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.