; long sequences of characters are reduced to a single token, and whitespace and comments are usually ignored by the parser.

5.2.2 *Scanner and Parser Generators*

Scanners and parsers can be generated from formal definitions of the lexical and context-free syntax of a language. Scanners are generated from regular expressions describing the tokens of the language and parsers are generated from context-free grammars (BNF). Conventional parser generators such as YACC, Bison, and CUP accept only a restricted class of context-free grammars such as LL, LR, or LALR. The advantage is that the complexity of parsers for such grammars is linear in the size of the input. Furthermore, grammars in these classes are not ambiguous; only one interpretation for any string is possible. The fact that a grammar does not belong in a certain class shows up as conflicts in the parse table. For example, in the case of LR parsing, shift-reduce and reduce-reduce conflicts indicate that the parser cannot make a decision about how to proceed based on the provided lookahead information. Solving such conflicts requires rewriting the grammar and sometimes changing the syntax of the language. Also, restricted classes of context-free grammars are not closed under composition.

5.2.3 *Lexical Context*

The uniform tokenization of the input string by means of regular expressions can be problematic, since the scanner does not consider the context in which a token occurs. This means that a particular sequence of characters is interpreted as the same token everywhere in the program text. For syntactically simple languages that have been designed with this restriction in mind this is not a problem. However, modern languages tend to become combinations of 'domain-specific' languages, each providing syntax appropriate for its domain. Because of the limitations of the ASCII character set, the same characters may play different roles in these languages.

One solution that is employed is the use of *lexical state*; the scanner operates in different modes depending on the context. This requires state switching when entering and leaving a context, and may require interaction between the scanner and the parser.

5.2.4 *Programmatic Parsers*

Another solution to bypass the restrictions posed by scanner and parser generators is the use of programmatic 'handwritten' parsers, usually according to the *recursive descent* (topdown parsing) approach. The advantage is that it is easy to escape the rigor of the parsing algorithm and customize it where necessary. Possible customizations are to drive tokenization from the parser to deal with lexical context or to provide customized error handling. A disadvantage of programmatic parsers is that a parser does not provide a declarative specification of the language; conversely, a formal grammar can serve as both documentation and implementation. Also, parser implementations are usually much larger in terms of lines of code, with all the implications for maintainability.

5.2.5 *Scannerless Generalized LR Parsing*

A *scannerless* parser does not make use of a separate scanner to tokenize the input [Salomon & Cormack 1995]; the parser directly reads the characters of the input string. Instead of a separate specification of lexical and context-free syntax as is customary in scanner-based parsing, a single grammar is used that defines all aspects of the language. Although there is no conceptual difference with scanner-based parsing, scannerless parsing is not in common use because it does not work with conventional parser generators. A grammar that describes the lexical as well as the context-free syntax of a language does not usually fit in the grammar classes supported by parser generators. The problem is that these algorithms need to make a decision on the first (few) token(s) in the input. In the case of scannerless parsing a decision may only be made after reading an unbounded number of characters. This problem is solved by the use of *Generalized LR* (GLR) parsing. GLR parsers use a parse table generated by a normal LR parser table generator, e.g. LALR(1) or SLR(1). At points in the input where the parser encounters a shift-reduce

or reduce-reduce conflict, there are multiple possible continuations. In that case a GLR parser simulates the execution of all possible LR parses in parallel. Scannerless GLR (SGLR) parsing adds a few disambiguation techniques to GLR parsing to make it suitable for scannerless parsing [Visser 1997a, Visser 1997b, van den Brand et al. 2002]. *Follow restrictions* define longest match disambiguation and *reject productions* express reserved word policies.

An advantage of SGLR parsing is that it deals naturally with the problem of lexical context. Rather than parsing a lexical entity in isolation, as is done with regular expressions, the parsing context acts naturally as lexical state. Thus, the same sequence of characters can be interpreted differently in different parts of a program.

In the following sections we closely examine the differences between scanner-based and scannerless parsing, by studying state-of-the-art implementations of parsers for AspectJ. In Section 5.4 we analyze the properties of the parsers of the `ajc` and `abc` compilers for AspectJ. In Section 5.8 we discuss a syntax definition for AspectJ using the declarative syntax definition formalism SDF2.

5.3 A QUICK INTRODUCTION TO ASPECTJ

To understand the examples and issues we discuss in this chapter, it is important to be somewhat familiar with the *syntactic structure* of an AspectJ program. This section briefly discusses the various constructs of AspectJ. (In this chapter we focus on the pointcut-advice mechanism of AspectJ.) Knowledge of their *semantics* is not necessary. For a more extensive account of the AspectJ language we refer to [AspectJ].

Figure 5.3 shows an AspectJ aspect for caching executions of the `calc` method of Fibonacci objects. It shows the concise syntax for defining pointcuts, an `around` advice, and how this is mixed with normal Java code (AspectJ keywords are in emphasized bold).

Aspect Declarations

Aspects can be declared, similar to Java classes, either as top-level entities or nested in Java classes [8]. An aspect declaration consists of a number of pointcut declarations and advices, as well as standard Java members (e.g. the `cache` field [9]).

Pointcuts

A pointcut is the specification of a pattern of join points of interest to a given aspect. Join points here are events in the dynamic execution of a program, e.g. a method call or an access to an object field. As such, the pointcut language of AspectJ is really a separate *domain-specific language* for identifying join points.

A pointcut is specified using a number of pointcut designators. Primitive pointcut designators refer to different kinds of operations in the execution of a program. For instance, `execution` refers to the execution of a method [11] — which method(s) is specified by giving a *method pattern* as explained below.

```
public aspect Caching { [8]
  private Map<Integer, Integer> cache = [9]
    new HashMap<Integer, Integer>();

  pointcut cached(int value): [10]
    execution(* Fib.calc(int)) && args(value); [11]

  int around(int value): cached(value) { [12]
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value); [13]
      cache.put(value, result);
      return result;
    }
  }
}
```

Figure 5.3 A sample caching aspect in AspectJ

Furthermore, some pointcut designators are used either to further restrict a pointcut, or to *bind* some values to pointcut formal parameters. In Figure 5.3, the pointcut is given a name (a *named pointcut*) and exposes one parameter of type `int` [10], which is bound via the `args` pointcut designator to the value of the argument to `calc` method executions [11].

Advice

Advice are pieces of code to execute when an associated pointcut matches. This piece of code, which is similar to a Java method body, can be executed before, after, or around the intercepted join point based on the *advice kind*. Since the caching aspect may actually replace the execution of `calc`, it is declared to be of the around kind [12]. As a consequence, its return type (`int`) must be specified. The caching advice is associated to the `cached` *named pointcut*, and it is parameterized by the value of the argument to `calc`. Within an around advice body, calling `proceed` results in the intercepted join point to be executed [13].

Patterns

Fundamental to the pointcut language of AspectJ are *patterns*. A *name pattern* is used to denote names (method names, field names, type names) in a declarative fashion, using wildcards such as `*` and `?`. A *type pattern* is used to denote types, e.g. `int` matches the primitive type `int`, while `A+` matches object type `A` and all its subtypes (`+`). A *method pattern* as in the `execution` pointcut designator in [11] identifies matching method signatures: a return type pattern (`*` in [11]), a declaring type pattern (`Fib` in [11]), a name pattern (`calc` in [11]), and then type patterns for the parameters (`int` in [11]).

In this section we give an overview of some of the challenges of parsing AspectJ. The overview is based on an analysis of the AspectJ language and a review of the source of the scanner and parser of the two major AspectJ implementations: the official AspectJ compiler `ajc`, and the `abc` compiler from the AspectBench Compiler project [Avgustinov et al. 2005]. The scanner and the parser of `abc` have partially been documented in [Hendren et al. 2004]. The purpose of this overview is to show that the parsers of the major implementations of AspectJ are not based on a declarative and complete definition of the language, which leads to minor differences between the two compilers and a lack of clarity about the exact language that each recognizes, as well as parsers that are not easy to maintain and extend.

The main source of the issues in parsing AspectJ is the difference between the lexical syntax of different parts of an AspectJ source file. Conventionally, parsers use a separate *scanner* (or *lexer*) for lexical analysis that breaks up the input character stream into a list of tokens, such as identifiers, literals, layout, and specific keywords such as `class`, `public`, and `try`. Usually this tokenization is applied uniformly to the text of a program, so at every offset in the input, all the same tokens of the language can be recognized by the scanner. However, this does not apply to AspectJ, which is in fact more like a mixture of three languages. Regular Java code, aspect declarations, and pointcut expressions each have a different lexical syntax.

For example, in Java, `get*` is an identifier followed by a multiplication operator, while in a pointcut expression it represents an *identifier pattern* that matches any identifier with prefix `get`. In the first case, the scanner should produce the tokens `get` `*`, while in the second case a single token `get*` would be expected. Similarly, the `+` in the pointcut `call(Foo+.new())` is not an addition operator, but a *subtype pattern* that matches any subclass of `Foo`. To complicate matters, Java code can also occur within a pointcut definition. For instance, the `if(...)` pointcut designator takes as an argument a plain Java expression.

The languages involved in AspectJ also have different *keywords*. Depending on the AspectJ implementation, these keywords might be *reserved* or not. For `ajc`, most keywords are not reserved, since at most places they are explicitly allowed as identifiers in the grammar. For example, `aspect` is a keyword, but it is allowed as the name of a local variable. Similarly, `around` is not allowed as the name of a method in an aspect declaration, but it is in regular Java code. On the other hand, `around` is allowed as the name of a local variable in regular Java as well as in aspect declarations. The `abc` compiler uses a different keyword policy. For example, `before` is a *keyword* in the context of an aspect declaration, but is an *identifier* in Java code *and* in pointcut expressions. In both compilers, pointcut expression keywords, such as `execution` and `get`, are allowed as elements of a pointcut name pattern, e.g. `Foo.execution` is a valid name pattern, and so is `get*`.

Hence, an AspectJ compiler needs to consider the *context* of a sequence of

characters to decide what kind of token they represent. Next, we discuss in detail how `abc` and `ajc` parse AspectJ.

5.5 THE AJC SCANNER AND PARSER

The official AspectJ compiler[1], `ajc`, extends the Eclipse compiler for Java, which is developed as part of the Eclipse Java Development Tools (JDT) [JDT Website]. The parser of `ajc` roughly consists of three components:

1. The *scanner* of `ajc` is a small extension of the regular Java scanner of the JDT. The JDT scanner and the extension in `ajc` are both written by hand. The scanner extension does nothing more than adding some keywords to the scanner.

2. The *parser* of `ajc` is generated from a grammar using the Jikes parser generator (these days also known as LPG, LALR Parser Generator). The grammar is a modified version of the JDT grammar for regular Java. It does not actually define the syntax of pointcut expressions: these are only scanned and parsed separately.

   The handwritten part of the JDT parser for constructing ASTs is extended as well. The original Java code has to be modified at some places, mostly for making the parser more flexible and extensible by introducing factory methods. Presumably, this could be merged with the JDT parser itself.

3. A handwritten recursive descent *pattern parser* is invoked to parse the pointcut expressions of AspectJ after the source file has been scanned and parsed by the previous components. Except for the `if` pointcut designator, the pattern parser works directly on the result of the `ajc` scanner, since the `ajc` parser parses pointcuts as a list of tokens.

5.5.1 *Parsing Pointcuts*

The most interesting part of the `ajc` parser is the handling of pointcuts.

Scanner

The `ajc` scanner is applied uniformly to the input program, which means that the same set of tokens is allowed at all offsets in the input. Note that the `ajc` scanner does not add tokens to the JDT scanner, except for some keywords, so the pointcuts are tokenized as any other part of the source file. For example, the pointcut of the caching aspect in Figure 5.3 is scanned to the following list of tokens:

```
execution ( * Fib .  calc ( int ) ) && args ( value )
```

This sequence of tokens is a correct tokenization of this pointcut, but our previous example of the simple name pattern `get*` is actually not scanned

[1] Our study is based on `ajc` version 1.5.0.

as the single token `get*`, but as the tokenization you would expect in the context of a regular Java expression: an identifier followed by a multiplication operator, i.e. the scanner produces the tokenization `get *`.

Still, this does not look very harmful, but actually scanning pointcuts and Java code uniformly can lead to very strange tokenizations. For example, consider the (somewhat artificial) pointcut `call(* *1.Function+.apply(..))`. For this pointcut the correct tokenization according to the lexical syntax of pointcuts is:

```
call ( * *1 . Function + . apply ( .. ) )
```

However, the `ajc` scanner produces the following list of tokens for this pointcut:

```
call ( * * 1.F unction + . apply ( . . ) )
```

Perhaps surprisingly, `Function` has been split up and the F is now part of the token `1.F`, which is a floating-point literal where the `F` is a floating-point suffix. Of course, a floating-point literal is not allowed at all in this context in the source file. As we will show later, the pattern parser needs to work around this incorrect tokenization.

Unfortunately, things can get even worse. Although rather uncommon, the first alpha-numerical character after the `*` in a simple name pattern can be a number (in fact, this is also the case in the previous floating-point example). The token that starts after the `*` will always be scanned as a number by the JDT scanner, and the same will happen in the `ajc` scanner. The JDT scanner checks the structure of integer and floating-point literals by hand and immediately stops parsing if it finds a token that should be a floating-point or integer literal according to the Java lexical syntax, but is invalid because certain parts of the literal are missing. This can result in error messages about invalid literals, while in this context there can never actually be a literal.

For example, scanning the pointcut `call(void *0.Ef())` reports an *"Invalid float literal number"* because the scanner wants to recognize `0.E` as floating-point literal, but the actual exponent number is missing after the exponent indicator `E`. As another example, scanning the pointcut `call(void Foo.*0X())` fails with the error message *"Invalid hex literal number"*, since `0X` indicates the start of a hexadecimal floating-point or integer literal, but the actual numeral is missing.

Parser

The `ajc` parser operates on the sequence of tokens provided by the scanner. Unfortunately, for pointcuts the parser cannot do anything useful with this tokenization, since it is not even close to the real lexical syntax of pointcuts in many cases. In a handwritten parser it might be possible to work around the incorrect tokenization, but the `ajc` parser is generated from a grammar using the Jikes parser generator. In a grammar workarounds for incorrect tokenizations are possible as well (as we will see later for parameterized types) but for pointcuts this would be extraordinarily difficult, if not impossible.

```
PointcutDeclaration ::=
  PcHeader FormalParamListopt ')' ':' PseudoTokens  ';'

DeclareDeclaration ::=
  DeclareAnnoHeader PseudoTokensNoColon ':' Anno ';'

PseudoToken ::=
  '(' | ')' | '.' | '*' | Literal | 'new'
  | JavaIdentifier | 'if' '(' Expression ')' | ...

ColonPseudoToken ::= ':'

PseudoTokens ::=
  one or more PseudoToken or ColonPseudoToken
PseudoTokensNoColon ::=
  one or more PseudoToken
```

Figure 5.4 Pseudo tokens in the `ajc` grammar for AspectJ

For these reasons, the parser processes pointcuts just as a list of tokens called *pseudo tokens* that are parsed separately by the handwritten *pattern parser*. In this way, the main parser basically just skips pointcuts and forwards the output of the scanner (with a twist for the `if` pointcut) to the pattern parser. It is essential that the parser can find the end of the pointcut without parsing the pointcut. Fortunately, this is more or less the case in AspectJ, since pointcuts cannot contain semicolons, colons, and curly braces, except for the expression argument of the `if` pointcut designator, which we will discuss later.

The handling of pointcuts using pseudo tokens is illustrated in Figure 5.4: the first production defines pointcut declarations, where the pointcut, recognized as a sequence of pseudo tokens, starts after the colon and is terminated by the semicolon. The second production for inter-type annotation declarations uses a somewhat smaller set of pseudo tokens, since it is terminated by a colon instead of a semicolon. Most of the Java tokens, except for curly braces, semicolons, and colons, but including keywords, literals, etc., are defined to be pseudo tokens.

The `if` pointcut designator is a special case, since it takes a Java expression as an argument. Of course, the pattern parser should not reimplement the parsing of Java expressions. Also, Java expressions could break the assumption that pointcuts do not contain colons, semicolons, and curly braces. For these reasons, the `if` pointcut designator is parsed by `ajc` parser as a special kind of pseudo token, where the expression argument is not a list of tokens, but a real expression node (see Figure 5.4).

Interestingly, this special pseudo token for the `if` pointcut designator reserves the `if` keyword in pointcuts, while all other Java keywords are allowed in name patterns. Hence, the method pattern `boolean *.if*(..)` is not allowed in `ajc` [2].

[2]This turned out to be a known problem, see bug 61535 in `ajc`'s Bugzilla: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=61535`, which has been opened in May 2004.

Pattern Parser

Finally, the handwritten pattern parser is applied to pointcuts, which have been parsed as a sequence of pseudo tokens by the parser. The pattern parser takes a fair amount of code, since the pointcut language of AspectJ is quite rich. Most of the code for parsing pointcuts is rather straightforward, though cumbersome to implement by hand. The most complex code handles the parsing of name patterns. Since the tokenization performed by the `ajc` scanner is not correct, the pattern parser cannot just consume the tokens. Instead, it needs to consider all the possible cases of incorrect tokenizations. For example, the pointcuts `call(* *1.foo(..))` and `call(* *1.f oo(..))` are both tokenized in the same way by the `ajc` scanner:

```
call ( * * 1.f oo ( . . ) ) ;
```

However, the token sequences for these two pointcuts cannot be handled in the same way, since the second one is incorrect, so a parse error needs to be reported. Therefore, the pattern parser checks if tokens are adjacent or not:

```
while(true) {
  tok = tokenSource.peek();
  if(previous != null) {
    if(!isAdjacent(previous, tok))
      break;
  }
  ...
}
```

The need for this adjacency check follows naturally from the fact that the pattern parser has to redo the scanning at some parts of the pointcut and a single AspectJ pointcut token can span multiple Java tokens, in particular in name patterns.

The special `if` pseudo tokens do not have to be parsed anymore. For this purpose, the `IToken` interface, of which `PseudoToken` and `IfPseudoToken` are implementations, is extended with a method *maybeGetParsedPointcut* that immediately returns the pointcut object. This method is invoked from the pattern parser:

```
public Pointcut parseSinglePointcut() {
  IToken t = tokenSource.peek();
  Pointcut p = t.maybeGetParsedPointcut();
  if(p != null) {
    tokenSource.next();
    return p;
  }
  String kind = parseIdentifier();
  ... // continue parsing the pointcut
}
```

5.5.2 *Parameterized Types*

Incorrect tokenization problems are not unique to AspectJ. Even in regular Java 5, a parser that applies a scanner uniformly to an input program has to

```
TypeArgs ::= '<' TypeArgList1

TypeArgList  -> TypeArg
TypeArgList  ::= TypeArgList ',' TypeArg
TypeArgList1 -> TypeArg1
TypeArgList1 ::= TypeArgList ',' TypeArg1
TypeArgList2 -> TypeArg2
TypeArgList2 ::= TypeArgList ',' TypeArg2
TypeArgList3 -> TypeArg3
TypeArgList3 ::= TypeArgList ',' TypeArg3

TypeArg  ::= RefType
TypeArg1 -> RefType1
TypeArg2 -> RefType2
TypeArg3 -> RefType3

RefType1 ::= RefType '>'
RefType1 ::= ClassOrInterface '<' TypeArgList2
RefType2 ::= RefType '>>'
RefType2 ::= ClassOrInterface '<' TypeArgList3
RefType3 ::= RefType '>>>'
```

Figure 5.5 Production rules for parameterized types in the `ajc` grammar, working around incorrect tokenizations of parameterized types

deal with incorrect tokenizations, namely of parameterized types. For example, the parameterized type `List<List<List<String>>>` is tokenized by the `ajc` scanner as:

`List < List < List < String >>>`

where `>>>` is the unsigned right shift operator [3]. Because of this, the grammar cannot just define type arguments as a list of comma-separated types between '<' and '>', since in some cases the final > will not actually be a separate token.

This tokenization problem has to be dealt with in two places: in the `ajc` grammar and in the handwritten pattern parser. For the `ajc` grammar, Figure 5.5 shows the production rules for type arguments. Clearly, this is much more involved than it should be [4]. For the pattern parser, incorrect tokenizations of >> and >>> are fixed by splitting the tokens during parsing when the expected token is a single >. Figure 5.6 show the code for this. The `eat` method is used in the pattern parser to check if the next token is equal to a specified, `expected` token. If a shift operator is encountered, but a > is expected, then the token is split and the remainder of the token is stored in the variable `pendingRightArrows`, since the remainder is now the next token.

5.5.3 *Pseudo Keywords*

For compatibility with existing Java code, `ajc` does not reserve all the keywords introduced by AspectJ. Yet, the scanner of `ajc` *does* add keywords to the

[3] In C++ this is not allowed: a space is required between the angle brackets.
[4] This workaround is documented in the GJ specification [Bracha et al. 1998].

```
private void eat(String expected) {
  IToken next = nextToken();
  if(next.getString() != expectedValue) {
    if(expected.equals(">") && next.getString().startsWith(">")) {
      pendingRightArrows = substring from 1 of next;
      return;
    }
    throw parse error
  }
}

private IToken pendingRightArrows;
private IToken nextToken() {
  if(pendingRightArrows != null) {
    IToken ret = pendingRightArrows;
    pendingRightArrows = null;
    return ret;
  }
  else {
    return tokenSource.next();
  }
}
```

Figure 5.6 Splitting shift operators in the `ajc` pattern parser to work around incorrect tokenizations of parameterized types

lexical syntax of Java (`aspect`, `pointcut`, `privileged`, `before`, `after`, `around`, and `declare`), which usually implies that these keywords cannot be used as identifiers since the scanner will report these tokens as keywords. However, in its grammar, `ajc` introduces `JavaIdentifier`, a new non-terminal for identifiers, for which these keywords are explicitly allowed:

```
JavaIdentifier -> 'Identifier'
JavaIdentifier -> AjSimpleName

AjSimpleName -> 'around'
AjSimpleName -> AjSimpleNameNoAround
AjSimpleNameNoAround -> 'aspect' or 'privileged' or
    'pointcut' or 'before' or 'after' or 'declare'
```

This extended identifier replaces the original `Identifier`, which can no longer be one of the AspectJ keywords, at most places in the grammar. For example, the following productions allow the AspectJ keywords as the name of a class, method, local variable, and field.

```
ClassHeaderName1 ::= Modifiersopt 'class' JavaIdentifier
MethodHeaderName ::= Modifiersopt Type JavaIdentifier '('
VariableDeclaratorId ::= JavaIdentifier Dimsopt
```

However, the extended identifier is not allowed everywhere. In particular, it cannot be the first identifier of a type name, which means that it is not allowed as a simple type name, and cannot be the first identifier of a qualified type name, which could refer to a top-level package or an enclosing class. For example, the first import declaration is not allowed, but the second one is [5]:

[5]This is related to `ajc` bug 37069 at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=37069`

```
import privileged.*;
import org.privileged.*;
```

If keywords would be allowed as simple type names, the grammar would no longer be LALR(1). The keywords as type names introduce shift-reduce and reduce-reduce conflicts. Hence, a qualified name is defined to be an `Identifier`, followed by one or more `JavaIdentifiers`:

```
ClassOrInterface ::= Name
SingleTypeImportDeclarationName ::= 'import' Name
Name -> SimpleName or QualifiedName
SimpleName -> 'Identifier'
QualifiedName ::= Name '.' JavaIdentifier
```

Pointcuts

The names of the primitive AspectJ pointcut designators, such as `get`, `set`, `call`, etc., are not declared as keywords. The scanner does not have any knowledge about pointcuts, so the names are parsed as identifiers, unless the pointcut designator was already a keyword, such as `if`. As we have seen earlier, the name `if` is still accidentally a reserved keyword, but the names of the other pointcut designators are not, so they can be used in pointcut expressions, for example in name patterns. However, a named pointcut with the same name as a primitive pointcut designator cannot be used (though surprisingly, it can be declared without warnings).

Around Advice Declarations

Around advice declarations introduce another complication. Whereas *after* and *before* advice declarations immediately start with the keywords `after` or `before`, *around* advice declarations start with a declaration of the return type. This introduces a shift-reduce conflict between an around advice declaration and a method declaration. For this reason, `ajc` does not allow methods named `around` in aspect declarations. Of course, it would not be acceptable to disallow the name `around` for all methods, including the ones in regular Java classes, so this restriction should only apply to aspect declarations (advice cannot occur in class declarations). Therefore, the `ajc` grammar needs to duplicate all the productions (19) from an aspect declaration down to a method declaration, where finally the name of a method is restricted to a `JavaIdNoAround`:

```
JavaIdNoAround -> 'Identifier'
JavaIdNoAround -> AjSimpleNameNoAround
MethodHeaderNameNoAround ::=
  Modifiersopt TypeParameters Type JavaIdNoAround '('
```

5.6 THE ABC SCANNER AND PARSER

The parser of `abc`[6] is based on Polyglot [Nystrom et al. 2003], which provides PPG, a parser generator for extensible grammars based on the LALR CUP parser generator. PPG acts as a front-end for CUP, by adding some extensibility and modularity features, which we will discuss later in Section 5.9.

[6] Our study is based on `abc` version 1.1.0, which supports `ajc` 1.2.1 with some minor differences

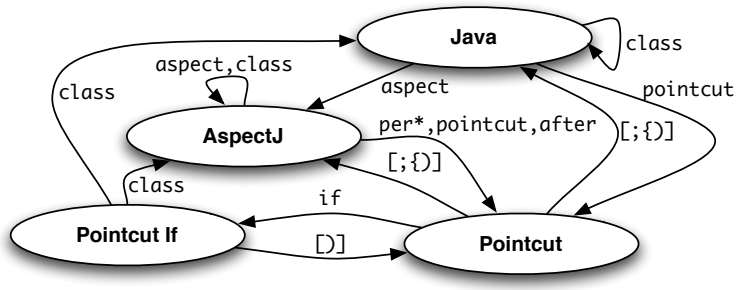


Figure 5.7 Lexical state transitions in the abc scanner

Polyglot's scanner for Java is implemented using the JFlex scanner generator. Polyglot does not feature an extensible scanner, so the abc compiler implements its own scanner for AspectJ, which takes an approach radically different from ajc. The abc scanner and parser can parse the entire source file in a single continuous parse. So, the Java, aspect, and pointcut language are defined in a single JFlex specification and CUP grammar. The abc scanner is designed to immediately produce the correct tokenization, so there is no need to fix incorrect tokenizations later. Also, the scanner does not interact with the parser.

5.6.1 *Managing Lexical State*

The abc scanner performs a rudimentary form of context-free parsing to recognize the global structure of the source file while scanning. The scanner keeps track of the current *state* (or *context*), by using a set of state transition rules that have been determined by a detailed analysis of the possible state switches in AspectJ. The lexical states and the transitions between them are illustrated in Figure 5.7. Some transitions have additional conditions, which we will explain later. Maintaining lexical state is not uncommon. It is widely used for scanning string literals and it is a standard feature of JFlex. Every lexical state has its own set of lexical rules, which means that a sequence of characters can be scanned as a different token in different states.

Pointcut Declarations

A simple example of such a state transition rule, is that a pointcut state is entered after the `pointcut` keyword and exited after a `";"` in pointcut context. For this example, the `pointcut` keyword and the semicolon indicates the start and end of a pointcut declaration, respectively. The exit of the pointcut state after a pointcut declaration is implemented in the flex specification by returning to the previous state (which is maintained on a stack) whenever the `";"` token is encountered in the pointcut state (`POINTCUT`):

```
<POINTCUT> {
  ";" {
```

```
      returnToPrevState();
      return op(sym.SEMICOLON);
    }
  }
```

For reasons of extensibility, keywords and their corresponding actions for *entering* lexical states are not specified in the flex specification, but are initialized from the Java code by means of a Java interface `LexerAction` whose instances can be registered with the scanner. `LexerActions` are always attached to keywords and can change the lexical state when the keyword has been scanned. For example, the following Java statement adds the keyword `pointcut`, which starts the pointcut declaration, to the scanner and specifies that the new lexical state after this keyword is pointcut.

```
lexer.addAspectJKeyword("pointcut",
  new LexerAction_c(new Integer(sym.POINTCUT),
    new Integer(lexer.pointcut_state())));
```

In this way, keywords are registered *per lexical state* in a `HashMap`. Initially, keywords are always scanned as identifiers and depending on the current lexical state, the identifier is turned into a keyword by a lexer action. As a side effect, the lexer action can modify the lexical state of the scanner. Figure 5.8 shows a fragment of the Java class `LexerAction_c` and the invocation of the lexer actions from the flex specification after an `Identifier` has been scanned. Note that keywords are automatically *reserved* in this way, since the identifier is always be turned in a keyword if there is a lexer action for it. Note that this design choice for reserved keywords is different from the pseudo keyword policy used by `ajc`.

If Pointcut Designator

The `pointcut` lexer action and the lexical rule for ; look rather concise, but unfortunately, most rules are more complex than this. For instance, the `if(..)` pointcut designator takes a Java expression as argument, which has the same lexical syntax as Java code in Java context, so the lexical state should be changed for the argument of the `if(..)`. Entering the lexical state is not very difficult: a lexer action for the `if` keyword can perform this state transition. The following Java statement adds the pointcut keyword `if` to the scanner and specifies that the new lexical state after this keyword is the special `POINTCUTIFEXPR` state:

```
lexer.addPointcutKeyword("if",
  new LexerAction_c(new Integer(sym.PC_IF),
    new Integer(lexer.pointcutifexpr_state())));
```

However, for recognizing the end of the `if(..)` pointcut designator, the scanner needs to find the closing parenthesis. Of course, a Java expression can contain parentheses as well. It would be incorrect to leave the special lexical state at the first closing parenthesis. Thus, the scanner needs to find the closing parenthesis that corresponds to the opening parenthesis after the `if`. For this purpose, the `abc` scanner maintains a variable `parenLevel` that is used to balance the parentheses. If a `")"` is encountered, the `parenLevel` is

```
<YYINITIAL,ASPECTJ,POINTCUTIFEXPR,POINTCUT> {
  {Identifier} {
    LexerAction la;
    switch(yystate()) {
      case YYINITIAL:
        la = javaKeywords.get(yytext());
        break;
      case ASPECTJ:
        la = aspectJKeywords.get(yytext());
        break;
      ...
    }

    if(la != null)
      return key(la.getToken(this));
    return id();
  }
}
```

```
class LexerAction_c implements LexerAction {
  public Integer token;
  public Integer nextState;

  public int getToken(AbcLexer lexer) {
    if(nextState != null)
      lexer.enterLexerState(nextState.intValue());
    return token.intValue();
  }
}
```

Figure 5.8 Lexer actions in the abc scanner

decremented and the new `parenLevel` is compared to the `parenLevel` of the `if` pointcut, for which the initial `parenLevel` has been saved in the entry on the `nestingStack`:

```
  <YYINITIAL,ASPECTJ,POINTCUTIFEXPR> {
    "(" {
      parenLevel++;
      return op(sym.LPAREN);
    }
    ")" {
      parenLevel--;
      if((yystate() == POINTCUTIFEXPR)
               && (parenLevel == nestingStack.peek().parenLevel))
        returnToPrevState();
      return op(sym.RPAREN);
    }
  }
```

Per-clause

There are more places where pointcuts can occur in an AspectJ program: aspect declarations optionally take a *per-clause*, which is used to control the instantiation of aspects. For example, declaring:

```
  aspect Foo perthis(pc) { ... }
```

entails that a new aspect instance of `Foo` is created for every `this` where the pointcut *pc* matches. Finding out the end of the pointcut of a per-clause is a bit more difficult than for normal pointcuts. The scanner again needs to find the matching closing parenthesis, but it also needs to know if it is actually scanning the pointcut of a per-clause or not. Instead of a new lexical state for per-clause pointcuts, the `abc` scanner uses a global boolean variable `inPerPointcut`. This variable is set to `true` by a lexer action for all per-clause keywords (`perthis`, `percflow`, etc.):

```
class PerClauseLexerAction_c extends LexerAction_c {
  ...
  public int getToken(AbcLexer lexer) {
    lexer.setInPerPointcut(true);
    return super.getToken(lexer);
  }
}
```

For a closing parenthesis in the pointcut lexical state, the scanner now needs to check if it is currently scanning a per-clause pointcut and if the closing parenthesis occurs at the same parenthesis level as the opening parenthesis that preceded the pointcut:

```
<POINTCUT> {
  ")" {
    parenLevel--;
    if(inPerPointcut &&
      parenLevel == nestingStack.peek().parenLevel) {
        returnToPrevState();
        inPerPointcut = false;
    }
    return op(sym.RPAREN);
  }}
```

Class Keyword

While the end of a lexical state is detected in the flex specification by a lexical rule for a token, the start of a context is declared in the lexer action of a keyword. In most cases, the start of a new lexical state is clearly indicated by a keyword. However, the `class` keyword does not unambiguously indicate the start of the Java lexical state for a class declaration, since it may also be used in class literals (e.g. `Foo.class`). To distinguish a class literal from a class declaration, the `abc` scanner maintains a special variable `lastTokenWasDot`. All tokens, except for the dot, set this variable to false. The rule for the `class` token can now determine whether it appears in a class literal or a class declaration and change the scanner state accordingly.

```
lexer.addGlobalKeyword("class",
  new LexerAction_c(new Integer(sym.CLASS)) {
    public int getToken(AbcLexer lexer) {
      if(!lexer.getLastTokenWasDot())
        lexer.enterLexerState(aspectj or java);
      return token.intValue();
    }
  });
```

It is interesting to observe the consequences for the scanner if a keyword no longer unambiguously indicates the next lexical state. In this case, the scanner needs to be updated for all tokens to maintain the `lastTokenWasDot` variable.

5.6.2 *Parser*

Thanks to the rudimentary context-free parsing in the scanner, the AspectJ grammar of `abc` is a clean modular extension of the basic Java grammar, implemented in PPG and based on the existing Polyglot grammar for Java. The grammar defines the entire AspectJ language, including pointcuts and name patterns, which is not the case in `ajc`.

Name Patterns

There is one interesting language construct for which some undesirable production rules have to be defined: name patterns. The grammar explicitly allows the reserved keywords of the pointcut lexical state as simple name pattern to allow name patterns such as `Foo.get`. Without explicitly allowing keywords, this would be forbidden, since `get` is a reserved keyword for pointcuts in `abc` and will therefore not be parsed as an identifier. The CUP production rules for this are:

```
simple_name_pattern ::=
  PC_MULT | IDENTIFIERPATTERN | IDENTIFIER
  | aspectj_reserved_identifier ;

aspectj_reserved_identifier ::=
  ASPECT | ... |  PC_GET | ... | PC_SET ... ;
```

This is somewhat unfortunate, because the keywords for pointcuts are hence defined in the grammar, as well as in the Java code, namely for adding lexer actions to the scanner. Extensions of AspectJ implemented in `abc` that introduce new pointcut keywords have to extend the `aspectj_reserved_identifier` production as well. Extensions may easily forget to do this and thereby reserve their keywords in name patterns. This extensibility issue will be discussed in more detail in Section 5.9.

Ideally, the `abc` scanner should enter a new lexical state for name patterns, since the lexical syntax of name patterns differs from pointcuts (i.e. the set of keywords is different). However, this will be more difficult to implement than the existing lexical states, since name patterns are not very explicitly delimited by certain tokens [7].

Parameterized Types

Although `abc` does not support AspectJ 5.0 and parameterized types, it is interesting to take a look at how the scanning problems for parameterized types would be solved in a similar setup of the scanner and parser. Currently, an extension of Polyglot for Java 5.0 is under development at McGill. In

[7] Indeed, very recently bug 72 has been created in the `abc` bugzilla, which proposes to introduce a lexer state for name patterns. See: `http://abc.comlab.ox.ac.uk/cgi-bin/bugzilla/show_bug.cgi?id=72`

```
reference_type_1 ::= reference_type GT
  | class_or_interface LT type_argument_list_2;
reference_type_2 ::= reference_type RSHIFT
  | class_or_interface LT type_argument_list_3;
reference_type_3 ::= reference_type:a URSHIFT;

wildcard   ::= QUESTION;
wildcard_1 ::= QUESTION GT;
wildcard_2 ::= QUESTION RSHIFT;
wildcard_3 ::= QUESTION URSHIFT;
```

Figure 5.9 Grammar production rules for parameterized types in Polyglot.

contrast to the approach of the abc compiler, the scanner of this extension does *not* always produce the correct tokenization for *regular Java*. Instead, the grammar works around the incorrect tokenization of parameterized types by encoding this in the definition of type arguments and reference types. To illustrate this workaround for incorrect tokenization, some production rules of this grammar are shown in Figure 5.9 (lots of details have been eluded). To resolve this issue a different lexical state should be used for types, since their lexical syntax is different from expressions. However, types will be very difficult to identify by a scanner in the input file, so this approach is rather unlikely to work.

Unfortunately, this grammar is now difficult to extend for reference types, since there are a large number of production rules involved, which encode the syntax of reference types in a rather tricky way.

5.7 SUMMARY AND DISCUSSION

We have discussed two approaches to parsing AspectJ. The ajc compiler uses a single scanner, but separate parsers (for 'regular' code and for pointcut expressions). The abc compiler uses a single parser with a stateful scanner. Based on our analysis we can make the following observations. Many rules on the syntax of AspectJ are only operationally defined in the implementation of the scanner and parser. As a consequence neither implementation provides a declarative formalization of the syntax of AspectJ, although the LALR grammar of abc [Hendren et al. 2004] is a step in the right direction. The ajc parser has undocumented implementation quirks because of the scanner implemented in and for plain Java. The abc parser improves over this by using a scanner with lexical states. The abc parser is also more predictable, but managing the lexical state in the parser is tricky and duplicates code and development effort. It is difficult to reason about the correctness and completeness of the context switching rules of the abc scanner. For example, the use of the global variable inPerPointcut happens to work correctly in case an anonymous class is used with aspect members in a per-pointcut, but a slight change or extension of the language may render this implementation invalid. Choices for introducing lexical states are guided by the complexity of determining this lexical state in the scanner. For example, a separate lexical state

for name patterns might be more appropriate. In conclusion, although the implementation techniques used in the parsers of `ajc` and `abc` are effective for parsing AspectJ, their implementations have several drawbacks.

5.8 A DECLARATIVE SYNTAX DEFINITION FOR ASPECTJ

In the previous sections, we have presented a range of implementation issues in parsing AspectJ, and the solutions for these in the two major AspectJ compilers, i.e. `ajc` and `abc`. As a consequence of these issues, the syntax of the language that is supported by these compilers is not clearly defined. We conclude that the grammar formalisms and parsing techniques that are used are not suitable for the specification of the AspectJ language. A complete and declarative definition of the syntax of the AspectJ language is lacking.

In this section, we present a definition of the syntax of AspectJ that is declarative, modular, and extensible. Our AspectJ syntax definition is based on the syntax definition formalism SDF and its implementation with scannerless generalized LR parsing. Thanks to these foundations, the definition elegantly deals with the extension and embedding of the Java language, the problems of context-sensitive lexical syntax, and the different keyword policies of the `ajc` and `abc` compilers. Indeed, the modularity of SDF allows us to define three variants of the AspectJ language:

- AJF, which is the most liberal definition, where only real ambiguities are resolved, for example by reserving keywords at very specific locations.
- AJC, which adds restrictions to the language to be more compatible with the official AspectJ compiler. The additional restrictions are mostly related to shift-reduce problems in the LALR parser of `ajc`.
- ABC, which reserves keywords in a context-sensitive way, thus defining the language supported by the `abc` compiler.

The AspectJ syntax definition modularly extends our syntax definition for Java 5 [8]. Also, the AJF, AJC, and ABC variants are all modular extensions of the basic AspectJ definition. Moreover, in Section 5.9 we will show that our syntax definition can easily be extended with new aspect features. In Section 5.10 we present benchmark results, which show that these techniques yield a parser that performs linear in the size of the input with an acceptable constant factor, at least for specification, research and prototyping purposes.

The core observation underlying the syntax definition is that AspectJ is a combination of languages, namely Java, aspects, and pointcuts. From this viewpoint, this work applies and extends previous work on combining languages for the purpose of domain-specific language embedding [Bravenboer & Visser 2004 (Chapter 2)] and metaprogramming with concrete object syntax [Bravenboer et al. 2005 (Chapter 3)] (see Section 5.11.1).

[8] Available at `http://java.syntax-definition.org`

5.8.1 *Integrating Lexical and Context-Free Syntax*

SDF integrates the definition of lexical and context-free syntax in a single formalism, thus supporting the *complete* description of the syntax of a language in a single definition. In this way, the lexical syntax of AspectJ can be integrated in the context-free syntax of AspectJ, which automatically leads to *context-sensitive* lexical syntax. Parsing of languages defined in SDF is implemented by the scannerless generalized LR parser SGLR [Visser 1997a], which operates on individual characters instead of tokens. Thus, recognizing the lexical constructs in a source file is actually the same thing as parsing. This solves most of the issues in parsing AspectJ.

Lexical syntax can be disambiguated in a declarative, explicit way, as opposed to the implicit, built-in heuristics of lexical analysis tools, such as a longest-match policy and a preference for keywords. Without explicit specification, keywords are *not* reserved and, for example, are perfectly valid as identifiers. Instead, keywords can be reserved explicitly by defining *reject productions*.

Java

Figure 5.10 illustrates the basic ideas of SDF with sample modules and productions from the Java syntax definition. Of course, the real syntax definition is much larger and spread over more modules. Note that the arguments of an SDF production are at the left and the resulting symbol is at the right, so an SDF production $s_1 \ldots s_n$ -> s_0 defines that an element of nonterminal s_0 can be produced by concatenating elements from nonterminals $s_1 \ldots s_n$, in that order. The modules of Figure 5.10 illustrate that modules have names [14] and can import other modules [15]. The module `Java` defines the composition of compilation units [16] from package declarations, import declarations, and type declarations. Note the use of optional (`?`) and iterated (`*`,`+`) nonterminals. The module `Expressions` defines expression names [17] (local variables, fields, etc), addition of expressions [18], which is declared to be `left` associative, and method invocation [19]. The production rule for method invocations uses `{`*s lit*`}*`, which is concise notation for a list of *s* separated by *lit*. The module `Identifiers` shows how *lexical syntax* is defined in the same syntax definition as the *context-free syntax*. To define lexical nonterminals such as identifiers SDF provides character classes to indicate sets of characters [20]. The `Identifiers` module also defines a longest-match policy for identifiers, by declaring that identifiers cannot directly be followed by one of the identifier characters [21]. Another difference with respect to other formalisms is that there may be multiple productions for the same nonterminal. This naturally leads to *modular* syntax definitions in which syntax can be composed by importing modules.

Aspects

Similar to the syntax definition of Java, SDF can be used to define modules for the languages of aspects, pointcut expressions, and patterns. Figure 5.11 presents a few productions for aspect declarations in AspectJ. The first two

```
module Java [14]
imports Statements Expressions Identifiers [15]
exports
   context-free syntax
     PackageDec? ImportDec* TypeDec+ -> CompilationUnit [16]
```

```
module Statements
exports
  context-free syntax
    "for" "(" FormalParam ":" Expr ")" Stm -> Stm
    "while" "(" Expr ")" Stm               -> Stm
```

```
module Expressions
exports
  context-free syntax
    ExprName                        -> Expr              [17]
    Expr "+" Expr                   -> Expr {left}       [18]
    MethodSpec "(" {Expr ","}* ")" -> Expr               [19]
    MethodName                      -> MethodSpec
    Expr "." TypeArgs? Id           -> MethodSpec
```

```
module Identifiers
exports
  lexical syntax
    [A-Za-z\_\$][A-Za-z0-9\_\$]* -> Id                   [20]

  lexical restrictions
    Id -/- [a-zA-Z0-9\_\$]                               [21]
```

Figure 5.10 Fragment of syntax definition for Java

productions define aspect declarations [22] and aspect declaration headers [23]. Both productions use nonterminals from Java, for example `Id`, `TypeParams` (generics), and `Interfaces`. The aspect header may have a per-clause, which can be used to control the instantiation scheme of an aspect. For instance, a `perthis` clause [24] specifies that one aspect instance is created for each currently executing object (`this`) in the join points matched by the pointcut expression given as a parameter. The per-clause `pertypewithin` [25], which has been added to the language in AspectJ 5, is used to create a new aspect instance for each type that matches the given type pattern. This is the only per-clause that does not take a pointcut expression as an argument.

Advice declarations [26] are mainly based on an advice specifier and a pointcut expression, where an advice specifier can be a before [27], after [28], or around [29] advice. Note that most of the productions again refer to Java constructs, for example `ResultType` and `Param` (an abbreviation of `FormalParam`).

Pointcuts

The AspectJ pointcut language is a language for concisely describing a set of join points. Pointcut expressions consist of applications of pointcut designators, which can be primitive or user-defined. Also, pointcut expressions

```
module AspectDeclaration
exports
  context-free syntax
    AspectDecHead AspectBody -> AspectDec    22
    AspectMod* "aspect" Id TypeParams? Super?
          Interfaces? PerClause? -> AspectDecHead    23

    "perthis"       "(" PointcutExpr ")" -> PerClause    24
    "pertypewithin" "(" TypePattern ")"  -> PerClause    25

    AdviceMod* AdviceSpec Throws? ":" PointcutExpr
                                    MethodBody -> AdviceDec    26

    "before" "(" {Param ","}* ")"              -> AdviceSpec    27
    "after"  "(" {Param ","}* ")" ExitStatus? -> AdviceSpec    28
    ResultType "around" "(" {Param ","}* ")"  -> AdviceSpec    29
    "returning" "(" Param ")" -> ExitStatus    30
```

Figure 5.11 Fragment of syntax definition for aspects and advice

```
module PointcutExpression
exports
  context-free syntax
    "call"   "(" MethodConstrPattern ")"    -> PointcutExpr  31
    "get"    "(" FieldPattern ")"           -> PointcutExpr  32
    "this"   "(" TypeIdStar ")"             -> PointcutExpr  33
    "cflow"  "(" PointcutExpr ")"           -> PointcutExpr  34
    "if"     "(" Expr ")"                   -> PointcutExpr  35
    PointcutName "(" {TypeIdStar ","}* ")" -> PointcutExpr  36
    Id -> PointcutName
```

Figure 5.12 Fragment of syntax definition for AspectJ pointcut expressions.

can be composed using boolean operators. Figure 5.12 shows some of the primitive pointcuts of AspectJ. The `call` 31 and `get` 32 pointcut designators take patterns of methods, constructors, or fields as arguments. The `this` 33 pointcut designator cannot be used with arbitrary type patterns. Instead, the argument must be a `Type`, an `Id` or a wildcard. The `if` 35 pointcut designator, which we have discussed before, takes a boolean Java expression as an argument. Finally, Figure 5.12 defines the syntax for user-defined pointcuts 36 in pointcut expressions, which have been declared somewhere in the program using a pointcut declaration.

Patterns

The AspectJ pattern language plays an important role: as we have already seen, most of the pointcut designators operate on patterns. Figure 5.13 shows some productions for the syntax of the pattern language. *Name patterns* are used to pick out names in a program. A name pattern is a composition of identifier patterns 44, which are used for matching identifiers (i.e. names without a dot) by adding a `*` wildcard to the set of identifier characters. The `..` wildcard 37 can be used to include names from inner types, subpackages, etc. Almost every pointcut uses *type patterns*, which are used for selecting types.

```
module Pattern
exports
  context-free syntax
    IdPattern                  -> NamePattern
    NamePattern "."  IdPattern -> NamePattern
    NamePattern ".." IdPattern -> NamePattern          37

    PrimType           -> TypePattern                  38
    TypeDecSpecPattern -> TypePattern
    TypeDecSpecPattern TypeParamsPattern -> TypePattern 39
    NamePattern        -> TypeDecSpecPattern           40
    NamePattern "+"    -> TypeDecSpecPattern           41

    FieldModPattern TypePatttern ClassMemberNamePattern
                                                  -> FieldPatttern 42
    MethodModPattern TypePattern ClassMemberNamePattern
        "(" {FormalPattern ","}* ")" ThrowsPattern? -> MethodPattern 43

lexical syntax
  [a-zA-Z\_\$\*][a-zA-Z0-9\_\$\*]* -> IdPattern  44
```

Figure 5.13 Fragment of syntax definition for AspectJ patterns

Any name pattern is a type pattern [40], but type patterns can also be used to match subtypes [41], primitive types [38], parameterized types [39], etc.

Method [43] and *field patterns* [42] combine name patterns, type patterns, modifier patterns, throw patterns and patterns for formal parameter into complete signature patterns that are used to match methods and fields by their signatures.

5.8.2 *Composing AspectJ*

We have now illustrated how the syntax of the sublanguages of AspectJ (Java, aspects, pointcuts, and patterns) can be defined as separate SDF modules. Next, we need to compose these modules into a syntax definition for AspectJ itself. In SDF, we can combine two syntax definitions by creating a new module that imports the main modules of the languages that need to be combined. The ease with which syntax definitions can be composed, is due to the two main features of the underlying parser: *scannerless parsing* and the use of the *generalized LR* algorithm.

First, in a setting with a separate scanner such a combination would cause conflicts as has extensively been discussed in Section 5.4. However, in the scannerless SDF setting this does not pose a problem. Since lexical analysis is integrated with parsing, context-sensitive lexical analysis comes for free. For example, when parsing `1+1` as a Java expression the `+` will be seen as an addition operator [18], but when parsing `Foo+` in the context of a pointcut expression, then the `+` will be interpreted as a subtype pattern [41].

Second, if LL, LR, or LALR grammars are used, then the combination of one or more languages is not guaranteed to be in the same subset, since these subsets of the context-free languages are not closed under composition. Indeed, if we combine method declarations from Java and advice declarations

```
module AspectJ [45]
imports
  Java AspectDeclaration PointcutExpression Pattern [46]
exports
  context-free syntax
    AspectDec     -> TypeDec            [47]
    ClassBodyDec -> AspectBodyDec       [48]
    AspectDec     -> ClassMemberDec     [49]
    PointcutDec   -> ClassMemberDec     [50]
```

Figure 5.14 SDF module combining Java, pointcut, and aspect declarations.

from aspects, then shift-reduce conflicts pop up since this combination is no longer LALR, as has been discussed in Section 5.5.3. Since SDF is implemented using generalized LR parsing, SDF supports the full class of context-free grammars, which *is* closed under composition. Hence, new combinations of languages will stay inside the same class of context-free grammars.

Nevertheless, in some cases there will be ambiguities in the new combination of languages where there are actually two or more possible derivations for the same input. These ambiguities can be solved in a declarative way using one of the SDF disambiguation filters [van den Brand et al. 2002], such as reject, priorities, prefer, and associativity. Section 5.9 presents examples of this in AspectJ extensions. However, this is not the case for AspectJ. For example, the around advice problem is not a real ambiguity: the syntax of around advice and method declarations are similar for the first few arguments, but the colon and the pointcut expression distinguishes the around advice syntactically from method declarations.

AspectJ

Figure 5.14 illustrates how the languages can be combined by importing [46] the modules of Java, aspects, pointcuts, and patterns [9]. In this way, most of the integration happens automatically: the productions for pointcut expressions already refer to patterns and aspect declarations already refer to pointcut expressions and patterns. By importing all modules, the symbols and productions of these modules will be combined, and as a result the pointcut expressions will automatically be available to the aspect declarations.

The integration of the languages can be extended and refined by adding more productions that connect the different sublanguages to each other. For instance, aspect declarations (`AspectDec`) are Java type declarations, since they can be used at the top-level of a source file [47] (see also the production rule for compilation units [16]). Furthermore, aspect declarations [49] and pointcut declarations [50] can occur *inside* a class, i.e. as members of a Java class declaration.

Just as aspects and pointcuts can be defined in regular Java code, the declarations of aspects can contain Java members such as constructors, initializers, fields, and method declarations. Thus, Java class body declarations

[9]The actual composition in the full definition is somewhat different, to make the definition more customizable. We will discuss this later.

(`ClassBodyDec`, i.e. elements of a Java class body) are allowed as aspect body declarations [48].

5.8.3 *Disambiguation and Restrictions*

We have not yet defined any reserved keywords or other restrictions for the syntax that we have presented. Next, we explain how the syntax definition can be extended in a *modular* way to impose additional restrictions on the language, such as different reserved keyword policies and requirements for being compatible with the language accepted by an LALR grammar. First, we discuss how keywords can be reserved in SDF. Next, we discuss the real ambiguities of the language that we have presented so far. The resulting syntax definition, which is the most liberal AspectJ syntax definition without ambiguities, is called AJF. After that, we extend the restriction to achieve the AJC and ABC variants, which are designed to be compatible with the AspectJ language as supported by the `ajc` and `abc` compilers, respectively.

Reserving Keywords

Scannerless parsing does not *require* a syntax definition to reserve keywords. Depending on the context, the same token can for example be interpreted as a keyword or as an identifier. However, in some cases a keyword is inherently ambiguous if it is not reserved. For example, the Java expressions `this` and `null` would be ambiguous with the identifiers `this` and `null` if they would not be reserved. In SDF reserved keywords are defined using *reject productions* [Visser 1997a], which are productions annotated with the `reject` keyword. The following two SDF productions illustrate this mechanism:

```
"abstract" | "assert" | ... | "while" -> Keyword
Keyword -> Id {reject}
```

The first production defines keywords and the second rejects these keywords as identifiers. Reject productions employ the capability of generalized LR parsers to produce all possible derivations. In case of a keyword, there will be two possible derivations: one using the real production for identifiers and one using the reject production. If the reject production is applicable, then all possible parses that produce the same nonterminal (in this case `Id`) are eliminated. In this way, the parse that uses the production for the real identifier is disallowed. Thus, in SDF reserved keywords are defined *per nonterminal*: in the example above, the keywords are *only* reserved for the `Id` nonterminal. If other identifier-like nonterminals would exist in Java (which is not the case), then keywords would not be reserved for that nonterminal. Because there is just a single identifier nonterminal for regular Java, this feature does not add much over a mechanism for global keywords, but the feature is most useful if languages are being combined: it can be used for defining context-sensitive keywords.

AJF

One of the few ambiguities in the syntax definition are the applications of user-defined [36] and primitive pointcut designators. For example, the pointcut expression `this(Foo)` can be parsed as the primitive pointcut `this`, but it can also parsed as a user-defined pointcut with the same name. To resolve this ambiguity, AJF rejects the names of primitive pointcuts as the name of a user-defined pointcut, which is similar to the behaviour of `ajc` and `abc`. To make the names of primitive pointcuts available to extensions and the other variants of AspectJ, we introduce a new nonterminal: `PrimPointcutName`. These names are rejected as the name of a used-defined pointcut.

```
"adviceexecution" | "args" | "call" | ...
    | "within" | "withincode" -> PrimPointcutName
PrimPointcutName -> PointcutName {reject}
```

Another ambiguity that needs to be resolved by reserving keywords occurs in type patterns. Type patterns are composed of name and identifier patterns, but we have not imposed any restrictions on these name patterns, which implies that a name pattern can just as well be one of the built-in types `int`, `float`, `void`, etc. We do not want to reject these types as identifier patterns in general, since there is actually no ambiguity there. To resolve this ambiguity more precisely, we can disallow keywords only for the name patterns that are used as type patterns, i.e. `TypeDecSpecPatterns` [40].

```
Keyword -> TypeDecSpecPattern {reject}
```

The final ambiguity is a bit more surprising. The `ajc` compiler does not reserve keywords in patterns, not even the regular Java keywords (except for the bug with the `if` pseudo token). For example, the method pattern `* try(String)` is accepted by `ajc`. Of course, this is not very useful since there can never be a method with this name, but for now we follow this decision. As a result of this, the identifier pattern `new` is allowed for the name of a method in a method pattern. Surprisingly, the constructor pattern `*Handler+.new()` can now also be parsed as a method pattern by splitting the `*Handler` identifier pattern after any of its characters. The part before the split then serves as a type pattern for the return type of the method. For example, one of the results of parsing are the method patterns `* Handler+.new()` and `*H andler+.new()`. The reason for this is that SDF does not by default apply a longest-match policy. Of course, this split is not desirable, so to disallow this, we define a longest-match policy *specifically* for identifier patterns using a follow restriction, which forbids derivations where an identifier pattern is followed by a character that can occur in a pattern.

```
IdPattern -/- [a-zA-Z0-9\_\$\*]
```

AJC *Compatibility*

In Section 5.5.3 we have discussed the pseudo keyword policy of `ajc` in detail. Basically, the pseudo keywords of AspectJ are only reserved for a few specific language constructs. This can concisely be expressed using reject productions,

which allow the definition of reserved keywords *per nonterminal*. Similar to the `PrimPointcutName` we introduced earlier, a new nonterminal for pseudo keywords can be used. For all the language constructs that cannot be pseudo keywords, a reject production is defined. For example:

```
"aspect" | "pointcut" | "privileged" | "before"
    | "after" | "around" | "declare" -> PseudoKeyword
PseudoKeyword -> TypeName          {reject}
PseudoKeyword -> PackageOrTypeName {reject}
```

The first production handles the case where a typename is a single identifier (e.g. `aspect`). The second case rejects pseudo keywords as the first identifier of the qualifier of a typename (i.e. a package- or typename), which corresponds to the behavior of `ajc`, where pseudo keywords are not allowed as the first identifier of a typename. Finally, to be more compatible with `ajc`, AJC could produce parse errors for incorrect floating-point literals in name patterns by defining the syntax of incorrect floating-point literals and the name patterns that contain them. These patterns can then be rejected as name patterns. If this behaviour were required, then this might be useful, but for now we leave this as an 'incompatibility'.

ABC *Compatibility*

While extending the syntax definition for compatibility with `ajc` was relatively easy, extending the definition (as we have presented it until now) to become compatible with `abc` is substantially more difficult, if undertaken without the appropriate solutions. First, we discuss how a relatively easy restriction of `abc` can be enforced. This leads to the explanation why other restrictions are impossible to solve concisely in the current setup. For this, and for the definition of AspectJ extensions, we present a novel method of combining languages using *grammar mixins*. Grammar mixins then arise as the key mechanism for composing the languages involved in AspectJ. After discussing grammar mixins, we return to the ABC compatibility.

KEYWORDS AND NAME PATTERNS In Section 5.6 we have discussed that the `abc` compiler reserves a different set of keywords per lexical state. For example, in the lexical state of a pointcut, `abc` reserves all the names of primitive pointcut designators. To support these keywords (such as the rather common `get` and `set`) in identifier patterns, they are explicitly allowed by the grammar of `abc` (Section 5.6.2). In SDF, this is not an issue: keywords are reserved *per nonterminal*, so keywords that have been reserved for identifiers are still allowed as identifier patterns. As opposed to `ajc`, `abc` does *not* allow regular Java keywords as identifier patterns, so the previous example of the method pattern `* try(String)` results in a syntax error. In our ABC compatible variant, this is handled by rejecting plain Java keywords as identifier patterns:

```
Keyword -> IdPattern {reject}
```

However, it is not obvious how the context-sensitive keywords of `abc` could be defined. For example, consider the following candidate for making primitive pointcut names keywords:

```
PrimPointcutName -> Keyword
```

Unfortunately, adding this production reserves keywords in every context, not just in pointcuts. The previous reject production for `IdPattern` illustrates why this is the case: we only have a single keyword nonterminal and in this way we cannot have context-specific sets of keywords. Moreover, we have just a single identifier nonterminal (`Id`), but an identifier can occur in every context, and for every context we need to reserve a different set of keywords. Since we cannot refer to an identifier in a specific context, it is impossible to define reserved keywords for it. Grammar mixins are a solution for this, but are more generally useful than just for defining reserved keywords.

5.8.4 *Grammar Mixins*

In the context of object-oriented programming, mixins are abstract subclasses that can be applied to different superclasses (i.e. are parameterized in their superclass) and in this way can form a family of related classes [Bracha & Cook 1990]. In the context of grammars, grammar mixins are syntax definitions that are parameterized with the context in which they should be used. The key observation that leads to the use of mixins for defining AspectJ is that the language uses multiple instances of Java, which are mixed with the new language constructs of AspectJ. For example, a Java expression in the context of an `if` pointcut is different from a Java expression in an advice declaration or in a regular Java class. Similarly, an identifier in the context of a pointcut is different from an identifier in an aspect body declaration. Therefore, it should be possible to handle them as separate units, which would make it possible to customize them separately.

Therefore, the Java language should be reusable in the definition of a new language, where the Java syntax effectively becomes part of the new syntax definition, i.e. if syntax definition $\mathcal{A}_1$ imports $\mathcal{B}$ and $\mathcal{C}$ using mixin composition, then the syntax of $\mathcal{B}$ and $\mathcal{C}$ should effectively become part of $\mathcal{A}_1$. A different language $\mathcal{A}_2$ should be able to compose itself with $\mathcal{B}$ or $\mathcal{C}$ and modify this new composition without affecting the other combination of $\mathcal{A}_1$, $\mathcal{B}$, and $\mathcal{C}$.

Grammar mixins provide a more flexible way of composing languages compared to the plain import mechanisms of SDF that we have been using until now. Using grammar mixins, Java can be mixed with pointcuts, name patterns, and aspects and each of these combinations is again a unit for composition. Also, it is possible to extend, customize, or restrict the Java language only for some specific combination. In particular, SDF grammar mixins flourish because the syntax definitions that are subject to mixin compositions are complete: the lexical as well as the context-free syntax is being composed and can *both* be customized for a specific composition. In the next section we will show how grammar mixins can be used to their full potential to combine AspectJ language extensions by unifying mixin compositions.

SDF Implementation For the implementation of grammar mixins we make use of a combination of existing SDF features whose applicability to syntax

```
module JavaMix[Ctx]   51
imports Java          52
  [ CompilationUnit => CompilationUnit[[Ctx]]   53
    TypeDec         => TypeDec[[Ctx]]
    ...
    FieldAccess     => FieldAccess[[Ctx]]
    MethodSpec      => MethodSpec[[Ctx]]
    Expr            => Expr[[Ctx]]  ]
```

Figure 5.15 SDF grammar mixin for Java.

```
module AspectJ[JavaCtx AspectCtx PointcutCtx PatternCtx]
imports
  JavaMix[JavaCtx]                              54
  JavaMix[AspectCtx]
  JavaMix[PointcutCtx]
  JavaMix[PatternCtx]
  aspect/Declaration[AspectCtx JavaCtx]         55
  pattern/Main[PatternCtx]                      56
  pointcut/Expression[PointcutCtx JavaCtx]      57
```

Figure 5.16 Main module of grammar mixin-based AspectJ

definition had not been fully explored previously: parameterized modules and parameterized symbols. Figure 5.15 shows the SDF implementation of the mixin module for Java. An SDF grammar mixin is an SDF module that has a formal parameter 51 that identifies a particular mixin composition. By convention this parameter is called `Ctx` (for context) and the module name has the suffix `Mix`. This grammar mixin module imports the real syntax definition 52 and applies a renaming 53 to all the nonterminals of the grammar, which places these nonterminals in the given `Ctx` by using a parameterized nonterminal. The list of renamings covers all the nonterminals of the language, which can be a very long list that is tedious to maintain. Therefore, we provide a tool `gen-sdf-mix` that generates a grammar mixin module given an SDF syntax definition. The grammar mixin is never modified by hand, so it can be regenerated automatically.

All grammar mixins that are imported using the same symbol for `Ctx` are subjected to mixin composition. In a way, the import statement of SDF and `Ctx` symbol are the mixin composition operators of grammar mixins. For grammar mixins, composition means that the grammars of the syntax definitions involved in a composition are fully automatically combined, based on the normal SDF grammar composition semantics (which are also applied to plain imports).

5.8.5 *AspectJ in the Mix*

Now we have revealed the actual design of the syntax definition, we need to revise the presentation of the AspectJ syntax. Figure 5.16 shows the imports of the main module of the syntax definition. The AJF, AJC, and ABC variants im-

```
AspectDec                 -> TypeDec[[JavaCtx]]                (see 47)
ClassBodyDec[[AspectCtx]] -> AspectBodyDec                     (see 48)
AspectDec                 -> ClassMemberDec[[JavaCtx]]         (see 49)
PointcutDec               -> ClassMemberDec[[JavaCtx]]         (see 50)

"before" "(" {Param[[AspectCtx]] ","}* ")" -> AdviceSpec       (see 27)
"if" "(" Expr[[JavaCtx]] ")" -> PointcutExpr                   (see 35)
 PrimType[[PatternCtx]]      -> TypePattern                    (see 38)
```

Figure 5.17 AspectJ productions updated to grammar mixins. The numbers refer to the productions mentioned earlier.

port this module and the variant specific modules. The `AspectJ` module itself has four contexts parameters, to make the mixin composition configurable for AspectJ extensions. `AspectJ` imports the grammar mixin `JavaMix` four times, once for every context. This makes all the nonterminals of Java available to AspectJ in these four contexts. The choice of the four contexts is somewhat arbitrary. For example, it might be a good idea to introduce an additional context for advice. Fortunately, this is very easy to do by just importing another instance of the Java grammar mixin with a symbol for that context. Our syntax definition has one context more than the `abc` scanner has lexical states: `abc` does not place patterns in a separate context.

Next, the modules for the sublanguages are imported, passing the required contexts as parameters to the modules. For example, pointcut expressions 57 need to know their own context, but also the context of regular Java expressions.

The imports of `JavaMix` and the sublanguage modules automatically compose all mixin compositions, but we still need to make some interactions explicit, like we did earlier in Figure 5.14. However, this time the productions also connect nonterminals from different contexts (mixin compositions). Figure 5.17 shows some of the production rules that we have discussed earlier, but this time using the context parameters. For example, aspect declarations are type declarations in the `JavaCtx` 47, but all the arguments of the aspect declaration will be in the context of aspects, so an aspect declaration changes the context from `JavaCtx` to `AspectCtx` in this case. The second production 48 defines that regular Java class body declarations from the aspect context can be used as aspect body declarations. The productions for aspect 49 and pointcut declarations 50 make these constructs available as class members in the regular Java context. Advice specifiers 27 use Java's formal parameters from the aspect context. The `if` pointcut expression takes an expression from the regular Java context as an argument. For Abc compatibility, we will later define reserved keywords *per context*. By using the expression from the Java context, aspect and pointcut-specific keywords will be allowed in this Java expression. Finally, the type pattern for primitive types 38 now uses a primitive type from the pattern context.

Note that the choice of the context of a symbol is completely up to the language designer: for every production argument we can choose the most

appropriate context. The choice of the context switches (lexical state transitions) is not influenced by the complexity of recognizing the context during lexical analysis. In the next section we show that this enables language designers to improve their language designs.

5.8.6 Abc *Compatibility Revised*

Thanks to the grammar mixins, we can now declare a different set of reserved keywords for each context. The AspectJ grammar now has four nonterminals for identifiers: `Id[[JavaCtx]]`, `Id[[AspectCtx]]`, `Id[[PointcutCtx]]`, and `Id[[PatternCtx]]`. Similarly, there are four nonterminals for keywords. Thus, the syntax definition can now reject a different set of reserved keywords for each specific context. The reject production is in fact already defined in the Java modules imported by the AspectJ definition, so we only need to extend the existing set of keywords. For the Java context, `abc` introduces three new keywords:

```
"privileged" | "aspect" | "pointcut" -> Keyword[[JavaCtx]]
```

For the aspect context, `abc` introduces a series of new keywords. Also, every keyword from the Java context is a keyword in aspect context.

```
"after" | ... |"proceed" -> Keyword[[AspectCtx]]
Keyword[[JavaCtx]] -> Keyword[[AspectCtx]]
```

However, `proceed` is now a reserved keyword in aspect declarations, so it is no longer allowed as the name of a method invocation, which now rejects the special `proceed` call for invoking the original operation in an `around` advice. To reintroduce the `proceed` call, we need to allow it explicitly as a method specifier in the aspect context (note that an advice context would be useful here, though that would not be compatible with `abc`, which is the whole point of this exercise).

```
"proceed" -> MethodSpec[[AspectCtx]]
```

In the context of pointcuts, `abc` reserves the Java keywords, primitive pointcut names, and some additional keywords from the context of aspects.

```
Keyword[[JavaCtx]]          -> Keyword[[PointcutCtx]]
PrimPointcutName            -> Keyword[[PointcutCtx]]
"error" | ... | "warning" -> Keyword[[PointcutCtx]]
```

Finally, we still need to define keywords for the context of patterns, since our syntax definition uses a separate context for that. In `abc`, these two states are merged, so defining pattern keywords is easy:

```
Keyword[[PointcutCtx]] -> Keyword[[PatternCtx]]
Keyword[[PatternCtx]]  -> IdPattern {reject}
```

We have now defined the keyword policy of `abc` in a declarative way as a modular extension of the basic syntax definition.

In the last few years, there has been a lot of research on extensions of AspectJ. For experimenting with aspect-oriented language features, an *extensible compiler* for AspectJ is most useful. One of the goals of the `abc` project is to facilitate this research by providing such an extensible compiler. The previous sections have highlighted a few challenges for the definition of the syntax of AspectJ and the implementation of an AspectJ parser. The result of this complexity is that the parsers of `ajc` and `abc` are more complex than usual, since the requirements imposed on the parser by the language do not match the conventional parsing techniques too well.

This section demonstrates these limitations through several existing extensions and their issues. We compare the implementation of the *syntax* of the extensions in `abc` to the definition of the syntax in SDF, based on the syntax definition for AspectJ that we presented in the previous section. We would like to emphasize that this discussion is all about the *syntax* of the extensions, and not about the other compiler phases. Our modular and declarative approach for the definition of the syntax of AspectJ does not suddenly make the *complete* implementation of AspectJ extensions trivial, since a lot of work is going on in later compiler phases.

5.9.1 *Issues in Extensibility*

The `abc` compiler is based on Polyglot [Nystrom et al. 2003], which provides PPG, a parser generator for extensible grammars based on CUP, a LALR parser generator. The extensibility features of PPG are based on manipulation of grammars, with features such as `drop` a symbol, `override` productions of a symbol, and `extend` the productions of a symbol. This way of extending a grammar works in practice for most of the language extensions that have been implemented for `abc` until now. Unfortunately this is not a truly modular mechanism, since LALR grammars do not compose, which means that the user of PPG has to make sure that the composed grammar stays in the LALR subclass of context-free grammars. For example, we have discussed the problem of around advice and method declarations with the name `around`. The `abc` compiler overcomes some of these issues by reserving keywords.

PPG does not feature an extensible scanner, so the `abc` compiler implements its own, stateful scanner as we have discussed in detail. This works fine for the basic AspectJ language, but it is inherently not modular. The rules for switching from context are based on knowledge of the entire language that is being scanned, which breaks down if the language is extended in an unexpected way. The `abc` scanner allows extensions to add keywords to specific states of the scanner. In this way, it is relatively easy to add keywords, but it is difficult to add operators and it is much more difficult to add new scanner states. For example, suppose that AspectJ did not define an `if(...)` pointcut. It would have been non-trivial to extend the scanner to handle this pointcut, since it requires the introduction of a new lexical state that affects several aspects of

```
module HelloWorld[JavaCtx AspectCtx PointcutCtx]
exports
  context-free syntax
    "cast" "(" TypePattern ")" -> PointcutExpr 58

    "global"":" ClassNamePattern ":" PointcutExpr ";" -> PointcutDec 59

    "cflowlevel" "(" IntLiteral[[JavaCtx]] "," PointcutExpr ")"
                                                    -> PointcutExpr 60
  lexical syntax
    "cast"       -> Keyword[[PointcutCtx]]
    "cflowlevel" -> Keyword[[PointcutCtx]]
    "global"     -> Keyword[[JavaCtx]]
    "global"     -> Keyword[[AspectCtx]]
```

Figure 5.18 Syntax of some abc extensions implemented in SDF

the scanner. In these situations, the scanner has to be copied and modified, which is undesirable for maintenance and composition of extensions.

The modular syntax definition we have presented solves many of these issues, since the definition itself can be extended in a modular way as well. Context or lexical state management is not based on rudimentary context-free parsing in the scanner, but fully integrated in the parser by the use of scannerless parsing. Moreover, contexts can be unified by mixin composition and ambiguities can be resolved in a modular way.

5.9.2 *Simple Extensions*

First, we discuss some small AspectJ extensions that are part of the EAJ (Extended AspectJ) extension of abc. The SDF implementation of the extensions is shown in Figure 5.18. Similar to the way Java is extended, the AspectJ syntax definition can be extended by creating a new module that imports AspectJ and adds new constructs.

Cast and Global Pointcuts

The cast pointcut designator 58 can be used to select points in the program where an implicit or explicit cast is performed. This is a very simple pointcut designator, yet this simple example already introduces a problem the implementer of the extension should be aware of. The keyword `cast` is reserved in the context of a pointcut, which means that it is no longer allowed as part of a name pattern (see Section 5.6.2). To resolve this, the keyword should be added to the *simple name patterns* explicitly, which has not been done for this extension in the abc implementation. The same problem occurs in the implementation of global pointcuts 59 (a mechanism for globally restricting some aspects by extending their pointcut definitions). In our syntax definition this is not an issue, since the keywords are reserved per nonterminal.

```
module AspectJMix[Ctx]
imports AspectJ
  [ AspectDec     => AspectDec[[Ctx]]
    AspectBodyDec => AspectBodyDec[[Ctx]]
    ...
    TypePattern   => TypePattern[[Ctx]]
    PointcutExpr  => PointcutExpr[[Ctx]] ]
```

Figure 5.19 Grammar Mixin for AspectJ

CFlow Level

The `cflowlevel` [10] pointcut designator is an extension used to select join points based on the level of recursion. The `cflowlevel` pointcut designator takes two arguments: a number for the recursion level and a pointcut. However, the lexical state for pointcuts in `abc` does not allow integer literals. To avoid the need for a new lexical state or other complex solutions, the syntax of the `cflowlevel` construct was changed to a string literal, which is supported in the pointcut lexical state [11]. Unfortunately, in this case the syntax of the extension was designed to fit the existing lexical states of the scanner. In the SDF implementation of this extension referring to an integer literal is not a problem.

5.9.3 *Open Modules*

Open modules were proposed by Aldrich [Aldrich 2005] to solve the coupling issues that arise between aspects and the code they advise. It provides an encapsulation construct that allows an implementation to limit the set of points to which external advice applies. Recently, an `abc` extension was proposed that extends open modules to full AspectJ ([Aldrich 2005] deals with a small functional language) and defines appropriate notions of module composition [Ongkingco et al. 2006]. The normal form of open modules as proposed in [Ongkingco et al. 2006] is as follows:

```
module ModuleName {
   class class name pattern
   friend list of friendly aspects
   expose : pointcut defining exposed join points
}
```

A module declaration applies to a set of classes as specified in the `class` part. It states that aspects can only advise join points matched by the pointcut specified in the `expose` part. *Friendly aspects*, listed in the `friend` part, have unrestricted access to the join points occurring within classes of the module. The exact syntax is more elaborate for notational convenience, and also includes constructs for restricting or opening modules upon composition.

The parsing of open modules requires a new lexical state. This need falls out of the designed extensibility of `abc`, as highlighted in Section 5.6. As a

[10] Available at `http://www.cs.manchester.ac.uk/cnc/projects/loopsaj/cflowlevel/`
[11] See http://abc.comlab.ox.ac.uk/archives/dev/2005-Aug/0003.html

```
module OpenModule[JavaCtx]
exports
  context-free syntax
    ModDec+ -> CompilationUnit[[JavaCtx]]
    Root? "module" Id "{" ModMember* "}" -> ModDec
    "class"     ClassNamePattern ";"  -> ModMember
    "friend"    {AspectName ","}+ ";" -> ModMember
    "open"      {Module ","}+ ";"     -> ModMember
    "constrain" {Module ","}+ ";"     -> ModMember
    Private? "expose"    ToClause? ":" PointcutExpr ";" -> ModMember
    Private? "advertise" ToClause? ":" PointcutExpr ";" -> ModMember
    "to" ClassNamePattern -> ToClause

lexical syntax
  "root" | "module"            -> Keyword[[JavaCtx]]
  "module" | ... | "advertise" -> Keyword
```

Figure 5.20 SDF module extending AspectJ with open modules.

consequence the full scanner has to be copied and modified. Although just 15 lines of code had to be modified in the copy, this introduces a maintenance problem: copying the scanner implies that the developer of the extension has to keep the extension in sync with the main scanner of abc, which is bound evolve, for example to introduce support for AspectJ 5.

Conversely, SDF allows the syntax of open modules to be concisely and modularly expressed, as illustrated in Figure 5.20. A new context can be introduced in a *modular* way. The implementation is based on the AspectJ grammar mixin module of Figure 5.19.

5.9.4 *Context-Aware Aspects*

We now consider the AspectJ syntax extensions for the pointcut restrictors proposed in [Tanter et al. 2006] for *context-aware aspects*. Context-aware aspects are aspects whose pointcuts can depend on external *context* definitions, and whose advices may be parameterized with context information. Contexts are stateful, parameterized objects: they are specified by implementing a context class with a method that determines whether the context is active or not at a given point in time; context activation can be based on any criteria, like the current control flow of the application, some application-specific condition, or input from environment sensors. A context is an object that may hold relevant state information (such as the value obtained from a given sensor).

Context-aware aspects [Tanter et al. 2006] propose a number of general-purpose and domain- or application-specific pointcut restrictors for restricting the applicability of an aspect based on some context-related condition. These pointcut restrictors are explained using an AspectJ extended syntax, although only a framework-based implementation is provided, based on the Reflex AOP kernel [Tanter & Noyé 2005].

The `inContext` pointcut restrictor is similar to an `if` pointcut designator, restricting the applicability of an aspect (e.g. `Discount`) to the application currently being in a certain context (e.g. `PromotionCtx`):

```
pointcut amount():
  execution(double Item.getPrice()) && inContext(PromotionCtx);
```

Also, context-aware aspects provide a mechanism to expose state associated to the context (e.g. a discount rate) as a pointcut parameter, subsequently it can be used in the advice. In the following example, the `rate` property of the `PromotionCtx` is exposed in the pointcut and subsequently used in the advice to compute the associated discount.

```
aspect Discount {
  pointcut amount(double rate):
    execution(* ShoppingCart.getAmount())
        && inContext(PromotionCtx(rate));

  double around(double rate): amount(rate) {
    return proceed() * (1 - rate);
  }
}
```

Context activation can be parameterized in order to foster reuse of contexts. For instance, a stock overload context can be parameterized with the ratio of stock overflow required to be considered active. In the following example, the `amount` pointcut matches only if the stock overload factor is superior to 80% when the rest of the pointcut matches.

```
pointcut amount():
  execution(* ShoppingCart.getAmount())
      && inContext(StockOverloadCtx[.80]);
```

An important characteristic of the approach presented in [Tanter et al. 2006] is the possibility to extend the set of pointcut restrictors, either general purpose or domain/application specific. Hence the set of context restrictors is *open-ended*. An example of a general-purpose restrictor is one that makes it possible to refer to past contexts, such as the context at creation time of an object. For instance, the `createdInCtx` restrictor in the next example refers to the context in which the currently-executing object *was* created. The `amount` pointcut matches if the current shopping cart was *created* in a promotional context, independently of whether the promotion context is still active at check-out time.

```
pointcut amount():
  execution(* ShoppingCart.getAmount()) && createdInCtx(PromotionCtx);
```

An example of application-specific restrictor is `putInCartInCtx`, which refers to the context at the time an item was put in the shopping cart:

```
pointcut amount():
  execution(* Item.getPrice()) && putInCartInCtx(PromotionCtx);
```

```
module CtxAspect
exports
  context-free syntax
    "inContext" "(" ActualCtx ")"    -> PointcutExpr
    "createdInCtx" "(" ActualCtx ")" -> PointcutExpr
    TypeName[[JavaCtx]] ACParams? ACValues? -> ActualCtx
    "[" {Expr[[JavaCtx]] ","}+ "]" -> ACParams
    "(" {CtxId[[JavaCtx]] ","}+   ")" -> ACValues

  lexical syntax
    "inContext" | "createdInCtx" -> Keyword[[PointcutCtx]]
```

```
module EShopCtxAspect
imports CtxAspect
exports
  context-free syntax
    "putInCartInCtx" "(" ActualCtx ")" -> PointcutExpr

  lexical syntax
    "putInCartInCtx" -> Keyword[[PointcutCtx]]
```

Figure 5.21 Two SDF modules for context-aware aspects: (top) general-purpose pointcut restrictors; (bottom) application-specific extension for the EShop.

Parsing Context-Aware Aspects

Extending AspectJ with the two general-purpose context restrictors `inContext` and `createdInCtx` can be defined in a `CtxAspect` SDF module (Figure 5.21 (top)). The context-free syntax section defines the new syntax: a context restrictor followed by the actual context definition; a context is a Java type name, with optional parameters and values (for state exposure). The lexical syntax section specifies that the new pointcut restrictors have to be considered as keywords in a pointcut context.

Figure 5.21 (bottom) shows a modular syntactic extension for context-aware aspects with the definition of the `putInCartInCtx` application-specific restrictor. Interestingly, it is not necessary to redefine the syntax for parameters and values in the new syntax extension definition (`ActualCtx` is visible from `EShopCtxAspect`).

5.10 PERFORMANCE

Deriving a production quality (i.e. efficient and with language-specific error reporting) parser from a declarative, possibly ambiguous, syntax definition is one of the open problems in research on parsing techniques. In particular, the area of scannerless parsing is relatively new and the number of implementations is very limited (i.e. about 2). This work does not improve the performance, error reporting or error recovery of these parsers in any way: besides the arguments for a declarative specification of AspectJ, it only provides a strong motivation for continued research on unconventional parsing

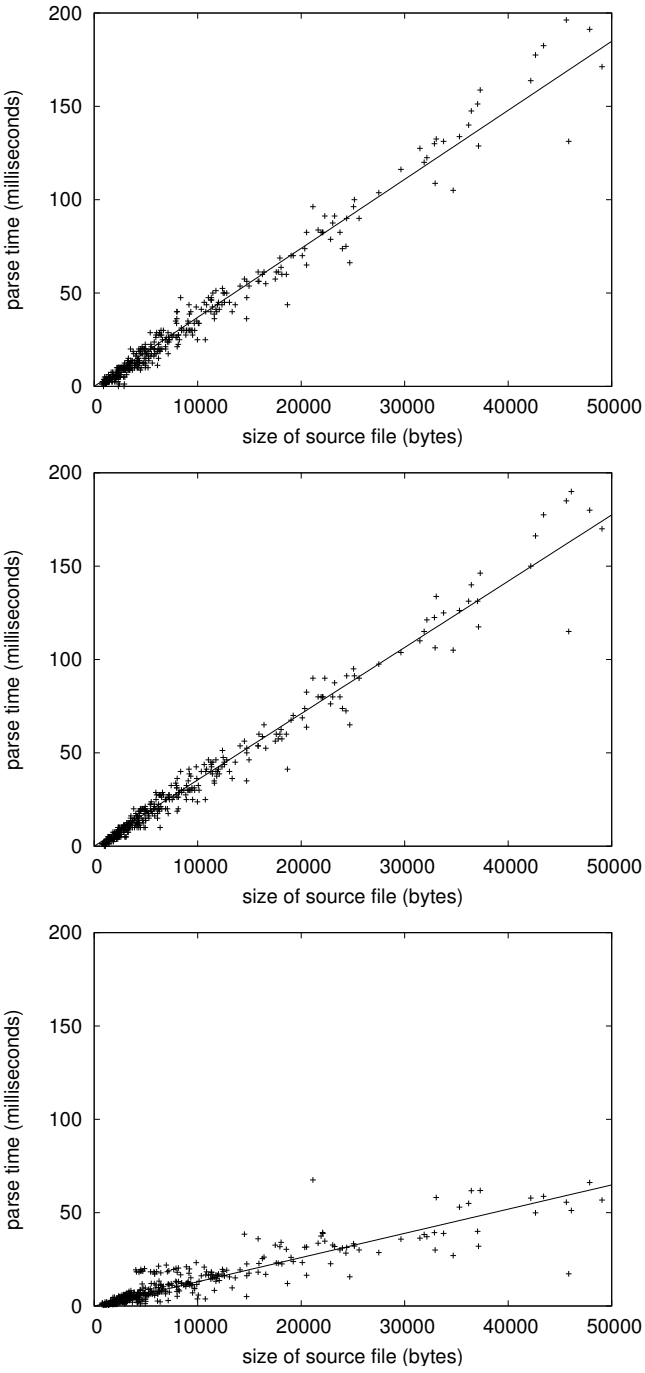


Figure 5.22 Benchmark of parsing Java source files. Top to bottom: sglr/ajc, sglr/java, and abc

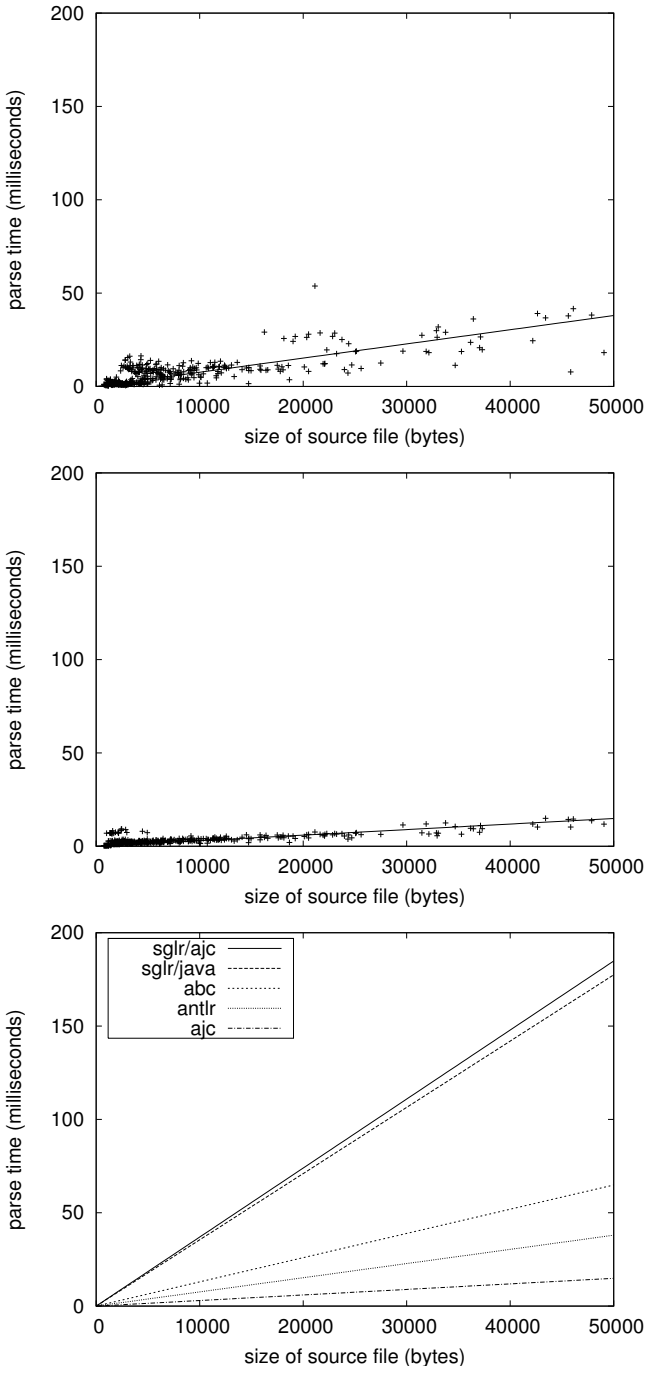


Figure 5.23 Benchmark of parsing Java source files. Top to bottom: ANTLR, `ajc`, and the trend lines of all the benchmarked parsers in a single graph

techniques. Although our current objectives are not to replace every single parser in a production compiler by a scannerless generalized LR parser, it is good to get an impression of the current state of a scannerless generalized LR parser compared to parsers used in existing compilers.

In order to evaluate the applicability of our approach beyond specification purposes, we have performed some benchmarks to estimate the efficiency of scannerless generalized LR parsing. It has been shown that although $O(n^3)$ in the worst case (with n the length of the input), generalized LR performs much better on grammars that are near-LR [Rekers 1992], and that the cost of scannerless parsing is linear in the length of the input, although with an important constant factor [Salomon & Cormack 1989]. There is little knowledge of how the integration of scannerless and generalized LR parsing performs. We hereby compare the cost of the SGLR parser with that of `abc`, `ajc`, and ANTLR [Parr] (an LL(k) parser generator) when parsing both a massive amount of Java code and the AspectJ testsuite of `abc`.

5.10.1 *Benchmark Setup*

The test machine is an Intel Pentium 4 3.2GHz CPU with 1GB memory, running SUSE 9.0. The `abc`, `ajc`, and ANTLR parsers use the Sun JDK 5.0. SGLR 3.15 is invoked with heuristic filters and cycle detection disabled. For all parsers, we only measure the actual parse time: this includes the construction of the parse tree, but no semantic analysis and I/O costs. In all benchmarks, the same source file is parsed 15 times and the first 10 parses are ignored to avoid start-up overhead (class loading and JIT compilation for Java, parse table loading for SGLR). For ANTLR we use version 3.0b3 and a recent Java 1.5 grammar written by Terence Parr.

5.10.2 *Benchmark Results*

Figure 5.23 shows the results of the Java benchmark: parsing of the source files of the Azureus Bittorrent client and Tomcat 5.5.12. This figure shows that parsing with all parsers is linear in the size of the input, illustrated by the trend lines (calculated using least-squares). SGLR parsing with the AspectJ grammar is about 4% slower than parsing with the Java grammar. The constant factor of parsing with `abc` is about 40% of the factor of SGLR. Clearly, the performance of `ajc` is superior to all the other parsers. The performance of the ANTLR Java parser is more or less between the `abc` and `ajc` parsers, but this is a plain Java parser. The creation of ANTLR parsers has been heavily optimized in this benchmark after noticing the substantial setup cost of ANTLR3 parsers. The absolute times are all fractions of second, which is only a very small portion of the total amount of time required for compiling an AspectJ program, since the most expensive tasks are in semantic analysis and actual weaving of aspects.

Figure 5.24 shows the results for parsing aspect code from the testsuite of `abc`. Note that the scales are different, since aspect sources are typically

smaller. Again, the parse time is linear in the size of the input, but the constant factor of `abc` is about 60% of the factor of SGLR. The performance of SGLR compared to `ajc` has improved as well. For both Java parsers, parsing source files close to 0 bytes is relatively expensive. The reason for this is JIT compilation, which still introduces start-up overhead after parsing the same file 10 times before the actual benchmark. At first, we ignored just the first two parses, which had a dramatic impact on the performance. Overall, the parse time is always smaller than 0.06 second, so the absolute differences are extraordinarily small for these tiny source files. We would have to benchmark larger aspect sources (which do not exist yet) to get more insight in the performance of parsing aspects and pointcuts.

As a matter of fact, a typical project consists of a lot of Java code with a few AspectJ aspects, so the Java benchmark is particularly relevant. To conclude, the absolute and relative performance of scannerless generalized LR parsing is promising for the considered grammars (Java and AspectJ). The fact that the parsers are implemented in Java versus C is not relevant, since the most important question is whether SGLR is fast enough in absolute time. Nevertheless, since there is virtually no competition in the area of scannerless parsing at present, there is ample opportunity for research on making the performance of scannerless parsing even more competitive.

5.10.3 *Testing*

The compatibility of the AJC syntax definition is tested heavily by applying the generated parser to all the valid source files of the testsuite of `ajc` 1.5.0. Testing invalid sources requires the examination of the full `ajc` testsuite to find out if tests should fail because of semantic or syntactic problems. This is a considerable effort, but will be very useful future work. The results of the testsuite are available from the web page mentioned in the introduction.

5.11 DISCUSSION

5.11.1 *Previous Work*

Although SDF has a long history [Heering et al. 1989], a more recent redesign and reimplementation as SDF2 [Visser 1997b, van den Brand et al. 2002] has made the language available for use outside of the algebraic specification formalism *ASF+SDF*. This redesign introduced the combination of scannerless and generalized LR parsing [Visser 1997a].

In [Bravenboer & Visser 2004 (Chapter 2)] we motivated the use of SGLR for parsing embedded domain-specific languages. This method, called MetaBorg, focuses on creating new language combinations, where it is important to support combinations of languages with a different lexical syntax. In [Bravenboer et al. 2005 (Chapter 3)] we presented the introduction of *concrete object syntax* for AspectJ in Java as a reimplementation of the code generation tool Meta-AspectJ [Zook et al. 2004]. In that project, we used the AspectJ syntax

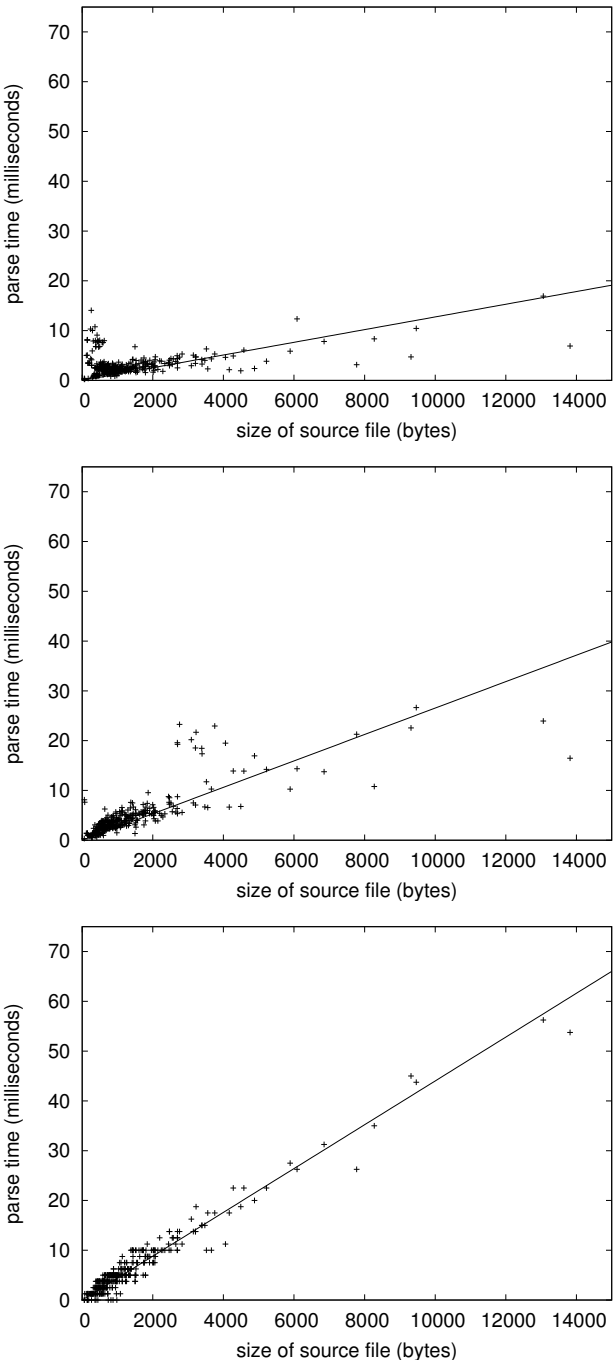


Figure 5.24 Benchmark of parsing AspectJ source files. Top to bottom: `ajc`, `abc`, sglr/ajc

definition of this chapter, but the design and benefits of the grammar were not discussed.

Compared to this earlier work, we have discussed the design of the syntax definition of AspectJ, which poses a challenge to parser generators. We discussed in detail how complex differences in lexical syntax of the involved languages can concisely be defined in SDF, and how this is related to a stateful lexer. The syntax definition for AspectJ provides a compelling example of the application of scannerless parsing to an existing language, where all of the following features of SDF prove their value: modular syntax definition, rejects for keywords, scannerless parsing, generalized LR parsing, parameterized symbols, and parameterized modules. Grammar mixins are a surprisingly useful application of parameterized modules and non-terminals. Also, we presented a solution for implementing context-sensitive keywords in SDF.

5.11.2 *Related Work*

Using a scannerless parser for AspectJ has not been proposed or even mentioned before in the literature. Concerning AspectJ implementations, we have discussed the `abc` and `ajc` scanners and parsers at length.

The advantage of separate scanners is their foundation on finite automata, which allows fast implementations. However, this characteristic also implies obliviousness to context, while processing languages such as AspectJ requires introducing context-sensitivity into scanners. There are essentially two options to make scanners context-sensitive. First, the scanner may interact with the parser to retrieve the context of a token. This is not very difficult to implement in a handwritten parser, but parser generators based on this approach are rare, and as far as we know none have been used to parse AspectJ. Blender [Begel & Graham 2004] uses GLR parsing with an incremental lexical analyzer that is forked together with the LR parsers in the GLR algorithm. A similar approach was also used by the implementation of SDF before the introduction of scannerless parsing [Heering et al. 1989]. Second, lexical analysis may be extended with a rudimentary form of context-free parsing to recognize the global structure of the source file while scanning by means of *lexical states*, without interaction with the parser. This approach is used in the `abc` scanner and parser for AspectJ.

DURA-LexYt [Blasband 2001] supports lexical analysis with multiple interpretations of the input. As opposed to scannerless parsing, a separate scanner with support for backtracking is used. In this way, choosing the correct lexical interpretation can be controlled by the parser without the need for managing lexical states in the scanner. DURA provides several levels of lexical backtracking to facilitate typical scenarios of tokenization (e.g. a single or multiple divisions into tokens), whereas scannerless parsing requires this to defined explicitly. Lexical backtracking can be used for context-sensitive lexical analysis, but does not facilitate context-specific reserved keywords. DURA-LexYt does not support inherent ambiguities: the parser always returns a single parse tree, which might also not be the one desired. More experience with

lexical backtracking is required to get insight in the performance compared to scannerless parsing.

JTS Bali [Batory et al. 1998] is a tool for generating parsers for extensions of Java. It supports composition of lexical syntax based on heuristics such that the best results are produced in the common cases. For example, keyword rules are put before the more general rules, such as for identifiers. This means that it cannot handle lexical state and it not suitable for defining AspectJ-like extensions of Java.

Parsing techniques with higher-order (parameterization) features, such as parser combinators in higher-order functional languages [Hutton 1992], allow reuse and abstraction over grammars, but do not support *unanticipated* reuse of a grammar. Grammar mixins, on the other hand, are modules based on unparameterized grammars (e.g. Java) that make this grammar reusable and allow unanticipated modification of the grammar in every context.

5.11.3 *Future Work*

Grammar Mixins

In this chapter we have applied grammar mixins and explained their functionality only in an informal way. In future work, we plan to make the notion of grammar mixins more formal. In particular, the semantics of mixin composition of grammars that already use mixins itself needs to be defined more precisely. Also, grammar mixins should be integrated in a syntax definition formalism. Currently, an external tool is used to generate grammar mixin modules, which is not desirable. Furthermore, a notion of interfaces for grammar mixins would be useful to separate the implementation of a mixin from its interface. Finally, multiple instantiations of a grammar mixin for a relatively large language, such as Java or AspectJ has a major impact on the performance of the parser generator, which could again be solved by integration of grammar mixins in the syntax definition formalism. Chapter 6 presents a first ingredient of this solution: *parse table composition*. Parse table composition enables separate compilation of grammars involved in a language conglomerate. The grammar of Java is compiled to a parse table component that can be instantiated multiple times in an efficient way. In this way, we avoid the application of the parser generator to all the separate instantiations.

Improvements to SDF and SGLR

As we have shown in this chapter, SDF provides a declarative approach to solving complex parsing problems. Yet, the formalism and tools are not in widespread use. What may be the reason for this (other than publicity) and what improvements can be made?

Rule Syntax SDF's reverse grammar production rules may make developers accustomed to BNF style rules uncomfortable. It might make sense to provide a version of SDF using such a conventional style.

Performance The benchmarks showed that the performance of the SGLR parser is a constant factor slower than the abc parser, which should be acceptable for use at least in research projects. However, there is good hope that the performance of SGLR can be much improved. There are alternative GLR implementations (e.g. [Aycock & Horspool 1999, McPeak & Necula 2004]) and alternative algorithms such as right-nulled GLR [Scott & Johnstone 2006] with better performance than SGLR. However, these techniques have not yet been extended to scannerless parsing, while scannerlessness is essential in our syntax definition for AspectJ. Even after these techniques are adopted, there remains a theoretical performance gap between GLR and LALR, since the complexity of GLR depends on the grammar. Therefore, it would be useful to develop profiling tools that help grammar developers to detect performance bottlenecks in grammars.

Error Reporting The current error reporting of SGLR is rather Spartan; it gives the line and column numbers where parsing fails. This may be improved using a technique along the lines of the Merr tool that generates error reporting for LR parsers from examples [Jeffery 2003]. This requires an adaption of the techniques where the set of parsing states at the failure point is interpreted.

Analyzing Ambiguities The disadvantage of LR-like parser generators is that the grammar developer is confronted with shift-reduce and reduce-reduce conflicts. However, this is also their advantage; the developer is forced to develop an unambiguous grammar. When using GLR there is no need to confront the developer, however, the conflicts are still there to inspect. It would be useful to develop heuristics that can be used to inspect the conflicts in the parse table and use these to point the developer to problematic parts in the grammar.

Platform A more mundane, not so scientific reason for lack of adoption may be the platform. The SDF parser generator and the SGLR parser are implemented in C and the distribution is Unix/Linux style. Furthermore, parse trees and abstract syntax trees are represented using ATerms, which requires linking with the ATerm library. Retargeting the SDF/SGLR implementation to other platforms, such as Java, may help adoption.

Applications of the AspectJ Syntax Definition

With respect to the AspectJ syntax definition itself, there are a number of applications to consider.

AspectJ Specification For widespread acceptance of aspect-oriented languages, a complete specification of the syntax and semantics of the language is important. In particular, concerns about modifying the semantics of the host language could be reduced by at least having a complete specification of the syntax of the language. If there is enough interest in the specification of the syntax and semantics of the AspectJ language, then we would like to work

with the AspectJ developers to make the current syntax definition even more compatible with `ajc` and make it the basis of such a specification.

As one of the first applications, the `abc` team has used our syntax definition of AspectJ in a definition of the semantics of static pointcuts, defined as a set of rewrite rules from AspectJ pointcuts to Datalog [Avgustinov et al. 2007].

Connecting to the AspectBench Compiler Considering the extensibility goals of `abc`, our modular and extensible definition of AspectJ would be most useful as part of the front-end of `abc`. Also, we have shown that pseudo keywords do not require a handwritten parser, so the `abc` compiler could be made more compatible with syntax accepted by `ajc`.

Multi-language AOP We are currently working on integrating the MetaBorg approach [Bravenboer & Visser 2004 (Chapter 2)] and the Reflex AOP kernel project [Tanter & Noyé 2005] for multi-language AOP. The current AspectJ syntax definition can be used to support AspectJ in Reflex, allowing AspectJ extensions to be prototyped conveniently.

5.12 CONCLUSION

We have presented the design of a modular syntax definition for the complete syntax of AspectJ, i.e. integrating the formalization of the lexical and the context-free syntax of the language. In addition, we have shown that scannerless parsing in combination with an expressive module system can elegantly deal with the context-sensitive lexical syntax of AspectJ. The result is a syntax definition that achieves a new level of extensibility for AspectJ, which is useful for research on aspect-oriented programming extensions. The performance of the scannerless generalized LR parser for this grammar turns out to be linear with an acceptable constant factor, which opens up possibilities for the integration of our solution in extensible compilers for AspectJ.

Furthermore, our work on syntax definition for AspectJ provides guidelines for approaching the current trend to design programming languages that are in fact mixtures of various sublanguages, for example for the integration of search capabilities or concrete object syntax (e.g. LINQ, E4X, XQuery, Cω). The convention of separating the parsing process into a scanner and a parser does not apply to such languages, requiring language designers and implementers to reconsider the parsing techniques to use.

With the syntax definition for AspectJ, we have shown that scannerless generalized LR parsing is not just useful for reverse engineering, meta programming, interactive environments, language prototyping, and natural language processing, but that scannerless generalized LR may at some point be used in compilers for modern general-purpose languages. AspectJ makes a strong case for the use of scannerless parsing to provide concise, declarative specification and implementation of the next-generation of programming languages.

This result provides a strong motivation for addressing the barriers to a wider adoption of scannerless generalized LR parsing that we observed in the previous section.

ACKNOWLEDGEMENTS

At Utrecht University this research was supported by the NWO/Jacquard project TraCE (638.001.201). Éric Tanter is partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile. We thank the abc team for the report on the abc scanner and parser. The description of lexical states was very useful in the development of our syntax definition. Pavel Avgustinov of the abc team provided extensive feedback on the syntax definition. We thank Arthur van Dam for his help with Gnuplot, and Jurgen Vinju for his advice on benchmarking SGLR. We thank Mark van den Brand, Jurgen Vinju and the rest of the SDF/SGLR team at CWI for their work on the maintenance and evolution of the SDF toolset. We thank Rob Vermaas for his help with the implementation of the syntax definition and benchmarking the generated parser. Finally, we would like to thank Eelco Dolstra, the anonymous reviewers of CC 2006, and the anonymous reviewers of OOPSLA 2006 for providing useful feedback on earlier versions of this chapter.

Parse Table Composition

6

ABSTRACT

Composition of context-free languages is useful in a wide range of scenarios, including program transformation and generation with concrete object syntax, language extension, domain-specific language embedding, and the definition of the syntax of language conglomerates such as AspectJ. The generation of parse tables for language combinations is expensive, which is a particular problem when the composition configuration is not fixed, as is for example the case in the instantiation of a template engine for a new target language. In this chapter we introduce an algorithm for parse table composition to support separate compilation of grammars. While the worst-case time complexity of parse table composition is exponential (like the complexity of parse table generation itself), for realistic language combination scenarios involving grammars for real languages, our parse table composition algorithm is an order of magnitude faster than computation of the parse table for the combined grammars, making online language composition feasible.

6.1 INTRODUCTION

Consider the following task: 'In Java, write a method implementing a web service that fetches data from a database, processes it in some way, and then sends the data as XML to the client.' This task involves three languages, i.e. Java, SQL, and XML. However, the SQL and XML code is not treated as code by a typical mainstream development environment. Instead, SQL and XML code fragments are generally composed using string composition with all the associated problems, such as a lack of syntactic and other static checks at compile-time, and vulnerabilities to injection attacks. This situation is not restricted to this particular combination, but holds for any combination of general-purpose (host) language (e.g. Java, C#, C, C++, Haskell) and special-purpose (guest) language (e.g. SQL, HQL, Shell, regular expressions, XPath, XQuery, LDAP).

The situation can be improved by making the host language syntactically and semantically aware of the guest languages. This type of *language composition* has many applications including program transformation and generation with concrete object syntax, language extension, domain-specific language embedding, and the definition of the syntax of language conglomerates such as AspectJ (Section 6.2 provides an extensive motivation). Given the combinatorial problem of possible language combinations, and the fact that new (special-purpose) languages are introduced regularly, it is not feasible to ex-

pect language developers (vendors) to build in support for each and every special purpose language.

Extensible compilers promise to make languages and their tools open to these types of extensions. The state-of-the-art in extensible compilers is represented by tools and frameworks such as Polyglot [Nystrom et al. 2003], MetaBorg [Bravenboer & Visser 2004 (Chapter 2)], Silver [van Wyk et al. 2007], and JastAdd [Ekman & Hedin 2004]. These tools focus on *source-level extensibility*, i.e. creating a language extension by extending the source code of the base compiler and compiling the source code of the extension together with the source code of the base compiler to build a binary compiler that supports the base language with this particular extension. Despite the advances in code reuse techniques applied or introduced by these extensible compiler frameworks, language extension is still a fairly heavyweight process and has not reached the state where language extensions can be deployed as separate plugins that can be added to the programming environment by an end user.

One of the challenges in realizing *binary extensible* compilers is binary extensibility of the syntax of the host language. Even if an extensible compiler framework supports the implementation of *independently extensible* [Szyperski 1996, Odersky & Zenger 2005] language extensions for later phases of the compiler (e.g. type-checking), then still a compound parser needs to be generated for every particular combination of language extensions. The generation of parse tables for language combinations is expensive, which is a particular problem when the composition configuration (i.e. the set of language extensions) is not fixed, and a parser needs to be generated at run-time (of the compiler).

In this chapter we introduce an algorithm for parse table composition to support separate compilation of grammars. As a result, an extensible compiler can be deployed using a parse table component for the base language. Plugins for language extensions provide a parse table component generated for the language extension only. A parser for a particular combination of languages is generated from the parse table components on the fly. While the worst-time complexity of parse table composition is exponential (like the complexity of parse table generation itself), for realistic language combination scenarios involving grammars for real languages, our parse table composition algorithm is an order of magnitude faster than computation of the parse table for the combined grammars, making online language composition feasible.

Contributions The technical contributions of this work are:

- The idea of parse table composition as symmetric composition of parse tables as opposed to incrementally adding productions, as done in work on incremental parser generation [Horspool 1990, Heering et al. 1990, Cardelli et al. 1994]
- A formal foundation for parse table modification based on automata
- An efficient algorithm for partial reapplication of NFA to DFA conversion, the key idea of parse table composition

- An efficient algorithm for SLR follow sets based on Digraph extended with optimizations for the characteristics of first and follow sets

6.2 MOTIVATION

In this section we motivate the need for parse table composition by considering a number of typical usage scenarios of language combination and analyzing their implementation requirements. These scenarios have in common that they require a combination of languages and that this combination is *not fixed*.

6.2.1 *Program Transformation and Generation*

The first type of application where language combination plays a prominent role is metaprogramming. Metaprograms are programs that manipulate programs as data, to transform existing programs or generate new ones. It is good practice to apply such manipulations on a *structured* representation of a program. For example, the Stratego program transformation language [Bravenboer et al. 2008] uses terms for the representation of abstract syntax trees and term rewriting for program transformation. The following Stratego rewrite rule, which lifts conditional expressions from return statements, uses terms to represent fragments of Java programs:

```
LiftConditionalFromReturn :
  Return(Some(Cond(e1, e2, e3))) ->
  If(e1, Return(Some(e2)), Return(Some(e3)))
```

The downside of the use of terms (and similar structured representations) is that the program fragments become very hard to read and write when they become larger. The mental gap between the abstract representation of program fragments and the object language syntax can be reduced by using the *concrete syntax* of the object language [Visser 2002]. The following Stratego rewrite rule uses concrete syntax to implement the exact same transformation as the rule above:

```
LiftConditionalFromReturn :
  |[ return e1 ? e2 : e3; ]| -> |[ if(e1) return e2; else return e3; ]|
```

While the program fragments are written in concrete syntax (text), the rule is interpreted as (translated to) a rewrite rule operating on terms, i.e. the abstract representation.

Template engines

Concrete syntax for program fragments is used heavily in template engines such as Velocity and StringTemplate, which are popular tools for program generation. Code generation with these tools is text-based, that is, program fragments are considered as character strings, rather than (abstract) syntax trees. As a consequence, templates are not checked syntactically, and there is no structured representation of the generated program to apply further transformations to. Exceptions include MetaAspectJ [Zook et al. 2004], which

extends Java with syntactically checked templates for generation of AspectJ code, and Repleo [Arnoldus et al. 2007], which uses the grammar of the target language to create a grammar of a template language.

Analysis

Mainstream template engines are text-based because the implementation of an engine supporting concrete syntax *and* a structured representation requires a parser for the template language and each target language. That is, the implementation of concrete object syntax requires the extension of the meta-language (e.g. Stratego, Velocity) with the syntax of the object language (e.g. Java, XML) for arbitrary combinations of object languages. Often it is necessary to support multiple languages in a single application of the metalanguage. For example, in the case of a code generator for a DSL, the grammars of the DSL itself and the grammars of the target languages (e.g. Java, SQL and XML) need to be available to produce the parser for this combination of languages. Fixing the meta-programming language to a particular set of languages reduces its scope.

Stratego, ASF+SDF [van den Brand et al. 2001] and Repleo provide a solution to parsing combinations of a metalanguage and object languages generic in the object language, based on the modular syntax definition formalism SDF [Visser 1997b]. Grammars for a particular combination of object language embeddings are compiled together with the grammar of the metalanguage into a parse table. The modularity features of SDF naturally allow the combination of arbitrary context-free languages in this way. Unfortunately, creating a parser for a different (but maybe overlapping) combination of embeddings requires *(1)* creating a new SDF module, *(2)* importing the involved extensions, and *(3)* generating a completely new parse table, i.e. without reusing the parse tables for the separate embedded languages. This entails that the instantiation of the metalanguage for a specific object language cannot be deployed as a separately compiled plugin. Rather, the parse tables of all the necessary combinations of the embeddings need to be deployed.

Deploying support for a specific object language as a single parse table for parsing the specific combination of this object language and the meta-language not only introduces a composition problem, but also hinders the evolution of the metalanguage. For example, it frequently leads to problems with the evolution of Stratego if extensions for a particular embedding are not updated when the Stratego language evolves. Indeed, the language embeddings should only depend on an *interface* of the metalanguage, not a particular implementation or revision.

6.2.2 *Language Extension and Embedding*

Another class of language combinations is that of language extension and embedding. The basic language extension scenario is the addition of new language constructs that provide some syntactic abstraction not previously available in the language, at least not with the same syntactic conciseness. A

```
JPanel panel = panel of border layout {
    north = label "Please enter your message"
    center = scrollpane of textarea {
      rows = 20
      columns = 40
    }
    south = panel of border layout {
      east = panel of grid layout {
        row = {
          button "Ok"
          button "Cancel"
        }
      }
    }
  };
```

Figure 6.1 Java variable declaration with initialization expression in Swing User interface Language (SWUL) to construct UI component

typical example is the addition of the for-each loop to Java 1.4, an extension that was added to the base language in Java 5. Characteristic of this type of extension is that the extension is relatively small in comparison to the base language (a few productions added) and that the lexical syntax of the base language is adopted in the extension.

Domain-specific Languages Embedding

A more disruptive type of language extension is the embedding of domain-specific languages in general-purpose languages, as implemented in approaches such as MetaBorg [Bravenboer & Visser 2004 (Chapter 2)] and Silver [van Wyk et al. 2007]. As an example consider the MetaBorg example program fragment in Figure 6.1. The fragment is a Java variable declaration with as initializer an expression in the Swing User interface Language (SWUL). The language provides a better notation than the usual series of statements needed to compose a user interface with Swing components, by following the hierarchical structure of components (e.g. create a panel consisting of label and a textarea). The embedding is realized by a syntactic extension of Java, along with an *assimilation* transformation mapping the extension to the base language.

Query and Script Embedding

A crossover between program generation and DSL embedding is the embedding of query and script languages in general-purpose languages. Many domain-specific languages are implemented by an interpreter (query/script engine) accessed through an API. Internally, the interpreter may use a structured representation for the interpreted programs, but the API often just provides a string based interface. This may be for convenience, to reduce the mental gap between concrete and abstract syntax, or the engine may run in a different process or even a different machine, requiring serialization of the query. Typical examples are SQL database queries, shell scripts, regular expressions, and XPath queries. The following Java fragment illustrates how a

query is composed from user input and passed to a database engine.

```
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0) ...
```

The generation of queries using string composition has several problems. First of all, the syntactic correctness of the query cannot be determined at compile-time and is only detected at run-time, during testing in the best case, or in production in the worst-case. Secondly, meta-characters of the host language (e.g. double quotes) need to be escaped, which can become particularly convoluted if the query languages has similar escape characters (e.g. regular expressions in Java string literals). But most importantly, the approach is vulnerable to injection attacks; user input that subverts the intended meaning of the query to gain access to restricted data. For example, passing the string `' OR 'x' = 'x` as `userName` to the query above turns the condition into a tautology, resulting in unintended access. Thus, user input should be rewritten to escape meta-characters such as the quote above. While database access APIs (e.g. JDBC) and object-relational mapping frameworks (e.g. Hibernate) provide a mechanism for passing parameters to SQL (HQL) queries that is safe for injection attacks, the use of this mechanism is not enforced, and most other query/script engines do not provide such mechanisms.

In the StringBorg approach [Bravenboer et al. 2007 (Chapter 4)] these problems are solved by embedding the syntax of the query language in the syntax of the host language and automating the escaping of user input strings. For example, the query above is written as follows with StringBorg:

```
SQL q = <| SELECT id FROM users
           WHERE name = ${userName} AND password = ${password} |>;
if (executeQuery(q.toString()).size() == 0) ...
```

For this example, the host language Java is extended with the syntax of SQL with queries as Java expressions of type `SQL`. A query is quoted between `<|` and `|>` and may escape to the host language using the `${...}` anti-quotation. The quotation mechanism ensures that only syntactically correct queries can be constructed, no character escaping is needed, and that string values passed as parameters via anti-quotation are properly escaped. A simple analysis then suffices to determine that only properly constructed query strings are passed to the query engine. The StringBorg method does not just work for SQL in Java, but can be applied to any combination of host language and guest languages.

Analysis

Embedding of domain-specific languages can be applied to any combinations of host and guest languages; it is not unreasonable for a compilation unit to use several DSLs. An important requirement in this type of extension is that the syntax of the embedded language is accurately described. In particular, the lexical syntax of the embedded language is often different from the lexical

syntax of the host language, which requires a technique that supports composition of lexical syntax. The state-of-the-art for realization of embeddings of complete languages (i.e. modular syntax definition and scannerless generalized LR parsing), relies on offline computation of a parse table for each combination of languages. This technology supports extensible compilers, in the sense that a compiler for the extended language can be defined without touching the implementation of the base compiler. However, it does not support binary extensibility, where a compiler can be extended by plugging in a separately developed and deployed language component. Hence, while the implementation of StringBorg is entirely generic in the object language and support for an object language is not specific to a particular metalanguage, it is still necessary to build a parse table for each combination of host and guest languages. Making the StringBorg approach really effective requires the separate deployment of syntax embedding plugins such that query engine vendors can provide a language extension with their product that does not require the compiler (or programming environment) to be rebuilt, and that allows users to combine components from different vendors. The perspective is that of DSL embeddings separately deployed as plugins to an open compiler. In order to use such embeddings, users should not have to recompile/rebuild the compiler for each combination of embeddings. Realization of this perspective requires binary extensibility of syntax and assimilation. The parse table composition mechanism introduced in this chapter provides the former.

6.2.3 *Language Conglomerates*

The language combinations in the previous scenarios are rather asymmetric; one language is the host (or meta) language, the other languages are the guest (or object) languages embedded in the former. *Language conglomerates* are languages that consist of a combination of several more or less 'equal' languages. An example is AspectJ, the aspect-oriented extension of Java, which introduces the concepts of aspect, pointcut, and advice to the language. Unlike the extensions discussed above, the extensions are not about domain-specific notation, and unlike more traditional language extension, the new concepts (especially pointcuts) syntactically diverge from the 'host' language. For example, the following pointcut definition uses a pointcut expression to indicate a set of join points.

```
pointcut cached(int value):
  execution(* Calc+.get*(int)) && args(value);
```

The syntactic structure and interpretation of pointcut expressions is completely different from regular Java expressions and symbols such as `*` and `+` have different roles. Also, depending on the context, different sets of identifiers should be considered as keywords (e.g. `execution` in a pointcut expression, but not in regular Java code).

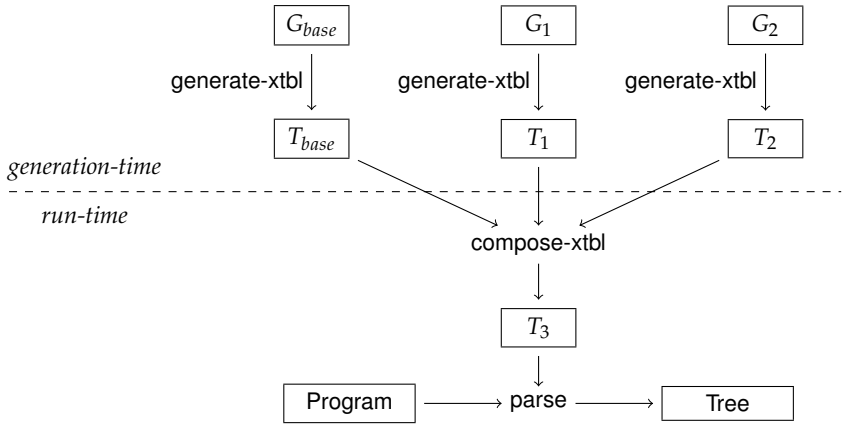


Figure 6.2 Architecture of parse table composition for a base language G_{base} with extensions/embeddings G_1 and G_2

Analysis

In [Bravenboer et al. 2006 (Chapter 5)] we describe the challenges AspectJ poses to traditional parsing techniques and how *grammar mixins* and scannerless generalized LR parsing can solve these challenges. In particular, we describe how the languages that make up AspectJ can be described as separate language components, composed into the full language using a mixin-like mechanism [Bracha & Cook 1990], which makes it possible to make multiple instantiations of a language component, for example, to make different sets of identifiers into keywords. Again, the composition is applied to the grammars for the language components, not to their parse tables. In the case of AspectJ that uses 5 instantiations of the base Java grammar, this results in a considerable cost for the parser generator (excessive memory consumption and one minute in the benchmark of Section 6.8). Separately compiling the Java grammar and then combining the AspectJ parse table using different instantiations of this single Java parse table component turns out to be much cheaper.

6.2.4 *Requirements*

In this section we have discussed several scenarios for combining languages. There is a considerable body of work on extensible compilers, addressing the implementation of language combination (e.g. MetaBorg [Bravenboer & Visser 2004 (Chapter 2)], Polyglot [Nystrom et al. 2003], Silver [van Wyk et al. 2007], JastAdd [Ekman & Hedin 2004]). The focus of this work is on optimizing the work of the compiler developer through source-level extensibility; that is, to create an implementation of an extended compiler by touching as little as possible the code of the base compiler. However, each extension (or combination of extensions) results in a different compiler. In this

work, syntactic extensibility is mostly based on the monolithic scenario discussed so far, where often less sophisticated parsing technologies are used than SDF/SGLR [Bravenboer et al. 2006 (Chapter 5)]. To support scenarios which require more dynamic configurations of language combinations, it is desirable to support *binary* extensibility. That is, extensibility that does not require rebuilding the compiler. This would make it possible to deploy a single version of the compiler, which users can then extend by plugging in separately deployed language components, where potentially each compilation unit uses a different combination of extensions.

To solve the syntactic aspect of this goal we have developed *parse table composition*, a mechanism for combining parse tables, rather than grammars. Figure 6.2 illustrates the workflow. At generation-time, grammars are compiled separately into *parse table components*. At run-time of a compiler, the parse table for the base language T_{base} is combined with a series of parse tables based on a user-selection of the desired extensions, e.g both T_1 and T_2. This results in a single parse table T_3 that is used by the parser to parse a source program. The main contribution is that composing a series of parse table components can be performed *very* efficiently, making the user of the compiler unaware of the parse table composition. Soon, the user will be familiar with the idea of a parser that parses a source file according to a series of parse table components, rather than a single one. After the parser generator has been applied to the individual components, linking the components typically just requires a minimal reconstruction of the parse table. This is a classical partial evaluation argument: as opposed to applying the full parser generation to the grammars G_{base}, G_1, and G_2, the parser generation has already been partially applied.

6.3 GRAMMARS AND PARSING

In this section we define the notions and notations for context-free grammars. Also, we review the basic concepts of the LR parsing algorithm and the generation of LR(0) and GLR parse tables.

6.3.1 *Context-free Grammars*

A context-free grammar G is a tuple $\langle \Sigma, N, P \rangle$, with Σ a set of terminal symbols, N a set of nonterminal symbols, and P a set of productions of the form $A \to \alpha$, where we use the following notation: V for the set of symbols $N \cup \Sigma$; A, B, C for variables ranging over N; X, Y, Z for variables ranging over V; a, b for variables ranging over Σ; v, w, x for variables ranging over Σ^*; and α, β, γ for variables ranging over V^*. The context-free grammar $G_1 = \langle \Sigma, N, P \rangle$ will be used throughout this chapter, where

$$\begin{array}{lcl} \Sigma & = & \{+, \mathtt{N}\} \\ N & = & \{E, T\} \\ P & = & \{E \to E + T,\ E \to T,\ T \to \mathtt{N}\} \end{array} \qquad (G_1)$$

The relation $\Rightarrow$ on V^* defines the derivation of strings by applying productions, thus defining the language of a grammar in a generative way. For a grammar G we say that $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if $A \to \gamma \in P(G)$. A series of zero or more derivation steps from α to β is denoted by $\alpha \Rightarrow^* \beta$. The relation $\Rightarrow_{rm}$ on V^* defines rightmost derivations, i.e. where only the rightmost nonterminal is replaced. We say that $\alpha Aw \Rightarrow_{rm} \alpha\gamma w$ if $A \to \gamma \in P(G)$. If $A \Rightarrow^*_{rm} \alpha$ then we say that α is a right-sentential form for A.

6.3.2 *LR Parsing*

We define an LR(0) parse table to be a tuple $\langle Q, \Sigma, N, \mathsf{start}, \mathsf{action}, \mathsf{goto}, \mathsf{accept}\rangle$ with Q a set of states, Σ a set of terminal symbols, N a set of nonterminal symbols, $\mathsf{start} \in Q$, action a function $Q \times \Sigma \to \mathit{action}$ where *action* is either $\mathsf{shift}\ q$ or $\mathsf{reduce}\ A \to \alpha$, goto a function $Q \times N \to Q$, and finally $\mathsf{accept} \subseteq Q$, where we use the following additional notation: q for variables ranging over Q; and S for variables ranging over $\mathcal{P}(Q)$.

An LR parser [Knuth 1965, Aho & Johnson 1974, Aho et al. 1986] is a transition system with as configuration a stack of states and symbols $q_o X_1 q_1 X_2 q_2 \ldots X_n q_n$ and an input string v of terminals. The next configuration of the parser is determined by reading the next terminal a from the input v and peeking the state q_n at the top of the stack. The entry $\mathsf{action}(q_n, a)$ indicates how to change the configuration. The entries of the action table are shift or reduce actions, which introduce state transitions that are recorded on the stack. A shift action removes the terminal a from the input, which corresponds to a step of one symbol in the right-hand sides of a set of productions that is currently expected. A reduce action of a production $A \to X_1 \ldots X_k$ removes $2k$ elements from the stack resulting in state q_{n-k} being on top of stack. Next, the reduce action pushes A and the new current state on the stack, which is determined by the entry $\mathsf{goto}(q_{n-k}, A)$.

Informally, a *handle* is the location in a string where a production can be applied in the reverse of a rightmost derivation, i.e. reducing a number of symbols to a nonterminal. Formally, a handle of a right-sentential form $\alpha\gamma w$ for S (i.e. $S \Rightarrow^*_{rm} \alpha\gamma w$) is a production $A \to \gamma$ in the position following α such that $S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\gamma w$. In the configuration of an LR parser, handles always appear on top of the stack. The symbols on the stack of an LR parser are always a prefix of a right-sentential form of a grammar. The set of possible prefixes on the stack is called the *viable prefixes*. A viable prefix is a prefix of a right-sentential form, where the prefix does not include any symbols after the rightmost handle of the right-sentential form. We do not discuss the LR parsing algorithm in further detail, since we are only interested in the generation of the action and goto tables.

6.3.3 *Generating LR Parse Tables*

The action and goto table of an LR parser are based on a deterministic finite automaton (DFA) that recognizes the viable prefixes for a grammar. The DFA

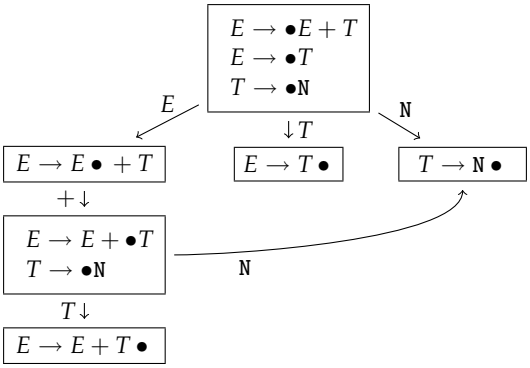


Figure 6.3 LR(0) DFA for grammar G_1

for recognizing the viable prefixes for grammar G_1 is shown in Figure 6.3. Every state of the DFA is associated to a set of items, where an item $[A \to \alpha \bullet \beta]$ is a production with a dot ($\bullet$) at some position of the right-hand side of a production. An item indicates the progress in possibly reaching a configuration where the top of the stack consists of $\alpha\beta$. If the parser is in a state q where $[A \to \alpha \bullet \beta] \in q$ then the α portion of the item is currently on top of the stack, implying that a string derivable from α has been recognized and a string derivable from β is predicted. For example, the item $[E \to E + \bullet T]$ represents that the parser has just recognized a string derivable from $E +$. We say that an item $[A \to \alpha \bullet X\beta]$ *predicts* the symbol X.

To deal with increasingly larger classes of grammars, various types of LR parse tables exist, e.g. LR(0), SLR, LALR, and LR(1). The LR(0), SLR, and LALR parse tables all have the same underlying DFA, but use increasingly more precise conditions on the application of reduce actions. We will return to SLR and LALR in Section 6.6. Figure 6.4 shows the standard algorithm for the generation of LR(0) parse tables. Because LR(0), SLR, and LALR parse tables have the same DFA the functions closure, move, and generate-dfa are the same for all these parse tables. The function generate-tbl is specific to LR(0). The main function generate-tbl first calls generate-dfa to construct a DFA. The function generate-dfa collects states as sets of items in Q and edges between the states in δ. The start state is based on an initial item for the start production. For each set of items, the function generate-dfa determines the outgoing edges by applying the function move to all the predicted symbols of an item set. The function move computes the *kernel* of items for the next state q' based on the items of the current state q by shifting the $\bullet$ over the predicted symbol X. Every kernel is extended to a closure using the function closure, which adds the initial items of all the predicted symbols to the item set.

The LR(0) specific generate-tbl procedure initializes the action and goto tables based on the set of item sets. Edges labelled with a terminal become shift actions. Edges labelled with a nonterminal are entries of the goto table.

```
function generate-tbl(A, G) =
 1  ⟨Q, δ, start⟩ := generate-dfa(A, G)
 2  for each q →_X q' ∈ δ
 3     if X ∈ Σ(G) then action(q, X) := shift q'
 4     if X ∈ N(G) then goto(q, X) := q'
 5  for each q ∈ Q
 6     if [A → α • eof] ∈ q then accept := accept ∪ {q}
 7     for each [A → α • ] ∈ q
 8        for each a ∈ Σ(G)
 9           action(q, a) := reduce A → α
10  return ⟨Q, Σ(G), N(G), start, action, goto, accept⟩

function generate-dfa(A, G) =
 1  start := closure({[S' → •A eof]})
 2  Q := {start}
 3  δ := ∅
 4  repeat until Q and δ do not change
 5     for each q ∈ Q
 6        for each X ∈ {Y | [B → α • Yβ] ∈ q}
 7           q' := closure(move(q, X))
 8           Q := Q ∪ {q'}
 9           δ := δ ∪ {q →_X q'}
10  return ⟨Q, δ, start⟩

function move(q, X) =
 1  return { [A → αX • β] | [A → α • Xβ] ∈ q}

function closure(q) =
 1  repeat until q does not change
 2     for each [A → α • Bβ] ∈ q
 3        q := q ∪ { [B → •γ] | B → γ ∈ P(G)}
 4  return q
```

Figure 6.4 LR(0) parse table generation for grammar G

Finally, if there is a final item, then for all terminal symbols the action is a reduce of this production.

Dealing with Parse Table Conflicts

LR(0) parsers require every state to have a deterministic action for every next terminal in the input stream. There are not many languages that can be parsed using an LR(0) parser, yet we focus on LR(0) parse tables for now. The first reason is that the most important solution for avoiding conflicts is restricting the application of reduce actions, e.g. using the SLR algorithm. These methods are orthogonal to the generation and composition of the LR(0) DFA. We discuss SLR tables in Section 6.6. The second reason is that we target a generalized LR parser [Tomita 1985, Rekers 1992], which already supports ar-

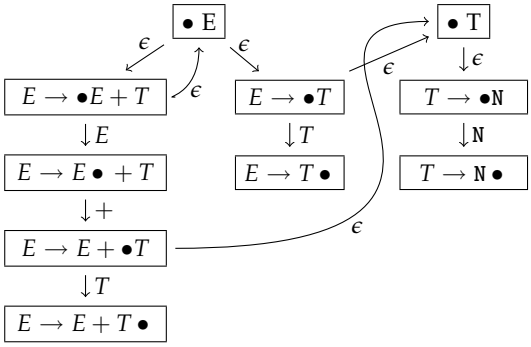


Figure 6.5 LR(0) ϵ-NFA for grammar G_1

bitrary context-free grammars by allowing a *set* of actions for every terminal. The alternative actions are performed pseudo-parallel, and the continuation of the parsing process will determine which action was correct. Our parse table composition method can also be applied to target deterministic parsers, but the composition of deterministic parse tables might result in new conflicts, which have to be reported or resolved.

6.4 LR PARSER GENERATION: A DIFFERENT PERSPECTIVE

The LR(0) parse table generation algorithm is very non-modular due to use of the closure function, which requires all productions of a nonterminal to be known at parse table generation time. If only a subset of the full grammar is known, then there is not much the algorithm of Figure 6.4 can do. To see a glimpse of a possible solution for separate compilation of grammars, we discuss a less common variation of the LR(0) algorithm in this section. This variation first constructs a non-deterministic finite automaton (NFA) with ϵ-transitions (ϵ-NFA) and converts the ϵ-NFA into an LR(0) DFA in a separate step using the standard subset construction algorithm [Grune & Jacobs 1990, Johnstone & Scott 2002]. The ingredients of this algorithm and the correspondence to the one discussed previously naturally lead to the solution to the modularity problem of LR(0) parse table generation.

6.4.1 *Generating LR(0) ϵ-NFA*

An ϵ-NFA [Hopcroft et al. 2006] is an NFA that allows transitions on ϵ, the empty string. Using ϵ-transitions an ϵ-NFA can make a transition without reading an input symbol. An ϵ-NFA A is a tuple $\langle Q, \Sigma, \delta \rangle$ with Q a set of states, Σ a set of symbols, and δ a transition function $Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$, where we use the following notation: q for variables ranging over Q; S for variables ranging over $\mathcal{P}(Q)$, but ranging over Q_D for a different automaton D; X for variables ranging over Σ; and $q_0 \to_X q_1$ for $q_1 \in \delta(q_0, X)$.

function generate-nfa(G) =
1 $Q := \{\bullet A \mid A \in N(G)\}$
2 **for** each $A \to \alpha \in P(G)$
3 $q := \{[A \to \bullet\alpha]\}$
4 $Q := Q \cup \{q\}$
5 $\delta := \delta \cup \{\bullet A \to_\epsilon q\}$
6 **repeat** until Q and δ do not change
7 **for** each $q = \{[A \to \alpha \bullet X\beta]\} \in Q$
8 $q' := \{[A \to \alpha X \bullet \beta]\}$
9 $Q := Q \cup \{q'\}$
10 $\delta := \delta \cup \{q \to_X q'\}$
11 **for** each $q = \{[A \to \alpha \bullet B\gamma]\} \in Q$
12 $\delta := \delta \cup \{q \to_\epsilon \bullet B\}$
13 **return** $\langle Q, \Sigma(G) \cup N(G), \delta\rangle$

Figure 6.6 LR(0) ϵ-NFA generation for grammar G

Figure 6.5 shows the LR(0) ϵ-NFA for the example grammar G_1 (observe the similarity to a syntax diagram). For every nonterminal A of the grammar, there is a *station state* denoted by $\bullet A$. All other states contain just a single LR item. The station states have ϵ-transitions to all the initial items of their productions. If an item predicts a nonterminal A, then there are two transitions: an ϵ-transition to the station state of A and a transition to the item resulting from shifting the dot over A. For an item that predicts a terminal, there is just a single transition to the next item.

Figure 6.6 shows the algorithm for generating the LR(0) ϵ-NFA for a grammar G. Note that states are singleton sets of an item or just a dot before a nonterminal (the station states). The ϵ-NFA of a grammar G accepts the same language as the DFA generated by the algorithm of Figure 6.4, i.e. the language of viable prefixes.

6.4.2 *Eliminating ϵ-Transitions*

The ϵ-NFA can be turned into a deterministic automaton by eliminating the ϵ-transitions using the subset construction algorithm [Hopcroft et al. 2006, Aho et al. 1986], well-known from automata theory and lexical analysis. Figure 6.7 shows the algorithm for converting an ϵ-NFA to a DFA. The function ϵ-closure extends a given set of states S to include all the states reachable through ϵ-transitions. The function move determines the states reachable from a set of states S through transitions on the argument X. The function labels is a utility function that returns the symbols (which does not include ϵ) for which there are transitions from the states of S. The main function ϵ-subset-construction drives the construction of the DFA by considering the current DFA states and for every state $S \subseteq Q_E$ determine the new subsets of states reachable by transitions from states in S.

Applying ϵ-subset-construction to the ϵ-NFA of Figure 6.5 results in the DFA

```
function ε-subset-construction(A, ⟨Q_E, Σ, δ_E⟩) =
1  Q_D := {ε-closure({•A}, δ_E)}
2  δ_D := ∅
3  repeat until Q_D and δ_D do not change
4    for each S ∈ Q_D
5      for each X ∈ labels(S, δ_E)
6        S' := ε-closure(move(S, X, δ_E), δ_E)
7        Q_D := Q_D ∪ {S'}
8        δ_D := δ_D ∪ {S →_X S'}
9  return ⟨Q_D, Σ, δ_D⟩

function ε-closure(S, δ) =
1  repeat until S does not change
2    S := S ∪ {q_1 | q_0 ∈ S, q_0 →_ε q_1 ∈ δ}
3  return S

function labels(S, δ) =
1  return {X | q_0 ∈ S, q_0 →_X q_1 ∈ δ}

function move(S, X, δ) =
1  return {q_1 | q_0 ∈ S, q_0 →_X q_1 ∈ δ}
```

Figure 6.7 Subset construction algorithm from ϵ-NFA E to DFA D

of Figure 6.3. This is far from accidental because the algorithm for LR(0) DFA generation of Figure 6.4 has all the elements of the generation of an ϵ-NFA followed by subset construction. The ϵ-closure function corresponds to the function closure, because ϵ-NFA states whose item predicts a nonterminal have ϵ-transitions to the productions of this nonterminal via the station state of the nonterminal. The first move function constructs the kernel of the next state by moving the dot, whereas the new move function constructs the kernel by following the transitions of the NFA. Incidentally, these transitions exactly correspond to moving the dot, see line 8 of Figure 6.6. Finally, the main driver function generate-dfa is basically equivalent to the function ϵ-subset-construction. Note that most textbooks call the closure function from the move function, but to emphasize the similarity we moved this call to the callee of move.

6.5 COMPOSITION OF LR(0) PARSE TABLES

We discussed the ϵ-NFA variation of the LR(0) parse table generation algorithm to introduce the ingredients of parse table composition. Obviously, LR(0) ϵ-NFA's are much easier to compose than LR(0) DFA's. A naive solution to composing parse tables would be to only construct ϵ-NFA's for every grammar at parse table generation-time and at composition-time merge all the station states of the ϵ-NFA's and run the subset construction algorithm. Unfortunately, this will not be very efficient because the subset construction aspect of LR(0) parse table generation is the expensive part of the algorithm.

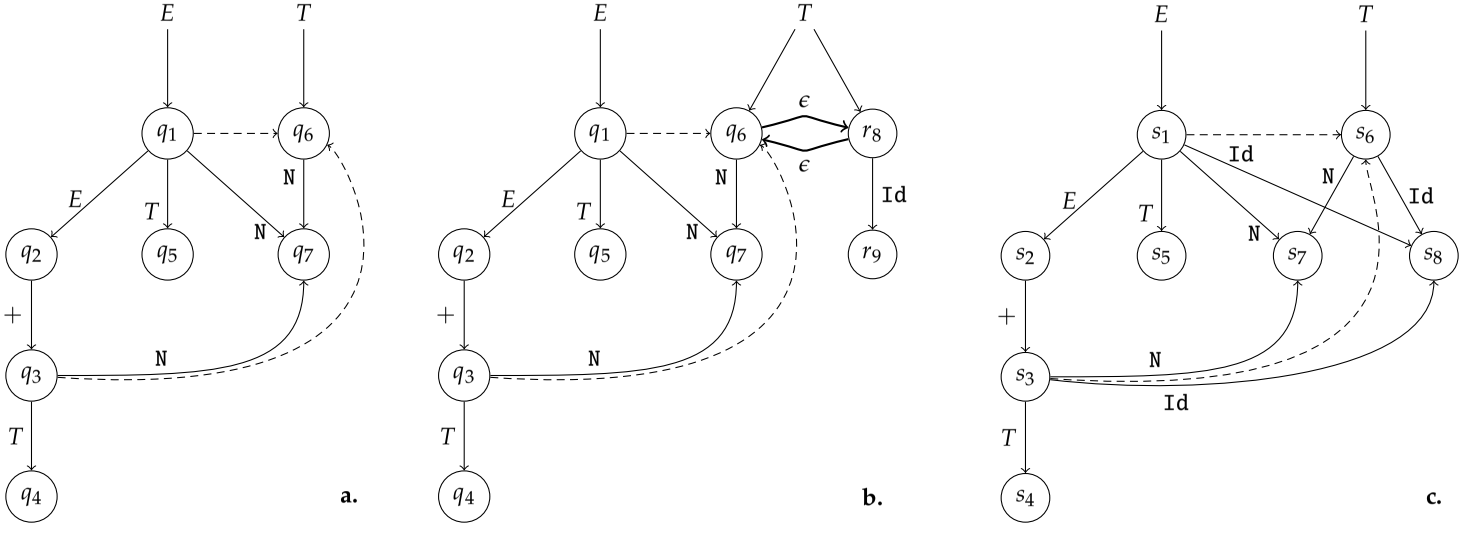


Figure 6.8 **a.** LR(0) ϵ-DFA for grammar G_1 **b.** Combination of ϵ-DFA's for grammars G_1 and G_2 **c.** ϵ-DFA after subset construction

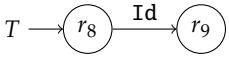
The ϵ-NFA's are in fact not much more than a different representation of a grammar, comparable to a syntax diagram.

The key to an efficient solution is to apply the DFA conversion to the individual parse table components at generation-time, but also preserve the ϵ-transitions as metadata in the resulting automaton, which we refer to as an ϵ-DFA. The ϵ-transitions of the ϵ-DFA can be ignored as long as the automaton is not modified, hence the name ϵ-DFA, though there is no such thing as a deterministic automaton with ϵ-transitions in automata theory. The ϵ-transitions provide the information necessary for reconstructing a correct DFA using subset construction if states (corresponding to new productions) are added to the ϵ-DFA. In this way the DFA is computed per component, but the subset construction can be rerun partially where necessary. The amount of subset reconstruction can be reduced by making use of information on the nonterminals that overlap between parse table components. Also, due to the subset construction applied to each component, many states are already part of the set of ϵ-NFA states that corresponds to a DFA state. These states do not have to be added to a subset again.

Figure 6.8a shows the ϵ-DFA for grammar G_1, generated from the ϵ-NFA of Figure 6.5. The composition algorithm is oblivious to the set of items (a subset) that resulted in a DFA state, therefore the states of the automaton no longer contain LR item sets. The E and T arrows indicate the closures of the station states for these nonterminals. The two dashed ϵ-transitions correspond to ϵ-transitions of the ϵ-NFA. The ϵ-DFA does not contain the ϵ-transition that would result in a self-edge on the station state E. Intuitively, an ϵ-transition from q_0 to q_1 expresses that station state q_1 is supposed to be closed in q_0 (i.e. the subset of q_0 is a superset of the subset of q_1) as long as the automaton is not changed, which makes self-edges useless since every state is closed in itself.

Figure 6.8b combines the ϵ-DFA of 6.8a with a second ϵ-DFA to form an automaton where the ϵ-transitions become relevant, thus being an ϵ-NFA. The parse table component adds variables to the tiny expression language of grammar G_1 based on the following grammar and its automaton.

$$\Sigma = \{\texttt{Id}\} \quad N = \{T\} \quad P = \{T \rightarrow \texttt{Id}\} \tag{G_2}$$



The combined automaton connects station states of the same nonterminal originating from different parse table components by ϵ-transitions, in this case the two station states q_6 and r_8 for T (bold). Intuitively, these transitions express that the two states should always be part of ϵ-closures (subsets) together. In a combination of the original ϵ-NFA's, the station state of T would have ϵ-transitions to all the initial items that now constitute the station states q_6 and r_8.

Figure 6.8c is the result of applying the subset *reconstruction* to Figure 6.8b, resulting in a deterministic automaton (ignoring the irrelevant ϵ-edges). State

q_1 is extended to s_1 by including r_8 because there is a new path from q_1 to r_8 over ϵ-transitions, i.e. r_8 enters the ϵ-closure of q_1. As a result of this extended subset, s_1 now has a transition on `Id` to s_8. Similarly, state q_3 is extended to s_3. Finally, station states q_6 and r_8 are merged into the single state s_6 because of the cycle of ϵ-transitions. Observe that five of the nine states from Figure 6.8b are not affected because their ϵ-closures have not changed.

6.5.1 *Generating LR(0) Parse Tables Components*

The visualizations of automata only show the states, station states and the transitions. However, LR parse tables also have reduce and accept actions and distinguish transitions over terminals (shift actions) from nonterminals (gotos). To completely capture parse tables, we define an LR(0) parse table component T to be a tuple

$$\langle Q, \Sigma, N, \delta, \delta^\epsilon, \mathsf{station}, \mathsf{predict}, \mathsf{reduce}, P, \mathsf{accept} \rangle$$

with Q a set of states, Σ a set of terminal symbols, N a set of nonterminal symbols, δ a transition function $Q \times (\Sigma \cup N) \to Q$, δ^ϵ a transition function $Q \to \mathcal{P}(Q)$ (visualized by dashed edges), station the function $N \to Q$ (visualized using arrows into the automaton labelled with a nonterminal), predict a function $Q \to \mathcal{P}(N)$, reduce a function $Q \to \mathcal{P}(P)$, P a set of productions of the form $A \to \alpha$, and finally accept $\subseteq Q$. Note that the δ function of a component returns a single state for a symbol, hence it corresponds to a deterministic automaton. The ϵ-transitions are captured in a separate function δ^ϵ. For notational convenience we do not explicitly restrict the range of the δ^ϵ function to station states in this definition. Parse table components do not have a specific start nonterminal, instead the station function is used to map all nonterminals to a station state. For the special start nonterminal, the station function will return the start state. We say that a station state q is *closed in* a state q' if the subset that was used to the construct q' includes q.

Figure 6.9 shows the algorithm for generating a parse table component, which is very similar to the subset construction algorithm of Figure 6.7. First, an ϵ-NFA is generated [1] for the grammar G. For every nonterminal A the ϵ-closure (defined in Figure 6.7) of its station state is added to the set of states Q_D [2] of the ϵ-DFA, thus capturing the closure of the initial items of all productions of A. Next, for every state, the nonterminals [8] predicted by the items of this subset are determined. Note that the predicted nonterminals of a state correspond to the ϵ-transitions to station states, or equivalently the transitions on nonterminal symbols from this state. The implementation could also preserve the full set of items, which can be used as another source of the predicted symbols. The set of predicted symbols of a state (predict) is not necessary for a naive implementation of composition, but it will help to improve the performance as we will discuss later. The ϵ-transitions [9] of a state are determined based on the predicted nonterminals, but it could also be based on δ_E. The self-edges are removed by subtracting the state itself. To drive the construction of the complete DFA, the next states [10] are determined by follow-

function generate-xtbl(G) =
1 $\langle Q_E, \Sigma, \delta_E \rangle$:= generate-nfa(G)
2 **for** each $A \in N(G)$
3 $S := \epsilon\text{-closure}(\{\bullet A\}, \delta_E)$
4 $Q_D := Q_D \cup \{S\}$
5 station$(A) := S$
6 **repeat** until Q_D and δ_D do not change
7 **for** each $S \in Q$
8 predict$(S) := \{A \mid q \in S, q \rightarrow_\epsilon \{\bullet A\} \in \delta_E\}$
9 $\delta_D^\epsilon(S) := \{\text{station}(A) \mid A \in \text{predict}(S)\} - \{S\}$
10 **for** each $X \in$ labels(S, δ_E)
11 $S' := \epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E)$
12 $Q_D := Q_D \cup \{S'\}$
13 $\delta_D := \delta_D \cup \{S \rightarrow_X S'\}$
14 reduce$(S) := \{A \rightarrow \alpha \mid [A \rightarrow \alpha \bullet] \in S\}$
15 **if** $[A \rightarrow \alpha \bullet \texttt{eof}] \in S$
16 accept := accept $\cup \{S\}$
17 **return** $\langle Q_D, \Sigma(G), N(G), \delta_D, \delta_D^\epsilon$, station, predict, reduce, $P(G)$, accept$\rangle$

Figure 6.9 LR(0) parse table component generation for grammar G

ing the transitions of the ϵ-NFA using the move function for all labels of this subset (see Figure 6.7). Finally, the reduce actions[14] of a state are all the productions for which there is an item with the dot at the last position. If there is an item that predicts the special `eof` terminal[15] that is part of an augmented production, then the state becomes an accepting state. Note that this definition requires the items of a subset to be known to determine accepting states and reduce actions, but this can easily be avoided by extending the ϵ-NFA with reduce actions and accepting states.

6.5.2 *Composing LR(0) Parse Table Components*

We first present a high-level version of the composition algorithm that does not take much advantage of the subset construction that has been applied to the individual parse table components. The algorithm is not intended to be implemented in this way, similar to the algorithms for parse table generation. In all cases the fixpoint approach is very inefficient and needs to be replaced by a worklist algorithm. Also, efficient data structures need to be chosen to represent subsets and transitions. Figure 6.10 shows the high-level algorithm for parse table composition. Again, the algorithm is a variation of subset construction. First, the combine-xtbl function is invoked to combine the components (resulting in Figure 6.8b of the example). The δ^ϵ functions of the individual components are collected into a transition function $\delta_E^{\epsilon+}$ [2]. To merge the station states this transition function $\delta_E^{\epsilon+}$ is extended to connect the station states of the same nonterminal in different components[4]. Finally, the relations station, predict, and reduce, and the set accept are combined. The domain

function compose-xtbl$(T_0, \ldots, T_k) =$

1 $\langle N_E, \delta_E, \delta_E^{\epsilon+}, \mathsf{stations}_E, \mathsf{predict}_E, \mathsf{reduce}_E, \mathsf{accept}_E \rangle \coloneqq$ combine-xtbl$(T_0, \ldots, T_k)$
2 **for** each $A \in N_E$
3 $\quad S \coloneqq \epsilon$-closure$(\mathsf{stations}_E(A)\}, \delta_E^{\epsilon+})$
4 $\quad Q_D \coloneqq Q_D \cup \{S\}$
5 $\quad \mathsf{station}(A) \coloneqq S$
6 **repeat** until Q_D and δ_D do not change
7 $\quad$ **for** each $S \in Q_D$
8 $\quad\quad \mathsf{predict}(S) \coloneqq \bigcup\{\mathsf{predict}_E(q) \mid q \in S\}$
9 $\quad\quad \delta_D^{\epsilon}(S) \coloneqq \{\mathsf{station}(A) \mid A \in \mathsf{predict}(S)\} - \{S\}$
10 $\quad\quad$ **for** each $X \in \mathsf{labels}(S, \delta_E)$
11 $\quad\quad\quad S' \coloneqq \epsilon$-closure$(\mathsf{move}(S, X, \delta_E), \delta_E^{\epsilon+})$
12 $\quad\quad\quad Q_D \coloneqq Q_D \cup \{S'\}$
13 $\quad\quad\quad \delta_D \coloneqq \delta_D \cup \{S \rightarrow_X S'\}$
14 $\quad\quad \mathsf{reduce}(S) \coloneqq \bigcup\{\mathsf{reduce}_E(q) \mid q \in S\}$
15 $\quad\quad$ **if** $(\mathsf{accept}_E \cap S) \neq \emptyset$
16 $\quad\quad\quad \mathsf{accept} \coloneqq \mathsf{accept} \cup \{S\}$
17 **return** $\langle Q_D, \bigcup_{i=0}^{k} \Sigma_i, N_E, \delta_D, \delta_D^{\epsilon}, \mathsf{station}, \mathsf{predict}, \mathsf{reduce}, \bigcup_{i=0}^{k} P_i, \mathsf{accept} \rangle$

function combine-xtbl$(T_0, \ldots, T_k) =$

1 $N_E \coloneqq \bigcup_{i=0}^{k} N(T_i)$
2 $\delta_E^{\epsilon+} \coloneqq \bigcup_{i=0}^{k} \delta^{\epsilon}(T_i)$
3 $\delta_E \coloneqq \bigcup_{i=0}^{k} \delta(T_i)$
4 **for** each $A \in N_E$
5 $\quad$ **for** $0 \leq i \leq k$
6 $\quad\quad$ **for** $0 \leq j \leq k, j \neq i$
7 $\quad\quad\quad$ **if** $A \in N(T_i) \wedge A \in N(T_j)$
8 $\quad\quad\quad\quad \delta_E^{\epsilon+} \coloneqq \delta_E^{\epsilon+} \cup \{\mathsf{station}(T_i, A) \rightarrow \mathsf{station}(T_j, A)\}$
9 $\mathsf{stations}_E \coloneqq \bigcup_{i=0}^{k} \mathsf{station}(T_i)$
10 $\mathsf{predict}_E \coloneqq \bigcup_{i=0}^{k} \mathsf{predict}(T_i)$
11 $\mathsf{reduce}_E \coloneqq \bigcup_{i=0}^{k} \mathsf{reduce}(T_i)$
12 $\mathsf{accept}_E \coloneqq \bigcup_{i=0}^{k} \mathsf{accept}(T_i)$
13 **return** $\langle N_E, \delta_E, \delta_E^{\epsilon+}, \mathsf{stations}_E, \mathsf{predict}_E, \mathsf{reduce}_E, \mathsf{accept}_E \rangle$

Figure 6.10 LR(0) parse table component composition

of the relations predict and reduce are states, which are unique in the combined automaton. Thus, the combined functions $\mathsf{predict}_E$ and reduce_E have the same type signature as the functions of the individual components. However, the domain of station functions for individual components might overlap, hence the new function $\mathsf{stations}_E$ is a function of $N \rightarrow \mathcal{P}(Q)$.

Back to the compose-xtbl function, we now initialize the subset reconstruction by creating the station states[2] of the new parse table. The new station states are determined for every nonterminal by taking the ϵ-closure of the sta-

tion states of all the components ($\mathsf{stations}_E$) over $\delta_E^{\epsilon+}$. The creation of the station states initializes the fixpoint operation on Q_D and δ_D [6]. The fixpoint loop is very similar to the fixpoint loop of parse table component generation (Figure 6.9). If the table is going to be subject to further composition, then the $\mathsf{predict}$ [8] and δ_D^{ϵ} [9] functions can be computed similar to the generation of components. For final parse tables this is not necessary. Next, the transitions to other states are determined using the move function [11] and the transition function δ_E. Similar to plain LR(0) parse table generation the result of the move function is called a *kernel* (but it is a set of states, not a set of items). The kernel is turned into an ϵ-$\mathsf{closure}$ using the extended set of ϵ-transitions, i.e. $\delta_E^{\epsilon+}$. Finally, the reduce actions are simply collected [14] and if any of the involved states is an accept state, then the composed state will be an accept state [15].

This algorithm performs complete subset reconstruction, since it does not take into account that many station states are already closed in subsets. Also, it does not use the set of predicted nonterminals in any way. The correctness of the algorithm is easy to see by comparison to the ϵ-NFA approach to LR(0) parser generation. Subset construction can be applied partially to an automaton, so extending a deterministic automaton with new states and transitions and applying subset construction subsequently is not different from applying subset construction to an extension of the original ϵ-NFA.

6.5.3 *Optimization*

In worst case, subset construction applied to a NFA can result in an exponential number of states in the resulting DFA. There is nothing that can be done about the number of states that have to be created in subset reconstruction, except for creating these states as efficiently as possible. As stated by research on subset construction [Leslie 1995, van Noord 2000], it is important to choose the appropriate algorithms and data structures. For example, the fixpoint iteration should be replaced, checking for the existence of a subset of states in Q must be efficient (we use uniquely represented treaps for subsets), and the kernel for a transition on a symbol X from a subset must be determined efficiently. In our implementation we have applied some of the basic optimizations, but have focused on optimizations specific to parse table composition. The performance of parse table composition mostly depends on *(1)* the number of ϵ-closure invocations and *(2)* the cardinality of the resulting ϵ-closures.

Avoiding Closure Calls

In the plain subset construction algorithm closure calls are inevitable for every subset. However, subset construction has already been applied to the parse table components. If we know in advance that for a given kernel a closure call will not add any station states to the closure that are not already closed in the states of the kernel, then we can omit the ϵ-closure call. For a kernel $\mathsf{move}(S, X, \delta_E)$ it is not necessary to compute the ϵ-closure if none of the states in the kernel predict a nonterminal that occurs in more than one parse table

component, called an *overlapping* nonterminal. If predicted nonterminals are not overlapping, then the ϵ-transitions from the states in this kernel only refer to station states that have no new ϵ-transitions added by combine-xtbl [4]. Hence, the ϵ-closure of the kernel would only add station states that are already closed in this kernel. Note that new station states cannot be found indirectly through ϵ-transitions either, because $\forall q_0 \rightarrow q_1 \in \delta^{\epsilon}(T_i) : \mathsf{predict}(q_0) \supseteq \mathsf{predict}(q_1)$. Thus, the kernel would have predicted the nonterminal of this station state as well.

To reduce the number of overlapping nonterminals it is useful to avoid unintentional overlap of nonterminals by supporting external and internal symbols. This is discussed in Section 6.7.2.

Reduce State Rewriting

If a closure ϵ-$\mathsf{closure}(\mathsf{move}(S, X, \delta_E), \delta_E^{\epsilon+})$ is a single state q, then q can be added directly to the states of the composed parse table without any updating of its actions and transitions. This is clear for the reduce and accept actions of a state, but the transitions from q also involve other states, which might not be single-state closures. Therefore, we need to choose the names of new states strategically. The state resulting from the closure of any single-state kernel (i.e. ϵ-$\mathsf{closure}(\{q\})$ is always given the same name as q. In this way, there is no need for updating transitions from single-state closures. This optimization makes it very useful to restrict the number of states in a closure aggressively. If the closure is restricted to a single state, then the compose algorithm only needs to traverse the transitions to continue composing states.

Reduce Closure Cardinality

Even if a kernel predicts one or more overlapping symbols (thus requiring an ϵ-closure call) it is not necessarily the case that any station states need to be added to the kernel. For example, if two parse table components T_0 and T_1 having an overlapping symbol E are composed and two states $q_0 \in Q(T_0)$ and $q_1 \in Q(T_1)$ predicting E are both in a kernel, then the two station states for E are already closed in this kernel. To get an efficient ϵ-closure implementation the closures could be pre-computed *per state* using a transitive closure algorithm, but this example illustrates that the *subset* in which a state occurs will make many station states unnecessary. Note that for a state q in table T not all station states of T are irrelevant. Station states of T might become reachable through ϵ-transitions by a path through a different component.

A first approximation of the station states that need to be added to a kernel S to form a closure is the set of states that enter the ϵ-closure because of the ϵ-transitions added by combine-xtbl [4]

$$S^{approx} = \epsilon\text{–}\mathsf{closure}(S, \delta_E^{\epsilon+}) - \epsilon\text{–}\mathsf{closure}(S, \bigcup \delta_i^{\epsilon})$$

However, another complication compared to an ordinary transitive closure is that the station states have been transitively closed already inside their components. Therefore, we are not interested in *all* states that enter the ϵ-closure, since many station states in S^{approx} are already closed in other station

states of S^{approx}. Thus, the minimal set of states that need to be added to a kernel to form a closure is the set of station states that *(1)* are not closed in the states of the kernel (S^{approx}) and are not already closed by another state in S^{approx}:

$$S^{min} = \{q_0 \mid q_0 \in S^{approx}, \nexists q_1 \in S^{approx} : q_1 \rightarrow q_0 \in \bigcup \delta_i^{\epsilon}\}$$

The essence of the problem is expressed by the *predict graph*, which is a subgraph of the combined graph shown in Figure 6.8b restricted to the ϵ-transitions and station states. Every δ_i^{ϵ} transition function induces an acyclic, transitively closed graph, but these two properties are destroyed in the graph induced by $\delta_E^{\epsilon+}$. We distinguish the existing ϵ-transitions and the new transitions introduced by combine-xtbl [4] in intra-component and inter-component edges, respectively.

Figure 6.11 shows the optimized ϵ-closure algorithm, which is based on a traversal of the predict graph. For a given kernel S, the procedure mark first marks the states that are already closed in the kernel. If a state q_1 in the subset is a station state, then the station state itself [3] and all the other reachable station states in the same component are marked [4]. Note that the graph induced by ϵ-transitions of a component is transitively closed, thus there are direct edges to all the reachable station states (intra-edges). For a state q_1 of the subset S that is not a station state, we mark all the station states that are closed in this non-station state q_1 [7]. Note that q_1 itself is not present in the predict graph. The station states are marked with a source state, rather than a color, to indicate which state is responsible for collecting states in this part of the graph. If the later traversal of the predict graph encounters a state with a source different from the one that initiated the current traversal, then the traversal is stopped. The source marker is just used to make the intuition of the algorithm more clear, i.e. a coloring scheme could be used as well.

Next, the function ϵ-closure initiates traversals of the predict graph by invoking the visit function for all the involved station states. The visit function traverses the predict graph, collecting [10] only the states reached *directly* through inter-component edges, thus avoiding station states that are already closed in an earlier discovered station state. For every state, the intra-component edges are visited first, and mark states BLACK as already visited. The visit function stops traversing the graph if it encounters a node with a different source (but continues if the source is NULL), thus avoiding states that are already closed in other station states.

6.6 EXTENSION TO SLR

For many languages, the LR(0) parse table generation algorithm results in states where the parser can perform a shift as well as a reduce action. If the parser generator targets a deterministic parser, then the parser generation fails at this point, or applies heuristics to resolve the conflict and issues a warning. To support a bigger class of languages, the SLR (Simple LR) algorithm [DeRemer 1971] extends LR(0) by guarding the application of a reduce action for a production $A \rightarrow \alpha$ by examining the next terminal in the input stream. An

```
procedure visit(v, src)
 1  color[v] := BLACK
 2  result[v] := ∅
 3  for each w ∈ intra–edges[v]
 4     if (source[w] = src ∨ source[w] = NULL) ∧ color[w] = WHITE
 5        visit(w, src)
 6        result[v] := result[v] ∪ result[w]
 7  for each w ∈ inter–edges[v]
 8     if source[w] = NULL ∧ color[w] = WHITE
 9        visit(w, src)
10        result[v] := {w} ∪ result[v] ∪ result[w]

procedure mark(S, δ^ε)
 1  for each q1 ∈ S
 2     if q1 is station state
 3        source[q1] := q1
 4        for each q2 ∈ intra-edges[q1]
 5           source[q2] := q1
 6     else
 7        for each q2 ∈ δ^ε(q1)
 8           source[q2] := q2
 9           for each q3 ∈ intra-edges[q2]
10              source[q3] := q2

function ε-closure(S, δ^ε) =
 1  color[*] := WHITE
 2  result[*] := NULL
 3  source[*] := NULL
 4  result := S
 5  mark(S, δ^ε)
 6  for each q ∈ S
 7     if q is station state
 8        maybe-visit(q)
 9     else
10        for each qA ∈ δ^ε(q)
11           maybe-visit(qA)
12  return result

13  local procedure maybe-visit(q) =
14     if source[q] = q ∧ color[q] = WHITE
15        visit(q, q)
16        result := result ∪ result[q]
```

Figure 6.11 Optimized ϵ-closure implementation

$$\mathsf{nullable}(X_1 \dots X_n) = \{X_1, \dots, X_n\} \subseteq \mathsf{nullable} \tag{R_3}$$

$$\frac{A \to \alpha \in P(G),\ \mathsf{nullable}(\alpha)}{A \in \mathsf{nullable}} \tag{R_4}$$

$$\mathsf{first}(t) = \{t\} \tag{R_5}$$

$$\frac{A \to \alpha X \beta \in P(G),\ \mathsf{nullable}(\alpha)}{\mathsf{first}(A) \supseteq \mathsf{first}(X)} \tag{R_6}$$

$$\frac{A \to \alpha B \beta X \gamma \in P(G),\ \mathsf{nullable}(\beta)}{\mathsf{follow}(B) \supseteq \mathsf{first}(X)} \tag{R_7}$$

$$\frac{A \to \alpha B \beta \in P(G),\ \mathsf{nullable}(\beta)}{\mathsf{follow}(A) \supseteq \mathsf{follow}(B)} \tag{R_8}$$

Figure 6.12 Rules for computing nullable nonterminals, the first set of a symbol, and the follow set of a nonterminal.

SLR parse table generator determines the set of terminals that can follow the nonterminal A and a reduce action is only applied if the next terminal in the input stream is in this set. If this is not the case, then there is no derivation possible with $A \to \alpha$ applied at this point of the input, so performing the reduce action is useless.

If using a deterministic parser, then the main reason for restricting the application of reduce actions (e.g. using SLR or LALR) is to support a larger class of grammars. For a GLR parser this is not necessary: a GLR parser can perform both actions of a shift/reduce conflict and the continuation of the parsing process will determine which action was correct. Thus, the GLR algorithm applied to (possibly non-deterministic) LR(0) parse tables already supports the full class of context-free grammars. However, for GLR it is still useful to reduce the number of conflicts to improve performance of the parser by avoiding the unnecessary application of reduce actions [Johnstone et al. 2006].

The SLR algorithm determines the follow set of nonterminals by analyzing the productions of the grammar. Unfortunately, the follow set of a nonterminal can (and usually will) change if new productions are added to a grammar. Thus, the calculation of the follow set of a nonterminal requires a global analysis of the grammar. Even if a nonterminal occurs in only a single parse table component, its follow set cannot be calculated before composition-time. For this reason, we do not guard reduce actions in parse table components by their actual follow set, but by a symbolic reference to the follow set of a nonterminal. The actual follow sets are calculated at composition-time.

The rules of Figure 6.12 define the set $\mathsf{follow}(A)$ of terminals that can directly follow A. The follow set of a nonterminal is determined using two other

properties of symbols, namely the set of terminals $\mathsf{first}(X)$ that can begin a string derived from X, and the set $\mathsf{nullable}$ of nonterminals that can derive the empty string. The first, follow, and nullable sets all require a global analysis of the grammar.

6.6.1 *Nullable Nonterminals*

The set $\mathsf{nullable}$ plays a significant role in the calculation of the first and the follow sets because it determines the dependencies between the first and follow sets of symbols. For example, Equation R_8 adds a dependency of the $\mathsf{follow}(A)$ on $\mathsf{follow}(B)$ on the condition that all symbols of β are nullable. It would be useful if the dependencies between the first and follow sets would be known at generation-time, but this would impose the unacceptable restriction that the nullable property of a nonterminal cannot be different across parse table components. Hence, we cannot avoid recalculating the nullable nonterminals at composition-time and change the dependencies between the first and follow sets accordingly.

The set $\mathsf{nullable}$ is usually calculated using a fixpoint algorithm, where productions are examined until the set no longer changes in an iteration. Such a fixpoint algorithm performs unnecessary computations, but whereas we can avoid a fixpoint algorithm for first and follow sets, the nullable relation is not a graph but a hypergraph. The time spent on calculating the nullable nonterminals at composition-time is negligible (see Figure 6.14), so we decided to avoid the complexity of an efficient algorithm on hypergraphs and stick to the plain fixpoint algorithm.

The set of productions that have to be considered at composition-time can be reduced substantially at generation-time. If a nonterminal is nullable in some component, then it will always be nullable in a composition. Therefore, the parser generator can split the nonterminals into two sets, known and not known to be nullable. To examine if nonterminals not known to be nullable have become nullable in a composition, we store a nullable relation in the parse table component. For every production $A \to \alpha$ where A is not known to be nullable and α does not contain a terminal, we store the relation $\beta \to A$, where $\beta = \{B \mid B \in \alpha, B \notin \mathsf{nullable}\}$. As an additional optimization, the relation could be stored in quasi-topological order to improve the performance of the composition-time fixpoint algorithm.

6.6.2 *First and Follow Sets*

The specification of first and follow sets of Figure 6.12 translates directly into a fixpoint algorithm, as is suggested by most compiler textbooks. However, benchmarks have shown that these algorithms are too slow for use at composition-time. To find a more efficient algorithm, we observe that the calculation of first and follow sets can be expressed as a transitive closure problem. The specification of Figure 6.12 introduces a superset relation over the first and follow sets of symbols. For the first set of symbols, Equation R_6

defines that $\mathsf{first}(A)$ has a superset relation with $\mathsf{first}(X)$. This relation induces a graph, which we call the *first graph*. Similarly, Equation R_8 introduces a superset relation over the follow sets of nonterminals, thus defining a *follow graph*. The first and follow graphs are typically sparse and can contain cycles.

Since the relations introduced by the specification of Figure 6.12 are all local to a production, the parser generator can determine the edges of the first and follow graphs for every parse table component. However, the nullable conditions of Equations R_6, R_7, and R_8 are not all known to be valid or not until composition-time. Therefore, the parse table components use two kinds of edges for the first and follow graphs: edges that are known at generation-time and *possible* edges. The possible edges are only used in the composed first and follow graphs if a set of symbols attached to the possible edge is nullable in the composition. For example, Equation R_8 will introduce an edge $A \rightarrow B$ if all symbols of β are nullable in the composition. If β contains a terminal, then β can never be nullable, so the generator does not produce possible edges in this case. Also, nonterminals that are already known to be nullable can be removed from the condition.

6.6.3 *Algorithm*

The main complication in the calculation of first and follow sets is the presence of cycles in the first and follow graphs. If these were acyclic graphs, then we could just traverse the graph, calculating the first or follow sets for every node by unifying the sets of their successors. Unfortunately, applying the same method during the depth-first traversal of a cyclic graphs will result in incorrect results for the strongly connected components. All the nodes of a strongly connected component should have the same first and follow sets, which will not be the case in a straightforward depth-first traversal. Alternatively, we could apply a transitive closure graph algorithm to the first and follow graphs and calculate the first and follow sets based on the resulting transitively closed edges (successor sets), but it is more efficient to combine the traversal of the graph with set computations to calculate the first and follow sets directly. This is the essence of the Digraph algorithm [DeRemer & Pennello 1979], which was originally used for the calculation of LALR(1) lookahead sets. Digraph modifies a transitive closure algorithm based on Tarjan's strongly connected components algorithm [Tarjan 1972, Eve & Kurki-Suonio 1966] to compute lookahead sets instead of successor sets. The use of an efficient transitive closure algorithm results in a great reduction of set computations compared to a naive fixpoint algorithm for first and follow sets.

The Digraph algorithm deals with cycles in dependency graphs by identifying the strongly connected components introduced by cycles, and calculating the lookahead sets over the collapsed *acyclic* graph induced by the strongly connected components and their edges. However, it does not actually construct the collapsed graph, rather it directly applies the set computations during the discovery of components. In our work, we have adapted the Digraph algorithm to be more efficient for typical first and follow graphs. Figure 6.13

```
procedure closure(G) =
 1  for v ∈ V(G)
 2     color[v] := WHITE
 3  time := 1
 4  stack := empty
 5  for v ∈ V(G)
 6     if color[v] = WHITE
 7        visit(v)

procedure visit(v) =
 1  var originalStack := stack
 2  color[v] := GRAY
 3  d[v] := time
 4  root[v] := d[v]
 5  time := time + 1
 6  result[v] := init[v]
 7  for w ∈ Adj(v)
 8     if color[w] = WHITE
 9        visit(w)
10        result[v] := result[v] ∪ result[w]
11     else
12        unless backward(v, w) ∨ forward(v, w)
13           result[v] := result[v] ∪ result[w]
14     if color[w] ≠ BLUE
15        if root[w] < root[v]
16           root[v] := root[w]
17
18  if root[v] = d[v]
19     color[v] := BLUE
20     while stack ≠ originalStack
21        w := pop(stack)
22        result[w] := result[v]
23        color[w] := BLUE
24  else
25     color[v] := BLACK
26     push(stack, v)

function backward(v, w) =
 1  return color[w] = GRAY

function forward(v, w) =
 1  return color[w] ∈ {BLUE, BLACK} ∧ d[v] < d[w]
```

Figure 6.13 Optimized transitive closure algorithm for computing first and follow sets, based on Tarjan's strongly connected component algorithm and the Digraph LALR lookahead set algorithm

shows the adapted transitive closure algorithm we use for calculating first and follow sets.

The basis of the Tarjan's strongly connected component algorithm (and hence the Digraph algorithm) is a depth-first traversal. During the traversal, the algorithm determines if the current node v is the *root* of a strongly connected component, i.e. if it is the first node of a component discovered during the traversal of the graph. This root node is important because the computed set (fist, follow, lookahead, successor, etc.) is correct for this root node, while the result for the other nodes of the component might be incorrect. To update these possibly incorrect results, the traversal collects the nodes that constitute a strongly component on a stack. Once the root of a component has been identified, the nodes of the component are popped from the stack and their results are updated to the value of the root node.

We observed that first and follow graphs typically contain only a few cycles, i.e. there are many trivial components and just a few non-trivial components. Therefore, it is very useful to optimize the algorithm for trivial components. In the original strongly connected components and Digraph algorithms the trivial components still use stack operations for collecting the nodes that constitute a component. However, because we always know that the node that is currently being visited is a member of the component, we can reduce the stack operations by not pushing the current node on the stack until we know that it is a member of a *non-trivial* component. In this way, the stack is only used if there actually is a non-trivial component.

Line by Line

To make the algorithm more clear, we use a different presentation than usual by applying an explicit coloring scheme. All nodes are initially white and become gray if they are being visited. The color blue is used to indicate a node that is part of a component that has completely been visited. The color black indicates a node has been visited, but the component of the node has not yet been visited completely.

The main function closure initializes the color of all the nodes to WHITE [2] and sets the stack for collecting nodes of a component to empty [4]. Next, closure visits every node of the graph [7], if the node has not been visited already. The function visit first records the original stack [1] to remember which nodes to pop from the stack if the component will be finished after visiting this node. The color of v is initialized to GRAY [2], and the discovery time of the node v is set to the current time [3]. For now, the visit function assumes that the root of the component of the node v is v itself. To identify the root, we use the discovery time rather than the node itself to avoid an unnecessary indirection if the discovery time of two candidate roots needs to be compared. To set the initial values of the special character class nodes (for computing first sets) and first set nodes (for computing follow sets), the result value of the current node is initialized to a specified value init$[v]$. For most nodes this will be the empty set. Next, we visit the adjacent nodes [9] if they have not been visited already (i.e. their color is still WHITE). After visiting the node w, the result is added to

current node v [10]. Note that the edge from v to w corresponds to a *tree edge* in depth-first traversal terminology.

If node w was already visited [11], then this is either a back, forward, or cross edge. We apply an optimization that there is no need for set computations for forward and backward edges [12]. An edge is a backward edge if the color of the node w is GRAY. An edge is a forward edge if the color of w is BLUE or BLACK and the visit time of v is smaller than the visit time of w. Forward edges can be ignored because there always exists a different path to w through which the result of w was already included in the result of v. Backward edges can be ignored because including the backward edge will not add anything to the result of the root of the component of v. The result of the other nodes of the component are irrelevant because they will be updated later with the result of the root.

Next, we reconsider if v is indeed a root of the current component. If w is not in a finished component [14], and the candidate root of w has been discovered before the current candidate root of v [15], then the root of w is a better candidate root. Therefore, we update the root of v to the root of w [16].

After visiting the adjacent nodes, we check if the root of v is still v itself [18]. If this is the case, then we have discovered the root of a component and we can finish the entire component. Usually, this will just be a trivial component, i.e. consisting solely of v. In this case, the original stack will immediately be equal to the current stack. However, if this is a non-trivial component. then the function visit unwinds the stack to its original value [20] and updates the results of all the nodes that are popped from the stack to the result of v [22]. Also, all nodes are marked BLUE to indicate that this component has been visited completely [23].

On the other hand, if the root of v is not v itself, then there is a different node that is currently being visited (i.e. its color is GRAY) that will turn out to be the root. Therefore, we mark v BLACK [25] and push v on the stack [26] so that its result can be updated later. Thus, only nodes that are part of a non-trivial component are pushed on the stack.

RESOURCES Both the stack operation optimization for trivial components and the optimization for avoiding set operations for backward and forward edges are based on Nuutila's series of optimizations for strongly connected component transitive closure algorithms [Nuutila 1995]. Several more optimizations are described in Nuutila's work, but these mostly optimize the set computations for non-trivial components, which is not that useful for first and follow graphs. Proofs of the correctness of Tarjan's strongly connected component algorithm, the Digraph algorithm, and its optimizations can be found in the cited literature.

6.7 EXTENSIONS FOR LEXICAL ANALYSIS

Before syntax analysis, a lexical analyzer splits the input stream of characters into a sequence of tokens that correspond to the terminals of a context-free grammar. The reason for the division of work is the use of different techniques

for the implementation of lexical analysis and syntax analysis. The lexical syntax of a language is usually specified using a set of regular expressions, which can be recognized by a deterministic finite automaton, whereas syntax analysis requires a pushdown automaton.

Until now we have ignored the composition of the lexical syntax definition, but any solution for extensible syntax needs to consider the lexical analysis phase as well. Unfortunately, if different languages are used together in the same source file, then the finite automata-based implementation techniques for lexical syntax get more problematic because finite automata are oblivious to context. The lexical syntax of the involved languages is usually different, i.e. they have a different set of tokens. For example, the languages can have different reserved keywords, literals, operators, and use different layout (whitespace and comments). In this way, the lexical syntax depends on the context in the source file and cannot be split into a single sequence of tokens without considering this context.

A possible solution is to ignore this problem and just unify the lexical syntax of the involved languages, resulting in a lexical analyzer that recognizes all tokens in every context. This will result in lexical ambiguities if no additional precedence rules are considered for tokens. On the other hand, if such precedence rules are applied, then the composition will often not have the desired effect, for example by reserving keywords globally while they were intended to be specific to one of the involved languages (see [Bravenboer & Visser 2004 (Chapter 2)] for an extensive discussion). Another solution is to require all the involved languages to use some subset of the same lexical syntax. This is rather restrictive, but for some applications it is still acceptable, for example in languages that want to allow users to extend the syntax with some minor sugar only.

Similar context problems already occur in lexical analyzers for 'single' programming languages, for example to use a different set of tokens in the context of string literals, *here documents*, or regular expressions. These language constructs are in fact different sublanguages of the programming language, leading to a language conglomerate. This is usually solved using a stateful lexical analyzer, which recognizes a different set of tokens in every context and switches the state of analyzer if it encounters certain tokens. As discussed extensively in [Bravenboer et al. 2006 (Chapter 5)], the transitions between the lexical states are based on a careful analysis of the *complete* language. If new tokens are added by composition with other components, then the transitions might no longer be correct.

6.7.1 *Scannerless Parsing*

A scannerless parser [Salomon & Cormack 1989] directly applies syntax analysis to the characters of the input stream, i.e. there is no separate lexical analyzer. Instead of a separate specification of lexical and context-free syntax, a single grammar is used that defines all aspects of the language. The terminals of these grammars are character classes. As argued in [Bravenboer & Visser

2004 (Chapter 2), Bravenboer et al. 2006 (Chapter 5)], scannerless parsers elegantly deal with lexical context issues in syntax embeddings and extensions. Rather than parsing a lexical entity in isolation, as is done with regular expressions, the parsing context acts naturally as lexical state. Therefore, the target parser of our prototype is a scannerless parser. Scannerless parsing has been integrated with GLR parsing in the implementation of SDF [Visser 1997b], adding a few disambiguation techniques to GLR parsing to define typical restrictions and disambiguations on the lexical syntax of a language. Therefore, SDF grammars and the scannerless GLR parser have a few unusual features, which we need to support in our parse table components.

Follow(1) Restrictions

To define a preference for the longest match of what would normally constitute a token in lexical syntax, follow restrictions can be defined for nonterminals. For example, nonterminals for identifiers typically have the restriction that they cannot be followed by a character that is allowed as an identifier. A follow restriction $A \not\succ cc$ specifies that there can be no derivation where the application of a production $A \rightarrow \alpha$ is directly followed by a character from the character class cc. In an SLR parse table, follow restrictions are trivial to implement by subtracting the character class of the restriction from the follow set that guards the application of a reduce action.

In parse table components the follow sets of nonterminals are not yet known, so they cannot be subtracted. Therefore, we store this information separately and subtract the restriction after computing the follow set at composition-time. In this way, follow restrictions can also be applied to the reduce actions of different parse table components, though it might also be useful to apply the restrictions only internally.

Follow(1+) Restrictions

The scannerless parser we target also supports follow restrictions of more than a single character lookahead. Because follow(1+) restrictions involve multiple characters, they cannot be applied to the follow sets of reduce actions. Therefore, follow(1+) restrictions are an attribute of the reduce action and are checked separately by the parser. Unfortunately, the application to the reduce actions is somewhat involved: a reduce action of the production $A \rightarrow \alpha$ with follow set cc_0 that is runtime restricted by $A \succ cc_1 \succ cc_2 \succ \ldots \not\succ cc_n$ needs to be split in two reduce actions: one with follow set $cc_0 - cc_1$ and one with the follow set cc_1 and the runtime restriction $cc_2 \succ \ldots \not\succ cc_n$.

The splitting of reduce actions is the only reason a closure of a single state needs to be modified by the composer. Follow(1+) restrictions are rather uncommon, so it might be useful to avoid this overhead by changing the way reduce action with follow(1+) restrictions are represented in the parse table.

Reserved Keywords

The scannerless parser allows the definition of reserved keywords for specific nonterminals using *reject productions* [Visser 1997b]. A reject production

$A \rightarrow \alpha$ {reject} declares that $A \Rightarrow^* \beta$ is invalid if also $\alpha \Rightarrow^* \beta$. Reject production are just normal productions with a special status; therefore the parse table composition algorithm automatically supports composition of parse table components involving reject productions. Hence, parse table composition also supports component specific reserved keywords.

6.7.2 *Context-Specific Layout*

Grammars for scannerless parsers define the syntax of a language completely down to the character level, including whitespace and comments. To keep the production rules concise the SDF grammar language (which targets the same scannerless parser we target) automatically inserts the nonterminal for optional layout between the symbols of a production. Unfortunately, the layout nonterminal that is used is a global nonterminal, which makes it difficult to have a different syntax for layout in different contexts. However, for many syntax embedding applications this feature is essential. For example, in a concrete object syntax embedding of Java in Stratego we want the whitespace and comments of Java in the quotations of Java code, but the layout of Stratego outside the quotations. The layout of one language can even conflict with the syntax of a different language. For example, Java uses // as end-of-line comments, but in XPath // is part of a path expression. Clearly, in embedding of XPath in Java, the Java end-of-line comment should not be layout in the context of an XPath expression.

For this reason, our parser generator supports making nonterminals internal to a parse table component. The set of productions for this nonterminal is not extended and other parse table components cannot refer to the nonterminal. Thus, the layout of the Java concrete object syntax embedding can be internal, avoiding overlap with the layout of the host language Stratego. Incidentally, parse table components that use this feature can also be composed much more efficiently. The runtime of the composer mostly depends on the number of overlapping symbols and their number of occurrences. Layout is used everywhere in a grammar, so if the layout nonterminals overlap between two parse table components, more LR states will need to be extended and result in closures of more than a single state.

6.8 EVALUATION

In the worst case scenario an LR(0) automaton can change drastically if it is combined with another LR(0) automaton. The composition with an automaton for just a *single* production can introduce an exponential number of new states [Horspool 1990]. Parse table composition cannot cirumvent this worst case. Fortunately, grammars of typical programming languages do not exhibit this behaviour. To evaluate our method, we measured how parse table composition performs in *typical* applications. We have applied an extensive benchmark to the applications that motivated this work. For these applications, parse table components correspond to languages that form a natural

sublanguage. These languages do not change the structure of the language invasively but hook into the base language at some points.

We compare the performance of our prototype implementation[1] to the SDF parser generator (sdf2-bundle-2.4pre212034) that targets the same scannerless GLR parser. SDF grammars are compiled by first normalizing the high-level features to a core language and next applying the parser generator. Our parser generator accepts the same core language as input. The prototype consists of two main tools: one for generating parse table components and one for composing them. As opposed to the SDF parser generator, the generator of our prototype has not been heavily optimized because its performance is not relevant to the performance of runtime parse table composition. We have implemented a generator and composer for *scannerless* generalized LR SLR parse tables. This affects the performance of the composer: grammars have more productions, more symbols, and layout occurs between many symbols. Also, handling of reduce actions is more complicated due to follow restrictions.

Figure 6.14 presents the results for a series of Stratego concrete object syntax embeddings, StringBorg grammars, and AspectJ. For Stratego and StringBorg, the number of overlapping symbols is very limited and there are many single-state closures. Depending on the application, different comparisons of the timings are useful. For runtime parse table composition, we need to compare the performance to generation of the full automaton. The total composition time (col. *16)* is only about 2% to 16% of the SDF parse table generation time (col. *4*). The performance benefit increases even more if we include the required normalization (col. *3*) (SDF does not support separate normalization). For larger grammars (e.g. involving Java) the performance benefit is bigger.

The AspectJ composition is clearly different from the other embeddings. This composition is not intended to be used at runtime, but serves as an example that separate compilation of grammars is a major benefit if multiple instance of the same grammar occur in a composition. The total time to generate a parse table from source using parse table composition (col. $10+11+16$) is only 7% of the time used by the SDF parser generator.

6.9 RELATED WORK

Modular Grammar Formalism

There are a number of parser generators that support splitting a grammar into multiple files, e.g. Rats! [Grimm 2006], JTS [Batory et al. 1998], PPG [Nystrom et al. 2003], and SDF [Visser 1997b]. They vary in the expressiveness of their modularity features, their support for the extension of lexical syntax, and the parsing algorithm that is employed. Many tools ignore the intricate lexical aspects of syntax extensions, whereas some apply scannerless parsing or context-aware scanning [van Wyk & Schwerdfeger 2007]. However, except for a few research prototypes discussed next, these parser generators

[1] available at www.strategoxt.org/ParseTableComposition

			sdf pgen		parse table composition											
	productions	symbols	normalization (ms)	table generation (ms)	overlapping symbols	$\sum$ states	composed states	% single state	$\sum$ normalization (ms)	$\sum$ generate-xtbl (ms)	compose-xtbl (ms)	nullables (ms)	first sets (ms)	follow sets (ms)	follow sets total (ms)	compose total (ms)
Stratego+XML	782	436	430	230	4	2983	2127	85	160	580	27	0	0	10	10	37
Stratego+Java	1766	836	3330	1790	3	6513	6612	93	2110	4250	67	0	0	10	20	80
Stratego+Stratego	1115	536	780	600	4	4822	4085	89	410	1530	37	0	0	7	7	47
Stratego+Stratego+XML	1420	745	1530	830	5	5752	4807	89	440	1600	60	0	3	10	13	77
Stratego+Stratego+Java	2404	1145	6180	2710	4	10405	9295	93	2390	5780	127	0	3	20	23	150
Java+SQL	1391	750	2800	1300	3	5175	3698	92	1350	2440	63	0	0	10	10	73
Java+XPath	1084	554	1560	780	3	4158	2848	90	1150	2010	60	0	0	3	3	63
Java+LDAP	1024	545	1550	760	3	3831	2467	89	1140	1940	43	0	0	3	3	53
Java+XPath+SQL	1580	853	3560	1290	3	5784	4272	93	1390	2530	63	0	0	10	13	83
Java+XPath+LDAP	1213	648	1910	1050	3	4440	3041	91	1170	2030	57	0	3	3	10	63
AspectJ + 5x Java	3388	2426	48759	10261	62	19332	8305	51	1460	1990	477	0	3	30	33	520

Figure 6.14 Benchmark of parse table composition compared to the SDF parser generator. *(1)* number of reachable productions and *(2)* symbols, *(3)* time for normalizing the full grammar and *(4)* generating a parse table using SDF pgen, *(5)* number of overlapping symbols, *(6)* total number of states in the components, *(7)* number of states after composition, *(8)* percentage of single-state closures, *(9)* total time for normalizing the individual components and *(10)* generating parse table components, *(11)* time for reconstructing the LR(0) automaton, *(12, 13, 14, 15)* time for various aspects of follow sets, *(16)* total composition time ($11 + 15$). We measure time using the clock function, which reports time in units of 10ms on our machine. We measure total time separately, which combined with the accuracy of clock explains why some numbers do not sum up exactly. All results are the average of three runs.

all generate a parser by first collecting all the sources, essentially resulting in whole-program compilation.

Extensible Parsing Algorithms

For almost every single parsing algorithm extensible variants have already been proposed. What distinguishes our work from all the existing work is the idea of separately compiled parse table components and a solid foundation on finite automata for combining these parse table components. The close relation of our principles to the LR parser generation algorithm makes our method easy to comprehend and optimize. All other methods focus on adding productions to an existing parse table, motivated by applications such as interactive grammar development. However, for the application in extensible compilers we do not need incremental but *compositional* parser generation. In this way, a language extension can be compiled, checked, and deployed independently of the base language in which it will be used. Next, we discuss a few of the most related approaches.

Horspool's [Horspool 1990] method for incremental generation of LR parsers is most related to our parse table composition method. Horspool presents methods for adding and deleting productions from LR(0), SLR, as well as LALR(1) parse tables. The work is motivated by the need for efficient grammar debugging and interactive grammar development, where it is natural to focus on addition and deletion of productions instead of parse table components. Interactive grammar development requires the (possibly incomplete) grammar to be able to parse inputs all the time, which somewhat complicates the method. For SLR follow sets Horspool uses an incremental transitive closure algorithm based on a matrix representation of the first and follow relations. In our experience, the matrix is very sparse, therefore we use Digraph. This could be done incrementally as well, but due to the very limited amount of time spend on the follow sets, it is hard to make a substantial difference.

IPG [Heering et al. 1990] is a lazy and incremental parser generator targeting a GLR parser using LR(0) parse tables. This work was motivated by interactive meta programming environments. The parse table is generated by need during the parsing process. IPG can deal with modifications of the grammar as a result of the addition or deletion of rules by resetting states so that they will be reconstructed using the lazy parser generator. Rekers [Rekers 1992] also proposed a method for generating a single parse table for a set of languages and restricting the parse table for parsing specific languages. This method is not applicable to our applications, since the syntax extensions are not a fixed set and typically provided by other parties.

Dypgen [Onzon 2007] is a GLR parser generator focusing on scoped modification of the grammar from its semantic actions. On modification of the grammar it generates a new LR(0) automaton.

Earley [Earley 1970] parsers work directly on the productions of a context-free grammar at parse-time. Because of this the Earley algorithm is relatively easy to extend to an extensible parser [Tratt 2005, Kolbly 2002]. Due to the

lack of a generation phase, Earley parsers are less efficient than GLR parsers for programming languages that are close to LR.

Maya [Baker & Hsieh 2002] uses LALR for providing extensible syntax but regenerates the automaton from scratch for every extension.

Cardelli's [Cardelli et al. 1994] extensible syntax uses an extensible LL(1) parser. Camlp4 [de Rauglaudre 2003] is a preprocessor for OCaml for the implementation of syntax extensions using an extensible top down recursive descent parser.

Automata Theory and Applications

The egrep pattern matching tool uses a DFA for efficient matching in combination with lazy state construction to avoid the initial overhead of constructing a DFA. egrep determines the transitions of the DFA only when they are actually needed at runtime. Conceptually, this is related to lazy parse table construction in IPG. It might be an interesting experiment to apply our subset reconstruction in such a lazy way.

Essentially, parse table composition is a DFA maintenance problem. While there has been a lot of work in the maintenance of transitive closures, we have have not been able to find existing work on DFA maintenance.

ACKNOWLEDGMENTS

We thank Emmanuel Onzon (of Dypgen) for his valuable comments on earlier renderings of the algorithms. We thank Arthur van Dam, Rob Vermaas and Shan Shan Huang for their comments on an earlier draft of this chapter. We thank Todd Veldhuizen for a useful discussion on the use of the treap data-structure in combination with ATerms.

Precedence Rule Recovery and Compatibility Checking

7

ABSTRACT

A wide range of parser generators are used to generate parsers for programming languages. The grammar formalisms that come with parser generators provide different approaches for defining operator precedence. Some generators (e.g. YACC) support precedence declarations, others require the grammar to be unambiguous, thus encoding the precedence rules. Even if the grammar formalism provides precedence rules, a particular grammar might not use it. The result is a collection of grammar variants implementing the same language. For the C language, the GNU Compiler uses YACC with precedence rules, the C-Transformers project uses SDF without priorities, while the SDF library does use priorities. For PHP, Zend uses YACC with precedence rules, whereas the PHP-front package uses SDF with priority and associativity declarations.

The variance between grammars raises the question whether the precedence rules of one grammar are compatible with those of another. This is usually not obvious, since some languages have complex precedence rules. Also, for some parser generators the semantics of precedence rules is defined operationally, which makes it hard to reason about their effect on the defined language. We present a method and tool for comparing the precedence rules of different grammars and parser generators. Although it is undecidable whether two grammars define the same language, this tool provides support for comparing and recovering precedence rules, which is especially useful for reliable migration of a grammar from one grammar formalism to another. We evaluate our method by the application to non-trivial mainstream programming languages, such as PHP and C.

7.1 INTRODUCTION

Defining the syntax of a programming language using a context-free grammar is one of the most established practices in the software industry and computer science. For various reasons a wide range of parser generators are used to generate parsers from context-free grammars. For almost every mainstream programming language there exists a series of parser generators, not only featuring different parsing algorithms, but also different grammar formalisms. These grammar formalisms often provide methods for declaring the precedence of operators, since the notions of priority and associativity are pervasive in the definition of the syntax of programming languages.

As early as 1975 Aho and Johnson recognized [Aho et al. 1975] that for many languages the most natural grammar is not accepted by the parser generators that are used in practice, since the grammar does not fall in the class of context-free grammars for which the parser generator can produce an efficient parser. Aho and Johnson proposed to define the syntax of a programming languages as an ambiguous grammar combined with disambiguation rules that tell the parser how to resolve a *parsing action conflict*, a method that was implemented in the now still dominant YACC parser generator [Johnson 1975]. Unfortunately, most of the work on separate precedence declarations has been guided by the underlying parsing technique and not by an analysis of the requirements and fundamentals of precedence declarations. Indeed, parser generators only support precedence rules that can efficiently be implemented in the parser. This is understandable from a practical point of view, yet the result is that little is known about the actual requirements for separate precedence declarations. Indeed, the semantics of separate precedence declarations is apparently so ill-defined that it is still not used in language specifications. Rather, language specifications prefer to encode precedence rules in the productions of the grammar. Sadly, it is difficult to disagree with this approach, since an encoding in productions is still the most precise, formal, and parsing technology independent way of defining precedences!

In this chapter, we argue that precedence rules need to be liberated from the idiosyncrasies of specific parser generators. The reasons for this are closely related to the efforts to work towards an engineering discipline for grammarware [Klint et al. 2005, Lämmel & Verhoef 2001, Sellink & Verhoef 2000, Lämmel 2001a]. Liberating grammars from concrete parser generators is not a new idea [Kort et al. 2002], however precedence rules have never been studied fundamentally outside of the context of specific parsing technologies or parser generators. Indeed, there is currently, for example, no solid methodology to

- recover precedence rules from 'legacy' grammar formalisms. For example, for PHP there is no language specification, only a YACC grammar. Due to the conflict resolution semantics of YACC precedence declarations, the exact precedence rules of PHP are currently very difficult to determine.
- compare the precedence rules of two grammars, whether they are defined in the same grammar formalism or not. For example, for the C language, the GNU Compiler uses YACC with precedence rules, the C-Transformers project [Borghi et al. 2006] uses SDF [Visser 1997b] without priorities, while the SDF library does use priorities. For PHP, Zend uses YACC with precedence rules, whereas PHP-front uses SDF with priority and associativity declarations. However, there is no way to check that the precedence rules of one grammar are compatible with those of another.
- reliably migrate a grammar from one grammar formalism to another including its precedence rules. This does not necessarily have to be done

completely automatic, but at least there can be support for recovering precedence rules and generating precedence declarations in the new formalism.

In this chapter we present a method and its implementation for recovering precedence rules from grammars. Our method is based on a core formalism for defining precedence rules, which is independent of specific parser generators. Based on this formalism and the recovery of precedence rules, we can compare precedence rules of different grammars, defined in different grammar formalism, and using different precedence declaration mechanisms. We have implemented support for recovering precedence rules from YACC [Johnson 1975] and SDF [Heering et al. 1989, Visser 1997b] (parser generators using different parsing algorithms) and present the details of an algorithm to check precedence rules against LR parsers.

Although it is in general undecidable whether two grammars define the same language, this tool provides support for comparing and recovering precedence rules, which is especially useful for reliable migration of a grammar from one grammar formalism to another. Also, the method can be used to analyze the precedence rules of a language, for example to determine if they can be defined using a certain grammar formalism specific precedence declaration mechanism. We evaluate our method by the application to the non-trivial mainstream programming languages C and PHP. For both languages we compare the precedence rules of three grammars defined in SDF or YACC. The evaluation was most successful and revealed several differences and bugs in the precedence rules of the grammars. The YACC and SDF implementations of the method that we present are implemented in Stratego/XT [Visser 2004] and available as open source software as part of the Stratego/XT Grammar Engineering Tools[1].

Contributions The contributions of this chapter are:

- A core formalism for precedence rules.
- A novel method for recovering precedence rules from grammars
- A method for checking the compatibility of precedence rules across grammars
- Implementations of the recovery method for YACC and SDF and an evaluation for non-trivial programming languages C and PHP.

Organization In Section 7.2 we introduce notations for context-free grammars and tree patterns. In Section 7.3 we introduce a running example and explain the precedence mechanisms of YACC and SDF. Section 7.4 is the body of the chapter, where we present our precedence rule recovery method. Section 7.5 discusses compatibility checking. In Section 7.6 we present our evaluation, and we conclude with a discussion of related work.

[1] `http://www.strategoxt.org/GrammarEngineeringTools`

In this section we define the notions and notations for context-free grammars and tree patterns as we will use them in this chapter.

7.2.1 *Context-Free Grammars*

A context-free grammar G is a tuple (Σ, N, P), with Σ a set of terminals, N a set of non-terminals, and P a set of productions of the form $A \to \alpha$, where we use the following notation: V for $N \cup \Sigma$; A, B, C for variables ranging over N; X, Y, Z for variables ranging over V; a, b for variables ranging over Σ; v, w, x for variables ranging over Σ^*; and α, β, γ for variables ranging over V^*. Context-free grammars are usually written in some concrete grammar formalism. Figure 7.1 gives examples of grammars for the same language in different grammar formalisms. The underlying structure is that of context-free grammars just defined. The augmentation of grammars with precedence mechanisms will be discussed in the next section.

7.2.2 *Parse Trees*

The family of valid parse trees T_G over a grammar G is a mapping from V to a set of trees, and is defined inductively as follows:

- if a is a terminal symbol, then $a \in T_G(a)$
- if $A_0 \to X_1 ... X_n$ is a production in G, and $t_i \in T_G(X_i)$ for $1 \leq i \leq n$, then $\langle A_0 \to t_1 ... t_n \rangle \in T_G(A_0)$.

For example, the tree $\langle E \to \langle E \to \langle T \to \langle F \to NUM \rangle \rangle \rangle + \langle T \to \langle F \to NUM \rangle \rangle \rangle$ is a parse tree for the addition of two numbers according to the left-most grammar in Figure 7.1.

7.2.3 *Parse Tree Patterns*

The family TP_G of parse tree patterns (or *tree patterns* for short) over a grammar G, is a mapping from grammar symbols in V to sets of parse trees over G extended with non-terminals as trees, which we define inductively as follows:

- if X is a terminal or non-terminal symbol in V, then $X \in TP_G(X)$
- if $A_0 \to X_1 ... X_n$ is a production in G, and $t_i \in TP_G(X_i)$ for $1 \leq i \leq n$, then $\langle A_0 \to t_1 ... t_n \rangle \in TP_G(A_0)$.

A parse tree pattern p denotes a *set* of parse trees, namely the set obtained by replacing each non-terminal A in p by the elements of $T_G(A)$. Basically, a parse tree pattern corresponds to the derivation tree for a sentential form. For example, the tree pattern $\langle E \to \langle E \to \langle T \to F \rangle \rangle + T \rangle$ denotes the set of trees for summation expressions where the first summand is a 'factor'. We denote a tree pattern with root $A \in N$ and yield α by $\langle A \rightsquigarrow \alpha \rangle$

```
%token NUM
E: E '+' T
 | T
T: T '*' F
 | F
F: NUM
```

```
%token NUM
%left '+'
%left '*'

E: NUM
 | E '+' E
 | E '*' E
```

```
context-free syntax
  E "+" E -> E
  E "*" E -> E
  NUM     -> E
context-free priorities
    E "*" E -> E {left}
  > E "+" E -> E {left}
lexical syntax
  [0-9] -> NUM
```

```
context-free syntax
  E "+" E -> E {left}
  T -> E
  T "*" T -> T {left}
  F -> T
  NUM -> F

lexical syntax
  [0-9] -> NUM
```

Figure 7.1 Grammars for a small arithmetic expressions language. Left to right: YACC using encoded precedence ($YACC_1$), YACC using precedence declarations ($YACC_2$), SDF using precedence declarations (SDF1), SDF using a mixture of encoding and precedence declarations (SDF2).

We use the notation $\langle A \sim B \rightarrow t^* \rangle$ to denote an *injection chain* from a tree pattern with root A to a node with non-terminal B and leaves t^*. Formally, $\langle A \sim B \rightarrow t^* \rangle$ is the subset of $TP_G(A)$ such that

- if $A \rightarrow B$ is a production in G, and $\langle B \rightarrow t^* \rangle \in TP_G(B)$, then $\langle A \rightarrow \langle B \rightarrow t^* \rangle \rangle \in \langle A \sim B \rightarrow t^* \rangle$
- if $A \rightarrow C$ is a production in G, and $\langle C \sim B \rightarrow t^* \rangle \in TP_G(C)$, then $\langle A \rightarrow \langle C \sim B \rightarrow t^* \rangle \rangle \in \langle A \sim B \rightarrow t^* \rangle$

For example, the expression $\langle E \rightarrow \langle E \sim F \rightarrow NUM \rangle + T \rangle$ abbreviates the injection chain in the tree pattern $\langle E \rightarrow \langle E \rightarrow \langle T \rightarrow \langle F \rightarrow NUM \rangle \rangle \rangle + T \rangle$.

Finally, to define the notion of precedence, we will need *one-level tree patterns*, which we define as follows:

- if $A \rightarrow \alpha B \gamma$ and $B \rightarrow \beta$ are productions, then $\langle A \rightarrow \alpha \langle B \rightarrow \beta \rangle \gamma \rangle \in TP^1_G(A)$
- if $A \rightarrow \alpha B \gamma$ is a production and $\langle B \sim C \rightarrow \beta \rangle \in TP_G(B)$ then $\langle A \rightarrow \alpha \langle B \sim C \rightarrow \beta \rangle \gamma \rangle \in TP^1_G(A)$

That is, one-level tree patterns are productions with a single subtree, with possibly an injection chain from the root production to the child production. Observe that $TP^1_G(A) \subseteq TP_G(A)$. The tree pattern $\langle E \rightarrow E + \langle T \rightarrow T * F \rangle \rangle$ is one-level, and so is $\langle E \rightarrow \langle E \rightarrow \langle T \rightarrow T * F \rangle \rangle + T \rangle$. However, $\langle E \rightarrow \langle E \rightarrow \langle T \rightarrow T * F \rangle \rangle + \langle T \rightarrow T * F \rangle \rangle$ is not a one-level tree pattern, since it has two non-chain subtrees.

7.3 PRECEDENCE MECHANISMS

In this chapter we focus on two grammar formalisms, their parser generators, and their precedence mechanisms. The first is YACC (Yet Another Compiler-Compiler) and the second is SDF (Syntax Definition Formalism). The parser targeted by the SDF parser generator has a different name: SGLR (Scannerless Generalized LR). Considering the combination of SDF and YACC is interesting for three reasons. First, the two grammar formalisms provide very different precedence declaration mechanisms. Second, the grammar formalisms

are implemented using different parsing techniques. Third, the conversion from YACC to SDF is a very common use case. We introduce the basics of the YACC and SDF precedence declaration mechanisms with a few grammars for a small arithmetic language, see Figure 7.1.

7.3.1 *YACC*

YACC [Johnson 1975] is *the* classic parser generator. It accepts grammars of the LALR(1) class of context-free grammars with optionally additional disambiguation rules. For our YACC-based tools we use Bison, the GNU version of YACC, however, we will refer to our use of Bison as YACC (on most systems `yacc` is actually an alias of `bison`). The first and the second grammar of Figure 7.1 are YACC grammars. The first grammar encodes the precedence rules of the arithmetic language in the productions of the grammar. The operators `+` and `*` are left-associative, since the grammar does not allow an occurrence of `+` at the right-hand side of a `+`. The operator `*` takes precedence over the operator `+`, since it is not possible at all to have a `+` at left or right-hand side of a `*`. The second grammar uses separate YACC precedence declarations [Aho et al. 1975]. Without disambiguation rules (and implicit conflict resultion), this grammar is ambiguous, e.g `1 + 2 * 3` has two different parse trees: $\langle E \rightarrow \langle E \rightarrow \langle E \rightarrow 1\rangle + \langle E \rightarrow 2\rangle\rangle * \langle E \rightarrow 3\rangle\rangle$ and $\langle E \rightarrow \langle E \rightarrow 1\rangle + \langle E \rightarrow \langle E \rightarrow 2\rangle * \langle E \rightarrow 3\rangle\rangle\rangle$ are both elements of $T_G(E)$.

As disambiguation rules, YACC allows declarations of the precedence of operators, which can be `%left`, `%right`, or `%nonassoc`. After the associativity comes a list of tokens. All tokens on the same line have the same precedence. The relative precedence of the operators is defined by the order of the precedence declarations. The operators in the first precedence declaration have lower precedence than the next. The semantics of the precedence declarations of YACC are defined in terms of parser generation. YACC produces an LALR parse table in which the action has to be deterministic for each state and lookahead. If there are multiple possible actions, then this results in shift/reduce or reduce/reduce conflicts. The precedence declarations are used by YACC to select the appropriate action if there is a conflict between two actions. If there is no precedence declaration for the involved tokens, then YACC will resolve the conflict by preferring a shift over a reduce. For a reduce/reduce conflict, YACC resolves the conflict by selecting the reduce of the first production in the input grammar. Later we will see in more detail what the consequence of this is for the precedence rules.

The main weakness of precedence declarations of YACC is that it is not really a precedence declaration mechanism, i.e. YACC has no notion of precedence of operators. Precedence declarations are a mechanism to resolve conflicts in the parse table, which can be used to *implement* operator precedence. Unfortunately, this requires understanding of LALR parsing and the way YACC generates a parser.

7.3.2 *SDF*

SDF [Heering et al. 1989, Visser 1997b] is a feature rich grammar formalism that integrates lexical and context-free syntax. SDF supports arbitrary context-free grammars, so grammars are not restricted to subclasses of context-free grammar, such as LL or LALR. The SDF parser generator generates a parse table for a scannerless generalized LR parser. For disambiguation, SDF supports various disambiguation filters [Visser 1997b, van den Brand et al. 2002], some of which are used to define precedence rules. The third grammar of Figure 7.1 uses the precedence declarations of SDF[2]. Similar to the second YACC grammar, the productions of this grammar define an ambiguous language. A separate definition of *priorities* is used to define that * takes precedence over +. Also, both operators are defined to be left associative by using the associativity attribute `left`.

The semantic of SDF priorities is well-defined in terms of the grammar, as opposed to operationally in the parser generator. SDF applies the transitive closure to the declared priority relation over productions (which introduces some limitations). Priority declarations generate a set $\mathit{conflicts}(\mathcal{G})$ of tree patterns of the form $\langle A \rightarrow \alpha\langle B \rightarrow \beta\rangle\gamma\rangle$. Note that this pattern has the same form as patterns from the set of one-level tree patterns, excluding injection chains. If $A \rightarrow \alpha B\gamma > B \rightarrow \beta$ is in the closure of the priority relation, then $\langle A \rightarrow \alpha\langle B \rightarrow \beta\rangle\gamma\rangle \in \mathit{conflicts}(\mathcal{G})$. The generated parser will never create a parse tree that matches one of the tree patterns in $\mathit{conflicts}(\mathcal{G})$.

The fourth grammar of Figure 7.1 illustrates that encoding precedence in productions is possible in all grammar formalisms, even if they provide separate precedence declarations. To make the example a bit more interesting, this grammar defines the priority of operators in productions, but uses associativity definitions for individual operators.

7.4 PRECEDENCE RULE RECOVERY

In previous sections, we have argued that there is a need for methods and tools for determining the precedence rules of a grammar. In this section, we present such a method for recovering the precedence rules as encoded in productions or defined using separate precedence declarations.

7.4.1 *A Core Formalism for Precedence Rules*

The recovered precedence rules need to be expressed in a certain formalism. To liberate the precedence rules from the idiosyncrasies of specific grammar formalisms, we need a formalization that is independent of specific parsing techniques. The formalism for precedence rules does not need to be concise or notationally convenient. Rather, it serves as a core representation of precedence rules of programming languages.

[2]SDF uses a reversed notation for production rules. We will only use this notation in verbatim examples of SDF. All other productions are written in conventional $A \rightarrow \alpha$ notation.

$\langle T \to \langle T \sim E \to E + T\rangle * F\rangle$ $\langle T \to T * \langle F \sim T \to T * F\rangle\rangle$ $\langle T \to T * \langle F \sim E \to E + T\rangle\rangle$ $\langle E \to E + \langle T \sim E \to E + T\rangle\rangle$	$\langle E \to \langle E \to E + E\rangle * E\rangle$ $\langle E \to E * \langle E \to E * E\rangle\rangle$ $\langle E \to E * \langle E \to E + E\rangle\rangle$ $\langle E \to E + \langle E \to E + E\rangle\rangle$
$\langle E \to \langle E \to E + E\rangle * E\rangle$ $\langle E \to E * \langle E \to E * E\rangle\rangle$ $\langle E \to E * \langle E \to E + E\rangle\rangle$ $\langle E \to E + \langle E \to E + E\rangle\rangle$	$\langle T \to \langle T \sim E \to E+ E\rangle *T\rangle$ $\langle T \to T * \langle T \to T * T\rangle\rangle$ $\langle T \to T * \langle T \sim E \to E +E\rangle\rangle$ $\langle E \to E + \langle E \to E + E\rangle\rangle$

Figure 7.2 Precedence rules for grammar of Figure 7.1. First row: $YACC_1$, $YACC_2$, second row: SDF1, SDF2

Inspired by previous work on SDF conflict sets defined by priorities [Heering et al. 1989, Visser 1997b], we use parse tree patterns to define precedence rules. Parse tree patterns denote a set of parse trees. Thus, a parse tree pattern can be used to define a set of invalid parse trees. For example, for the grammar SDF1 in Figure 7.1 the tree pattern $\langle E \to \langle E \to E + E\rangle * E\rangle$ denotes a set of invalid parse trees according to the precedence rules of this grammar. However, the precedence rules for a grammar $\mathcal{G}$ cannot just be defined as a subset of $TP_{\mathcal{G}}$. The reason for this is that for grammars that encode precedence in productions, there will be no tree patterns that denote invalid parse trees. Such grammars have a series of expression non-terminals that are only allowed at specific places. For example, in grammar $YACC_1$ of Figure 7.1, the expression E is not allowed at the right-hand side of the operator $+$ in the production $E \to E + T$. Nevertheless, we are interested in precedence rules over such grammars. Therefore, we define the set of precedence rules for $\mathcal{G} = (\Sigma, N, P)$ to be a subset of $TP_{\underrightarrow{\mathcal{G}}(N_E)}$, where $\underrightarrow{\mathcal{G}}(N_E)$ is an **extended context-free grammar** of the grammar $\mathcal{G}$ where $N_E \subseteq N$ and $\underrightarrow{\mathcal{G}}(N_E) = (\Sigma, N, P')$ where $P' = P \cup \{A \to B | A \in N_E, B \in N_E, A \neq B\}$. For example, for the grammar $YACC_1$ in Figure 7.1, $\underrightarrow{YACC_1}(\{E, T, F\})$ contains the injections $E \to F$, $T \to E$, $F \to E$, and $F \to T$ in addition to the productions of $YACC_1$.

Based on this definition we can now introduce the precedence rules for the grammars of Figure 7.1 that are presented in Figure 7.2. First, note that an injection chain $\langle A \to \alpha\langle B \sim C \to \beta\rangle\gamma\rangle$ is used when the symbol C of the nested production is not equal to the symbol B at the place where the nested production is used. Second, note that for the grammar $YACC_1$ the tree pattern $\langle T \sim E \to E + T\rangle$ is not actually valid. This is exactly where $\underrightarrow{YACC_1}$ comes in, since the injection $T \to E$ is present in $\underrightarrow{YACC_1}$.

There is no relation defined between the tree patterns that are members of the precedence rule set, e.g we do not take the transitive closure of a precedence relation over productions. If a precedence declaration for operators needs to be transitively closed for a language, then this should be expressed by having all combinations in the set. A precedence rule set is not by defini-

tion required to be minimal. This means that some tree patterns can define precedence rules that are already implied by other tree patterns.

Precedence rules defined by tree patterns are closely related to the set of conflicts $\mathit{conflicts}(\mathcal{G})$ defined by SDF priority and associativity declarations. One important difference is that the set of conflicts of SDF is transitively closed, since it is defined by a priority relation that is a strict partial ordering between productions. Another difference is that we do not restrict the tree patterns used in the precedence rule sets to trees of two productions. As mentioned before, we do not assume anything about (the feasibility of) a concise notation for the set of tree patterns.

7.4.2 *Tree Pattern Generation*

We recover precedence rules from grammars by generating a set of tree patterns involving expression productions and checking if a parse is possible that will result in a parse tree matching the tree patterns. By default, we generate the set of one-level tree patterns $TP^1_{\underrightarrow{\mathcal{G}}}(N_E)$ for a grammar $\mathcal{G}$ with P restricted to $P = \{A \rightarrow \alpha \in P | A \in N_E\}$, i.e a set of tree patterns involving two productions for all combinations of expression productions. For example, the set of one-level tree patterns for two productions $E \rightarrow E + E$ and $E \rightarrow \&E$ is $\langle E \rightarrow \&\langle E \rightarrow \&E\rangle\rangle$, $\langle E \rightarrow \langle E \rightarrow \&E\rangle + E\rangle$, $\langle E \rightarrow \langle E \rightarrow E + E\rangle + E\rangle$, $\langle E \rightarrow \&\langle E \rightarrow E + E\rangle\rangle$, $\langle E \rightarrow E + \langle E \rightarrow \&E\rangle\rangle$, $\langle E \rightarrow E + \langle E \rightarrow E + E\rangle\rangle$. One-level tree patterns are sufficient to express the precedence rules of most operator languages. Indeed, our case studies in Section 7.6 are based on one-level tree patterns. However, some languages require precedence rules that include 3 or more productions. For this, the precedence recovery tool supports configuration of the number of levels that is to be generated.

Next, the question is how to check if a grammar allows a parse that matches a tree pattern. If the pattern is accepted, then there are valid parse trees for this pattern. If not, then it denotes invalid parse trees and it will be an element of the resulting precedence rule set. Clearly, checking tree patterns is parser generator specific, since we need intimate knowledge about the semantics of the grammar formalism that is used by the parser generator. Based on the requirements for our case studies and our practical needs, we implemented the validation of tree patterns for YACC and SDF. However, the algorithm and the approach that is used can easily be ported to different (Generalized) LR parser generators.

7.4.3 *Precedence Rule Recovery: YACC*

For YACC, the precedence rules are difficult to determine from the grammar directly, since the semantics of precedence declarations in YACC is defined operationally. The precedence declarations are used to resolve conflicts during parser generation, which means that precedence rules are only applied if there is actually a conflict. Also, YACC applies implicit conflict resolution mechanisms, i.e. preference for a shift over a reduce, and preference for a

reduce of the first production in the grammar. Furthermore, grammars can encode the precedence rules in productions and combine this with precedence declarations, an issue that is not YACC specific. Hence, checking the *grammar* for possible matches of tree patterns is complex and requires intimate knowledge of YACC parser generation and conflict resolution. A much more general solution is to validate tree patterns against the *parse table* generated by YACC. Of course, a parser generated by YACC can not parse tree patterns. To check if a tree pattern is valid, we simulate the parsing of a sentential form that results in a parse tree matching the tree pattern. If this is possible, then the tree pattern is valid, otherwise it is invalid.

A shift reduce parser is a transition system with as configuration a stack and an input string. The configuration is changed by shift and reduce actions. A shift action moves a symbol from the input to the stack, which corresponds to a step of one symbol in the right-hand sides of a set of productions that is currently expected. A reduce removes a number of elements from the stack and replaces them by a single element, which corresponds to the application of a grammar production. In an LR parser [Knuth 1965], the information on the actions to perform is stored in an action table. Both a shift and a reduce introduce state transitions, which are recorded on the stack and are based on information in the action and goto table. After popping elements from the stack in a reduce, the goto entries of the state on top of the stack are consulted to find the new state to push on the stack.

To recognize tree patterns, we change the input of the LR parser to a string of tree patterns and symbols. The tree patterns are translated into LR actions and all changes in the configuration of the parser are checked against the actions that are allowed to derive a parse tree that matches the tree pattern. Figure 7.3 lists the transition rules that implement the modified LR parser for recognizing tree patterns. The configuration of the parser, denoted by $\mid stack \mid input \mid$, is rewritten by the transition rules. The stack grows to the right, the input grows to the left. The variable e ranges over all possible input symbols, which is the set $N \cup \Sigma \cup TP_{\overrightarrow{G}}(N_E) \cup \mathcal{R}(P) \cup \overrightarrow{\mathcal{R}}(N)$. Hence, the input of the parser consists of a sequence of non-terminals, terminals, tree patterns, and two special elements for representing reduces. $\mathcal{R}(A \to \alpha)$ represents a reduction of the production $A \to \alpha$. $\overrightarrow{\mathcal{R}}(A)$ represents a reduction of any chain production $B \to C$, until B is A. The function *head* finds the first non-reduce element in its list of arguments.

Equation R_1 defines a shift of a terminal a. This definition is not different from a shift in a normal LR parser. A terminal is removed from the input and a new state is pushed on the stack. Equation R_2 defines the unfolding of a tree pattern $\langle A \to \alpha \rangle$. This transition rule does not exist for an LR parser, since the normal input is a sequence of terminals. The unfolding of a tree pattern involves adding α and a reduce of $\langle A \to \alpha \rangle$ to the input. The reduction is denoted by $\mathcal{R}(A \to \alpha)$. Equation R_3 defines the unfolding of a tree pattern $\langle A \sim B \to \alpha \rangle$. After the unfolding of $\langle B \to \alpha \rangle$, a reduce $\overrightarrow{\mathcal{R}}(A)$ for arbitrary chain productions is inserted. Thanks to the unfolding of productions, the input of

$$\frac{action(s_m, a) = \texttt{shift}(s_{m+1})}{\mid s_0 \dots s_m \mid a, e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m, s_{m+1} \mid e_i \dots e_n \mid} \quad (R_1)$$

$$\frac{}{\mid s_0 \dots s_m \mid \langle A \to \alpha \rangle \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid \alpha, \mathcal{R}(A \to \alpha) \dots e_n \mid} \quad (R_2)$$

$$\frac{}{\mid s_0 \dots s_m \mid \langle A \sim B \to \alpha \rangle \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid \langle B \to \alpha \rangle, \overrightarrow{\mathcal{R}}(A) \dots e_n \mid} \quad (R_3)$$

$$\frac{goto(s_m, A) = s_{m+1}}{\mid s_0 \dots s_m \mid A, e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m, s_{m+1} \mid e_i \dots e_n \mid} \quad (R_4)$$

$$\frac{action(s_{m+k}, head(e_i \dots e_n)) = \texttt{reduce}(A \to X_1 \dots X_k)}{\mid s_0 \dots s_m \dots s_{m+k} \mid \mathcal{R}(A \to X_1 \dots X_k), e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid A, e_i \dots e_n \mid} \quad (R_5)$$

$$\frac{action(s_{m+1}, head(e_i \dots e_n)) = \texttt{reduce}(A \to B)}{\mid s_0 \dots s_m, s_{m+1} \mid \overrightarrow{\mathcal{R}}(A), e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid A, e_i \dots e_n \mid} \quad (R_6)$$

$$\frac{action(s_{m+1}, head(e_i \dots e_n)) = \texttt{reduce}(B \to C), B \neq A}{\mid s_0 \dots s_m, s_{m+1} \mid \overrightarrow{\mathcal{R}}(A), e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid B, \overrightarrow{\mathcal{R}}(A), e_i \dots e_n \mid} \quad (R_7)$$

Figure 7.3 Transition rules for checking tree patterns for a YACC parser

the system can now contain non-terminals. This is the reason for a separate transition rule R_4 for performing a goto, which is usually considered to be a part of the reduce action. The goto transition rule removes a non-terminal from the input and pushes a new state on the stack, determined by the *goto* function. The reason why this works is that we can assume that the non-terminal A is productive, which means that there will always be a production for A that will finally reduce to state s_m, which would lead to exactly the same goto.

Equation R_5 defines a reduce action. The transition system only allows reduces if a reduce is explicitly identified in the input. This method of *checked reduces* is used to enforce the structure of the tree pattern on the parser, i.e. it is not possible to recognize the leafs of the tree pattern with a parse tree that has a different internal structure. The definition of the reduce action reuses the separate transition rule of *goto* by inserting a non-terminal in front of the list. The equations R_6 and R_7 define the reduction of chain productions, which is allowed if there is an $\overrightarrow{\mathcal{R}}(A)$ in front of the input. If the reduce is applied for $A \to B$ then $\overrightarrow{\mathcal{R}}(A)$ is removed from the input and A is added. If the chain production does not produce A, then more chain productions might be necessary. Therefore, the $\overrightarrow{\mathcal{R}}(A)$ is preserved and B is pushed in front of the

unfold	$0 \mid \langle E \rightarrow E + \langle E \rightarrow E * E \rangle \rangle \mid$
goto	$0 \mid E, +, \langle E \rightarrow E * E \rangle, \mathcal{R}(+) \mid$
shift	$0,3 \mid +, \langle E \rightarrow E * E \rangle, \mathcal{R}(+) \mid$
unfold	$0,3,5 \mid \langle E \rightarrow E * E \rangle, \mathcal{R}(+) \mid$
goto	$0,3,5 \mid E, *, E, \mathcal{R}(*), \mathcal{R}(+) \mid$
shift	$0,3,5,7 \mid *, E, \mathcal{R}(*), \mathcal{R}(+) \mid$
goto	$0,3,5,7,6 \mid E, \mathcal{R}(*), \mathcal{R}(+) \mid$
reduce	$0,3,5,7,6,8 \mid \mathcal{R}(*), \mathcal{R}(+) \mid$
goto	$0,3,5 \mid E, \mathcal{R}(+) \mid$
reduce	$0,3,5,7 \mid \mathcal{R}(+) \mid$
goto	$0 \mid E \mid$
accept	$3,0 \mid \mid$

unfold	$0 \mid \langle E \rightarrow \langle E \rightarrow E + E \rangle * E \rangle \mid$
unfold	$0 \mid \langle E \rightarrow E + E \rangle, *, E, \mathcal{R}(*) \mid$
goto	$0 \mid E, +, E, \mathcal{R}(+), *, E, \mathcal{R}(*) \mid$
shift	$0,3 \mid +, E, \mathcal{R}(+), *, E, \mathcal{R}(*) \mid$
goto	$0,3,5 \mid E, \mathcal{R}(+), *, E, \mathcal{R}(*) \mid$
error	$0,3,5,7 \mid \mathcal{R}(+), *, E, \mathcal{R}(*) \mid$

Figure 7.4 LR configuration sequences for a valid and invalid tree pattern

input to trigger a goto.

Using the extended LR parser that operates on tree patterns, the parsing of an actual input of the form of a tree pattern is simulated in detail. To illustrate the validation of tree patterns, Figure 7.4 shows the configuration of a parser generated from grammar $YACC_2$ (Figure 7.1) for every application of a transition rule. The $\mathcal{R}(*)$ and $\mathcal{R}(+)$ inputs are abbreviations for the complete productions of these operators. The tree pattern on the left is valid. The tree pattern on the right is invalid, since in the last configuration the lookahead is the terminal *. For this lookahead, a reduce of the + operator is not allowed, since that would give the + operator precedence over *. Thus, parsing fails and the tree pattern is invalid.

By working on the parse table generated by YACC, the recovery supports all precedence rules of a YACC grammar: encoded in productions, defined using precedence declarations, and even implicit conflict resolution. Indeed, if we remove the precedence declarations from $YACC_2$, then the precedence rule recovery returns $\langle E \rightarrow \langle E \rightarrow E * E \rangle * E \rangle$, $\langle E \rightarrow \langle E \rightarrow E + E \rangle * E \rangle$, $\langle E \rightarrow \langle E \rightarrow E * E \rangle + E \rangle$, $\langle E \rightarrow \langle E \rightarrow E + E \rangle + E \rangle$, which illustrates that YACC prefers a shift over a reduce.

Bison has a detailed report function that provides information about the generated LR parse table, item sets, shifts, gotos, reduces, and conflicts. We parse this output to get a representation of the parse table. The tree pattern parser is implemented in Stratego. The transition rules of Figure 7.3 directly correspond to rewrite rules in the Stratego implementation, which are applied using a rewriting strategy. The configurations of the parser can be inspected, which was used to produce the examples of configuration sequences of Figure 7.4. The implementation of the transition system takes 55 lines of code.

7.4.4 *Precedence Rule Recovery: SDF*

For recovering precedence rules from SDF grammars, an analysis of the grammar would be feasible, since the precedence declarations of SDF are not op-

erationally defined in terms of parser generation. Yet, supporting a mixture of encoded and separately defined precedence declarations can still be rather involved. Based on the success of the approach that we used for recovering YACC precedence rules, we chose the same method for SDF grammars. Thus, precedence rules are recovered by checking generated tree patterns up to a certain level against the parse table generated from an SDF grammar.

We cannot reuse the transition system (a modified LR parser) that we defined for checking tree patterns against YACC parse tables, since SDF is implemented using a scannerless generalized LR parser, called SGLR. Because the parser uses the generalized LR algorithm, there will be cases where multiple actions are possible in some configuration, for example a shift as well as a reduce action. To handle the alternatives, the GLR configuration needs to be forked, where in the end one of the alternatives has to succeed to make a tree pattern valid. Furthermore, the scannerless generalized LR parser generator uses a different method for applying precedence declarations to the parse table. Whereas YACC uses precedence declarations to resolve conflicts between shift and reduce actions, SDF effectively prunes the goto table of a parse table. SGLR refines the goto table from gotos based on symbols to gotos based on productions, i.e the goto table is now a table of states and productions instead of states and symbols [Visser 1997b]. This slightly complicates the definition of the transition system, since the system applies gotos that are not introduced by a reduce, but by a non-terminal in the tree pattern. For this reason, we distinguish such a goto from a goto induced by a reduce. In the case of a goto caused by a non-terminal in the input, we consider all possible gotos for this non-terminal. We determine the set of possible states where the parser can goto from the current configuration and fork the GLR configuration to check all alternatives. Our method supports ambiguous grammars, which is illustrated by the case studies of Section 7.6, where two ambiguous grammars for C are compared.

The implementation of the precedence recovery tool for SDF is a very basic and somewhat naive GLR parser. However, for the size of tree patterns this is not an issue at all. Again, the checker is implemented in Stratego using rewrite rules that rewrite the GLR configuration.

7.5 PRECEDENCE COMPATIBILITY

Comparing the language defined by two grammars is undecidable, but this does not mean that nothing can be said about the compatibility of two grammars. Static analysis tools, such as our precedence rule recovery tool, can be used to extract information from different grammars and compare the results, even if they are written in different grammar formalisms.

While the precedence rules are represented in a grammar formalism independent formalism, this does not imply that precedence rules can be compared directly in a useful way after recovering them from two different grammars. Grammars usually have different naming conventions, different names for lexical symbols, and often also have a different structure at some points.

The recovered precedence rules can still be compared by first applying grammar transformations to the precedence rules to achieve a common representation. After this, the comparison of precedence rules is a simple set comparison.

7.5.1 *Grammar Transformation*

Precedence rule recovery usually results in rather big sets of tree patterns. Trying to transform this huge set of tree patterns to a common representation is usually not a good idea. To avoid working with this large set of precedences, it is a good idea to first extract the productions from the precedence rules and compare and transform the set of productions in order to find the required set of grammar transformations that achieves a common representation. Also, this is the most convenient way to identify language extensions that are only present in one of the two grammars.

The relationship between two grammars is something that has to be custom defined for a particular combination of grammars. Typically, one of the grammar transformations that needs to be applied to the precedence rules is the renaming of all expression symbols to a single expression symbol. Note that it is essential that this renaming is applied to the precedence rules and not to the original grammar, since that would most likely change the precedence rules of the language or even make it impossible to generate a parser.

Similar to the renaming of expression symbols, injections caused by the application of chain productions are no longer useful. To achieve a common representation, all injection chain nodes $\langle B \sim C \to \beta \rangle$ are transformed to $\langle C \to \beta \rangle$

In the comparison of a YACC grammar and an SDF grammar a common issue is that the YACC precedence rules use names for the operators of the language (e.g. `ANDAND` instead of `&&`). This is usually a straightforward renaming where the lexical specification can be consulted if necessary.

Another common difference between grammars are different factorizations. For example, the first grammar might have a single assignment production using a nonterminal `AssignmentOperator` and separate productions for the various assignment operators (e.g. `=`, `*=`, `+=`), whereas the second grammar might have separate productions for all these assignments. Such structural differences between the grammars can be solved by inlining the alternatives of the `AssignmentOperator`. The grammar transformations can be implemented generically, which we have indeed done for our case studies.

7.6 EVALUATION

We have evaluated the method for precedence rule recovery and compatibility checking by applying the implementation for YACC and SDF to a set of grammars for the C and PHP languages. Both languages have a large number of operators and non-obvious precedence rules. The size and complexity of the languages makes this compatibility check a good benchmark for our method.

The case studies of C and PHP are motivated by practical problems we experience. For C99, there are two high quality SDF grammars available. Currently, it is difficult to choose between these grammars, since it is unclear if both are standard compliant and if there are any differences in the language they define. To compare the grammars, a method and tool for determining the precedence rules is most useful. For PHP, we experienced the problem that the precedence rules are not very well defined. PHP has many operators (many more than C or Java) and the operators are also a bit unusual. For example, PHP features a unary operator with very low precedence. There is no specification of PHP, so the parser of the official PHP distribution defines the syntax of PHP. This parser is generated from a YACC grammar, and as we have argued the precedence rules are difficult to determine from a YACC grammar without having intimate knowledge of the parser generator.

7.6.1 *C99*

We have compared three grammars for C99:

- The C compiler of the the GNU Compiler Collection uses a parser generated from a YACC grammar[3]. The YACC grammar uses a mixture of precedence declarations and encoding of priorities in productions.
- The Transformers project provides a C99 SDF grammar [Borghi et al. 2006]. This grammar is a direct translation of the standard to SDF [4]. The grammar does not use SDF precedence declarations. Instead, it uses an encoding of precedence in productions as specified by the standard. The grammar is designed to be ambiguous where the C syntax is ambiguous.
- The SDF Library provides an ANSI C SDF grammar [5]. Unlike C-Transformers, this grammar uses SDF precedence declarations. The grammar is designed to be ambiguous.

The precedence tools reported various differences between the grammars. All the reports have been verified as being real differences, i.e there were no false positives. Examples of the reported differences are:

$\langle E \rightarrow \texttt{sizeof}\langle E \rightarrow \texttt{(}\mathit{TypeName}\texttt{)}\ E\rangle\rangle$
: A cast as an argument of `sizeof` is forbidden in GCC and C-Transformers, which is correct, but it is allowed in the SDF Library, which is a bug.

$\langle E \rightarrow \texttt{++}\langle E \rightarrow \texttt{(}\mathit{TypeName}\texttt{)}\ E\rangle\rangle \quad \langle E \rightarrow \texttt{--}\langle E \rightarrow \texttt{(}\mathit{TypeName}\texttt{)}\ E\rangle\rangle$
: GCC and SDF Library allow a cast as an argument of ++ and --. The C-Transformers do not, which corresponds to the standard. The standard defines ++ and -- separate from unary operators, while GCC and the SDF Library ignore this difference.

[3] In GCC 4.1 the Bison-generated C parser has been replaced with a hand-written recursive-decent parser. We use the Bison grammar for GCC 4.03.

[4] We used revision 1611 of the transformers-c-tools package. The one bug we found has been fixed in revision 1613.

[5] We used revision 20649 of the sdf-library package for our evaluation.

$E \rightarrow$ `sizeof(`*TypeName*`)`
: Though not a precedence problem, our tools reported this missing production in the SDF library grammar. This means that some `sizeof` expressions that should be parsed ambiguously are currently unambiguous.

$\langle E \rightarrow E\,\texttt{?}\,E\,\texttt{:}\,\langle E \rightarrow E = E\rangle\rangle$
: This tree pattern of an assignment in the else branch of the conditional is forbidden in GCC and the SDF Library, but is allowed in C-Transformers. This is a bug in C-Transformers: the else branch of the conditional operator uses the wrong non-terminal

$\langle E \rightarrow \langle E \rightarrow E\,\texttt{?}\,E\,\texttt{:}\,E\rangle = E\rangle$ $\langle E \rightarrow \langle E \rightarrow \texttt{(}\,\mathit{TypeName}\,\texttt{)}\,E\rangle = E\rangle$
: A conditional or a cast in the left-hand side of an assignment is allowed by GCC and the SDF Library. For GCC this is a legacy feature that now produces a semantic error. C-Transformers forbids this, which is correct. The same issue holds for many more binary operators (`||`, `&&`, `|`, `^`, `&`, `!=`, `==`, `>=`, `<=`, `>`, `<`, `<<`, `>>`, `-`, `+`, `%`, `/`, `*`). The C standard only supports unary operators in the left-hand side of an assignment.

7.6.2 *PHP 5*

We compared three grammars for PHP:

- The official PHP distribution comes with a YACC grammar for PHP, as part of the Zend engine. The grammar makes heavy use of YACC precedence declarations [6].
- The open source PHP compiler PHC comes with a YACC grammar that has been forked from the PHP distribution [7].
- PHP-front provides a syntax definition for PHP 4.0 and 5.0 in SDF.

For PHP YACC versus PHC YACC the precedence tool reported several major bugs in the PHC YACC grammar: several operator precedences have been inverted since the fork of the grammar. For example, in PHC the `||`, `OR`, and `XOR` operators had precedence over respectively `&&`, `AND`, and `AND`. This issue was reported by our tools as a missing precedence rule $\langle E \rightarrow \langle E \rightarrow E\,\texttt{||}\,E\rangle\,\texttt{\&\&}\,E\rangle$ in PHC. For each precedence rule in PHC that was not in PHP, there was a corresponding rule in PHC that was not in PHP. For example, the rule corresponding to the previous pattern is $\langle E \rightarrow E\,\texttt{||}\,\langle E \rightarrow E\,\texttt{\&\&}\,E\rangle\rangle$.

For PHP versus the PHP-front SDF grammar we expected many differences in the precedence rules. We were already aware of various issues in the precedence of operators of the PHP-front grammar. Actually, the uncertainty about the exact precedence rules of PHP was the primary motivation to develop this

[6] We used PHP 5.2.0 for our evaluation.

[7] We used PHC 0.1.7 for our evaluation. All bugs have been fixed by the developers of PHC after our reports.

method of precedence rule recovery. One of the questions that we want to answer in this project is whether the PHP precedence rules can actually be expressed in SDF. The PHP operators are a bit unusual since PHP has very weak as well as very strong binding unary operators. The transitive closure of priorities in SDF results in various cases where we could not find a solution by hand. In future work, we plan to analyse precedence rule sets to extract characteristics and hopefully determine automatically whether these precedence rules can be expressed using grammar formalism specific precedence declaration mechanisms, in this case SDF priorities.

7.7 RELATED WORK

7.7.1 *Grammar Engineering Vision*

Several researchers have suggested that there is a strong need for proper foundations and practices for grammar engineering [Lämmel & Verhoef 2001, Klint et al. 2005, Sellink & Verhoef 2000, Lämmel 2001a]. In particular, [Klint et al. 2005] presents an extensive research agenda for grammar engineering. Our method for recovery and compatibility checking of precedence rules is highly related to several of the presented research challenges, such as maintaining consistency between the incarnations of conceptually the same grammar. Also, our precedence rules help to abstract from the idiosyncratic precedence mechanisms provided by the various parser generators in use. Our precedence rule recovery method is very useful in the semi-automatic grammar recovery process [Lämmel & Verhoef 2001] from language references and existing compilers. In particular, more automation of grammar recovery is now possible, since precedence declarations can be checked during the life-time of a grammar.

7.7.2 *Grammar Engineering Tools*

The Grammar Deployment Kit (GDK) [Kort et al. 2002] targets the process of producing a working parser from a specification. An important goal is parser generator independence. The GDK provides tools to generate parser generator specific grammars from a universal grammar formalisms, called LLL. The GDK does not provide more advanced grammar analysis tools, such as our precedence recovery tool. Parser generator independence can be increased by our representation of precedence rules, for which there is no comparable concept available in the GDK. Sellink and Verhoef [Sellink & Verhoef 2000] present the vision and implementation of a set of tools for grammar reengineering, such as assessment (metrics) and conversion tools. BNF2SDF automatically improves the resulting grammar by using EBNF list notations, but does not consider precedence rules. Early versions of Stratego/XT provided a similar tool `yacc2sdf` [de Jonge & Monajemi 2001]. Our precedence recovery for YACC should be integrated in such a tool. Lämmel [Lämmel 2001a] discusses a formal approach to grammar transformation based on a concise

set of primitives and combinators for refactoring, extension, and restriction of grammars. This work does not consider grammars that use separate disambiguation mechanisms. Also, the authors define some equivalency notions for grammars. Comparing precedence rules is a very restricted form of structural equivalence, which is much easier than comparing grammars in general. Schatborn [Schatborn 2005] presents a feature rich grammar transformation language for SDF, providing modules, functions, variables, types, and patterns to facilitate the development of grammar independent grammar transformations, based on a detailed case study of the transformation from ANSI C from YACC to SDF. Advanced grammar analysis tools, such as our precedence recovery, would be valuable in combination with such a language.

7.7.3 *Grammar Testing*

Lämmel [Lämmel 2001b] contributes grammar coverage analysis techniques, combined with test set generation, applied to grammar recovery. Checking the correct implementation of precedences could be implemented using test set generation accompanied by code coverage requirements. Our method just exercises the parsing of operators using sentential forms, ignoring the actual values of the expression. Also, tests need a description of the expected result, which is usually parser specific (not just parser *generator* specific, like our method). Our method does not actually run the parser, which makes it easier in practice to test expressions in isolation. In this way, we also have very precise control over the correct behaviour of the parser, which makes a comparison to the result of the parser unnecessary.

7.7.4 *Pretty-Printing*

The ASF+SDF Meta-Environment provides a tool `restorebrackets` that inserts parenthesis where necessary according to priorities defined in a grammar. Stratego/XT's `sdf2parenthesize` is similar to the tool `restorebrackets`. However, both tools only support precedence rules defined using SDF priorities and associativity. As a result, they fail to insert parenthesis for grammars that encode precedence rules in grammar productions. Compared to this earlier work, our new method identifies all conflicts between operators, not just those that are specified using precedence declarations.

7.8 CONCLUSION

We have presented a method for recovering precedence rules from grammars. We have presented the algorithm for YACC and implemented the method in tools for YACC and SDF. As far as we know, this is the first effort to develop methods and tools for reliably assisting grammar developers with the recovery of precedence rules, migration of grammars with precedence rules, and compatibility checking of grammars. Although there are many open issues and opportunities for further research, the evaluation of our current proto-

types has already clearly demonstrated the value of the tools that we have presented.

ACKNOWLEDGMENTS

This research was supported by the NWO/JACQUARD project *TraCE: Transparent Configuration Environments* (638.001.201). The development of the SDF grammar for PHP was sponsored by the *Google Summer of Code 2006*. We thank Mark van den Brand, Giorgios Robert Economopoulos, Jurgen Vinju, and the rest of the SDF/SGLR team for their work on SDF. We thank Valentin David and the Transformers team at the EPITA Research & Development Laboratory for the SDF grammar for C99. We thank the anonymous reviewers of LDTA 2007 for providing useful feedback on an earlier version of this chapter.

Conclusion

8

In this dissertation we have studied techniques for the principled combination of multiple textual software languages into a single, composite language. The applications we have studied motivate the need for composing languages, e.g for syntactic abstraction, syntactic checking of metaprograms with concrete object syntax, and the prevention of injection attacks. We have extensively evaluated the application of modular syntax definition, scannerless parsing, generalized parsing, and parse table composition for composing languages.

Our case studies provide strong evidence that the aggregate of these techniques is the technique for the principled combination of languages into a single, composite language. First, using these techniques we have been able to formally define the syntax of a series of composite languages, in particular AspectJ, a complex, existing language conglomerate for which no formal syntax definition was available. Second, we have explained in detail how employing scannerless and generalized parsing techniques elegantly deals with the issues in parsing such combinations of languages, in particular context-sensitive lexical syntax. Third, we have shown for various applications that we can avoid the need to spend considerable effort on crafting specific combinations of languages. The resulting genercity of these applications is due to techniques that allow the syntax of a composite language to be defined as the principled combination of its sub-languages. Fourth, we have introduced and evaluated parse table composition as a technique that enables the separate compilation and deployment of language extensions. This allows the separate deployment of syntax embeddings as plugins to a compiler. Hence, end-programmers can select syntax embeddings for use in their source programs, but do not have to wait for the parse table to be compiled from scratch.

Next, we summarize the contributions of this dissertation. We refer to the individual chapters for a more detailed account of our contributions.

MetaBorg

> We present the MetaBorg method for introducing domain-specific syntactic abstractions for existing class libraries in a general-purpose language. The method is distinguished by its generality, i.e. the lack of restrictions on either the syntax or the semantics of embedding and assimilation.

StringBorg

> We introduce the StringBorg method for preventing injection attacks by embedding the syntax of guest languages in a host language. Because of the combination of generative programming and a principled and

generic approach to syntax embedding, it takes effort $\Theta(N + M)$ rather than $\Theta(N \times M)$ to support N guest languages in M host languages.

Syntax Definition for Language Conglomerates

We provide an in-depth analysis of the intricacies of parsing AspectJ, a prototypical example of a language conglomerate. To improve upon the lack of a formal definition of the AspectJ syntax and the issues with existing parsers, we have formally defined the full syntax of AspectJ. This is also a case study showing the applicability of scannerless generalized LR parsing to complex programming languages.

Motivating Scannerless Parsing

For various applications, we have given an account of the application of scannerless parsing to elegantly deal with context-sensitive lexical syntax. This has resulted in significantly more insight in the applications of scannerless parsing.

Grammar Mixins

We introduce a mixin-like mechanism for combining syntactic extensions and instantiating sub-languages for use in different contexts. Grammar mixins are applied in all our applications of syntax embeddings.

Parse Table Composition

We introduce the idea of parse table composition as symmetric composition of parse tables, thus enabling the separate compilation and deployment of syntax embeddings. Parse table composition has a formal foundation in finite automata, i.e. the algorithm partially reapplies the conversion of an NFA to a DFA.

Generalized Type-based Disambiguation

We introduced a generic method for disambiguation of concrete object syntax embeddings. By leveraging the type system of the host language the method removes the need for explicit tagging of quotations and antiquotations in metaprogramming with concrete object syntax. The disambiguation method is generic in the embedded object language.

Lightweight Disambiguation

For StringBorg we introduced a more lightweight disambiguation method that is also applicable in dynamically typed languages. Similar to type-based disambiguation this method preserves ambiguities in the parsing phase. However, the ambiguities are not resolved statically, rather the method only checks at run-time if antiquoted values syntactically fit in the object language fragment.

Precedence Rule Recovery and Migration

We define a core formalism for specifying precedence rules of expression languages and describe a novel method for recovering precedence

rules from grammars. The recovered specification of precedence rules can be used to check whether grammars have compatible precedence rules or to migrate grammars to a different grammar formalism. Our precedence rule migration method supported the development of a formal and declarative definition of the PHP syntax.

8.1 FUTURE WORK

Our applications of modular syntax definition and scannerless generalized parsing are a strong motivation for continued (or renewed) research into fully automatic parser generation. For example, for our solution to preventing injection attacks to become mainstream, it is crucial for the generated parsers to feature production quality, language-specific error reporting, error recovery, and acceptable performance. Surprisingly, deriving a production quality parser from a declarative, possibly ambiguous, syntax definition is still one of the open problems in research on parsing techniques. Perhaps, one of the reasons is that the main application of parser generators has been compilers for single programming languages that are designed to be parsed using algorithms that can easily be implemented by hand. Considering the user-base of compilers, it is worth the effort to spend considerable time on handcrafting such a compiler. For these use cases, there is no strong argument for using parser generators, in particular because the generated parsers usually do not have a better user experience [1].

However, crafting a parser for specific combinations of a host language and its extensions does not scale to the full vision of applications such as StringBorg. As a result, there is a strong motivation for renewed research into bringing the user experience of fully automatic generated parsers closer, or possibly beyond, the user experience of handcrafted parsers.

8.1.1 *Scannerless Generalized LR Parser User Experience*

Error Reporting and Recovery

Scannerless and generalized LR parsing have attractive properties for parsing language conglomerates, yet they introduce complications in error reporting and recovery. Due to the forking LR parsers of the generalized LR algorithm, it is not immediately obvious how existing techniques for error reporting and recovery of LR parsers can be applied.

Scannerless parsing introduces another challenge for error reporting, since errors are usually reported in terms of tokens in conventional scanner-based parsers. Scannerless parsers have no notion of tokens, i.e. they operate on individual characters and nonterminals. Currently, the implementation of SGLR reports errors in terms of characters, without any knowledge about sequences of characters the user might experience as a token. The parser does

[1] Surprisingly, PHP users accept the very poor error reports of the generated PHP parser. The parser reports errors in terms of symbolic names for tokens, e.g. "parse error: unexpected T_ECHO in foo.php on line 2", where figuring out the definition of T_ECHO is left as an exercise to the user.

not even report expected characters. Moreover, reporting expected tokens is usually more informative than reporting expected characters. While the ideal situation would be to have an error reporting strategy that is completely based on a grammar, error messages may improve considerably by providing language-specific examples [Jeffery 2003].

The current implementation of SGLR does not perform any error recovery. Declarative mechanisms are needed to specify strategies for error recovery. In practice, grammars are often extended to handle syntactic errors and gracefully continue parsing to report as many errors as possible. If languages are being extended, these error recovery rules might become invalid and might conflict with the language extensions. Preferably, error recovery should be fully automatic [Charles 1991], but again example-based approach might be a valuable edition.

Valkering [Valkering 2007] has done early experiments with applying error reporting and recovery techniques for LR parsers to scannerless generalized LR parsers.

Performance

The performance of scannerless generalized LR parsers is one of the most frequently asked questions. Indeed, the performance of scannerless generalized LR parsers has an air of mystery about it. One of the reasons is that there is no definite answer: the performance of the generalized LR algorithm depends heavily on the grammar. It has been shown that although $O(n^3)$ in the worst case (with n the length of the input), generalized LR performs much better on grammars that are near-LR [Rekers 1992]. Our benchmark of the scannerless generalized LR parser applied to the AspectJ syntax definition shows that the performance of the SGLR parser for this parse table is linear and a constant factor slower than the abc parser. However, for production quality parsers this constant factor is still not acceptable. Fortunately, there is good hope that the performance of SGLR can be much improved. There are alternative GLR implementations (e.g. [Aycock & Horspool 1999, McPeak & Necula 2004]) and alternative algorithms such as right nulled generalized LR [Scott & Johnstone 2006] with better performance than SGLR. However, these techniques have not yet been extended to scannerless parsing, while scannerlessness is essential for our applications.

The performance of parsing lexical syntax might improve substantially with a parser that operates in three modes: generalized LR for non-deterministic parts of the grammar, LR for deterministic parts, and finite automata for lexical syntax that has a regular grammar. The hybrid of generalized LR and LR already proved to be successful for the Elkhound generalized LR parser [McPeak & Necula 2004].

To evaluate the performance of scannerless generalized LR parsing, the implementation should be benchmarked extensively using a range of domain-specific and mainstream programming languages. This benchmark should include an evaluation of parse table formats (i.e. SLR, LALR, and LR(1)). Johnstone et al. conclude that the performance benefit of LR(1) tables over

LALR or SLR is minor in the context of generalized LR parsers [Johnstone et al. 2004]. Rekers concluded LALR tables are not useful in generalized LR parsers either [Rekers 1992]. However, it is unclear whether these results apply to scannerless generalized LR parsers as well.

Even after all these techniques are adopted, there remains a theoretical performance gap between generalized LR and LALR, since the complexity of GLR depends on the grammar. Therefore, it would be useful to develop profiling tools that help grammar developers to detect performance bottlenecks in grammars.

8.1.2 *Syntax Definition for Real Life Programming Languages*

Many mainstream programming languages have one or more syntactic features that cannot be defined in the current version of the SDF syntax definition formalism. For the future development of new programming methods based on syntax embeddings, it is crucial that grammar formalisms actually support the syntactic features that are used by programming languages and domain-specific languages, since the applications of syntax embeddings require the syntax of the host as well as the guest languages to be expressible in a grammar formalism.

Unicode Support

The current implementation of the SDF syntax definition formalism only supports characters from the ASCII character set, which is a major limitation for applications involving languages such as Java, C#, and XML. The scannerless generalized LR parser implementation should be extended to support Unicode, which is not entirely trivial due to the integration of the scanner in the parser. Research is necessary to determine if techniques for lexical analysis of inputs of large character sets are applicable to scannerless parsers.

Lexical Translations

Some programming languages (e.g. Java) support Unicode escape sequences for arbitrary characters in the input. Before lexical analysis the Unicode escape sequences are translated to their corresponding Unicode character. The SDF syntax definition for Java that is used in the implementation of our examples and prototypes only supports Unicode escape sequences in string literals, since the SDF syntax formalism has no facilities to define such *lexical translations*. Preprocessing the input of the parser is a poor workaround, because lexical translations could be context-sensitive when parsing language conglomerates.

Context-Sensitive Syntax

Unfortunately, some languages do not have a context-free grammar. Many languages have slightly context-sensitive language constructs, such as *here documents* (e.g. shell, PHP), or an indentation rule (e.g. Haskell, Python). SDF does not fully support the definition of the syntax of such languages. A po-

tential solution to the problem of indentation rules is to parse these programs using an ambiguous grammar or add basic features for context-sensitive languages to the parser.

Precedence Mechanisms

As illustrated in Chapter 7 some programming languages have rather complex precedence rules. While precedence rule recovery is a solution for the migration of grammars from one grammar formalism to another, we explicitly do not claim that our precedence rule sets are better than existing precedence declaration mechanism, such as an encoding of the precedence rules in nonterminals. However, our precise definition of precedence rules could be used to evaluate the requirements for precedence declaration mechanisms in grammar formalisms. Also, grammar engineering support should be developed to find a suitable encoding of the precedence rules of a grammar in nonterminals. In particular, this is necessary if a grammar formalism has no precedence mechanism that corresponds to our precedence rules.

8.1.3 *Module Systems for Grammar Formalisms*

Unfortunately, there are not many parser generators that support a module system as part of their input grammar formalism. Due to the limited support for module systems, there is a major lack of insight in the design space of modularity features that are needed in a grammar formalism. For example, grammar mixins proved to be very useful in our applications of syntax embeddings, while grammar mixins are not supported by the SDF module system. We have contributed two important ingredients for a revision of the module system and its implementation (i.e. grammar mixins and parse table composition), but we have not yet designed a new module system that integrates them in the grammar formalism. Currently, an external tool is used to *generate* SDF grammar mixin modules.

For the case studies of Chapter 6 syntax definitions are compiled to parse table components without any specification of the external components they depend on. All symbols are assumed to be possibly overlapping, i.e. they are all public and can be extended by other components. Furthermore, when compiling a syntax definition to a parse table component there is no check whether symbols are defined in other components at all. This might result in error messages about undefined symbols at composition (linking) time. Clearly, there is a need for the introduction of *grammar interfaces* in the module system of the grammar formalism. The use of grammar interfaces can also make sure that symbols do not accidentally overlap, e.g. by making all non-public symbols internal. Thus, introducing grammar interfaces in a module system separates the implementation of the syntax of a language from its interface.

Parse table composition has been developed for application at parse time, but the composition algorithm could also be applied by the parser generator itself. For example, if a single parse table component uses a grammar mixin

in multiple contexts, then it might be useful to compile the grammar mixin separately first.

8.1.4 *Grammar Engineering*

To make the applications presented in this dissertation work in practice, grammars need to become a software artifact with solid engineering practices and supporting tools. For example, reliable methods are necessary to migrate a grammar from one grammar formalism to another. Unfortunately, tool support for semi-automatic grammar migration is currently restricted to ad-hoc tools. In Chapter 7 we presented grammar engineering support for the reliable migration of precedence rules, which was one of the major obstacles we encountered in our case studies. However, more work is necessary in this direction. For example, there should be tool support for deriving an encoding of precedence rules in different expression nonterminals from a set of precedence rules. Also, a complete semi-automatic migration tool from a lexical analyzer specification (e.g. flex) and grammar (e.g. yacc) to an SDF syntax definition would be most useful.

Also, grammar engineering support is necessary for testing, profiling, and analyzing grammars. The advantage of LR-like parser generators is that the developer is forced to develop an unambiguous grammar, i.e. the existence of an LR grammar is a proof that the grammar is unambiguous. In general, it is not possible to prove this for arbitrary context-free grammars. Hence, grammars developed for a generalized LR parser generator can always turn out to be ambiguous in some obscure, unexpected cases. To minimize this risk, it is important to develop grammar analysis and testing tools. A first step in this direction is the SDF unit testing tool *parse-unit*, which is part of Stratego/XT [Stratego Website].

8.1.5 *Composition of Assimilations*

In this dissertation we mostly focus on the issues in defining the syntax of combinations of languages and parsing syntax embeddings. For the assimilation of language extensions, some of our applications (such as StringBorg, Chapter 4) use *generic* assimilations, which have the attractive property that they can be applied to programs with multiple guest language embeddings without any additional effort for specific combinations of embeddings.

However, MetaBorg assimilations can also be specific to a certain syntax embedding (e.g. the assimilation of Swul, Section 2.3.2) and the transformation these assimilations apply to the source program is not restricted in any way. Such assimilations can usually still be combined with assimilations of other language embeddings, but often some metaprogramming effort and understanding of the individual assimilations is required to compose them. This effort for specific combinations of language extensions is not desirable, since the *end-programmer* should be able to select language extensions he wants to use in his programs. Therefore, composing the assimilations of a configuration of

language extensions should be fully automatic. Nevertheless, some language extensions are inherently not compatible with each other. In such cases, the end-programmer should get a clear error report instead of incorrectly assimilated programs. Obviously, checking the compatibility of language extensions should work for *arbitrary* extensions, not just the extensions the developer of the extension is aware of.

Designing a more high-level domain-specific transformation language might be a solution for this. This domain-specific language would need to support typical assimilation scenarios at a higher level of abstraction, which might enable a composition tool to analyze whether assimilations can be composed.

Bibliography

[Aasa et al. 1988] Aasa, A., Petersson, K., & Synek, D. (1988). Concrete syntax for data objects in functional languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, (pp. 96–105). ACM Press. (Cited on page 61.)

[Aho & Johnson 1974] Aho, A. V. & Johnson, S. C. (1974). LR parsing. *ACM Computing Surveys*, *6*(2), 99–124. (Cited on page 154.)

[Aho et al. 1975] Aho, A. V., Johnson, S. C., & Ullman, J. D. (1975). Deterministic parsing of ambiguous grammars. *Communications of the ACM*, *18*(8), 441–452. (Cited on pages 184 and 188.)

[Aho et al. 1986] Aho, A. V., Sethi, R., & Ullman, J. (1986). *Compilers: Principles, Techniques, and Tools*. Reading, Massachusetts: Addison Wesley. (Cited on pages 154 and 158.)

[Aldrich 2005] Aldrich, J. (2005). Open modules: Modular reasoning about advice. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *LNCS*, (pp. 144–168). Springer. (Cited on page 130.)

[Allan et al. 2005] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., & Tibble, J. (2005). Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, (pp. 345–364)., San Diego, California, USA. ACM Press. (Cited on page 94.)

[Anley 2002a] Anley, C. (2002a). Advanced SQL injection. `http://www.ngssoftware.com/papers/advanced_sql_injection.pdf`. (Cited on page 90.)

[Anley 2002b] Anley, C. (2002b). (more) Advanced SQL injection. `http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf`. (Cited on page 90.)

[Arnoldus et al. 2007] Arnoldus, B. J., Bijpost, J. W., & van den Brand, M. G. J. (2007). Repleo: A syntax-safe template engine. In proceedings GPCE 2007 [Lawall 2007]. (Cited on page 148.)

[ASF+SDF MetaEnv] ASF+SDF Meta-Environment website. `http://www.meta-environment.org`. (Cited on page 36.)

[AspectJ] AspectJ. *AspectJ Documentation*. With links to the AspectJ Programming Guide and the AspectJ 5 Developer's Notebook. (Cited on page 99.)

[Avgustinov et al. 2005] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., & Tibble, J. (2005). abc: an extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'04)*, (pp. 87–98)., New York, NY, USA. ACM Press. (Cited on pages 94 and 101.)

[Avgustinov et al. 2007] Avgustinov, P., Hajiyev, E., Ongkingco, N., de Moor, O., Sereni, D., Tibble, J., & Verbaere, M. (2007). Semantics of static pointcuts in AspectJ. In Felleisen, M. (Ed.), *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (pp. 11–23)., New York, NY, USA. ACM Press. (Cited on page 142.)

[Aycock & Horspool 1999] Aycock, J. & Horspool, R. N. (1999). Faster generalized LR parsing. In *Proceedings of 8th International Conference on Compiler Construction (CC'99)*, volume 1575, (pp. 32–46)., Amsterdam. Springer-Verlag. (Cited on pages 141 and 206.)

[Bachrach & Playford 2001] Bachrach, J. & 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)*, (pp. 31–42). ACM Press. (Cited on pages 42 and 61.)

[Baker & Hsieh 2002] Baker, J. & Hsieh, W. (2002). Maya: multiple-dispatch syntax extension in Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, (pp. 270–281). ACM Press. (Cited on pages 62 and 181.)

[Batory et al. 1998] Batory, D., Lofaso, B., & Smaragdakis, Y. (1998). JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse (ICSR'98)*, (pp. 143–153). IEEE Computer Society Press. (Cited on pages 6, 11, 39, 47, 48, 61, 67, 73, 85, 140, and 178.)

[Baxter et al. 2004] Baxter, I. D., Pidgeon, C., & Mehlich, M. (2004). DMS®: Program transformations for practical scalable software evolution. In proceedings ICSE 2004 [Estublier & Rosenblum 2004], (pp. 625–634). (Cited on pages 48 and 61.)

[BCEL Website] Byte code engineering library (BCEL). `http://jakarta.apache.org/bcel/`. (Cited on page 44.)

[Begel & Graham 2004] Begel, A. & Graham, S. L. (2004). Language analysis and tools for input stream ambiguities. In proceedings LDTA 2004 [Hedin & Wyk 2004], (pp. 75–96). (Cited on pages 11, 38, and 139.)

[Bierman et al. 2005] Bierman, G., Meijer, E., & Schulte, W. (2005). The essence of data access in Cω. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference*, volume 3586 of *LNCS*, (pp. 287–311). Springer. (Cited on page 88.)

[Blasband 2001] Blasband, D. (2001). Parsing in a hostile world. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, (pp. 291)., Washington, DC, USA. IEEE Computer Society. (Cited on page 139.)

[Bodden & Stolz] Bodden, E. & Stolz, V. J-LO, the Java Logical Observer. `http://www-i2.informatik.rwth-aachen.de/Research/RV/JLO/`. (Cited on page 94.)

[Borghi et al. 2006] Borghi, A., David, V., & Demaille, A. (2006). C-Transformers: a framework to write C program transformations. *Crossroads*, *12*(3), 3–3. (Cited on pages 184 and 197.)

[Bouma] Bouma, F. LLBLGen Pro, the n-tier generator and O/R mapper for .NET. `http://www.llblgen.com`. (Cited on page 13.)

[Bourret 2007] Bourret, R. (2007). XML data binding resources. `http://www.rpbourret.com/xml/XMLDataBinding.htm`. (Cited on page 12.)

[Bouwers et al. 2007] Bouwers, E., Bravenboer, M., & Visser, E. (2007). Grammar engineering support for precedence rule recovery and compatibility checking. In Johnstone, A. & Sloane, T. (Eds.), *Proceedings of the Seventh Workshop on Language Descriptions, Tools and Applications (LDTA'07)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers. (Cited on page 8.)

[Brabrand & Schwartzbach 2002] Brabrand, C. & Schwartzbach, M. I. (2002). Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'02)*, (pp. 31–40). ACM Press. (Cited on pages 11, 41, 42, and 47.)

[Brabrand et al. 2003] Brabrand, C., Schwartzbach, M. I., & Vanggaard, M. (2003). The Metafront system: Extensible parsing and transformation. In proceedings LDTA 2003 [Bryant & Saraiva 2003]. (Cited on page 42.)

[Bracha & Cook 1990] Bracha, G. & Cook, W. (1990). Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming systems, Languages, and Applications (OOPSLA/ECOOP '90)*, (pp. 303–311)., New York, NY, USA. ACM Press. (Cited on pages 124 and 152.)

[Bracha et al. 1998] Bracha, G., Odersky, M., Stoutamire, D., & Wadler, P. (1998). GJ specification. (Cited on page 106.)

[van den Brand et al. 2000] van den Brand, M. G. J., de Jong, H., Klint, P., & Olivier, P. (2000). Efficient annotated terms. *Software: Practice & Experience*, *30*(3), 259–291. (Cited on pages 13, 29, 32, and 39.)

[van den Brand et al. 2001] van den Brand, M. G. J., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J.,

Vinju, J., Visser, E., & Visser, J. (2001). The ASF+SDF Meta-Environment: a component-based language laboratory. In Wilhelm, R. (Ed.), *Compiler Construction, 10th International Conference, CC 2001*, volume 2027 of *LNCS*, (pp. 365–368). Springer. (Cited on pages 37 and 148.)

[van den Brand et al. 2003] van den Brand, M. G. J., Klusener, S., Moonen, L., & Vinju, J. (2003). Generalized parsing and term rewriting - semantics directed disambiguation. In proceedings LDTA 2003 [Bryant & Saraiva 2003]. (Cited on page 62.)

[van den Brand et al. 2005] van den Brand, M. G. J., Moreau, P., & Vinju, J. (2005). A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software*, *152*(2), 70–78. (Cited on page 52.)

[van den Brand & Perigot 2001] van den Brand, M. G. J. & Perigot, D. (Eds.). (2001). *Proceedings of LDTA'01, First Workshop on Language Descriptions, Tools and Applications*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers. (Cited on pages 216 and 220.)

[van den Brand & Ringeissen 2000] van den Brand, M. G. J. & Ringeissen, C. (2000). ASF+SDF parsing tools applied to ELAN. In *Third International Workshop on Rewriting Logic and Applications*, ENTCS. (Cited on page 60.)

[van den Brand et al. 2002] van den Brand, M. G. J., Scheerder, J., Vinju, J. J., & Visser, E. (2002). Disambiguation filters for scannerless generalized LR parsers. In Horspool, R. N. (Ed.), *Compiler Construction, 11th International Conference, CC 2002*, volume 2304 of *LNCS*, (pp. 143–158). Springer. (Cited on pages 12, 30, 31, 32, 37, 48, 60, 62, 95, 99, 120, 137, and 189.)

[Bravenboer et al. 2006] Bravenboer, M., de Groot, R., & Visser, E. (2006). MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In Lämmel, R. & Saraiva, J. (Eds.), *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, LNCS, Braga, Portugal. Springer. (Cited on page 6.)

[Bravenboer et al. 2007 (Chapter 4)] Bravenboer, M., Dolstra, E., & Visser, E. (2007). Preventing injection attacks with syntax embedding – a host and guest language independent approach. In proceedings GPCE 2007 [Lawall 2007]. (Cited on pages 6, 8, and 150.)

[Bravenboer et al. 2008] Bravenboer, M., Kalleberg, K. T., Vermaas, R., & Visser, E. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming. Special Issue on Experimental Software and Toolkits (EST)*. (To appear). (Cited on page 147.)

[Bravenboer et al. 2006 (Chapter 5)] Bravenboer, M., Tanter, E., & Visser, E. (2006). Declarative, formal, and extensible syntax definition for AspectJ – A case for scannerless generalized-LR parsing. In *Proceedings of the 21st ACM*

SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006), New York, NY, USA. ACM Press. (Cited on pages 6, 8, 73, 152, 153, 175, and 176.)

[Bravenboer et al. 2005 (Chapter 3)] Bravenboer, M., Vermaas, R., Vinju, J. J., & Visser, E. (2005). Generalized type-based disambiguation of meta programs with concrete object syntax. In proceedings GPCE 2005 [Glück & Lowry 2005], (pp. 157–172). (Cited on pages 6, 8, 67, 73, 86, 115, and 137.)

[Bravenboer & Visser 2004 (Chapter 2)] Bravenboer, M. & Visser, E. (2004). Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Schmidt, D. C. (Ed.), *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, (pp. 365–383). ACM Press. (Cited on pages 6, 8, 50, 61, 69, 73, 78, 90, 115, 137, 142, 146, 149, 152, 175, and 176.)

[Bryant & Saraiva 2003] Bryant, B. & Saraiva, J. (Eds.). (2003). *Proceedings of the Third Workshop on Language Descriptions, Tools and Applications (LDTA'03)*, volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers. (Cited on pages 213 and 214.)

[Buehrer et al. 2005] Buehrer, G. T., Weide, B. W., , & Sivilotti, P. A. (2005). Using parse tree validation to prevent SQL injection attacks. In *Fifth International Workshop on Software Engineering and Middleware (SEM 2005)*. ACM Press. (Cited on page 89.)

[Cardelli 1993] Cardelli, L. (1993). An implementation of $F_{<:}$. SRC Research Report 97, Digital Equipment Corporation, Systems Research Center. (Cited on page 37.)

[Cardelli et al. 1994] Cardelli, L., Matthes, F., & Abadi, M. (1994). Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, Palo Alto, California. (Cited on pages 11, 37, 146, and 181.)

[Castor Website] Castor databinding framework for Java. `http://www.castor.org`. (Cited on page 13.)

[Charles 1991] Charles, P. (1991). *A Practical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, New York Univeristy. (Cited on page 206.)

[Christensen et al. 2003] Christensen, A. S., Møller, A., & Schwartzbach, M. I. (2003). Precise analysis of string expressions. In *Static Analysis Symposium (SAS 2003)*, volume 2694 of *LNCS*, (pp. 1–18). Springer. (Cited on page 89.)

[Cocoon Website] The Apache Cocoon project. `http://cocoon.apache.org`. (Cited on page 15.)

[Cook & Rai 2005] Cook, W. R. & Rai, S. (2005). Safe query objects: Statically typed objects as remotely executable queries. In proceedings ICSE 2005 [Roman et al. 2005], (pp. 97–106). (Cited on pages 69 and 88.)

[Cordy et al. 1991] Cordy, J., Halpern-Hamu, C., & Promislow, E. (1991). TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, *16*(1), 97–107. (Cited on pages 48 and 61.)

[CPP Manual] The C preprocessor manual. `http://gcc.gnu.org/onlinedocs/cpp/`. (Cited on page 42.)

[de Jonge & Monajemi 2001] de Jonge, M. & Monajemi, R. (2001). Cost-effective maintenance tools for proprietary languages. In *Proceedings International Conference on Software Maintenance (ICSM 2001)*, (pp. 240–249)., Los Alamitos, CA, USA. IEEE Computer Society Press. (Cited on page 199.)

[de Jonge et al. 2001] de Jonge, M., Visser, E., & Visser, J. (2001). XT: A bundle of program transformation tools. In proceedings LDTA 2001 [van den Brand & Perigot 2001]. (Cited on pages 12 and 37.)

[de Rauglaudre 2003] de Rauglaudre, D. (2003). *Camlp4 Reference Manual*. (Cited on pages 62 and 181.)

[DeRemer 1971] DeRemer, F. (1971). Simple LR(k) grammars. *Communications of the ACM*, *14*(7), 453–460. (Cited on page 167.)

[DeRemer & Pennello 1979] DeRemer, F. & Pennello, T. J. (1979). Efficient computation of LALR(1) look-ahead sets. In *Proceedings of the 1979 SIGPLAN symposium on Compiler Construction (CC 1979)*, (pp. 176–187)., New York, NY, USA. ACM Press. (Cited on page 171.)

[van Deursen et al. 1996] van Deursen, A., Heering, J., & Klint, P. (Eds.). (1996). *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. Singapore: World Scientific. (Cited on pages 37, 48, and 60.)

[Earley 1970] Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, *13*(2), 94–102. (Cited on pages 34 and 180.)

[Ekman & Hedin 2004] Ekman, T. & Hedin, G. (2004). Rewritable reference attributed grammars. In Odersky, M. (Ed.), *Proceedings of 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *LNCS*, (pp. 144–169)., Oslo, Norway. Springer. (Cited on pages 146 and 152.)

[Estublier & Rosenblum 2004] Estublier, J. & Rosenblum, D. S. (Eds.). (2004). *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, Washington, DC, USA. IEEE Computer Society Press. (Cited on pages 212 and 217.)

[Eve & Kurki-Suonio 1966] Eve, J. & Kurki-Suonio, R. (1966). On computing the transitive closure of a relation. *Acta Informatica*, *8*(4), 303–314. (Cited on page 171.)

[Filman 2006] Filman, R. (Ed.). (2006). *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, New York, NY, USA. ACM Press. (Cited on pages 218 and 222.)

[Fischer & Visser 2004] Fischer, B. & Visser, E. (2004). Retrofitting the AutoBayes program synthesis system with concrete object syntax. In proceedings DSPG 2004 [Lengauer et al. 2004], (pp. 239–253). (Cited on page 37.)

[Freeman & Pryce 2006] Freeman, S. & Pryce, N. (2006). Evolving an embedded domain-specific language in Java. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, (pp. 855–865)., New York, NY, USA. ACM Press. (Cited on page 4.)

[Gagnon & Hendren 1998] Gagnon, E. M. & Hendren, L. J. (1998). SableCC, an object-oriented compiler framework. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS'98)*, (pp. 140–154). IEEE Computer Society. See also `http://www.sablecc.org`. (Cited on page 29.)

[Glück & Lowry 2005] Glück, R. & Lowry, M. (Eds.). (2005). *Generative Programming and Component Engineering: 4th International Conference (GPCE 2005)*, volume 3676 of *LNCS*, Tallinn, Estonia. Springer. (Cited on pages 215 and 224.)

[GNU M4 Website] GNU m4. `http://www.gnu.org/software/m4/`. (Cited on page 42.)

[Gould et al. 2004a] Gould, C., Su, Z., & Devanbu, P. (2004a). JDBC checker: A static analysis tool for SQL/JDBC applications. In proceedings ICSE 2004 [Estublier & Rosenblum 2004], (pp. 697–698). (Cited on pages 69 and 88.)

[Gould et al. 2004b] Gould, C., Su, Z., & Devanbu, P. (2004b). Static checking of dynamically generated queries in database applications. In proceedings ICSE 2004 [Estublier & Rosenblum 2004], (pp. 645–654). (Cited on pages 69 and 88.)

[Grimm 2006] Grimm, R. (2006). Better extensibility through modular syntax. In Cook, W. R. (Ed.), *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006)*. ACM Press. (Cited on page 178.)

[Grune & Jacobs 1990] Grune, D. & Jacobs, C. J. H. (1990). *Parsing Techniques - A Practical Guide*. Upper Saddle River, NJ, USA: Ellis Horwood. (Cited on page 157.)

[Halfond & Orso 2005] Halfond, W. G. & Orso, A. (2005). AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, (pp. 174–183)., Long Beach, California, USA. (Cited on pages 66, 69, and 89.)

[Halfond et al. 2006] Halfond, W. G., Orso, A., & Manolios, P. (2006). Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Foundations of Software Engineering (FSE 14)*. (Cited on pages 69 and 89.)

[Halfond et al. 2006] Halfond, W. G., Viegas, J., & Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering (ISSSE 2006)*. (Cited on pages 65 and 81.)

[Harbulot & Gurd 2006] Harbulot, B. & Gurd, J. (2006). A join point for loops in AspectJ. In proceedings AOSD 2006 [Filman 2006]. (Cited on page 94.)

[Hedin 2003] Hedin, G. (Ed.). (2003). *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of *LNCS*. Springer. (Cited on page 222.)

[Hedin & Wyk 2004] Hedin, G. & Wyk, E. V. (Eds.). (2004). *Proceedings of the Fourth Workshop on Language Descriptions, Tools and Applications (LDTA'04)*, volume 110 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers. (Cited on pages 212 and 219.)

[Heering et al. 1989] Heering, J., Hendriks, P. R. H., Klint, P., & Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, *24*(11), 43–75. (Cited on pages 37, 39, 137, 139, 185, 189, and 190.)

[Heering et al. 1990] Heering, J., Klint, P., & Rekers, J. (1990). Incremental generation of parsers. *IEEE Transactions on Software Engineering*, *16*(12), 1344–1351. (Cited on pages 146 and 180.)

[Hendren et al. 2004] Hendren, L., de Moor, O., Christensen, A. S., & the abc team (2004). The abc scanner and parser, including an LALR(1) grammar for AspectJ. Techrep, Programming Tools Group,Oxford University and the Sable research group, McGill University. (Cited on pages 94, 101, and 114.)

[Hopcroft et al. 2006] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley. (Cited on pages 72, 157, and 158.)

[Horspool 1990] Horspool, R. N. (1990). Incremental generation of LR parsers. *Computer Languages*, *15*(4), 205–223. (Cited on pages 146, 177, and 180.)

[Hosoya & Pierce 2000] Hosoya, H. & Pierce, B. C. (2000). XDuce: A typed XML processing language. In *The World Wide Web and Databases, Third International Workshop WebDB 2000, Selected Papers*, volume 1997 of *Lecture Notes in Computer Science*, (pp. 226–244). Springer. (Cited on pages 4 and 45.)

[Huang et al. 2004] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., & Kuo, S.-Y. (2004). Securing web application code by static analysis and runtime protection. In *13th International World Wide Web Conference (WWW2004)*, (pp. 40–52). ACM Press. (Cited on pages 66 and 88.)

[Hudak 1996] Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys*, *28*(4es). (Cited on page 11.)

[Hutton 1992] Hutton, G. (1992). Higher-order functions for parsing. *Journal of Functional Programming*, (2(3)), 323–343. (Cited on page 140.)

[ISO 1992] ISO (1992). *ISO/IEC 9075:1992: Database Language SQL*. (Cited on pages 4, 67, and 90.)

[JavaCC Website] Java Compiler Compiler (JavaCC). `https://javacc.dev.java.net/`. (Cited on pages 29 and 39.)

[JAXB Website] Java architecture for XML binding JAXB. `https://jaxb.dev.java.net`. (Cited on page 13.)

[JDT Website] Eclipse Java Development Tools (JDT) website. `http://www.eclipse.org/jdt/`. (Cited on pages 51, 59, and 102.)

[Jeffery 2003] Jeffery, C. L. (2003). Generating LR syntax error messages from examples. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *25*(5), 631–640. (Cited on pages 141 and 206.)

[Johnson 1975] Johnson, S. C. (1975). YACC—yet another compiler-compiler. Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, N.J. (Cited on pages 184, 185, and 188.)

[Johnstone & Scott 2002] Johnstone, A. & Scott, E. (2002). Generalised reduction modified LR parsing for domain specific language prototyping. In *35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, (pp. 282)., Washington, DC, USA. IEEE Computer Society Press. (Cited on page 157.)

[Johnstone et al. 2004] Johnstone, A., Scott, E., & Economopoulos, G. (2004). The grammar tool box: A case study comparing GLR parsing algorithms. In proceedings LDTA 2004 [Hedin & Wyk 2004], (pp. 97–113). (Cited on page 207.)

[Johnstone et al. 2006] Johnstone, A., Scott, E., & Economopoulos, G. (2006). Evaluating GLR parsing algorithms. *Science of Computer Programming*, *61*(3), 228–244. (Cited on page 169.)

[de Jong & Olivier 2004] de Jong, H. & Olivier, P. (2004). Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, *59*(1-2), 35–61. See also `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ApiGen`. (Cited on pages 13 and 29.)

[Kats 2007] Kats, L. (2007). Supporting language extension and separate compilation by mixing Java and bytecode. Master's thesis, Utrecht University, Utrecht, The Netherlands. (Cited on pages 25 and 44.)

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). An overview of AspectJ. In Lindskov Knudsen, J. (Ed.), *ECOOP 2001: Object-Oriented Programming: 15th European Conference*, volume 2072 of *LNCS*, (pp. 327–353). Springer. (Cited on page 94.)

[Kirby et al. 1998] Kirby, G., Morrison, R., & Stemple, D. (1998). Linguistic reflection in Java. *Software: Practice & Experience*, *28*(10), 1045–1077. (Cited on pages 43 and 44.)

[Klint et al. 2005] Klint, P., Lämmel, R., & Verhoef, C. (2005). Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, *14*(3), 331–380. (Cited on pages 184 and 199.)

[Klint & Visser 1994] Klint, P. & Visser, E. (1994). Using filters for the disambiguation of context-free grammars. In Pighizzini, G. & San Pietro, P. (Eds.), *Proceedings of ASMICS Workshop on Parsing Theory*, (pp. 1–20). Tech. Rep. 126, Università di Milano. (Cited on page 62.)

[Knuth 1965] Knuth, D. E. (1965). On the translation of languages from left to right. *Information and Control*, *8*(6), 607–639. (Cited on pages 154 and 192.)

[Kolbly 2002] Kolbly, D. M. (2002). *Extensible Language Implementation*. PhD thesis, University of Texas at Austin. (Cited on page 180.)

[Kort et al. 2002] Kort, J., Lämmel, R., & Verhoef, C. (2002). The grammar deployment kit – system demonstration. In van den Brand, M. G. J. & Lämmel, R. (Eds.), *Proceedings of LDTA 2002, Second Workshop on Language Descriptions, Tools and Applications*, volume 65 of *Electronic Notes in Theoretical Computer Science*, (pp. 117–123). Elsevier Science Publishers. (Cited on pages 184 and 199.)

[Kuipers & Visser 2001] Kuipers, T. & Visser, J. (2001). Object-oriented tree traversal with JJForester. In proceedings LDTA 2001 [van den Brand & Perigot 2001]. (Cited on page 29.)

[Lämmel 2001a] Lämmel, R. (2001a). Grammar Adaptation. In *Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, (pp. 550–570). Springer. (Cited on pages 184 and 199.)

[Lämmel 2001b] Lämmel, R. (2001b). Grammar testing. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001*, volume 2029 of *LNCS*, (pp. 201–216). Springer. (Cited on page 200.)

[Lämmel & Verhoef 2001] Lämmel, R. & Verhoef, C. (2001). Semi-automatic grammar recovery. *Software: Practice & Experience*, *31*(15), 1395–1438. (Cited on pages 184 and 199.)

[Lawall 2007] Lawall, J. (Ed.). (2007). *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, New York, NY, USA. ACM Press. (Cited on pages 211, 214, and 225.)

[Lea 2000] Lea, D. (2000). *Concurrent Programming in Java, Second Edition, Design Principles and Patterns*. Addison Wesley Longman Publishing Co., Inc. (Cited on page 44.)

[Leavenworth 1966] Leavenworth, B. M. (1966). Syntax macros and extended translation. *Communications of the ACM*, *9*(11), 790–793. (Cited on pages 11, 40, 41, and 61.)

[Lehman 1980] Lehman, M. M. (1980). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, *1*(3), 213–231. (Cited on page 93.)

[Leijen & Meijer 1999] Leijen, D. & Meijer, E. (1999). Domain specific embedded compilers. In *2nd USENIX Conference on Domain-Specific Languages (DSL)*. USENIX. (Cited on page 88.)

[Lengauer et al. 2004] Lengauer, C., Batory, D., Consel, C., & Odersky, M. (Eds.). (2004). *Domain-Specific Program Generation*, volume 3016 of *LNCS*, Dagstuhl Castle, Germany. Springer. (Cited on pages 217 and 225.)

[Leslie 1995] Leslie, T. (1995). Efficient approaches to subset construction. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada. (Cited on page 165.)

[Livshits & Lam 2005] Livshits, V. B. & Lam, M. S. (2005). Finding security vulnerabilities in Java applications with static analysis. In *14th USENIX Security Symposium*, (pp. 271–286). USENIX. (Cited on pages 66 and 88.)

[Maor & Shulman 2004] Maor, O. & Shulman, A. (2004). SQL injection signatures evasion. White paper, `http://www.imperva.com/`. (Cited on page 87.)

[Masuhara & Kawauchi 2003] Masuhara, H. & Kawauchi, K. (2003). Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *LNCS*, (pp. 105–121). Springer. (Cited on page 94.)

[McClure & Krüger 2005] McClure, R. A. & Krüger, I. H. (2005). SQL DOM: Compile time checking of dynamic SQL statements. In proceedings ICSE 2005 [Roman et al. 2005], (pp. 88–96). (Cited on pages 66, 69, and 87.)

[McPeak & Necula 2004] McPeak, S. & Necula, G. C. (2004). Elkhound: A fast, practical GLR parser generator. In Duesterwald, E. (Ed.), *Proceedings of 13th International Conference on Compiler Construction (CC'04)*, volume 2985 of *LNCS*, (pp. 73–88)., Berlin. Springer. (Cited on pages 141 and 206.)

[Meijer & Schulte 2003a] Meijer, E. & Schulte, W. (2003a). Programming with rectangles, triangles, and circles. In *Proceedings of XML Conference & Exposition 2003 (XML 2003)*. IDEAlliance. (Cited on pages 4 and 45.)

[Meijer & Schulte 2003b] Meijer, E. & Schulte, W. (2003b). Unifying tables, objects, and documents. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL'03)*. (Cited on pages 4 and 45.)

[Meijer & van Velzen 2001] Meijer, E. & van Velzen, D. (2001). Haskell Server Pages: Functional programming and the battle for the middle tier. In *2000 ACM SIGPLAN Haskell Workshop*, volume 41/1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers. (Cited on page 88.)

[Mernik et al. 2005] Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, *37*(4), 316–344. (Cited on pages 10 and 11.)

[Møller 2005] Møller, A. (2005). dk.brics.automaton — finite-state automata for Java. `http://www.brics.dk/automaton/`. (Cited on page 75.)

[Moreau et al. 2003] Moreau, P.-E., Ringeissen, C., & Vittek, M. (2003). A pattern matching compiler for multiple target languages. In proceedings CC 2003 [Hedin 2003], (pp. 61–76). (Cited on page 40.)

[van Noord 2000] van Noord, G. (2000). Treatment of epsilon moves in subset construction. *Computational Linguistics*, *26*(1), 61–76. (Cited on page 165.)

[Nuutila 1995] Nuutila, E. (1995). *Efficient Transitive Closure Computation in Large Digraphs*. PhD thesis, Helsinki University of Technology. (Cited on page 174.)

[Nystrom et al. 2003] Nystrom, N., Clarkson, M. R., & Myers, A. C. (2003). Polyglot: An extensible compiler framework for Java. In proceedings CC 2003 [Hedin 2003], (pp. 138–152). (Cited on pages 108, 128, 146, 152, and 178.)

[Odersky & Zenger 2005] Odersky, M. & Zenger, M. (2005). Independently extensible solutions to the expression problem. In *Proceedings of the Twelth International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*. (Cited on page 146.)

[Ongkingco et al. 2006] Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O., & Sittampalam, G. (2006). Adding open modules to AspectJ. In proceedings AOSD 2006 [Filman 2006]. (Cited on pages 94 and 130.)

[Onzon 2007] Onzon, E. (2007). Dypgen: Self-extensible parsers for ocaml. `http://dypgen.free.fr`. (Cited on pages 38 and 180.)

[Parr] Parr, T. ANTLR Parser Generator. `http://www.antlr.org`. (Cited on page 136.)

[Rekers 1992] Rekers, J. (1992). *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam. (Cited on pages 31, 34, 95, 136, 156, 180, 206, and 207.)

[Roman et al. 2005] Roman, G.-C., Griswold, W. G., & Nuseibeh, B. (Eds.). (2005). *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, Washington, DC, USA. IEEE Computer Society Press. (Cited on pages 215 and 221.)

[Sakurai et al. 2004] Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., & Komiya, S. (2004). Association aspects. In Lieberherr, K. (Ed.), *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, (pp. 16–25)., Lancaster, UK. ACM Press. (Cited on page 94.)

[Salomon & Cormack 1989] Salomon, D. J. & Cormack, G. V. (1989). Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI 1989)*, (pp. 170–178). ACM Press. (Cited on pages 31, 36, 95, 136, and 175.)

[Salomon & Cormack 1995] Salomon, D. J. & Cormack, G. V. (1995). The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canada. (Cited on pages 95 and 98.)

[Schatborn 2005] Schatborn, E. P. (2005). GTL, a grammar transformation language for SDF specifications. Master's thesis, Programming Research Group, Faculty of Science, University of Amsterdam, Amsterdam. (Cited on page 200.)

[Scott & Johnstone 2006] Scott, E. & Johnstone, A. (2006). Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *28*(4), 577–618. (Cited on pages 141 and 206.)

[SDF Website] http://www.syntax-definition.org. (Cited on pages 12, 31, and 36.)

[Sellink & Verhoef 2000] Sellink, M. & Verhoef, C. (2000). Development, assessment, and reengineering of language descriptions. In Ebert, J. & Verhoef, C. (Eds.), *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, (pp. 151–160). IEEE Computer Society Press. (Cited on pages 184 and 199.)

[Shalit 1996] Shalit, A. (1996). *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc. (Cited on pages 11 and 41.)

[Smaragdakis & Batory 2000] Smaragdakis, Y. & Batory, D. (2000). Application generators. *Encyclopedia of Electrical and Electronics Engineering*. J.G. Webster (Ed.), John Wiley and Sons. (Cited on page 11.)

[Stratego Website] http://www.stratego-language.org. (Cited on pages 12, 20, 37, and 209.)

[Su & Wassermann 2006] Su, Z. & Wassermann, G. (2006). The essence of command injection attacks in web applications. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (pp. 372–382)., New York, NY, USA. ACM Press. (Cited on pages 66, 70, and 83.)

[Szyperski 1996] Szyperski, C. (1996). Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia. (Cited on page 146.)

[Tanter et al. 2006] Tanter, É., Gybels, K., Denker, M., & Bergel, A. (2006). Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, LNCS, (pp. 227–249). Springer. (Cited on pages 94, 131, and 132.)

[Tanter & Noyé 2005] Tanter, É. & Noyé, J. (2005). A versatile kernel for multi-language AOP. In proceedings GPCE 2005 [Glück & Lowry 2005], (pp. 173–188). (Cited on pages 131 and 142.)

[Tao et al.] Tao, K., Wang, W., & Palsberg, J. Java Tree Builder (JTB). `http://compilers.cs.ucla.edu/jtb/`. (Cited on pages 13 and 29.)

[Tarjan 1972] Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*(2), 146–160. (Cited on page 171.)

[Tomita 1985] Tomita, M. (1985). *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers. (Cited on pages 31, 34, 95, and 156.)

[Tratt 2005] Tratt, L. (2005). The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London. (Cited on page 180.)

[Valkering 2007] Valkering, R. (2007). Syntax error handling in scannerless generalized LR parsers. Master's thesis, Programming Research Group, University of Amsterdam, Amsterdam, The Netherlands. (Cited on page 206.)

[Vinju 2005] Vinju, J. (2005). A type driven approach to concrete meta programming. Technical Report SEN-E0507, Centrum voor Wiskunde en Informatica. (Cited on pages 54, 60, and 61.)

[Visser 1997a] Visser, E. (1997a). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam. (Cited on pages 30, 31, 36, 95, 99, 116, 121, and 137.)

[Visser 1997b] Visser, E. (1997b). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam. (Cited on pages 12, 30, 31, 37, 48, 60, 69, 71, 95, 99, 137, 148, 176, 178, 184, 185, 189, 190, and 195.)

[Visser 2002] Visser, E. (2002). Meta-programming with concrete object syntax. In Batory, D., Consel, C., & Taha, W. (Eds.), *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, volume 2487 of *LNCS*, (pp. 299–315). Springer. (Cited on pages v, 6, 7, 21, 37, 48, 50, 60, 61, 67, 73, 85, and 147.)

[Visser 2004] Visser, E. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In proceedings DSPG 2004 [Lengauer et al. 2004], (pp. 216–238). (Cited on pages 12, 20, 37, 60, 69, 71, and 185.)

[Visser et al. 1998] Visser, E., Benaissa, Z.-e.-A., & Tolmach, A. (1998). Building program optimizers with rewriting strategies. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, (pp. 13–26). ACM Press. (Cited on page 37.)

[Visser 2001] Visser, J. (2001). Visitor combination and traversal control. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, (pp. 270–282). ACM Press. (Cited on page 44.)

[Weise & Crew 1993] Weise, D. & Crew, R. (1993). Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI'93)*, (pp. 156–165). ACM Press. (Cited on pages 11 and 41.)

[van Wyk et al. 2007] van Wyk, E., Krishnan, L., Schwerdfeger, A., & Bodin, D. (2007). Attribute grammar-based language extensions for Java. In *European Conference on Object Oriented Programming (ECOOP 2007)*, LNCS. Springer. (Cited on pages 146, 149, and 152.)

[van Wyk & Schwerdfeger 2007] van Wyk, E. & Schwerdfeger, A. (2007). Context-aware scanning for parsing extensible languages. In proceedings GPCE 2007 [Lawall 2007]. (Cited on page 178.)

[Xie & Aiken 2006] Xie, Y. & Aiken, A. (2006). Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium*, (pp. 179–192). USENIX. (Cited on pages 66 and 88.)

[Xtatic] The Xtatic project: Native XML processing for C#. `http://www.cis.upenn.edu/~bcpierce/xtatic/`. (Cited on pages 4 and 45.)

[Zook et al. 2004] Zook, D., Huang, S. S., & Smaragdakis, Y. (2004). Generating AspectJ programs with Meta-AspectJ. In Karsai, G. & Visser, E. (Eds.), *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, volume 3286 of *LNCS*, (pp. 1–19)., Vancouver, Canada. Springer. (Cited on pages 4, 7, 47, 48, 62, 67, 73, 86, 137, and 147.)

Samenvatting

HOOFDSTUK 2: CONCRETE SYNTAX VOOR OBJECTEN

Interfaces voor applicatieprogrammeurs (APIs) bieden toegang tot domeinkennis die geëncapsuleerd is in klassebibliotheken. Ze bieden echter vaak geen optimale notatie om programma's samen te stellen voor dit domein. Omdat object-georiënteerde talen zijn ontworpen voor uitbreidbaarheid en hergebruik, zijn de taalconstructies vaak voldoende voor het uitdrukken van domeinabstracties op een semantisch niveau. Op een syntactisch niveau bieden deze talen desalniettemin niet de juiste abstractie. In dit hoofdstuk beschrijven we MetaBorg, een methode voor het bieden van *concrete syntax* (*syntaxis*) voor domeinabstracties aan applicatieprogrammeurs. De methode bestaat uit het *embedden* van een domein-specifieke taal in een algemene programmeertaal en het *assimileren* van de geëmbedde code in de omringende hostcode. In plaats van het uitbreiden van de implementatie van de hosttaal implementeert de assimilatiefase de domeinabstracties in termen van bestaande APIs, waardoor de hosttaal verder ongemoeid wordt gelaten. MetaBorg kan zo gezien worden als een methode om APIs naar het niveau van de taal te promoveren. De methode wordt ondersteund door bestaande en bewezen technologie, namelijk het grammaticaformalisme SDF en de programmatransformatietaal en tools van Stratego/XT. We illustreren de methode met toepassingen in drie verschillende domeinen: code generatie, XML generatie, en het implementeren van gebruikersinterfaces.

HOOFDSTUK 3: TYPE-GEBASEERDE DISAMBIGUATIE

Bij metaprogrammeren met concrete objectsyntax worden programma's van de objecttaal samengesteld uit fragmenten die geschreven zijn in de concrete syntax van de objecttaal. Het gebruik van kleine programmafragmenten in zulke quotes en het gebruik van metaexpressies in deze fragmenten (antiquotes) leidt vaak tot syntactische ambiguïteiten. Dit probleem wordt meestal opgelost door de programmeur de fragmenten expliciet te laten disambigueren, wat zorgt voor aanzienlijke syntactische overhead. Enkele systemen vermijden dit door typeinformatie te gebruiken tijdens het parseren (*ontleden*). Doordat dit lastig te realiseren is met traditionele parseertechnieken zijn deze systemen specifiek voor een combinatie van één meta- en één objecttaal. Ook zijn de implementaties van deze systemen niet herbruikbaar.

In dit hoofdstuk generaliseren we deze aanpak en presenteren een *taalonafhankelijke* methode voor het introduceren van concrete objectsyntax zonder expliciete disambiguatie te vereisen. De methode maakt gebruik van het *generalized LR* algoritme voor het parseren van metaprogramma's met concrete objectsyntax. Dit resulteert in een woud van alle mogelijke manieren waarop

het programma te parseren is. Dit woud wordt gereduceerd tot een enkele boom door een disambiguerende typechecker voor de metataal. Ter validatie hebben we diverse objecttalen opgenomen in Java, waaronder AspectJ en Java zelf.

HOOFDSTUK 4: PREVENTIE VAN INJECTIEAANVALLEN

Software geschreven in een bepaalde programmeertaal moet vaak zinnen construeren in een andere programmeertaal, bijvoorbeeld SQL-zoekacties, XML-documenten, of Shellcommando's. Dit wordt vrijwel altijd gedaan met onhygiënische stringmanipulatie, waarbij constante strings en gebruikersinvoer samengevoegd worden. Een gebruiker kan in deze situatie een kwaadaardige string invoeren die ervoor zorgt dat de uiteindelijk gegenereerde zin geïnterpreteerd wordt op een door de programmeur onbedoelde wijze. Dit is een injectieaanval.

Wij presenteren in dit hoofdstuk een meer natuurlijke stijl van programmeren die code oplevert die ongevoelig is voor injectieaanvallen. Onze aanpak garandeert door de manier waarop zinnen geconstrueerd worden dat de uiteindelijk gegenereerde zin altijd overeenkomt met de zinnen die de programmeur bedoelde te genereren. Voor onze aanpak embedden we een gasttaal (bijvoorbeeld SQL) in een hosttaal (bijvoorbeeld Java) en genereren uit de geëmbedde fragmenten van de gasttaal automatisch code die deze omzet naar code in de hosttaal die deze zinnen veilig construeert. Waar nodig worden automatisch functies aangeroepen die karakters met een voor de hosttaal speciale betekenis omzetten. Onze methode is generiek doordat het relatief eenvoudig toegepast kan worden op willekeurige combinaties van host- en gasttalen.

HOOFDSTUK 5: GRAMMATICA VAN ASPECTJ

Aspect-georiënteerd programmeren (AOP) geniet belangstelling vanuit zowel de academische wereld als de industrie. Dit wordt geïllustreerd door de alsmaar groeiende populariteit van AspectJ, de standaard AOP uitbreiding van Java. Vanuit het perspectief van compilerbouw is AspectJ interessant omdat het een typisch voorbeeld is van een taalagglomeraat, een taal die bestaat uit een aantal subtalen met ieder een verschillende syntax. Naast Java bevat AspectJ namelijk ook talen voor het definiëren van patronen, *pointcuts*, en *advice*. Dergelijke samenstelling van talen is een uitdaging voor conventionele parseertechnieken. Het samenvoegen van twee of meer talen met een verschillende lexicale syntax zorgt voor nogal wat complexiteit in de lexicale toestanden waarmee scanners vaak werken. Ook is er nog steeds actief onderzoek naar nieuwe taaleigenschappen voor AOP. Dergelijke voorstellen zijn vaak uitbreidingen van AspectJ, waardoor er behoefte is aan een uitbreidbare grammatica van AspectJ.

In dit hoofdstuk laten we zien hoe parseren zonder een scanner (*scannerless parsing*) elegant deze problemen met het gebruik van conventionele par-

seertechnieken voor AspectJ oplost. We presenteren het ontwerp van een modulaire, uitbreidbare, en formele definitie van zowel de lexicale als de context-vrije syntax van AspectJ in het grammaticaformalisme SDF. Parsers die gegenereerd worden uit SDF grammatica's maken gebruik van het scannerless generalized LR algoritme. Verder introduceren we *grammar mixins*, een nieuwe toepassing van het SDF modulesysteem. Grammar mixins maken het mogelijk om de verschillende keyword policies (*beleid* voor *gereserveerde woorden*) van AspectJ-compilers declaratief te beschrijven. We illustreren de modulaire uitbreidbaarheid van onze definitie met syntactische uitbreidingen die voorgesteld zijn in recent onderzoek naar aspecttalen. Tot slot laten benchmarks zien dat de snelheid van de scannerless generalized LR parser acceptabel is voor deze grammatica.

HOOFDSTUK 6: SAMENSTELLEN VAN PARSEERTABELLEN

Modulesystemen, gescheiden compilatie, het afleveren van binaire componenten, en dynamisch linken zijn algemeen geaccepteerd in programmeertalen en systemen. De syntax van een taal is daarentegen meestal niet modulair gedefiniëerd, kan niet gescheiden gecompileerd worden, kan niet eenvoudig gecombineerd worden met de syntax van andere talen, en kan niet afgeleverd worden als een component die later gecombineerd wordt met andere componenten. Grammaticaformalismen die wel modules ondersteunen compileren uiteindelijk toch alle modules gezamenlijk.

De huidige uitbreidbare compilers zijn ontworpen om uitgebreid te worden op het niveau van hun broncode. De gebruiker moet hierdoor de *compiler* compileren voor elke specifieke configuratie van uitbreidingen. Voor elke combinatie moet hiervoor ook een samengestelde parser gegenereerd worden. Het genereren van een parser is echter duur, wat vooral een probleem is wanneer de combinaties niet vastliggen en de gebruiker zelf taaluitbreidingen kan kiezen.

In dit hoofdstuk introduceren we een algoritme voor het samenstellen van parseertabellen. Dit algoritme ondersteunt het gescheiden compileren van grammatica's naar *parseertabelcomponenten*. Parseertabelcomponenten kunnen efficiënt samengesteld (gelinkt) worden juist voordat de tabellen gebruikt worden voor het parseren. De complexiteit van het samenstellen van parseertabellen is in het slechtste geval exponentieel (zoals ook de complexiteit van parseertabelgeneratie), maar voor realistische scenario's van het combineren van talen is ons algoritme substantieel sneller dan het berekenen van een parseertabel uit de gecombineerde grammatica's.

HOOFDSTUK 7: AFLEIDEN VAN GROEPERINGSREGELS

Er zijn veel verschillende parsergeneratoren in gebruik. De grammaticaformalismen van deze parsergeneratoren bieden verschillende methoden voor het definiëren van groeperingsregels (*precedence rules*). Sommige generatoren (zoals bijvoorbeeld YACC) ondersteunen declaratie van groeperingsregels, ter-

wijl andere generatoren vereisen dat groeperingsregels afgedwongen worden door de productieregels van de grammatica. Zelfs wanneer een grammaticaformalisme declaraties voor groeperingsregels ondersteunt, zou een specifieke grammatica deze taalconstructie kunnen negeren en de groeperingsregels toch in de productieregels verwerken.

Het resultaat is een verzameling varianten van grammatica's die allemaal dezelfde taal definiëren. Voor de taal C gebruikt de GNU Compiler de parsergenerator YACC met groeperingsregeldeclaraties, het C-Transformers project gebruikt SDF zonder prioriteiten, terwijl de C-grammatica van de SDF bibliotheek deze prioriteiten wel gebruikt. Voor PHP gebruikt Zend YACC met groeperingsregeldeclaraties, terwijl het PHP-front pakket SDF met prioriteiten associativiteitdeclaraties gebruikt. Deze variatie in grammatica's roept de vraag op of de groeperingsregels van de ene grammatica wel overeenkomen met die van de andere. In het algemeen is dit vaak niet direct duidelijk, omdat sommige talen zeer complexe groeperingsregels hebben. Voor sommige parsergeneratoren is de semantiek van groeperingsregeldeclaraties ook louter operationeel gedefiniëerd, waardoor het lastig is om te redeneren over het effect van deze declaraties op de gedefiniëerde taal.

In dit hoofdstuk presenteren we een methode en een tool voor het vergelijken van groeperingsregels van verschillende grammatica's en parsergeneratoren. Alhoewel de vraag of twee grammatica's dezelfde taal definiëren onbeslisbaar is, ondersteunt onze methode het vergelijken en afleiden van groeperingsregels. Dit is in het bijzonder nuttig voor het betrouwbaar migreren van een grammatica naar een ander grammaticaformalisme. We evalueren onze methode door deze toe te passen op enkele niet-triviale programmeertalen, namelijk PHP en C.

Curriculum Vitae

PERSONAL DATA

Martin Bravenboer
Place of Birth: Epe, The Netherlands
Date of Birth: February 11, 1979

EDUCATION

M.Sc. in Computer Science (1997-2004)
Utrecht University
Department of Information and Computing Science
Foundation year examination diploma June 26, 1998 (cum laude)
Final examination diploma February 7, 2004 (cum laude)

Pre-university Education (1991-1997)
Johannes Fontanus College, Barneveld
Final examination diploma June 11, 1997

EMPLOYMENT

Postdoctoral Researcher (November 2007 - February 2008)
Delft University of Technology
Department of Software Technology

Research Assistant (February 2007 - October 2007)
Delft University of Technology
Department of Software Technology

Research Assistant (November 2003 - January 2007)
Utrecht University
Department of Information and Computing Sciences

Teaching Assistant (January 2003 - March 2003)
Utrecht University
Department of Information and Computing Sciences

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in µCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

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