Verifiable Composition of Deterministic Grammars

August C. Schwerdfeger Eric R. Van Wyk

Department of Computer Science and Engineering, University of Minnesota
Minneapolis, Minnesota

schwerdf@cs.umn.edu, evw@cs.umn.edu

Abstract

There is an increasing interest in extensible languages, (domain-specific) language extensions, and mechanisms for their specification and implementation. One challenge is to develop tools that allow non-expert programmers to add an eclectic set of language extensions to a host language. We describe mechanisms for composing and analyzing concrete syntax specifications of a host language and extensions to it. These specifications consist of context-free grammars with each terminal symbol mapped to a regular expression, from which a slightly-modified LR parser and context-aware scanner are generated. Traditionally, conflicts are detected when a parser is generated from the composed grammar, but this comes too late since it is the non-expert programmer directing the composition of independently developed extensions with the host language.

The primary contribution of this paper is a modular analysis that is performed independently by each extension designer on her extension (composed alone with the host language). If each extension passes this modular analysis, then the language composed later by the programmer will compile with no conflicts or lexical ambiguities. Thus, extension writers can verify that their extension will safely compose with others and, if not, fix the specification so that it will. This is possible due to the context-aware scanner's lexical disambiguation and a set of reasonable restrictions limiting the constructs that can be introduced by an extension. The restrictions ensure that the parse table states can be partitioned so that each state can be attributed to the host language or a single extension.

Categories and Subject Descriptors D3.4 [*Processors*]: Parsing, Compiler generators; F4.2 [*Grammars and Other Rewriting Systems*]: Parsing; F4.3 [*Formal Languages*]: Classes defined by grammars; D3.2 [*Language Classifications*]: Extensible Languages

General Terms Languages, Algorithms, Verification

Keywords LR Parsing, context-aware scanning, language composition, grammar composition, extensible languages

1. Introduction.

1.1 Motivation.

There is a rising amount of interest in the related areas of domain-specific languages, extensible languages, and in the tools and techniques used to specify and implement them. Of particular interest are systems that allow new syntax or semantic analysis to be added modularly to an extensible language framework. Ideally, it is done in a way that allows a non-expert programmer to extend his or her language with several eclectic extensions addressing the different aspects of the programming problem at hand.

Consider the simple program in Figure 1. It is written in a version of Java to which two extensions have been added (25). The first adds the `using ... query ...` and `connection ...` constructs to extend Java with the database query language SQL. Such an extension statically detects syntax and type errors in the query and also adds a `foreach` construct, which iterates over results from the query and extracts values from each query result. The import-like `connection` construct sets up the connection to the database and retrieves database type schemas to type-check the query. The second extension adds a construct for representing boolean conditions in a tabular form, inspired by similar constructs in modeling languages such as SCR (13). It consists of a keyword `table` followed by a list of rows, each consisting of a Java expression followed by a colon and several truth-indicators (`T`, `F`, `*`) indicating if the expression is expected to be true, or false, or if it does not matter. In this case, the table evaluates to true if `a > 18` is true and `z == 10001` is false, or if `a > 18` is false (the value of `z == 10001` does not matter). This extension checks that the expression in each row is of type `boolean` and that each row has the same number of truth-indicators. To support these types of extensions, language extension frameworks and tools must allow *new concrete syntax* to be added to the language as well as *new semantic analysis* to type-check the extension constructs.

Ideally, these extensions can be developed by separate parties, unaware of each other's extensions, and the non-expert programmer is provided with some mechanism to compose the host language (Java in this case) automatically with the language extensions. Although the development of such language extensions may require some knowledge of programming language implementation tools and techniques, their use and composition should not.

```
connection db_c with table person ;
class Demo {
 boolean m ( ) {
   rs = using db_c query { SELECT age, zip
                           FROM person
                           WHERE state = "NY" } ;
   foreach (int a, int z) in rs {
      res = res && table ( a > 18     : T F
                           z == 10001 : F * ) }
   return res ;
 }
}
```

Figure 1. Code written in an extended version of Java.

PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06...$5.00

A number of language processing tools and extensible language frameworks have been proposed. Polyglot (18) is a collection of Java classes that implement the core front-end functions of a Java 1.4 compiler; one can add new classes and extend the existing ones to add new language constructs and analyses. Others have investigated the use of attribute grammars for building extensible specifications of languages. The JastAdd extensible Java compiler (9) is implemented in the JastAdd attribute grammar system (10). To extend this compiler, one writes new attribute grammar fragments that the system combines to create a attribute grammar specification for the extended language. From this specification a compiler for the extended language is automatically generated. We have developed ableJ (25) a similar system based on our Silver (24) attribute grammar system. Xoc (8) is an extensible language framework for C, also inspired by attribute grammars.

These systems use standard LALR(1) parser and scanner generator technology such as Yacc and Lex (though they can be easily adapted to use other types of parser and scanner generators). These are notoriously brittle under composition and extension; merely adding or modifying a production can remove the grammar from the desired LALR(1) class.

In the MetaBorg (5) system, where semantic processing is done by term rewriting in Stratego (29), the concrete syntax is instead implemented using a scannerless GLR parser generator (28). Since GLR parsers can parse any context free grammar the composition of the host language grammar and language extension grammars can always be parsed. But the composed grammar may contain ambiguities; the programmer has no assurance that the parser created for their composed language will be deterministic and not, on occasion, return more than one parse tree. We seek an approach that guarantees the determinism of the composed grammar.

Parsing Expression Grammars (11) are closed under composition and thus satisfy this requirement and an extensible specification of C based on PEGs has been constructed (12). The determinism comes at a cost, however; order matters in the composition of PEGs. The concrete syntax tree returned by the parser may be different if the language extension grammars are composed in different orders. Productions that have the same nonterminal on the left hand side are disambiguated by the order in which they appear in the specification. Determining the proper order of applying language extensions is the sort of implementation-level knowledge that we do not want to require of the programmer. Furthermore, the problem of determining if altering production order alters the recognized language is undecidable (11).

1.2 Summary of results.

In this paper, we consider the case of extending an existing host language H with some (unordered) set of language extensions $\{E_1, E_2, ..., E_n\}$. We have previously studied the issues of semantics (25) and believe that they are critical, but our concern here is only with their concrete syntax. Our goal is to generate a parser and scanner for the language $H \cup \{E_1, E_2, ..., E_n\}$. These specifications consist of a context free grammar that specifies the parser and an association that maps each terminal in the grammar to a regular expression, to specify the scanner. The specification of a language L and an extension E are denoted, respectively, as Γ^L and Γ^E. Grammars and the grammar composition operator ($\cup^*_G$) are defined formally in Section 2 but $\cup^*_G$ is just the component-wise unions of the sets of terminals, nonterminals, productions, and terminal/regular-expression mappings defined in the grammars.

We would like the resulting parser to be deterministic; our parsing approach uses slightly modified LALR(1) parsing and parse-table generation techniques, ensuring no conflicts (shift-reduce or reduce-reduce) in the generated LR parse table. We write $\mathit{conflictFree}(\Gamma^L)$ to indicate that the parse table generated from grammar Γ^L contains no conflicts and thus can be parsed with a deterministic LALR(1) parser. Also, we would like the scanner generated from Γ^L to be ambiguity free, *i.e.,* on any input the scanner will return exactly one token or, in the case of a lexical error, zero tokens. We indicate this as $\mathit{lexAmbigFree}(\Gamma^L)$. A language specification is deterministic, indicated by $\mathit{det}(\Gamma^L)$, iff $\mathit{conflictFree}(\Gamma^L) \wedge \mathit{lexAmbigFree}(\Gamma^L)$.

Traditionally these analyses are performed on the complete language specification, *i.e.,* after the extensions have been added to the host language. In the approach to extensible languages outlined above, programmers can plug extensions from independent sources into their host language. Thus, an error message reporting a shift-reduce conflict in the generated parser or a lexical ambiguity in the generated scanner comes too late, as the programmer may not be able to fix it; this is properly the job of the extension designer.

Verifiable composition of grammars. In this paper we introduce an analysis that can verify that the grammar for the composed language $\Gamma^C = \Gamma^H \cup^*_G \{\Gamma^{E_1}, \Gamma^{E_2}, ..., \Gamma^{E_n}\}$ will be deterministic ($\mathit{det}(\Gamma^H)$) if each extension grammar Γ^{E_i} individually passes a modular analysis. This modular analysis, denoted $\mathit{det}_m(\Gamma^H, \Gamma^{E_i})$, can be described by the implication

$$(\forall i \in [1, n], \mathit{det}_m(\Gamma^H, \Gamma^{E_i})) \implies \mathit{det}(\Gamma^H \cup^*_G \{\Gamma^{E_1}, ..., \Gamma^{E_n}\}).$$

This states that if each extension grammar Γ^{E_i} passes the modular determinism test (with respect to the host grammar Γ^H) then the composition of Γ^H and all of the extensions is deterministic. The implication of this is that extension writers can "*certify*" their extensions as composable without needing to test against other language extensions. (This is not that much unlike library writers "certifying" libraries by compiling them to ensure that they contain no type errors.)

As will be seen, the modular test $\mathit{det}_m(\Gamma^H, \Gamma^E)$ does put some restrictions on the type of constructs that an extension E can add to H. For example, language extensions cannot specify syntax such that new terminals, except for marking terminals (defined below), are added to the follow-sets of host language nonterminals. In our experience, these are reasonable for many language extensions. But there are some that do not fit, such as the addition of new infix binary operators to the host language H, though these can be added as part of an embedded language as is done in the SQL extension. We will limit productions added by some extension E with a host language nonterminal on the left hand side to a single production $h \rightarrow \mu_E\ s_E$, where μ_E is a *marking terminal* and s_E is the extension's "start" nonterminal. The effect of such a production is that it causes the parser, upon shifting μ_E, to enter a parse state that is the domain of the language extension. Extensions typically add several productions whose left hand side is an extension-introduced nonterminal and the few restrictions placed on them are much less severe. Critically, the right hand sides off these extension productions *can contain terminals and nonterminals defined in the host language grammar* Γ^H.

LALR(1)'s "brittleness" and context-aware scanners. It may seem unlikely that this guarantee can be achieved, given that LALR(1) parsers can be rather brittle under composition and extension. Adding or modifying a production can easily remove the grammar from the desired LALR(1) class. A large part of this brittleness can be mitigated by the use of a context-aware scanner (21; 7; 27), which takes context into account when scanning. In this paper we have extended our notion of *parse-state-based context-aware scanning* as implemented in *Copper*, our LALR(1) parser and context-aware scanner generator (27). Each time the scanner is called by the parser it is passed the set of *valid lookahead*: terminals that can be accepted by the parser at that point in the parse. These terminals are those whose action in the LR parse table for the current state are *shift*, *reduce*, or *accept*, but not *error*. The scanner will then scan the input and only return a token that is in this set of valid lookahead. This allows terminals to have overlapping regular expressions as long as they appear in different parse-state contexts: it is the parse state context that disambiguates them. It does require the LR parsing algorithm to be slightly modified to pass parse-state information to the scanner when it is called to return the next token (27).

Consider scanning the type expression `List<List<T>>` in a language that includes Java-like parameterized types and a right bit shift operator `>>`. If the `>>` operator is not valid in parse states where the closing bracket of a type expression is valid, then `>` will be in the valid lookahead set but `>>` will not. Thus the scanner will not return `>>` when parsing types and the grammar can be simplified. This is useful when extending languages, as one extension may introduce new terminals whose regular expressions overlap with those in other extensions but occur in different contexts.

Note that the restrictions we place on extension grammars would be unreasonable in a traditional scanning approach. With context-aware scanning, they are much more reasonable because extension writers can create their own terminal symbols that may have overlapping regular expressions (with terminals introduced by other extensions) but this often does not cause conflicts in the composed language. For example, the `table` terminal introduced by the SQL extension will not be in the same context as Java keywords or identifiers and thus the lexeme `table` can be used as an identifier or a token for some other language extension, as it is in the table expression extension.

An appealing aspect of this approach is that we do not need to develop new parsing and scanning techniques from scratch to be able to certify language extensions as composable. The LR parsing technology used here is only slightly modified from the established traditional approach. The scanner, by making it aware of its context, can be more discriminating in the tokens that it returns and thus language designers do not need to re-use the same token in many different contexts. Different terminals with the same regular expression can be used. This has a rather dramatic effect on the parser — the additional tokens simplify the grammar and make it much less brittle and make practical the modular, conflict-free composability analysis that is proposed here.

Paper Outline: Section 2 provides the formal specifications of grammars used in this analysis and background material on LR parsing and context aware scanners. Section 3 describes the primary contribution of this paper: a modular analysis of language extension specifications that ensures that when a collection of extensions are added to a host language, if each one individually passes the modular analysis, then the composed language has no parse-table shift-reduce or reduce-reduce conflicts. This analysis partitions LR DFA of the extended language into sets of states that are either the purview of the host language or of an individual language extension. This section also provides a discussion of the algorithm's correctness. Section 4 describes the lexical ambiguity analysis and how these techniques can be applied in the presence of "practical" specifications such as operator and lexical precedence. Section 5 discusses related work and briefly explains how this partitioning of the LR DFA can allow extension grammars to be separately compiled down to parse tables that are composed by the programmer; thus allowing extensions to be distributed in a pre-compiled format. This section also discusses limitations of this approach, and argues that the benefits of safe language composition outweigh the moderate loss of expressibility imposed by the restrictions of the modular analysis. Finally we discuss Silver (24), an attribute grammar system, and Copper (27), an LALR(1) parser and context-aware scanner generator we have developed to support the specification and implementation of extensible languages and language extensions.

2. Grammar composition and parser and scanner generation.

The problem we address is how to ensure determinism while combining a *host* grammar with several *extension* grammars, assuming no communication between the writers of the extensions. The host grammar is a context-free grammar in its own right, while extension grammars may reference host language terminals and nonterminals. Extensions, thus, are not defined to extend multiple host languages. While it may be appealing to see how languages such as SQL can be embedded into multiple host languages (3), our interests are in extensions that are more closely tied to the host language, both syntactically (in that extension constructs may include host language constructs, such as in the SQL foreach and table extensions) and also semantically. Detecting a type error in the table construct requires type-checking the Java expressions in it.

2.1 Context-free grammars and grammar composition.

For the purposes of compiling a parser and scanner, a context-free grammar is embellished with a mapping ($regex$) that associates a regular expression (over some alphabet) with each terminal symbol. Thus, a language grammar is a 5-tuple $\langle T, NT, P, s \in NT, regex\colon T \rightarrow Regex\rangle$. Let CFG_L denote the set of such context-free grammars. Below, we fix $\Gamma^H = \langle T_H, NT_H, P_H, s_H, regex_H\rangle$ to be the host language grammar. The grammars that define language extensions are similar to those for defining a complete (host) language, with one exception. Instead of having a start nonterminal, they have a *bridge* production that connects the extension-defined language to the host language. We define *extension* grammars to be of the following form:

$$\Gamma^E = \langle T_E, NT_E, P_E, nt_H \rightarrow \mu_E s_E, regex_E\rangle$$

where $s_E \in NT_E, nt_H \in NT_H, dom(regex_E) = T_E \cup \{\mu_E\}$. The production $nt_H \rightarrow \mu_E s_E$ is the *bridge* production. Its left hand side is a host language nonterminal; its right hand side is the extension's *marking terminal* (μ_E), a terminal introduced by E but not in T_E. It is followed by a nonterminal in NT_E, the start-symbol of the embedded language. We can be less restrictive (but choose not to in order to simplify the presentation and discussion) and allow more than one bridge production — each with a distinct marking terminal — and any non-empty sequence of host and extension terminals and nonterminals following marking terminals. If $p_E \in P_E$, then symbols on the right hand side of p_E are in $T_H \cup NT_H \cup T_E \cup NT_E$, but symbols on the left-hand side must be in NT_E.

We say that Γ^E *extends* Γ^H if Γ^E satisfies these conditions with respect to Γ^H. Grammar composition is only defined when Γ^E extends Γ^H. Let CFG_E denote the set of context-free grammars defining language extensions. We will often use the unqualified term "grammar" but it will be clear from the context if the grammar is a language or extension grammar.

Examples: Figure 2 shows a small portion of the grammars for Java 1.4 and its SQL and tables extensions. Each grammar declares the specified nonterminals, terminals, and productions. The marking terminal for the SQL query extension is the terminal $Using$, which has the regular expression `/using/`; the extension's start symbol is **Sql**. The marking terminal of the tables extension is Tbl. Note that while the SQL productions do not use host language constructs (except for a few terminals) the tables extension allows Java expressions to be in table rows (**TRow**). Thus, it is syntactically correct (but not semantically so) to allow an SQL query or other phrase derived from **Expr** to appear at the beginning of a row.

The SQL extension is split into two grammars here to conform to the grammar structure used in the proof of the modular analysis in Section 3, but in practice these are combined into one grammar.

Java 1.4:

Nonterminals: **Expr**, **PrimaryExpr**, **Dcl**
Terminals: $\mathit{Question}$ `/?/`, Colon `/:/`, Comma `/,/` Semi `/;/`,
LParen `/(/`, RParen `/)/`, LBrk `/{/`, RBrk `/}/`,
Id `/[A-Za-z][A-Za-z0-9]*]/`
Expr $\rightarrow$ **Expr** $\mathit{Question}$ **Expr** Colon **Expr**
Expr $\rightarrow$ **PrimaryExpr**
PrimaryExpr $\rightarrow$ Id
Dcl $\rightarrow$ ...

SQL Connection:

Nonterminals: **ConnDcl**
Terminals: $\mathit{Connection}$ `/connection/`, SqlId `/[A-Za-z]+/` ,
With `/with/`, Table `/table/`
Dcl $\rightarrow$ $\mathit{Connection}$ **ConnDcl**
ConnDcl $\rightarrow$ $\mathit{SqlId}\ \mathit{With}\ \mathit{Table}\ \mathit{SqlId}\ \mathit{Semi}$

SQL Query:

Nonterminals: **Sql**, **SqlQ**, **SqlIds**, **SqlExpr**
Terminals: Using `/using/`, Query `/query/`, Select `/SELECT/`,
From `/FROM/`, Where `/WHERE/`, SqlId `/[A-Za-z]+/`
Expr $\rightarrow$ Using **Sql**
Sql $\rightarrow$ $\mathit{SqlId}\ \mathit{Query}\ \mathit{LBrk}$ **SqlQ** RBrk
SqlQ $\rightarrow$ Select **SqlIds** $\mathit{From}\ \mathit{SqlId}\ \mathit{Where}$ **SqlExpr**
SqlIds $\rightarrow$ SqlId
SqlIds $\rightarrow$ $\mathit{SqlId}\ \mathit{Comma}$ **SqlIds**
SqlExpr $\rightarrow$ ...

Tables:

Nonterminals: **BTable**, **TRows**, **TRow**
Terminals: Tbl `/table/`
PrimaryExpr $\rightarrow$ Tbl **BTable**
BTable $\rightarrow$ LParen **TRows** RParen
TRows $\rightarrow$ **TRow**
TRows $\rightarrow$ **TRow TRows**
TRow $\rightarrow$ **Expr** Colon **TFStarList**

Figure 2. Sample grammar productions from host and extensions.

Also, in practice we do not need to use a start nonterminal in the extensions; in the case of the Tables extension we use the production **PrimaryExpr** $\rightarrow$ $\mathit{Tbl}\ \mathit{LParen}$ **TRows** RParen instead. The proof generalizes to capture both of these modifications, but the simpler proof is presented below and thus the grammars in the figure match that format.

Composition of grammars. Let $\cup_G : \mathit{CFG}_L \times \mathit{CFG}_E \mapsto \mathit{CFG}_L$ be a non-commutative, non-associative operation on context-free grammars. If Γ^E extends Γ^H, then $\Gamma^C = \Gamma^H \cup_G \Gamma^E = \langle T_C, NT_H \cup NT_E, P_C, s_H, \mathit{regex}_C\rangle$ where:

- $T_C = T_H \cup T_E \cup \{\mu_E\}$. μ_E is the extension's marking terminal.
- $P_C = P_H \cup P_E \cup \{h \rightarrow \mu_E s_E\}$, where $h \rightarrow \mu_E s_E$ is the bridge production in Γ^E.
- $\mathit{regex}_C(t) = \begin{cases} \mathit{regex}_H(t) & \text{if } t \in T_H \\ \mathit{regex}_E(t) & \text{if } t \in T_E \text{ or } t = \mu_E \end{cases}$

The operation for composing an unordered set of extensions, $\cup^*_G : \mathit{CFG}_L \times \mathcal{P}(\mathit{CFG}_E) \rightarrow \mathit{CFG}_L$, which is written $\Gamma^H \cup^*_G \{\Gamma^E_1, \ldots, \Gamma^E_n\}$, is the straightforward generalization of $\cup_G$. There is no notion of ordering when composing multiple extensions with the host language; they can be seen as being applied all at once. Thus, the following equivalences hold: $\Gamma^H \cup^*_G \{\Gamma^E_1, \Gamma^E_2\} \equiv (\Gamma^H \cup_G \Gamma^E_1) \cup_G \Gamma^E_2 \equiv (\Gamma^H \cup_G \Gamma^E_2) \cup_G \Gamma^E_1$.

The names of nonterminal and terminal symbols in host and extension grammars can be assumed to be distinct. One way to achieve this is to name grammars in the same way that Java packages are uniquely named (based on Internet domain names) and append symbol names to their defining grammar name.

2.2 Background on parser generation.

This subsection provides some background on LR parsing, LALR(1) parsers, parse tables, and monolithic analysis of grammars. Readers familiar with these topics may wish to skim this section.

Traditionally, we would compose the host language and extension grammars to create the composed language grammar $\Gamma^C = \Gamma^H \cup^*_G \{\Gamma^E_1, \ldots\Gamma^E_n\}$ and then create a parser for that grammar. If LR parsing is used, we would create an LR DFA M^C from the composed grammar (16; 17; 1). This LR DFA can be used to generate an LR parse table which is then checked for shift-reduce and reduce-reduce conflicts. From a context free grammar Γ, an LR parser generator creates an LR DFA from which an LR parse table is directly constructed. We do not review how LR DFAs are constructed from a context free grammar, but only the structure of these DFAs and how they are used. The above references discuss the construction of LR DFAs.

LALR(1) DFA. An LALR(1) DFA for a grammar Γ^L is a 4-tuple $M^L = \langle \Gamma^L, \mathit{States}_L, s_L \in \mathit{States}_L, \delta_L\rangle$, where States_L is the set of states, s_L is the DFA's start state, and $\delta_L : \mathit{States} \times (T_L \cup NT_L) \rightarrow \mathit{States}$ is the DFA's transition function. An LR DFA state is a pair $S = (\mathit{Items}, la\colon \mathit{Items} \rightarrow \mathcal{P}(T_L))$, where Items is the set of items in that state and la maps each item to its lookahead set. An item is a production in Γ^L with a marker placed on its right hand side, to indicate the state of parsing. An item of the form $x \rightarrow \alpha \bullet t\beta, \{t_1, t_2\}$ in a state indicates that the parser has consumed input, a suffix of which can be derived from α. If the next terminal symbol consumed is t and it is shifted onto the parse stack the DFA moves to a state with the item $x \rightarrow \alpha t \bullet \beta, \{t_1, t_2\}$.

In the discussion of the modular analysis and argument for its correctness we will have need to compare different LR DFA states; we introduce these comparisons here.

Given two LR DFA states s and t: • s is an ***I-subset*** of t, written $s \subseteq_{\mathrm{I}} t$, if $\mathit{Items}_s \subseteq \mathit{Items}_t$. • They are ***LR(0)-equivalent***, written $s \equiv_0 t$, if $s \subseteq_{\mathrm{I}} t$ and $t \subseteq_{\mathrm{I}} s$ — *i.e.*, they have the same item set. We use the term ***LR(0)-equivalent*** because in an LR(0) DFA, where there are no lookahead sets, two LR(0)-equivalent states would be equal. • s is an ***IL-subset*** of t, written $s \subseteq_{\mathrm{IL}} t$, if $s \subseteq_{\mathrm{I}} t$ and $\forall i \in \mathit{Items}_s.\,(la_s(i) \subseteq la_t(i))$. • They are ***LR(1)-equivalent***, written $s \equiv_1 t$, if $s \subseteq_{\mathrm{IL}} t$ and $t \subseteq_{\mathrm{IL}} s$ — *i.e.*, the states' item sets and all lookahead sets are equal. Note that if $s \subseteq_{\mathrm{IL}} t$, and t produces a conflict-free parse state, so does s.

LR DFAs are converted to parse tables, a more convenient representation, for use in parsing. See (1) for details on how this is done.

Parse tables. Let States denote the set of all rows of all parse tables. A parse table is defined as a 4-tuple $PT = (\Gamma_{PT}, \mathit{States}_{PT}, \pi_{PT}, \gamma_{PT}, n^S_{PT} \in \mathit{States}_{PT})$, where:

- Γ_{PT} is a grammar that this parse table parses correctly.
- $\pi_{PT} : \mathit{States}_{PT} \times T \rightarrow \mathcal{P}(\mathit{Actions})$, where $\mathit{Actions} = \{\mathit{accept}\} \cup \{\mathit{reduce}(p)\colon p \in P\} \cup \{\mathit{shift}(x)\colon \in \mathit{States}\}$.
- $\gamma_{PT} : \mathit{States}_{PT} \times NT_{PT} \rightarrow \mathcal{P}(\mathit{Goto})$, where $\mathit{Goto} = \{\mathit{goto}(x) :\in \mathit{States}\}$.

If for some parse table pt, some $n \in \mathit{States}$ and $t \in T$, $\pi_{pt}(n,t) = \emptyset$, that cell contains an error action. A cell (n,t) in a parse table PT has a conflict if $|\pi_{PT}(n,t)| \geq 2$. A state n is conflict-free if for all $t \in T_{PT}$, cell (n,t) does not have a conflict:

$\forall t \in T_{PT}.\,(|\pi_{PT}(n,t)| \leq 1)$. A parse table PT is conflict-free if all $n \in States_{PT}$ are conflict-free.

The programmer can run this monolithic analysis on the composed grammar to ensure that there are no parse-table conflicts or lexical ambiguities. This comes too late, however, as the programmer will not necessarily have the skills to modify the grammars to fix the problems. What is needed is a modular analysis that the extension writer can use on his or her extension to ensure that when it is composed, by the programmer, with other extensions that also pass the modular analysis, the resulting composed grammar will pass the monolithic test.

2.3 Context-aware scanners.

The context-aware scanners used here are an extension of those described in (27). The scanner can be constructed as described there or alternatively one can generate a scanner for each parse state that only matches those in the valid lookahead set. How this scanner is constructed is not of concern; it is sufficient that it return only tokens in the valid lookahead set for the current parse state.

Given a parse table PT (with terminals T), valid lookahead sets are represented collectively by the function $validLA : States_{PT} \rightarrow \mathcal{P}(T)$, where $validLA(n) = \{t : \pi_{PT}(n,t) \neq \emptyset\}$ (the terminals with non-error actions). For this parse table and the function $scan : \mathcal{P}(T) \times \Sigma^\star \rightarrow \mathcal{P}(T) \times \Sigma^\star$, it holds that $scan(X,w) = (X',w')$ where X' is the set of terminals matched by the scanner and w' the lexeme they match. X is the valid lookahead set and w is the input. Thus w' is a prefix of w and $X' \subseteq X$. If $scan(X,w) = \emptyset$, it means that no $t \in X$ matches a prefix of w.

A *lexical ambiguity* occurs when $\exists n \in States_{PT}, w \in \Sigma^\star.\,(|scan(validLA(n),w)| \geq 2)$. An *ambiguity-free scanner* is one for which this is untrue of every $(n,w) \in States \times \Sigma^\star$. This can be verified for all w by analyses on scanner DFAs (27).

Lexical disambiguation. Context is the primary way in which terminals with overlapping regular expressions are disambiguated. Notions of lexical precedence (*e.g.,* indicating that a keyword takes precedence over an identifier terminal) can also be used. Using marking terminals from different extensions may lead to lexical ambiguities because they may appear in the same context (*e.g.,* at the beginning of an expression) and no lexical precedence settings can be specified to disambiguate them, since they are in different extensions. A notion of "transparent prefixes" (see Section 4) allows programmers to disambiguate them by prefixing them with the grammar name, the same way that a Java class name is prefixed with its package name when two imported packages define a class with the same name. Other precedence setting techniques are also described in Section 4. These are used to ensure that a composed language Γ^C will have no lexical ambiguities that prevent the use of a deterministic scanner. Thus, in the following discussion of the modular analysis, we can focus our attention on the parser and assume that there are no lexical ambiguities.

3. The modular determinism analysis.

The modular parser analysis $isComposable(\Gamma^H, \Gamma^E_i)$ analyzes the context-free grammar of an extension with respect to the host language being extended. If the extension passes this analysis it can be considered "certified" and safely composed with other certified extensions by the programmer. Formally, this is expressed as

$$\begin{aligned}&(\forall i \in [1,n].isComposable(\Gamma^H, \Gamma^E_i) \wedge \\ &\qquad conflictFree(\Gamma^H \cup_G \Gamma^E_i)) \\ &\Longrightarrow conflictFree(\Gamma^H \cup^\star_G \{\Gamma^E_1, \ldots, \Gamma^E_n\})\end{aligned}$$

3.1 The modular analysis $isComposable$.

The restrictions we place on what kind of constructs can be added in an extension to a host language are not restrictions on the extension grammar but on the LR DFA generated when compiling $\Gamma^H \cup_G \Gamma^E_i$ as compared to the LR DFA generated when compiling Γ^H alone, *i.e.*, what states, items, and lookahead can be added to the LR DFA of Γ_H to yield the LR DFA for $\Gamma^H \cup_G \Gamma^E_i$. The key factor is that items and lookahead added by one extension cause conflicts neither with the host language nor with any other extensions that are added later (and have also passed this modular analysis). In brief, the test $isComposable(\Gamma^H, \Gamma^E)$ must ensure that the grammar $\Gamma^H \cup_G \Gamma^E$, and the LR DFA generated for it, have the following properties:

1. For all $nt_H \in NT_H$, no new terminals appear in its *follow* set $follow(nt_H)$, except the marking terminal μ^E_i.
2. In states also appearing in the DFA for Γ^H, no new terminals appear in the lookahead sets, except the marking terminal μ^E_i.
3. For states that do not appear in the DFA for Γ^H, but contain only host syntax items and thus could potentially be generated by several extensions, their item set and lookahead sets are subsets of those of some state also appearing in the DFA for Γ^H that contains no conflicts, ensuring that these types of states (even when combined with similar types of states generated by other extensions) will contain no conflicts.

In describing the analysis and argument for its correctness we will need to consider various LR DFAs and different states thereof:

1. The LR DFA for the original host language, denoted M^{orig};
2. The LR DFA generated for $\Gamma^H \cup_G \Gamma^E_i$ by the designer of extension E_i, denoted M^{E_i}; and
3. The LR DFA generated for $\Gamma^H \cup^\star_G \{\Gamma^E_1, ..., \Gamma^E_n\}$ by the programmer, denoted M^C.

The superscripts $orig$, E_i, and C indicate when the LR DFA is constructed: when the host language is defined, when an extension designer specifies an extension and performs the modular analysis, and when a programmer combines multiple extensions.

The extension designer will perform the modular analysis, which takes Γ^H and Γ^E_i as inputs: $isComposable(\Gamma^H, \Gamma^E_i)$. The analysis generates $\Gamma^H \cup_G \Gamma^E_i$ and builds the two LR DFAs M^{orig} and M^{E_i} (in the usual way) and then proceeds in two phases. First, it checks that no follow sets of host nonterminals have changed except to add the marking terminal. Formally, it checks if

$$\forall nt \in NT_H.\left(follow_{\Gamma^H \cup_G \Gamma^{E_i}}(nt) \setminus follow_{\Gamma^H}(nt) \subseteq \left\{\mu^E_i\right\}\right).$$

If additional (non-marking) terminals exist in the follow sets in $\Gamma^H \cup_G \Gamma^{E_i}$, then the analysis fails. Consider the production **TRow** $\rightarrow$ **Expr** $Colon$ **TFStarList** from the tables extension and the conditional-expression production **Expr** $\rightarrow$ **Expr** $Question$ **Expr** $Colon$ **Expr** from the host language Java grammar, both shown in Figure 2. The extension does not add to the follow set of **Expr** since terminal $Colon$ is already there due the the conditional-expression production. If new terminals are added to the follow sets of host language nonterminals two different extensions may compose individually with the host language to create conflict-free parsers, but when combined together, the conflict can arise.

It is worth noting that this restriction is more easily satisfied in syntactically rich fully-developed languages (such as Java), since their nonterminals have larger follow sets, than in small toy languages with smaller follow sets.

In the second phase, which is used only if the follow sets have not expanded, the analysis compares the two constructed LR DFAs M^{orig} and M^{E_i} to determine if M^{E_i} can be constructed by adding new LR DFA states to M^{orig} in a specific way (defined below), and adding new items and lookahead elements corresponding to this addition to the states already existing in M^{orig}. The constraints of the modular analysis only allow such additions to the states of the host LR DFA M^{orig} as to allow a partitioning of the states in M^{E_i}. Such a partitioning assigns "ownership" of the states of M^{E_i} to either the host language, the extension, or neither. The LR DFA M^{E_i} consists of the states in the three DFA partitions: $M^{E_i}_{E_i}$ (states owned by the extension), $M^{E_i}_{H}$ (states owned by the host language), and $M^{E_i}_{NH}$ (states owned by neither). The notion of ownership is indicated by the subscript. A diagram of this partitioned LR DFA is shown in Figure 3(a).

It is possible that these three partitions have no transitions between them except for the single transition from $M^{E_i}_{H}$ into $M^{E_i}_{E_i}$ labeled with μ^E_i, the marking terminal. If one of the extension's productions contains a reference to a host language nonterminal in its right hand side there may also be other transitions, labeled with host terminals, between the partitions. These possible transitions are indicated by the dotted lines in Figure 3.

The analysis inspects each state n in M^{E_i}. If each state can be assigned to one of these partitions, then the extension passes the modular analysis $isComposable$, and when several extensions are combined to form M^C the parse table generated therefrom will be conflict-free.

Extension owned states $M^{E_i}_{E_i}$***:*** A state $n \in M^{E_i}$ is assigned to the partition $M^{E_i}_{E_i}$ if it has at least one item ($i \in Items_n$) whose left hand side symbol nt is an Γ^E_i nonterminal (*i.e.*, $nt \in NT_{E_i}$). States assigned to this partition are those used for parsing the embedded language of Γ^E_i. For the SQL extension, these states parse the embedded SQL language. These states are said to be *owned* by Γ^E_i. There are few restrictions on these states since they contain parts of the embedded language introduced by the extension, and hence, when extensions are merged, these states will not be merged with the states owned by other extensions.

Formally, $\forall n \in M^{E_i}.(\exists\,(nt \to \cdots \bullet \cdots) \in Items_n.(nt \in NT_{E_i}) \Rightarrow n \in M^{E_i}_{E_i})$.

Let $n^S_{E_i}$ be the state in M^{E_i} that contains the item $h \to \mu^E_i \bullet s^E_i$. This is the extension start state $n^S_{E_i} \in M^{E_i}_{E_i}$.

In the LR DFA for the language composed by the programmer, these states may have new items added to them. These items will only be bridge production items of the form $h \to \bullet\mu_{E_j} s_{E_j}$ for some other extension E_j.

Host owned states $M^{E_i}_{H}$***:*** These states are said to be owned by the host language H. The analysis attempts to construct a bijection $m : M^{E_i}_{H} \to M^{orig}$ that satisfies the criteria discussed below. The inverse function is denoted m^{-1}. If such a bijection cannot be constructed, the analysis fails. If one can, then each state in $M^{E_i}_{H}$ has a unique corresponding state in M^{orig}. They are then further classified by if and how their item sets and lookahead are (safely) extended by the extension. If a state in M^{orig} cannot be so classified the analysis fails.

In constructing m, elements in its domain (*i.e.*, $M^{E_i}_{H}$) are further partitioned into 3 sets. The first is $nochange(M^{E_i}_{H})$. These are states that have not changed. A state $n \in M^{E_i}$ is a member of $M^{E_i}_{H}$ and assigned to the partition $nochange(M^{E_i}_{H})$ if $\exists n_0 \in M^{orig}.n_0 \equiv_1 n$. In this case we specify that $m(n) = n_0$. This checks if n_0 and n have the same set of items and each item has the same lookahead. If n_0 had no conflicts, then clearly n has no conflicts. This is one way in which we *maintain* the determinism of M^{orig} in M^C.

The second is $markingLA(M^{E_i}_{H})$. These are states that do not have new items, but may have the extension's marking terminal μ^E_i in the lookahead set of existing items. A state $n \in M^{E_i}$ is assigned to be in $markingLA(M^{E_i}_{H})$ if $\exists n_0 \in M^{orig}.n_0 \equiv_0 n \wedge (\forall i \in Items_{n_0}.(la_n(i) \setminus la_{n_0}(i) \subseteq \{\mu^E_i\}))$. This checks that some state n_0 in the original host LR DFA has the same set of items as n ($n_0 \equiv_0 n$) and for each item, either its lookahead sets are the same in both states or its lookahead in n has the single additional element μ^E_i. This does not cause any conflicts; see lemma 3, below.

This may result in shift and reduce actions based on this marking terminal to appear in the parse table of M^{E_i}, but any conflicts will be detected by the extension writer when checking that this table is conflict free. This check is part of the analysis as specified at the beginning of Section 3. Note the similarity of this test to the follow-set test. Both work along the same basic principle; however, instead of entirely new symbols, this test aims to exclude from a state symbols that were already in the follow set of some nonterminal but did not show up as lookahead in this state.

The third is $bridge(M^{E_i}_{H})$. These are states that have exactly one new item, of the form $h \to \bullet\mu^E_i s^E_i, [z]$, and also allow the marking terminal μ^E_i in the lookahead of existing items. A state $n \in M^{E_i}$ is in $M^{E_i}_{H}$ and is assigned to be in $bridge(M^{E_i}_{H})$ if $\exists n_0 \in M^{orig}.(Items_n = Items_{n_0} \cup \{h \to \bullet\mu^E_i s^E_i\} \wedge (\forall i \in Items_{n_0}.(la_n(i) \setminus la_{n_0}(i) \subseteq \{\mu^E_i\}))$. This checks that some state n_0 in the original host LR DFA has the same set of items as n (excluding the bridge item $h \to \bullet\mu^E_i s^E_i$) and that the lookahead on the items that n and n_0 have in common are identical except that the marking terminal μ^E_i may be in the lookahead of items in n.

These items cause the parser to shift on the marking terminal of that extension to an extension language state in $M^{E_i}_{E_i}$, as indicated by the two edges labeled μ^E_i in Figure 3(a). If n is a bridge state then $\delta(n, \mu_E) = n^S_\mu$; it shifts to the start state of the extension. Extensions only add items of the form $h \to \bullet\mu^E_i s^E_i$ to host language states. Thus, since the marking terminals for each extension are by definition different we will not have any conflicts in the parse table of the programmer composed language.

"New" host states $M^{E_i}_{NH}$: Extensions do interact with the host language by including host language constructs in the extension-introduced constructs. Thus, productions in P_{E_i} may include host language nonterminals and terminals on the right hand side.

These productions may cause states to be generated that contain items consisting of only host language terminals and nonterminals, but do not correspond to states in $M^{E_i}_{H}$ in one of the ways described above. We need to ensure that these new states do not have conflicts and are consistent with existing host language states, so that the determinism of the host language is maintained in these states. This is needed to ensure that in creating the composed language LR DFA M^C no conflicts are introduced based on how the extensions use host language terminals and nonterminals.

A state $n \in M^{E_i}$ is assigned to the partition $M^{E_i}_{NH}$ if:

1. It cannot be assigned to the partition $M^{E_i}_{H}$ in any of the three ways described above;
2. Each item $i \in Items_n$ consists only of host language terminals and nonterminals, or is of the form $h \to \mu^E_i s^E_i$;
3. $\exists n' \in M^{E_i}_{H}.(n \subseteq_{\mathrm{IL}} n')$; and
4. $\forall n' \in M^{E_i}_{H}.(n \subseteq_{\mathrm{I}} n' \Rightarrow n \subseteq_{\mathrm{IL}} n')$.

Conditions 3 and 4 ensure that productions that generate new-host states from different extension DFAs ($M^{E_i}_{NH}$ and $M^{E_j}_{NH}$), when combined, generate compositions of these states in the new-host partition of the composed LR DFA M^C, as in Figure 3(b).

If a state $n \in M^{E_i}$ cannot be classified as belonging to one of the three partitions $M^{E_i}_{E_i}$, $M^{E_i}_{H}$, or $M^{E_i}_{NH}$, then the modular analysis

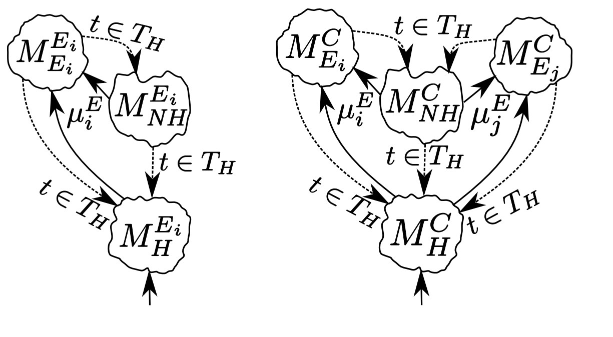

Figure 3. Abstract diagram of merging the host language with one extension (a) and with two extensions (b).

fails. Since the partitions are, by these definitions, disjoint, the order in which the state $n \in M^{E_i}$ is checked against them does not matter.

If each extension passes the modular analysis and the host language extended with the extension has a conflict free parser, then the host language extended with all such extensions has a conflict free parser as well. In Section 4 we define the modular lexical ambiguity checker $isLexComp$. In tandem, they ensure that "certified" language extensions can be safely composed by the programmer and recognized with a deterministic parser and scanner. Next we provide some examples and then explain why this claim is true.

3.2 Examples of passing and failing grammars.

Composable extension. Consider the host, extension, and bridge productions below:

- $P_H = \{E \to T + E, E \to T, T \to x\}$
- $P_{E_i} = \{S \to S_1\ E, S_1 \to \#\}$, bridge production $E \to \mu_E\ S$

This grammar passes the modular determinism analysis because the only reference to a host nonterminal inside the extension syntax occurs at the end of the first production in P_{E_i}. Therefore, the only symbols added to $follow_{\Gamma^C}(E)$ are those in $follow_{\Gamma^C}(S)$. Since S itself appears only at the end of a production derived from E, this adds no new terminals to $follow_{\Gamma^C}(E)$.

Extension adding to the follow set of a host nonterminal. Consider an extension with the following productions, which fails the analysis:

- $P_H = \{E \to T + E, E \to T, T \to x\}$
- $P_{E_i} = \{S \to S_1\ E, S_1 \to T\}$, bridge production $E \to \mu_E\ S$

In this extension, an expression derived from E can occur immediately after an expression derived from T. This means that all terminals in $follow_{\Gamma^H}(E)$ are added to $follow_{\Gamma^C}(T)$. This includes $x \notin follow_{\Gamma^H}(T)$; thus, the extension fails the analysis.

Extension producing a "non-IL-subset" condition. Consider another failing extension, with these following productions:

- $P_H = \{E \to T + E, E \to T, E \to x\ !, T \to x\}$
- $P_{E_i} = \{S \to S_1\ z\ E, S_1 \to T\}$, bridge $E \to \mu_E\ S$

In the host grammar, both E and T can derive expressions beginning with x. This results in there being a state inside M^{orig} containing the items $E \to x\bullet!, \{\$\}$ and $T \to x\bullet, \{\$, +\}$. The extension contains a reference to T; the placement adds z to $follow_{\Gamma^C}(T)$, *per se* causing the extension to fail. However, unlike in the host grammar there are no other references to x there; if x occurs in an extension expression it is derived from T. This means that there is inside the extension DFA a state containing only the item $T \to x\bullet, \{z\}$. This state is a "new host" state. This is an I-subset of the state named above, but is not an IL-subset of it, causing this extension to fail the analysis on a second count.

3.3 Correctness of the modular analysis $isComposable$.

Composability Theorem: The modular conflict-free analysis $isComposable$ performed on the LR DFAs for Γ^H and Γ^E and traditional conflict-free analysis performed on $\Gamma^H \cup_G \Gamma^E$ ensure that when the extension grammar E is composed with H and other extensions passing these analyses, the resulting grammar will have a conflict-free LR parse table. Formally, this is expressed as

$$\begin{aligned}&(\forall i \in [1,n].isComposable(\Gamma^H, \Gamma^E_i) \wedge \\ &\qquad\qquad conflictFree(\Gamma^H \cup_G \Gamma^E_i)) \\ &\implies conflictFree(\Gamma^H \cup^\star_G \{\Gamma^E_1, \ldots, \Gamma^E_n\})\end{aligned}$$

A sketch of the proof of this theorem follows. It is based on the fact that the partitioning of states in M^{E_i}, as seen in Figure 3(a), is extended to the partitioning of states in the composed language DFA M^C, as seen in Figure 3(b). A full version of the proof appears in an accompanying technical report (23). The following lemmas are used in the proof; for brevity, their proofs are summarized.

Lemma 1: No items from two extensions in any state. If a state n has $(nt_i \to \alpha \bullet \beta) \in Items_n$, then if $nt_i \in NT_{E_i}$, $\{nt : (nt \to \gamma \bullet \delta) \in Items_n\} \subseteq NT_H \cup NT_{E_i}$, *i.e.*, no state has items with left-hand sides from more than one extension.

Proof summary: The only transition path from the start state of the composed LR DFA M to any state with such an item in it must, by construction, pass through a state seeded from item $h \to \mu^E_i \bullet s^E_i$, that is, the extension start state $n^S_{E_i}$. This state functions as a "bottleneck" that filters out syntax from all other extensions.

Lemma 2: The follow sets of host nonterminals in the grammar Γ^C add only marking terminals to the follow sets in Γ^H.

Proof summary: Any non-marking terminals introduced to the follow sets would have to have been put there by a single extension, which means that it would have been caught in the modular analysis.

Lemma 3: The class of conflict-free states is closed under introduction of bridge items and marking terminal lookahead.

Proof summary: Since marking terminals are only defined with one production $h \to \mu^E_i s^E_i$, wherever μ^E_i can be shifted, so can the other terminals of *first*(h) (the set *first*(h) is the set of terminals that begin any phrase derivable from h). Similarly, if μ^E_i is in a lookahead set, so is everything else in *first*(h). It follows that if there is a parse table conflict on μ^E_i, an identical conflict would occur on all other terminals in *first*(h). Therefore, if a state is conflict-free, conflicts cannot be introduced to it by adding bridge items or marking terminal lookahead.

Proof sketch of the composability theorem: The proof shows that the states in LR DFA M^C built from the composed grammar $\Gamma^H_{\cup^*_G} \{\Gamma^E_1, \Gamma^E_2, ...\Gamma^E_n\}$ can be partitioned into the sets M^C_H, $M^C_{E_1}$, $M^C_{E_2}$, ..., $M^C_{E_n}$, and M^C_{NH}. This partitioning is straightforward and depends one the structure of the state. The proof then shows that states in each of these sets are conflict-free. Each of these partitions is described below and it is argued that the states in these partitions are necessarily conflict-free.

***Host state partition* M_H^C:** M_H^C will contain states n that

1. are not in any $M_{E_i}^C$ (see below), and
2. There exists a path through only non-extension states from the start state of M^C to this state, *i.e.*, there exists a sequence $\{n_0, (n_1, \sigma_1), \ldots, (n_k, \sigma_k)\}$ with $n_0 = s_{M^C}, \delta_{M^C}(n_{i-1}, \sigma_i) = n_i$, $n_k = n$, and $\forall i, j. n_j \notin M_{E_i}^C$.

To reach this state from the start state of M^C, we followed the path $(\sigma_1, \ldots, \sigma_k)$. None of these σ_is are extension symbols or marking terminals, since each $\delta_{M^C}(n_{i-1}, \sigma_i)$ points to a host state. Therefore, in each of the LR DFAs M^{E_i}, $i \in [1, n]$, in which the host and a single extension are composed, we could follow the path $\sigma_1, \ldots, \sigma_k$ and in each one end up in a state in $M_H^{E_i}$ that, if bridge items and marking terminal lookahead are not considered, is identical to n.

This follows by construction, since host syntax is common to all M^{E_i}(for each i) and any states along the same sequence of transitions on host-only symbols must have an identical set of host-only items and host-only lookahead. The state in $M^C$is a union of the item set and lookahead sets of all of the states in M^{E_i}. It only differs from any of these in that it contains additional bridge production items and marking terminal lookahead from multiple extensions. Since the marking terminals are distinct, the state in M^Cwill not have any conflicts even though it may have several bridge production items or lookahead sets with several marking terminals. If there were conflicts in this state, that conflict would have to exist in one of the DFAs M^{E_i}, for some i.

***Extension state partitions* $M_{E_i}^C$:** The LR DFA $M_{E_i}^C, i \in [1, n]$, will consist of states containing items with an extension nonterminal $nt \in NT_{E_i}$ on the left hand side. Note that by lemma 1, these subsets are all disjoint.

Suppose that the bridge production for Γ_i^E is $h \rightarrow \mu_i^E s_i^E$. By construction, the only paths to any state $n \in M_{E_i}^C$ from the start state of M^C run through the extension start state $n_{E_i}^S$. Furthermore, such a path must exist, since no state in the DFA is isolated.

If there is some state n_I on a path between $n_{E_i}^S$ and n for which $n_I \notin M_{E_i}^C$, then there is no syntax in its items from Γ_i^E in it, so by construction $n_{E_i}^S$ must be on the path between n_I and n. This constitutes a cycle. Since there is by definition an acyclic path between those two states, every such acyclic path must consist entirely of states in $M_{E_i}^C$.

The sequence of symbols labeling this path are all in $T_H \cup NT_H \cup T_{E_i} \cup NT_{E_i}$. If one were not, then it is in some $T_{E_j} \cup NT_{E_j}$. But this means that the symbol in question must be in the state preceding the transition marked with that symbol, which is a contradiction. This means that $M_{E_i}^C$ forms an unbroken "block" of states connected by transition from $n_{E_i}^S$ along paths marked solely with symbols in $T_H \cup NT_H \cup T_{E_i} \cup NT_{E_i}$.

Consider the properties of $n_{E_i}^S$. It is seeded from the single item $i_E = h \rightarrow \mu_i^E \bullet s_i^E$, and is the only state containing this item. There is a corresponding state $n_0 \in M_{E_i}^{E_i}$ seeded from i_E. By construction $la_{n_0}(i_E) \subseteq la_{n_{E_i}^S}(i_E)$. Since by construction there is a transition to $n_{E_i}^S$ from any state in which there are items $h \rightarrow \bullet \cdots$, the lookahead on i_E is exactly the whole follow set of h: $la_{n_{E_i}^S}(i_E) = \mathit{follow}_{\Gamma^C}(h)$. Analogously, $la_{n_0}(i_E) = \mathit{follow}_{\Gamma^H \cup_G \Gamma^{E_i}}(h)$. Therefore, the difference between the two lookahead sets is the same as the difference between the two follow sets: $la_{n_{E_i}^S}(i_E) \setminus la_{n_0}(i_E) = \mathit{follow}_{\Gamma^C}(h) \setminus \mathit{follow}_{\Gamma^H \cup_G \Gamma^{E_i}}(h)$. By lemma 2, this difference consists entirely of marking terminals.

Using the above process of following transitions simultaneously, this time in M^C and $M^{E_i}$and from $n_{E_i}^S$ and n_0, it follows by construction that for every state in $n \in M_{E_i}^C$ there is a corresponding state in $n' \in M_{E_i}^{E_i}$, accessible via the same path. Since every acyclic path from $n_{E_i}^S$ to n goes entirely through states in $M_{E_i}^C$, structural induction shows that new bridge items are the only sort of new items that can appear, and $n_{E_i}^S$ and these bridge items being the sole source for any new lookahead, marking terminals are the only new lookahead.

Therefore, by lemma 3, no state in $M_{E_i}^C$ has a conflict.

***"New" host partition* M_{NH}^C:** M_{NH}^C will contain all states not in the above subsets, which by elimination:

1. are not in any $M_{E_i}^C$, and
2. all paths from M^C's start state to n pass through some extension start state $n_{E_i}^S$.

Firstly, note that it is altogether possible that there are paths from *several* such $n_{E_i}^S$s to n, not counting paths through other extension start states $n_{E_j}^S$. Call the set of grammars with such a path $\mathit{Contrib}_n$. $|\mathit{Contrib}_n| \geq 1$. No such path contains any marking terminals (that would put it through some $n_{E_j}^S$) or terminals not from the particular extension from whose start state it originates (in consequence of there being no marking terminals). Neither does it go through any state $n_H \in M_H^C$: as all transitions out of such states, except those labeled with marking terminals, have another state in M_H^C as a destination, it follows that every state between n_H and n is in M_H^C. But this would put n in M_H^C, which it is not.

It is now established that for each $\Gamma_i^E \in \mathit{Contrib}_n$, there is a path with every transition in $T_H \cup NT_H \cup T_{E_i} \cup NT_{E_i}$ leading from $n_{E_i}^S$ to n. This means that there is an identical path in M^{E_i} leading to a state n_i that, ignoring bridge items, is LR(0)-equivalent to n. This state is in neither $M_H^{E_i}$ nor $M_{E_i}^{E_i}$, hence must be in $M_{NH}^{E_i}$. For the same reasons as above, in addition to the new marking terminal lookahead, n contains exactly all the lookahead from all such n_i. This is the set of lookahead that could potentially cause conflicts in this state.

By conditions 3 and 4 of the analysis that assigns states to $M_{NH}^{E_i}$, there is some $n_i' \in M_{NH}^{E_i}$ with $n_i \subseteq_{\mathrm{I}} n_i'$, and furthermore for all such n_i', $n_i \subseteq_{\mathrm{IL}} n_i'$. Furthermore, since the most the item sets of the n_is differ is one bridge item, and the addition of a bridge item to n_i would imply that it was also added to any I-superset of n_i, it follows that the space of I-supersets of each n_i maps to the same set of states in M^{orig}, which are these states shorn of their bridge items and marking terminal lookahead. Furthermore, each hypothetical state consisting of n_i shorn of its one possible bridge item and marking terminal lookahead is an IL-subset of each of these states in M^{orig}.

Now the subset relation is closed under union. This means that if a state consists of the union of the items and lookahead of several IL-subsets of the same state, the union state is itself an IL-subset of that state. It follows immediately that the hypothetical state consisting of n shorn of all its bridge items and lookahead is an IL-subset of each of the M^{orig} states. This state, therefore, is conflict-free, and the addition of marking terminal lookahead and bridge items will not add conflicts; therefore, n is also conflict-free.

4. Lexical disambiguation and practical concerns.

4.1 Resolving lexical ambiguities.

Above, we have assumed that there are no lexical ambiguities, emphasizing that if each language extension chosen by the programmer passes the modular analysis $\mathit{isComposable}$, then the composed language parse table will be deterministic. In practice, regular expressions for terminal symbols do overlap and the language designer resolves them, typically by specifying some sort of lexical precedence so that, for example, keyword terminals are preferred over identifier terminals for lexemes that match both.

With context-aware scanners such as Copper (27), the parse-state-based context used to disambiguate the lexical syntax allows the composed language to have terminal symbols that have overlapping regular expressions (those that share at least one common lexeme) as long as those terminals are not in the same valid lookahead set for any parse state. Based on the partitioning of M^C described in Section 3 we know that non-marking terminals of different extensions cannot cause lexical ambiguities in the composed language since they never occur in the same valid lookahead set. (Lexical ambiguities between terminals in a single extension and/or the host language can be resolved by the extension designer.) For example, the Table keyword terminal introduced by the SQL extension will never be in the same context as the table extension's Tbl terminal even though both have the same regular expression `/table/`; therefore, that causes no lexical ambiguity.

But since bridge-production items can be (safely) added to parse states owned by the host language or other extensions we have the possibility for lexical ambiguities in two ways. The first is between marking terminals from different extensions. For example, the SQL marking terminal Using and the "tables" marking terminal Tbl can both appear at the beginning of an expression and there are thus in the same valid lookahead set for several parse states.

The second occurs more rarely, in parse states owned by an extension E between its non-marking terminals and other extension marking terminals. (Note that this does not occur in the SQL extension since its Table terminal cannot appear in the same location as a Java expression, which can begin with the table extension's Tbl marking terminal.)

Most scanner generators, including context-aware ones such as Copper, allow a "global" precedence relation to be specified between terminals, so if s takes precedence over t, no string matching s will match t, even in contexts where s is invalid. This type of static precedence does not respect boundaries of parse state: if some terminal $t_e \in T_i^E$ is made to take precedence over a host terminal $t_h \in T_H$, no lexeme matching t_e can match t_h — even in a state belonging to some other extension Γ_j^E. For this reason, extension writers should avoid defining static precedence relations between host terminals and extension terminals, though it is reasonable for static precedence to be specified on host language keyword terminals. If an extension writer does this, no additional conflicts or ambiguities will occur, but the presence of Γ_i^E will alter the language of Γ_j^E. Also, extensions may not define any new precedence relations between host terminals.

Transparent prefixes (27) provide a solution for disambiguating marking tokens. The technique is similar to how class names are disambiguated in Java programs when two packages that define a class with the same name are imported into a Java program; the package name (based on the unique Internet domain name of the package author) is prepended to the class name to indicate the desired class. Grammar names, which can also be based on Internet domain names, can be used to disambiguate marking tokens. This approach is taken by Copper and Silver. The grammar names are added to the valid lookahead tokens passed to the scanner. If the input matches such a name, the scanner does not return it to the parser, but instead uses this extension name to remove terminals defined in other extensions from the valid lookahead set and scans again from the point in the input after the grammar name. Now, only terminals from the extension and the host language are in the valid lookahead set so there will be no lexical ambiguities. (If there were, they would have been resolved by the extension writer.) Thus, if the scanner does report a lexical ambiguity to the programmer, it can be easily resolved by the programmer by adding the extension name before the marking token. This is the same burden that is placed on Java programmers and thus we do not feel that it is unreasonable. Our lexical ambiguity analysis reports these possible ambiguities, but these do not prevent the extension from passing the modular lexical ambiguity analysis. The analysis does, of course, check for lexical ambiguities between extension and host language terminals, which are then resolved by the extension writer.

In addition to providing transparent prefixes, the extension writer must specify a "default" behavior that use of the prefix can preempt. This is done by indicating that a marking terminal μ_E is of one of the following sorts:

- "Reserve against other terminals." This means that static precedence relations will be formed with μ_E taking precedence over any terminals with which it conflicts lexically, and no string matching μ_E's regular expression can match any of these terminals in any context. The use of this option should be avoided for the same reason as other static precedence relations between host and extension terminals should be avoided.
- "Prefer over host terminals." This means that wherever μ_E causes an ambiguity with another terminal or terminals, the ambiguity is resolved in favor of μ_E.
- "Avoid in favor of host terminals." This means that wherever μ_E causes an ambiguity with another terminal, the ambiguity is resolved in favor of the other terminal. If one has a set of terminals X disambiguated via this mechanism, and a new marking terminal μ_i^E is introduced, a new ambiguity — $X \cup \{\mu_i^E\}$ — is resolved the same way X was. N.B.: The use of this option mandates the use of μ_E's transparent prefix to match it.

Thus, there are a number of ways to design the host and extension languages to handle lexical disambiguation. The modular lexical ambiguity checking analysis $\mathit{isLexComp}$ verifies that no lexical ambiguities are possible in the composed language *except* for those involving marking terminals that can be disambiguated by the programmer. Thus, a deterministic scanner can still be used and it can be designed to give helpful error messages when a lexical ambiguity occurs. It can rescan the input with all terminals in the valid lookahead set, see which match, determine which extensions defined those terminals and suggest to the programmer that a transparent prefix naming one of these extensions is needed. This disambiguation process requires no implementation-level knowledge of the composed language parser or scanner and is essentially the same as disambiguation done for ambiguous Java class names. Thus we do not consider it a significant burden on the programmer.

4.2 Operator precedence.

We prove above that the introduction of a marking terminal μ_i^E, where $h \to \mu_i^E s_i^E$, cannot cause parse-table conflicts because the conflict in question would also occur on other members of $\mathit{first}(h)$ and, therefore, be caught by the modular analysis. However, this does not hold true if the conflicts on the non-marking terminal cells have been resolved by setting operator precedence rules on these other members of $\mathit{first}(h)$, which do not apply to μ_i^E. Given that marking terminals are "prefixes" of a sort, this is unlikely to occur in practice. We have not seen any instance where it occurs, but for operator precedence to be used in this approach, one of the following two solutions could be applied.

Specify a blanket precedence rule. The extension writer could provide a blanket precedence rule specifying how to resolve such conflicts should they occur. This would simply be an ordinary operator precedence specified on a "placeholder" marking terminal, $\mu^\star$, standing in for any μ_i^E that are introduced.

Tighten the test. Extensions that reference a host nonterminal h could also be subjected to more stringent tests. While compiling the LR DFA for $\Gamma^H \cup_G \Gamma^E_j$, it is possible to keep track of what nonterminals contribute lookahead to items in which states. Let $Interlopers_j$ signify every host nonterminal contributing lookahead to any state owned by Γ^E_j. Then ensure that each of these nonterminals has a symbol in its *first* set with no operator precedence defined on it. Define a bijection

$$mark\ :\ \left\{\mu^\star_1, \ldots, \mu^\star_{|Interlopers_j|}\right\} \rightarrow Interlopers_j,$$

mapping a fresh new marking terminal to every member of the set $Interlopers_j$. Then compile a grammar consisting of $\Gamma^H \cup_G \Gamma^E_j$ combined with a set of productions $mark(\mu^\star_i) \rightarrow \mu^\star_i$. If this compiles without conflicts, the validity of the proof is restored.

5. Discussion.

In this section we discuss some opportunities for future work based on the partitioning of the LR DFA described in Section 3 as well as some of the related work. We then describe our experience in building various language extensions and the limitations imposed by the modular *isComposable* analysis. Finally we comment on the importance of static analyses in the adoption of extensible languages and tools we have developed to support extensible languages.

5.1 Future work.

Parse table composition. The strict separation of the parse states described above suggests that it may be possible to compile an extension grammar into a parse table *fragment* that could be composed with the host language parse table (and other extension parse table fragments) at the direction of the programmer when he or she selects the set of extensions with which to extend the host language. Because of this strict separation, the items $h \rightarrow \bullet\mu^E_i s^E_i$ are the only additions to states in host language partition of the LR DFA M^C_H for the composed language. Thus, in those states, any new actions introduced by adding an extension are only added in the μ^E_i columns of the parse table. States associated with the extensions (in $M^C_{E_i}$) are entirely separate. It follows that, if $\Gamma^H \cup_G \Gamma^E_i$ has passed the modular test, one can take parse tables for M^{orig} and M^{E_i}, concatenate their rows, and add a new column μ^E_i with appropriate actions, one will have a parse table for $\Gamma^H \cup_G \Gamma^E_i$, verified correct and free of conflicts. Furthermore, one can concatenate a parse table for M^{orig} with those of *several* extensions, adding a new column for *each* marking terminal; the resulting parse table would then parse M^C and also be conflict-free.

Extension-specific lexical static precedence. Static lexical precedence, as used to specify that keywords take precedence over identifiers, is a convenient mechanism for disambiguating lexical syntax. However, as discussed in Section 4, its use is not recommended for indicating that extension introduced terminals have precedence over those defined in the host language, since such precedence specifications have effect in all parse states. The strict separation of parse states may also be useful here in that it would allow an extension-specific static precedence that only has effect in the parse states owned by the extension in which the keyword is defined.

5.2 Related work.

Context-aware scanning. In the TICS algebraic compiler framework (20) the notion of context is used in the pattern-matching parser and in the scanner. The scanner can take into account the results of the n previous scans in determining how to recognize the current input (21); the value of n is determined when the scanner is generated. This is a lexical notion of context and is more limited than parse-state-based context-aware scanning such as used in Copper, which provides contextual information based on an unbounded number of tokens to the left of the current point in the file. However, the context used in the TICS scanner is more general in two ways: the scanner also considers the context of what can *follow* the current token and it introduces the notion of *non-context* in which terminals can specify contexts in which they are *not* valid. These can be used, for example, to distinguish an integer constant terminal from a label. They may have the same regular expression, but the label has a following context of a colon and the integer has a colon in its following non-context specification (21).

The Tatoo parser and scanner generator (7) has two innovations of relevance here. First, it uses a *lookahead activator* implementing an independently developed notion of parse-state-based context aware scanning. However, the expressiveness of parse-state-based context-aware scanning appears not to have been fleshed out in Tatoo, as the lookahead activator is presented as a scanner optimization.

Separate parse table-based approaches. The second innovation of Tatoo is that it also supports rapid composition of extensions without the need for regenerating parse tables. The system can switch between different pre-compiled parse tables and thus support some notion of parse table composition. But Tatoo's concept of extensions is different from ours: while we conceive of a fully independent host grammar supplemented by an unspecified set of extensions, in Tatoo, certain "holes" are explicitly left in the host grammar, and users *must* "fill" each of these with one of a possible selection of extensions written to fill that particular "hole." Therefore, the extensions to a Tatoo grammar are not optional, are of a fixed number, and are of a more restricted character.

Component LR-parsing (30) (CLR) is similar to Tatoo's approach in that multiple separate parse tables are used, but CLR introduces two new actions: *switch* and *return*. When a component parser enters an *error* state it inspects the current state and will either switch to another component parser, return to the parser that called it, or backtrack. This point at which the calling parser fails is where, in our approach, a shift on a marking terminal would occur. The priorities of these new actions are fixed by the parsing algorithm and the order in which component parsers are called is determined by the textual-order in which they appear in the specification. Backtracking is used when a component parser fails and the system backtracks to try another component parser. It would be interesting to add backtracking to our approach, but limit it to the states at which a marking terminal is shifted. This would allow extensions with overlapping marking terminals, at the expense of backtracking.

Arbitrary parse table composition. Bravenboer and Visser (6) outline a strategy for composing the parse tables of *arbitrary* extensions into a single GLR (specifically, GLR(0)) table. This approach is based upon a construct called an "ϵ-NFA" — a nondeterministic LR(0) finite automaton that allows ϵ-transitions. ϵ-NFAs being very easy to glom together in a composition, they are made use of as an intermediate step in the process of producing composable parse tables. The ϵ-NFA for the host or a particular extension is determinized into an "ϵ-DFA," which is an ordinary DFA with the ϵ-transitions retained as metadata. This allows the addition of new items to an ϵ-DFA state (*i.e.*, the introduction of new extensions) without the need to recompute the entire closure of the state. Most of the information from the ϵ-DFA is then included with the parse table. The generality of this method is at once a strength and a weakness: although it is able to do on-the-fly composition of a host grammar with *any* extension, there is no way to guarantee that even *one* such extension, let alone several unrelated ones, will compose deterministically or without other issues. As most of this method concentrates on the potentially inefficient process of recomputing closures on-the-fly (entirely unneeded when using our approach of marking tokens) and ignores scanner issues (being designed for a scannerless GLR parser), its results, but not its methods, are similar to ours.

Incremental generation of LR parse tables. Many have studied the problem of incrementally generating LR parse tables. Horspool (15), for example, presents a similar method to Bravenboer's for addition and deletion of productions *in situ* in *deterministic* parse tables. However, Horspool's method was designed for interactive development of grammars, where a whole grammar is being modified and debugged all in one place and a monolithic determinism analysis would be of much more use.

5.3 Experience with the modular analysis restrictions.

We have built parsers and scanners in Copper for Java 1.4 and ANSI C and designed several language extensions to these host languages that pass the modular analysis. For example, to the Java 1.4 host language we have added the significant subset of SQL and the boolean-expressions tables mentioned in Section 1 (25). We have also implemented an extension that adds algebraic data types to Java in a manner similar to that of Pizza (19), specifying concrete syntax for defining different cases of a class and for pattern matching over them. Further examples include dimension-types used to check for errors in computations over physical measurements (*e.g.*, to check that a length measurement is not added to a mass or acceleration measurement (26)). All of these extensions pass the modular analysis *isComposable*; in fact, many of them were designed before the modular analysis was.

Thus, our experience shows that the restrictions imposed by *isComposable* are not too severe. That said, there are some limitations. For example, adding a new infix binary operator (*e.g.*@) to the host language is not allowed since a production of the form $Expr ::= Expr'@'Expr$ does not have a marking terminal. (New infix binary operators can, however, be specified in the languages defined in extensions.) Many extensible language frameworks (10; 24) do support type-based overloading of existing host language operators. Thus, adding a new numeric type (*e.g.*, complex or rational numbers) may require new syntax to define the type but no new syntax for arithmetic operators over these values.

If in the extension in Figure 1 we replaced the extension keyword `foreach` with the host language keyword `for`, we would not have a marking terminal in the bridge production of this extension and our analysis would thus reject this extension. This type of extension would be possible with traditional LALR(1) parsers, PEGs and GLR parsers. In our approach, one could also write the production using the host language *for* terminal, but now we must rely on the monolithic analysis to detect any conflicts. Thus, one does not lose determinism completely, but the ability of the extension writer to ensure it is lost. In this case one could also design the host language to support overloading of the enhanced for loop (similar to operator overloading). Such an overloading would be appropriate since the intention of iterating over each element returned from the query is consistent with the intuitive understanding of the construct. Thus the extension would not need to add new concrete syntax specifications. Therefore, one area of future work is to study how to best design host languages to support different types of overloading to mitigate this limitation to some degree.

AspectJ. We have also extended our Java 1.4 specification to create a specification for AspectJ, a language that provides aspect constructs in Java. This language has historically proven difficult to parse using traditional methods. The AspectJ Bench Compiler (`abc`) (14) uses a traditional LALR(1) parser, but uses a moded scanner that switches modes based on whether or not an aspect construct is being parsed. This allows different keywords to be reserved based on the scanner mode, but the hand-written mode-switching specifications are not declarative. More recently, Visser *et al.* (4) have devised a declarative parser for AspectJ in their nondeterministic scannerless-GLR framework — although they had to add a new feature, *grammar mix-ins*, to handle the problem of the different sets of keywords.

We have adapted the LALR(1) grammar used in `abc` to extend the Java grammar in our ableJ framework. As it happens, with a context-aware scanner, the `abc` version of AspectJ can be parsed deterministically and declaratively (22). Essentially, context-aware scanning provides a more fine-grained version of the mode-switching that is done manually in the `abc` scanner.

However, AspectJ does not pass our modular determinism analysis, for two reasons. First, AspectJ introduces large numbers of new keywords, placed in such a way that they are allowed to follow host Java constructs (*e.g.*, type constructs). This adds these terminals to the follow sets of host nonterminals and causes the test to fail. Second, AspectJ extends some Java host nonterminals that derive phrases beginning with an access modifier such as `public` or `protected`. For example, Java specifies the production

Dcl → **Modifiers Type** Id ...

for methods and AspectJ adds the production

Dcl → **Modifiers** $Aspect\ Id$ ...

to define certain aspect constructs. The new $Aspect$ keyword is not a marking terminal at the beginning of the productions right-hand side. However, the host Java grammar could be refactored, and the extension productions modified, so that they satisfy the requirements of *isComposable*. Thus writers of a host grammar may be able to increase the number of extensions that pass the analysis by designing the grammar in a particular way. Our Java 1.4 grammar was directly derived from the freely available JavaCup version (2) with no such modifications, and our extensions (except AspectJ) passed the analysis with that host grammar. Thus it seems that the way one naturally writes grammars does lead to a high degree of extensibility.

Alternate restrictions. The set of restrictions imposed by the *isComposable* analysis are not the only ones that we have considered. For example, we experimented with tighter but simpler restrictions defined on the grammar (25), instead of on the LR DFA. One required beginning and ending marking terminals. However, these proved too restrictive to admit many of our previously implemented extensions. We also considered relaxing, but complicating, the restrictions as follows: the analysis would, given a subset A of host nonterminals, only guarantee that the extension would compose if all the other extensions with which it was composed had a bridge production with its left-hand side in A. This exploited the fact that few host nonterminals would be used on left-hand sides of bridge productions (e.g., ”expression” and ”statement”). However, it is unclear if this is worth the added complexity.

5.4 Restrictions on expressiveness versus safe composition.

It may still be asked, *Are the restrictions imposed by this analysis too severe?* We argue that the importance of a static analysis, *performed by the extension writer*, outweighs the moderate loss of expressibility imposed by the modular analysis restrictions. Other parsing techniques that support language extension do allow some constructs not allowed by our analysis and it is appropriate to compare these approaches, as we have. It is also appropriate to compare all of these approaches to the accepted mechanism that programmers currently use to “extend” their language with new abstractions: *libraries*. They provide no new syntactic constructs (or new semantic analysis), but because the library writer can compile and type-check the code before it is distributed, *the programmer is assured that he or she can pick any combination of libraries needed* *to address the particular problem, and use them as needed in a program.* It is this level of assurance that we seek, and our approach provides, allowing a wide range of new expressive syntactic constructs to be safely added to the host language.

The requirements for truly extensible languages are different from those for traditional language design in which a language expert is expected to understand the language and the parsing and scanning technology. Here, one may reasonably choose to use a GLR parser to simplify the grammar rules and accept the responsibility of extensively testing and manually analyzing the grammar to ensure that no ambiguities exist at the top-level of the grammar. For PEGs, though it is unlikely that two extensions will introduce the exact same syntax, it is possible, and then the order in which the extensions are added will determine which construct is recognized by the PEG parser and must be managed by someone familiar with PEGs.

If extensible languages are to become widely used, we need static analyses that let extension writers *certify* their language extensions to provide a *guarantee* that language extensions can be *safely* composed by the non-expert programmer.

5.5 Tool support.

Copper is an integrated LALR(1) parser and context-aware scanner generator that we developed to address the challenges in parsing and scanning extensible languages (27). Copper also implements the modular analysis *isComposable* described in this paper. Copper serves as the parser and scanner generator for our attribute grammar system, Silver (24), which was used to implement ableJ (25).

Copper, Silver, and the host language and language extension specifications mentioned in this paper are available on the web at `http://melt.cs.umn.edu`.

Acknowledgments

We thank the anonymous reviewers for their helpful and insightful comments. This work was partially funded by the McKnight Foundation and by the National Science Foundation under grants #0347860 and #0429640.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] S. Ananian. Java 1.4 LALR(1) grammar. Available at `http://www2.cs.tum.edu/projects/cup/`.

[3] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In *Proc. of the Intl. Conf. on Generative programming and component engineering (GPCE)*, pages 3–12. ACM, 2007.

[4] M. Bravenboer, Éric Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. In *Proc. of Conf. on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 209–228. ACM, 2006.

[5] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 365–383. ACM, 2004.

[6] M. Bravenboer and E. Visser. Parse table composition - separate compilation and binary extensibility of grammars. In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, 2008.

[7] J. Cervelle, R. Forax, and G. Roussel. Tatoo: an innovative parser generator. In *Proc. Principles and practice of programming in Java (PPPJ)*, pages 13–20. ACM, 2006.

[8] R. Cox, T. Bergany, A. T. Clements, F. Kaashoek, and E. Kohlery. Xoc, an extension-oriented compiler for systems programming. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[9] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proc. Conf. on Object oriented programming systems and applications (OOPSLA)*, pages 1–18. ACM, 2007.

[10] T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69:14–26, December 2007.

[11] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proc. of Symp. on Principles of Programming Languages (POPL)*, pages 111–122. ACM, 2004.

[12] R. Grimm. Better extensibility through modular syntax. In *Proc. of Conf. on Programming Language Design and Implementation (PLDI)*, pages 38–51. ACM Press, 2006.

[13] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. of Tenth Annual Conf. on Computer Assurance (COMPASS)*, 1995.

[14] L. Hendren, O. de Moor, A. S. Christensen, and the abc team. The abc scanner and parser, including an LALR(1) grammar for AspectJ. Available at `http://abc.comlab.ox.ac.uk/documents/scanparse.pdf`, September 2004.

[15] R. N. Horspool. Incremental generation of LR parsers. *Computer Languages*, 15(4):205–223, 1990.

[16] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[17] W. R. LaLonde. An efficient LALR parser generator. Technical Report 2, Computer Systems Research Group, University of Toronto, 1971.

[18] N. Nystrom, M. R. Clarkson, and A. C. Myer. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Conf. on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152. Springer-Verlag, 2003.

[19] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *Proc. of Symp. on Principles of Programming Languages (POPL)*, pages 146–159. ACM Press, 1997.

[20] T. Rus. A unified language processing methodology. *Theoretical Computer Science*, 281(1-2):499–536, 2002.

[21] T. Rus and T. Halverson. A language independent scanner generator. Paper available at `http://www.uiowa.cs.edu/~rus`, 1998.

[22] A. Schwerdfeger. A declarative specification of a deterministic parser and scanner for AspectJ. Technical Report 09-007, University of Minnesota, 2009. Available at `http://www.cs.umn.edu`.

[23] A. Schwerdfeger and E. Van Wyk. Verifiable composition of deterministic grammars. Technical Report 09-008, University of Minnesota, 2009. Available at `http://www.cs.umn.edu`.

[24] E. Van Wyk, D. Bodin, L. Krishnan, and J. Gao. Silver: an extensible attribute grammar system. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 203(2):103–116, 2008. Originally in LDTA 2007.

[25] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *European Conf. on Object Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer-Verlag, July 2007.

[26] E. Van Wyk and Y. Mali. Adding dimension analysis to java as a composable language extension. In *Post Proc. of Generative and Transformational Techniques in Software Engineering (GTTSE)*, number 5235 in LNCS, pages 442–456. Springer-Verlag, 2008.

[27] E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*. ACM Press, October 2007.

[28] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, Aug. 1997.

[29] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.

[30] X. Wu, B. R. Bryant, J. Gray, and M. Mernik. Component-based LR parsing. *Computer Languages, Systems & Structures*, 2009. In press.