Verifiable Parse Table Composition for Deterministic Parsing*

August Schwerdfeger and Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN
{schwerdf,evw}@cs.umn.edu

Abstract. One obstacle to the implementation of modular extensions to programming languages lies in the problem of parsing extended languages. Specifically, the parse tables at the heart of traditional LALR(1) parsers are so monolithic and tightly constructed that, in the general case, it is impossible to extend them without regenerating them from the source grammar. Current extensible frameworks employ a variety of solutions, ranging from a full regeneration to using pluggable binary modules for each different extension. But recompilation is time-consuming, while the pluggable modules in many cases cannot support the addition of more than one extension, or use backtracking or non-deterministic parsing techniques.
We present here a middle-ground approach that allows an extension, if it meets certain restrictions, to be compiled into a parse table fragment. The host language parse table and fragments from multiple extensions can then always be efficiently composed to produce a conflict-free parse table for the extended language. This allows for the distribution of deterministic parsers for extensible languages in a pre-compiled format, eliminating the need for the "source code" of the grammar to be distributed. In practice, we have found these restrictions to be reasonable and admit many useful language extensions.

1 Introduction

In parsing programming languages, the usual practice is to generate a single parser for the language to be parsed. A well known and often-used approach is LR parsing [**?**] which relies on a process, sometimes referred to as grammar compilation, to generate a monolithic *parse table* representing the grammar being parsed. The LR algorithm is a generic parsing algorithm that uses this table to drive the parsing task. However, there are cases in which it is desirable to generate different portions of a parser separately and then put them together without any further monolithic analysis. An example can be found in the case of extensible programming languages, wherein a *host* language such as C or Java is composed with several *extensions*, each possibly written by a different party.

* This work was partially funded by the National Science Foundation grants #0347860 and #0429640.

This is done by distributing the grammar of the host language for use in writing language extension grammars by extension developers. Finally, the user of the extensible language will collect the set of grammars defining the desired language

```
connection tripdb with table trip_log ;
class TripLogData {
 boolean examine_trips ( ) {
   rs = using tripdb query { SELECT dist, time FROM trips
                                                WHERE time > 600 } ;
   boolean res = false ;
   foreach (int dist, int time) in rs {
     Unit<time 1 sec, int> t = time ;
     Unit<length 1 meter, int> d = distance ;
     Unit<length 1 meter  time -1 sec, int> r = d / t ;
     res = res || table ( t < 3600  : T F
                          d > 10000 : F * ) }
   return res ;
 }
}
```

Fig. 1. Code written in an extended version of Java using the SQL, foreach, tables, and dimension analysis extensions.

extensions and provide them to a compiler generator that merges the host and extension grammars and builds a monolithic parser from this composition. Of course, extensible languages must also provide a means for describing the semantics of the language extensions. This can be accomplished by rewriting rules as in MetaBorg [**?**] or by composable attribute grammar specifications [**?**,**?**,**?**], but these are not discussed in this paper.

Consider the small, slightly contrived, sample program in Figure 1 written in a version of Java to which five extensions have been added [**?**]. The first and second add the `using ... query ...` and `connection ...` constructs, respectively, to extend Java with the database query language SQL, for static detection of syntax and type errors in the query. The import-like `connection` construct sets up the connection to the database and retrieves database type schemas to type-check the query. (In practice, these two extensions are bundled in a single SQL-embedding package, but grammatically remain separate to satisfy the restrictions placed on extensions, which are discussed further below.) The third extension is a facsimile of the *foreach* loops added in Java 1.5. A fourth extension implements dimension and unit analysis to ensure that values representing physical measurements are used correctly. In this program it copies the integer values for time and distance into variables associated with dimension and unit information; *e.g.*, `d` represents a length in meters and `r` represents a rate in meters per second (note the negative exponent for time). This extension overloads arithmetic operators for these types to check that units of measurement are used

correctly and that no attempts are made to, *e.g.*, add a length in meters to a length in feet or add a time to a distance.

The fifth extension adds a construct for representing boolean conditions in a tabular form, inspired by similar constructs in modeling languages such as RSML^{-e} [**?**]. The keyword `table` is followed by rows 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 `t < 3600` is true and `d > 10000` is false, or if `t < 3600` is false (the value of `d` does not matter). This extension checks that each row has the same number of truth-indicators and that the expression in each row is of type `boolean`. 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 typecheck the extension constructs.

Ideally, these extensions can be developed by separate parties, unaware of each other's extensions, and the non-expert programmer would be provided with some mechanism to compose the host language (Java in this case) automatically with the language extensions. This approach requires monolithic composition of the grammars and thus has some significant problems: (1) This monolithic composition allows little room for failure. For example, the composed grammar may contain ambiguities or the generated LR parse table may contain conflicts. (2) It requires that the grammar's "source code" be released to any end user of the extensible system. This may be undesirable if an extension writer or the writer of the host grammar only wish to release the grammars in binary form (*i.e.*, the parse tables). (3) The parser generation process is too time consuming for some applications.

Our previous work [**?**] addressed the first problem by presenting some reasonable restrictions on the form and substance of the extension grammars, which if adhered to, provide a guarantee to each extension writer that the monolithic grammar compilation of any composition of extensions that meet the restrictions will not fail. Thus, in that approach, the monolithic grammar compilation by the end-user can be automated, and is guaranteed to result in a conflict-free LALR(1) parser, but is still time-consuming and requires the user to be in possession of the source code for all constituent grammars.

Contributions. In this paper we present a corollary of that work, based on a relaxed version of the same grammar restrictions, that allows parse tables to be composed *after* they have been compiled, provided that a modest amount of additional metadata is provided with each parse table. We also specify the process of composing the host language parse table and extension parse table fragments, and provide a time-complexity analysis of this process. Finally, we describe a technique for using a different scanner for each language extension and for the host language, obviating the problem of generating a new scanner from the composed language specifications.

The structure of the rest of this paper is as follows. Section 2 discusses background, including formal definitions of grammars and parse tables, examples of useful extensions, and a summary of context-aware scanning. Section 3 discusses

the restrictions needed on the extension grammars (which we have found to be broad enough to admit many interesting extensions to Java, including all those mentioned above). Section 4 discusses the specifics of the parse table composition, including the metadata that needs to be retained. Section 5 discusses specific data structures that can be used to expedite the composition process, as well as providing a time-complexity analysis for it. Section 6 discusses related work, and section 7 concludes.

2 Background.

This section defines the notations used in the remainder of the paper for constructs used in LR-style parsing: grammars, LR finite automata, parse tables, etc. It also discusses some examples of language extensions and provides some background on context aware scanning [**?**], upon which our approach relies.

2.1 Formal definitions of the extension system.

Host and extension grammars.

Definition 1. (*Language grammar*) A language grammar is defined as a 5-tuple $\langle T,\ NT,\ P,\ s \in NT,\ regex\colon T \rightarrow Regex\rangle$. As expected, T is the finite set of terminal symbols, NT is the finite set of nonterminal symbols, P is the finite set of productions of the form $NT \rightarrow (T \cup NT)^*$, and s is the start symbol. *regex* is a mapping that associates a regular expression (over some alphabet) with each terminal symbol.[1] We use CFG_L to denote the set of all language grammars.

Definition 2. (Γ^H) We use the symbol $\Gamma^H = \langle T_H, NT_H, P_H, s_H, regex_H\rangle$ to designate the host language grammar.

For extensions, we use a slightly modified grammar form; instead of a start nonterminal, extensions have a *bridge* production connecting them to their host language.

Definition 3. (*Extension grammar*) An extension grammar is defined as a 5-tuple $\Gamma^E = \langle T_E, NT_E, P_E, nt_H \rightarrow \mu_E s_E, regex_E\rangle$ where $s_E \in NT_E$, $nt_H \in NT_H$, and $dom(regex_E) = T_E \cup \{\mu_E\}$. Let CFG_E be the set of all extension grammars.

The production $nt_H \rightarrow \mu_E s_E$ is the bridge production, its left hand side being a host language nonterminal and its right hand side the extension's *marking terminal* (μ_E) — a terminal introduced by the extension but not in T_E. It is followed by a nonterminal (s_E) in NT_E, which thus becomes the extension's start symbol. It is possible to allow more than one bridge production, each with a distinct marking terminal, and also allow any non-empty sequence of host and

[1] Generally, the regular expressions associated with terminals are not included in context-free grammars in this manner. We include them because when using a context aware scanner, the scanner and parser are tightly coupled and both are generated from this type of specification.

extension terminals to follow the marking terminal. However, we have taken the more restrictive approach to simplify presentation. This simplified approach does still allow host language nonterminals on the right hand side of extension productions. If $p_E \in P_E$, its left hand side symbol must be in NT_E, but symbols on its right hand side need only be in $T_H \cup NT_H \cup T_E \cup NT_E$.

Definition 4. (Γ^E *extends* Γ^H) If (and only if) Γ^E satisfies the restrictions of Definition 3 with respect to Γ^H, we say that Γ^E *extends* Γ^H.

Note that the term "grammar" will be used without the adjectives "host," "language," or "extension" when the context makes the distinction clear.

Parse tables.

Definition 5. (*States*) Let *States* denote the set of all rows of a parse table.

Definition 6. (*Parse table*) A *parse table* is a 4-tuple $PT = (\Gamma^{PT}, States_{PT}, \pi_{PT}, \gamma_{PT}, n^S_{PT} \in States_{PT})$. Γ^{PT} is a grammar that this parse table parses correctly; $\pi_{PT} : States_{PT} \times T \to \mathcal{P}(Actions)$, where $Actions = \{accept\} \cup \{reduce(p) \colon p \in P\} \cup \{shift(x) \colon x \in States\}$; $\gamma_{PT} : States_{PT} \times NT_{PT} \to \mathcal{P}(Goto)$, where $Goto = \{goto(x) : x \in States\}$. n^S_{PT} is the start state.

Note that parse table fragments for extensions can refer to states in the host language parse table. Also, the grammar Γ^{PT} is mentioned here only to reference components of it (*e.g.*, T^{PT}) used in parse table construction; it is not necessary to distribute it along with the parse table.

Definition 7. (*Error action*) An error action is represented by an empty cell in the parse table; *i.e.*, when for some parse table pt, some $n \in States$ and $t \in T$, $\pi_{pt}(n, t) = \emptyset$.

Definition 8. (*Parse table conflicts*) 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 no cells in its row have a conflict; *i.e.*, $\forall t \in T_{PT}.(|\pi_{PT}(n, t)| \leq 1)$. A parse table pt is *conflict-free* if all $n \in States_{PT}$ are conflict-free.

Definition 9. ($\cup_G$ *and* $\cup^*_G$) Let $\cup_G : CFG_L \times CFG_E \mapsto CFG_L$ be a non-commutative, non-associative operation on context-free grammars. If Γ^E extends Γ^H, then $\Gamma^C = \Gamma^H \cup_G \Gamma^E = \langle T_C, NT_H \cup NT_E, P_C, s_H, regex_C \rangle$ where:

- $T_C = T_H \cup T_E \cup \{\mu_E\}$, μ_E being the extension's marking terminal.
- $P_C = P_H \cup P_E \cup \{nt_H \to \mu_E s_E\}$, ($nt_H \to \mu_E s_E$ is the Γ^E bridge production).
- $regex_C(t) = \begin{cases} regex_H(t) \text{ if } t \in T_H \\ regex_E(t) \text{ if } t \in T_E \text{ or } t = \mu_E \end{cases}$

$\cup^*_G$ is an operation on a host grammar and an unordered set of extensions, generalizing in the intuitive manner.

Java:

Terminals: **Question** `/?/`, **Colon** `/:/`, **Comma** `/,/` **Semi** `/;/`,
LParen `/(/`, **RParen** `/)/`, **LBrk** `/{/`, **RBrk** `/}/`,
LT `/</`, **GT** `/>/`, **Id** `/[A-Za-z][A-Za-z0-9]*/`
Nonterminals: *Expr*, *PrimaryExpr*, *Dcl*, *Type*

Expr	→	*Expr* **Question** *Expr* **Colon** *Expr*
Expr	→	*PrimaryExpr*
PrimaryExpr	→	**Id**
Type	→	... Java type expressions ...

SQL Connection:

Terminals: **Connection** `/connection/`, **SqlId** `/[A-Za-z]+/` ,
With `/with/`, **Table** `/table/`
Nonterminals: *ConnDcl*

Dcl	→	**Connection** *ConnDcl*
ConnDcl	→	**SqlId With Table SqlId Semi**

SQL Query:

Terminals: **Using** `/using/`, **Query** `/query/`, **Select** `/SELECT/`,
From `/FROM/`, **Where** `/WHERE/`, **SqlId** `/[A-Za-z]+/`
Nonterminals: *Sql*, *SqlQ*, *SqlIds*, *SqlExpr*

Expr	→	**Using** *Sql*
Sql	→	**SqlId Query LBrk** *SqlQ* **RBrk**
SqlQ	→	**Select** *SqlIds* **From SqlId Where** *SqlExpr*
SqlIds	→	**SqlId**
SqlIds	→	**SqlId Comma** *SqlIds*
SqlExpr	→	...

Tables:

Terminals: **Tbl** `/table/`
Nonterminals: *BTable*, *TRows*, *TRow*, *TFStarList*

PrimaryExpr	→	**Tbl** *BTable*
BTable	→	**LParen** *TRows* **RParen**
TRows	→	*TRow*
TRows	→	*TRow TRows*
TRow	→	*Expr* **Colon** *TFStarList*

Dimension analysis:

Terminals: **Units** `/Unit/`
Nonterminals: *DimensionExpr DimS*

Type	→	**Units** *DimS*
DimS	→	**LT** *DimensionExpr* **Comma** *Type* **GT**
DimensionExpr	→	... list of dimension and unit specifications

Fig. 2. Sample grammar productions from host and extensions (adapted from [**?**]).

2.2 Grammar examples.

Figure 2 shows some of the nonterminals, terminals, and productions declared in the grammars for Java and the SQL, tables, and dimension analysis extensions. This is Java 1.4, to which a notion of generic types has been added. The SQL Query extension's marking terminal is **Using**, with the regular expression `/using/`; the SQL Connection extension's, **Connection**; the tables extension's, **Tbl**; the dimension analysis extension's, **Units**. The SQL Query extension's start symbol is *Sql*; the tables extension's, *BTable*; the dimension-analysis extension's, *DimS*. Except for a few terminals, the SQL Query extension does not use any host language constructs. However, Java expressions form a part of table rows (*TRow*) in the tables extension, while Java types form a part of type expressions in the dimension-analysis extension. Thus, it is syntactically (but not semantically) correct for an SQL query to appear at the beginning of a table row. In practice, extension start nonterminals are not necessary; in the case of the Tables extension we use the production *PrimaryExpr* → **Tbl LParen** *TRows* **RParen** instead.

2.3 Context-aware scanning.

The context-aware scanners discussed here are identical to those described in [**?**], an extension of the basic idea found in [**?**] and implemented in *Copper*, our parser and context-aware scanner generator. However, while in discussing the composition of grammars the exact building process for the scanner does not matter, it is of more relevance here, as discussed in section 4.2. At each scan, a context-aware scanner will only match terminals in this *valid lookahead set*; this is the set of terminals that have valid parse actions in the current parse state (*i.e.*, shift, reduce, or accept, but not error). For example, in Fig. 1, the scanner will recognize "`table`" as either a SQL keyword or a Tables keyword, based on the context of the parser. Similarly, "`time`" is recognized as either an identifier or dimension specifier (as in the type of `t` and `r`) based on the current parse state. This can be achieved by building a different scanner for each parse state, but this is inefficient so in practice the valid lookahead set information is "compiled into" the scanner DFA and the parser then passes the valid lookahead set for the current parse state to the scanner when it is called for the next token.

Context-aware scanning is useful in modular composition of grammars, because with a different valid lookahead set for each parse state, the partitioning of the LR DFA into disjoint sections for each extension, as described below, also partitions the lexical syntax of extensions and ensures that they do not interfere with each other lexically.

3 Extension restrictions for parse table composition.

Previously, we defined an analysis, $isComposable(\Gamma^H, \Gamma^{E_i})$, that language extension developers can apply to their extension grammars, in effect "certifying"

the extension's grammar as being safe to compose with other identically certified extensions [**?**]. This can be formulated as follows:

$$(\forall i \in [1,n].[\mathit{isComposable}(\Gamma^H, \Gamma^{E_i}) \wedge \mathit{conflictFree}(\Gamma^H \cup_G \Gamma^{E_i})]) \\ \implies \mathit{conflictFree}(\Gamma^H \cup_G^\star \left\{\Gamma^{E_1}, \ldots, \Gamma^{E_n}\right\})$$

If each extension grammar Γ^{E_i} passes this analysis, and results in a conflict-free parse table when combined with the host language, then the composition of the host language and all the extensions $\Gamma^{E_1}, \ldots, \Gamma^{E_n}$ passing this analysis will also result in a conflict-free parse table.

In this paper we introduce a very similar analysis, $\mathit{isComposable}_{PT}(\Gamma^H, \Gamma^{E_i})$. If the analysis passes, it guarantees that the portions of the parse table for $\Gamma^H \cup_G \Gamma^{E_i}$ specific to the extension can be extracted and distributed as a parse table fragment. The user of the extensible language can then direct the supporting tool to *compose parse tables* instead of grammars. The resulting parse table is conflict free. This analysis can be formulated as follows:

$$(\forall i \in [1,n].[\mathit{isComposable}_{PT}(\Gamma^H, \Gamma^{E_i}) \wedge \mathit{conflictFree}(\Gamma^H \cup_G \Gamma^{E_i})]) \\ \implies \mathit{conflictFree}(PT^H \cup_T^\star \left\{PT^{E_1}, \ldots, PT^{E_n}\right\})$$

The parse table merging operation $\cup_T^*$ is defined in Section 4; in this section we focus on the restrictions imposed by $\mathit{isComposable}_{PT}$. These restrictions are less restrictive than (in fact, a subset of) those imposed by *isComposable*, since the process of merging parse tables produces a parse table that is not quite so monolithic as that produced by compiling $\Gamma^C = \Gamma^H \cup_G^\star \left\{\Gamma^{E_1}, \ldots, \Gamma^{E_n}\right\}$ directly; this is described in more detail below. We first present some background definitions of relations on states of LR DFAs, then the restrictions imposed by *isComposable*, and finally the relaxation that defines $\mathit{isComposable}_{PT}$.

3.1 Definitions relating LR finite automata states.

Definition 10. (*LR item, lookahead*) Formally, with respect to a grammar Γ, an *LR item* is a pair $(p, n) \in P^\Gamma \times \mathbb{N}$ representing a production in the grammar and a marker that can be placed before or after any symbol on the production's right hand side. Therefore, the minimum value for n is zero, and the maximum is the number of symbols on the right hand side.

In LALR(1) DFAs, an LR item is always accompanied by a set of *lookahead*, represented formally as a map $la : \mathit{Items} \to \mathcal{P}(T^\Gamma)$. An LR item is usually written with the marker in place rather than specified by a number, with the lookahead after it. For example, the LR item $i = (A \to \alpha, 0)$ with $la(i) = z$ is written $A \to \bullet\alpha, z$, and the LR item $(A \to \alpha\beta, 2)$ is written $A \to \alpha\beta\bullet, z$.

In the following s and t are LR DFA states.

Definition 11. (*I-subset,* $\subseteq_\mathrm{I}$) s is an *I-subset* of t, written $s \subseteq_\mathrm{I} t$, if $\mathit{Items}_s \subseteq \mathit{Items}_t$, where the sets *Items* represent all LR items in their respective states.

Definition 12. (*LR(0)-equivalence,* $\equiv_0$) s and t are *LR(0)-equivalent*, written $s \equiv_0 t$, if $s \subseteq_{\mathrm{I}} t$ and $t \subseteq_{\mathrm{I}} s$, *i.e.*, the two states 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.

Definition 13. (*IL-subset,* $\subseteq_{\mathrm{IL}}$) s is an *IL-subset* of t, written $s \subseteq_{\mathrm{IL}} t$, if $s \subseteq_{\mathrm{I}} t$ and $\forall i \in Items_s.\, (la_s(i) \subseteq la_t(i))$.

Definition 14. (*LR(1)-equivalence,* $\equiv_1$) s and t 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.

3.2 The *isComposable* restrictions.

The analysis consists of checking the forms of the host grammar

$$\Gamma^H = (T_H, NT_H, P_H, s_H, regex_H)$$

and the extension grammar

$$\Gamma^{E_i} = \left(T_{E_i}, NT_{E_i}, P_{E_i}, nt_H \rightarrow \mu_i^E s_i^E, regex_{E_i}\right)$$

and then compiling two LR DFAs — M^{orig} from Γ^H and M^{E_i} from $\Gamma^H \cup_G \Gamma^{E_i}$ — to verify that they conform to certain restrictions, which serve to partition the LR DFA (and, by extension, the derivative parse table) into separate sections for parsing host constructs and extension constructs; this guarantees that when Γ^H is composed with any number of extensions that meet the restrictions into $\Gamma^C = \Gamma^H \cup_G^\star \left\{\Gamma^{E_1}, \ldots, \Gamma^{E_n}\right\}$, the resulting LR DFA M^C will be able to be partitioned in an analogous manner and its derivative parse table will also be conflict-free.

The first restriction is that an extension must contain only one production with a host-language nonterminal on its left hand side, called the *bridge production*, of the form $nt_H \rightarrow \mu_i^E s_i^E$, where μ_i^E is a unique *marking terminal* and s_i^E is a nonterminal defined by Γ^{E_i}. That the extensions satisfy this restriction can be seen in Figure 2.

The second restriction is that the follow sets of the nonterminals in NT_H are identical in $\Gamma^H \cup_G \Gamma^{E_i}$, disregarding the marking terminal μ_i^E. The tables extension satisfies this because anytime it derives *Expr*, the expression is followed by a colon, and the follow set of *Expr* already contains the colon in the host language (on account of the `?:` conditional expression). Similarly, in the dimension analysis example the *Type* nonterminal is always followed by the greater-than symbol.

The third restriction is that every state in the LR DFA M^{E_i} must be able to be placed into one of three classes of states:

1. $M_H^{E_i}$ — states "owned" by the host grammar. A state is in this class if, disregarding bridge production items ($nt_H \rightarrow \bullet \mu_i^E s_i^E$) and the marking terminal, it is identical to some state in M^{orig}.

2. $M_{E_i}^{E_i}$ — states owned by the extension grammar. A state is in this class if it contains some item with a nonterminal $nt_{E_i} \in NT_{E_i}$ on its left hand side.
3. $M_{NH}^{E_i}$ — states that are not owned by either host or extension. A state n is in this class if it cannot be placed in the other two, and (disregarding marking terminals and bridge productions):
   - there is a state $n' \in M_H^{E_i}$ such that $n \subseteq_{\text{IL}} n'$, and
   - for every state $n_H \in M_H^{E_i}$, if $n \subseteq_{\text{I}} n_H$, then $n \subseteq_{\text{IL}} n_H$.

These restrictions and partitions of states ensure that when the host and several extensions are composed and built into the LR DFA M^C, this DFA can be split into analogous partitions M_H^C, M_{NH}^C, and $M_{E_i}^C$ for each Γ^{E_i}.

It is the third class $M_{NH}^{E_i}$ of states in the third restriction (the "new-host" states) with which we are most concerned here. It is possible that the same host-language construct can be derived from more than one extension: if, for example, two extensions to Java each embed Java expressions. The classes $M_{E_i}^{E_i}$ cannot share any states because, since they contain items with extension nonterminals on the left hand side, no pair of states in $M_{E_i}^{E_i} \times M_{E_j}^{E_j}$ will have identical item sets (be LR(0)-equivalent). This is not true of the classes $M_{NH}^{E_i}$, which may have LR(0)-equivalent states. When building an LALR(1) DFA, all LR(0)-equivalent states are merged; those pairs of states in $M_{E_i}^{E_i} \times M_{E_j}^{E_j}$ that have identical item sets may be merged into new states $n \in M^C$ with the same item sets. This restriction ensures that this new state does not produce parse-table conflicts.

3.3 The *isComposable*$_{PT}$ restrictions.

When composing parse tables instead of grammars, to avoid combining parse table rows, the states in the third class, $M_{NH}^{E_i}$, will not be merged, but will remain separate as they would in an LR(1) DFA. Therefore, the restrictions on the third class (item 3 in the enumerated list above) are not required and can be safely dropped without affecting the conflict-free composability guarantee provided by the analysis; it can simply be those states that are not in $M_H^{E_i}$ but contain no item with an extension nonterminal on its left hand side.

An example is our Java extension for dimension analysis, in which some Java constructs are embedded. This causes states containing no items of extension syntax to be made, but since the Java constructs are being used outside their normal syntactic context, these states do not fit into the $M_H^{E_i}$ class. However, they also happen to fail the criteria for the $M_{NH}^{E_i}$ class and thus this grammar fails the analysis *isComposable*.

The restrictions of $M_{NH}^{E_i}$ are meant to guarantee that if, by chance, another extension should generate a state that is LR(0)-equivalent to some state in that class, a conflict will not be introduced when merging these two LR(0)-equivalent states during grammar compilation. But in this case, the states remain separate in their different extension parse tables and are not merged at all, obviating the need for the restrictions. This means that the dimension analysis extension, which *only* fails the test on $M_{NH}^{E_i}$, passes the analysis *isComposable*$_{PT}$.

LALR(1) parsers are preferred to LR(1) parsers precisely because they do this merging, because LR(1) parse tables having not been compacted this way are much larger. But the circumstances in which mergable states remain unmerged in this case are uncommon; two extensions produce LR(0)-equivalent new-host states rather infrequently, and the restrictions of $M_{NH}^{E_i}$ are in place to handle corner cases rather than common ones, so this duplication will not produce significant overhead.

4 Merging parse tables and scanners.

In this section we describe the process of merging parse tables and parse table fragments, the metadata that is required in this process, and how to create separate scanners to use with the separately created parse tables.

4.1 Merging parse tables.

The restrictions of *isComposable* and $isComposable_{PT}$, as described above, enforce a strict separation of states that are used for parsing the host language and those that are used for parsing extensions. Consequently, the items $nt_H \rightarrow \bullet\mu_i^E s_i^E$ are the only additions to states in M_H^C, the host language states in the LR DFA for the composed language. Thus, in the parse states corresponding to these host language states, it is only in the columns for extension marking terminals (μ_i^E) that any new actions need to be introduced when adding an extension — states specific to an extension E_i, in $M_{E_i}^C$, are entirely separate. It follows that if $isComposable_{PT}$ ($\Gamma^H \cup_G \Gamma^{E_i}$) then one can take the rows for the parse table for M^{orig} and the parse table rows for extension states (corresponding to state in $M_{E_i}^{E_i}$ and $M_{NH}^{E_i}$) from M^{E_i} and concatenate them, add a new column μ_i^E with appropriate actions, and end up with 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. The goto-table (columns labeled by nonterminals) do not change during composition and can be composed without any modifications. Figure 3 provides a graphical representation of how the composed parse table is put together.

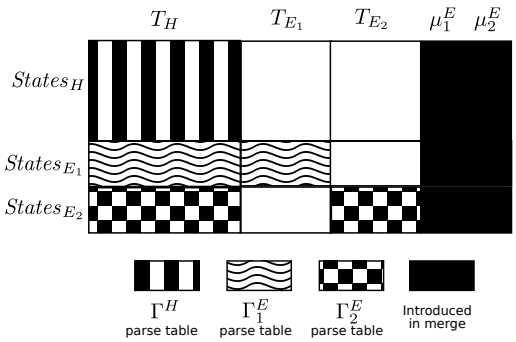

Fig. 3. A graphical representation of the source of all actions in the composed parse table.

Definition 15. (PT_i^E) With respect to a composed parse table PT, the parse table fragment for an extension Γ^{E_i}, labeled PT_i^E, consists of those rows in the parse table corresponding to states $M_{E_i}^{E_i}$ and $M_{NH}^{E_i}$, *i.e.*, the rows that "belong" to Γ^{E_i}.

Definition 16. (*The parse table merge operation* $\cup_T$) Let $\cup_T : ParseTables \times ParseTables \rightarrow ParseTables$ be the parse table composition operator analogous to $\cup_G$, the grammar composition operator. ($PT^H \cup_T PT^E$) is defined iff Γ^E extends Γ^H. Then the composed parse table $PT^H \cup_T PT^E = (\Gamma^H \cup_G \Gamma^E, States_C, \pi_C, \gamma_C, n_H^S)$, where:

- $States_C = States_H \cup States_E$.
- $\pi_C(n,t) = \begin{cases} \pi_H(n,t) & \text{if } n \in States_H \\ & \text{and } t \in T_H \\ \pi_E(n,t) & \text{if } n \in States_E \\ & \text{and } t \in T_H \cup T_E \\ \pi_\mu(n,t) & \text{if } t = \mu_E \end{cases}$

  where π_μ (see Definition 19 below) represents all *new* parse actions, *viz.*, shifts on marking terminals, reductions on the bridge productions, and reductions on marking-terminal lookahead.
- $\gamma_C(n,t) = \begin{cases} \gamma_H(n,nt) & \text{if } n \in States_H \\ & \text{and } nt \in NT_H \\ \gamma_E(n,nt) & \text{if } n \in States_E \\ & \text{and } t \in NT_H \cup NT_E \end{cases}$.

Additional metadata needed for composition. Some metadata not stored in parse tables is needed to add new shift and reduce actions for computing π_μ. Thus, we briefly review LALR(1) construction to clarify its definition.

How LALR(1) shift actions are put into the parse table. According to the usual rules for constructing LALR(1) parse tables, a shift action is put into table cell (n,t) when an item $A \rightarrow \alpha \bullet t\beta, z$ is in state n of the LR DFA from which the table was built. If α is empty (*i.e.*, the item is of the form $A \rightarrow \bullet t\beta, z$), the shift action in question will only show up if some item $B \rightarrow \bullet A\gamma, y$ is also present in state n. The nonterminals A in such items need to be kept track of for each state.

Definition 17. (*initNTs*) Let $initNTs : States \rightarrow \mathcal{P}(NT)$ represent these nonterminals (A above) for each state.

How LALR(1) reduce actions are put into the parse table. Again according to the usual rules, a reduce action $Reduce(A \rightarrow \alpha)$ is put into table cell (n,t) when an item in $A \rightarrow \alpha\bullet, z$, with $t \in z$, is in state n of the LR DFA. The lookahead sets are in turn derived from the union of the *first* sets of several nonterminals. The precise method of calculation is specified in the closure rule for building

LR(1) DFA states. If there is an item $A \to \alpha \bullet X\beta, z$ already in a state, items for all productions with X on the left hand side are added to the state, and are given lookahead consisting of all terminals in the *first* set of β. If β is nullable, the lookahead in z is passed on as well. If this β is a nonterminal, it needs to be kept track of so that if it turns up on the left hand side of a bridge production, the marking terminal can be added to the appropriate lookahead sets.

Definition 18. (*laSources*) Let $laSources : States \times NT \mapsto \mathcal{P}(P)$ represent these sources for each reduce action in a state; specifically, if the nonterminal nt has been found to be a source of the lookahead in the item $A \to \alpha\bullet$ in the state n, as described above, then $A \to \alpha$ will be placed in $laSources(n, nt)$.

Definition 19. (π_μ) We can thus define π_μ as follows:

- π_μ must include *shift actions* on the new marking terminal. If the glue production is $nt_H \to \mu_E s_E$, then $\forall n \in States_C.\ (h \in initNTs(n) \Rightarrow Shift(n_E^S) \in \pi_\mu(n, \mu_E))$.
- π_μ must include reduce actions on *marking-terminal lookahead* — if h contributes lookahead to a set of reduce actions in state n, μ_E must be added to that set. Therefore, $p \in laSources(n, h) \Rightarrow Reduce(p) \in \pi_\mu(n, \mu_E)$.

Definition 20. (*The operation* $\cup_T^*$) Merging a host language parse table with *multiple* extension parse table fragments is the straightforward generalization of the operator $\cup_T$. We refer to this generalization as $\cup_T^* : ParseTables \times \mathcal{P}(ParseTables) \to ParseTables$.

Concealment of source code. In an implementation, the two maps *initNTs* and *laSources* can be stored explicitly in the form presented above, but should the writers of the host grammar or an extension grammar wish to conceal its "source code," this is also feasible in our approach. The different sections of the parse table can be encoded in the traditional manner by using integers to represent terminals and nonterminals. The additional metadata can be encoded as follows. *initNTs* can be inferred, if necessary, from the *goto* tables (if for any state n there is an action in the appropriate goto table — host or extension, depending on the n — for a nonterminal h, $h \in initNTs(n)$). *laSources* only needs to store enough information to remove the proper number of elements from the parse stack when a reduce action takes place, and to look up the proper action in the goto table after that. Thus, the minimum that is needed is the left hand side of the production to be reduced upon and the number of symbols on its right hand side. In practice, a code fragment might be specified that takes the elements removed from the stack and replaces them with some computed result such as the corresponding abstract syntax tree.

4.2 Composition of scanners.

As the operation of merging parse tables offers a rapid way to compose parsers, it is apropos to discuss ways of composing *scanners* to feed tokens to the composed

parser. There are two primary problems to be resolved when composing scanners. First is the problem of bundling each of the component parse tables with appropriate scanner DFAs in such a way that they can be assembled quickly to be used with the composed parser, but with a minimum of duplication of scanner DFA states. Specifically, the DFA(s) built for the host language (or more precisely for the parse states of the host language) will not be able to recognize terminals defined in an extension, but the DFA(s) built for the extension states may have to recognize host-language terminals. This happens when, for example, an extension construct derives a host-language nonterminal h and any terminal that can begin a string derived from h must be scanned for. Second is the problem of building the DFAs at composition time to scan for the various marking terminals introduced by multiple extensions. Extension-language non-marking terminals will only have to be scanned for in the states of that extension, but any marking terminal may have to be scanned for in any state in the composed parse table. Again, this occurs when an extension construct derives a host nonterminal h; if the extension defines a bridge production with h as the left hand side, the marking terminal will be among the terminals that can begin strings derived from h. These must be recognized in the host language scanner DFA and possibly in other extension scanner DFAs.

Scanner bundling problem. One solution to this problem (which we do not presently recommend) is to use a different parse-state-based context-aware scanner for each state with different valid lookahead. A better solution is to compile a different scanner not for each state, but for each extension. Then the scanner for a particular extension Γ^{E_i} would be called only when the parser is in one of the parse states in $States_{E_i}$, and would only need to scan for the terminals in the valid lookahead sets of those states. This approach involves some duplication of states, although we find that the amount of duplication necessary for our extensions is not wholly unreasonable.

ableJ extension	SQL	Tables	Foreach	Dimension analysis
Terminals in composed grammar ($T_H \cup T_E$)	194	161	158	181
Terminals containing actions in $n \in States_E$ (% of above total)	132 (68%)	69 (43%)	95 (60%)	105 (58%)
States in scanner for *States*	930 (120 new)	852 (42 new)	813 (3 new)	856 (46 new)
States in scanner for $States_E$ (% of above total)	549 (59%)	253 (30%)	411 (51%)	345 (40%)

Table 1. Scanners for extension states.

See Table 1 for some test data, comparing this approach with the approach of building a single new scanner DFA for the entire composed language in the context of *ableJ*, our extensible version of Java [**?**]. The columns in this table represent several of our modular extensions.

- The first row gives the number of terminals in the host Java grammar and the given extension grammar together. The Java grammar alone has 157 terminals. The "foreach" extension (providing the same functionality as the foreach loops added to Java 1.5) introduces no new terminals except its marking terminal, while the extension for embedded SQL introduces a large number of new terminals for SQL syntax.
- The second row gives the proportion of the totals in the first that are in some valid lookahead set of an extension state. It is evident that these are made up mainly of host terminals. This is because in Java, a very large number of keywords can begin expressions or statements, which are derived by various extension constructs, and hence anything that can begin an expression or statement must be scanned for.
- The third row gives the number of states in the DFA that can scan on valid lookahead sets in both host and extension states (*i.e.*, the scanner that would be built if the composed grammar were compiled from source).
- The fourth row gives the number of states in the DFA that can scan just on those terminals enumerated in the second row (and hence can be bundled with the extension for use in scanning on any state within it).

The total number of scanner DFA states needed in the composed parser, therefore, is all the states in the fourth row put together, plus the number of states in the scanner DFA for the host grammar compiled alone. In this case, the Java host grammar produces a scanner DFA with 810 states, so the total number of states will be $810 + 549 + 253 + 411 + 345 = 2368$. To compare, the scanner DFA generated when the composition of all these extensions is compiled from source contains 1020 states. This significantly increases the number of scanner DFA states required, but may be a reasonable price to pay for the advantages of quickly composing previously generated parse tables and scanners.

Marking terminal problem. The arrangements above do not, however, offer any solution to the marking terminal problem. The brute force approach is simply to rebuild the DFAs of each scanner for every affected state to include the marking terminals, but this is potentially expensive, as is all building of arbitrary DFAs. It goes so far as to make useless the approach of building a single DFA for each extension, since all such single DFAs would have to be rebuilt. A more efficient option would be to use a "double-state" DFA, wherein the regular expressions corresponding to all the necessary marking terminals are compiled into a separate DFA — this is a polynomial-time process, since these regular expressions usually match only one string, and thus the NFA generated from their union is acyclic — and the old and new DFAs are then run in parallel when the scanner is called. One or both will then return a match; if the DFA scanning for

marking terminals finds the match, we may choose to prefer that over another possible match by the pre-existing scanner DFA.

5 Implementation/time complexity analysis.

5.1 Data structures.

To summarize the problem, the composition process must take several parse tables, as well as the maps *initNTs* and *laSources*, and assemble them into the finished parse table according to the rules put forth in the previous section. One could do this in the naïve manner and generate the same parse table that would be generated if the grammar were recompiled from the ground up, but this approach is somewhat inefficient.

The two maps are arranged by state, and must be stored piecemeal with the maps for each chunk of states being bundled along with the parse table for the corresponding extension. It is very straightforward to use *hash-tables* to hold the maps. It is common practice when building parse tables for practical use to assign a number to each grammar symbol and then use a two-dimensional array to index parse actions. This is, furthermore, the only data structure that is efficient enough for practical use in that application. In this case, however, to expedite the process of merging we will use *several* two-dimensional arrays, which can then be used as a large "virtual" two-dimensional array by mapping different indices of the "virtual" array to each "real" table.

Let us go back to Figure 3. The leftmost column in that table — the actions on terminals in T_H — would form its own array. This would allow the parse table for the host language to be copied byte-for-byte into the composed parse table, and the same for those columns of the extension parse tables containing actions on terminals in T_H. The rightmost column — the actions for marking terminals — would also form its own array.

There are no actions on terminals in T_{E_i} except in the states inside $States_{E_i}$. This is illustrated by the bottommost two white squares in Figure 3. It follows that these columns of each extension parse table can be stored in the same columns of the composed parse table; more specifically, starting at column $|T_H|+1$.[2] This means that at the time extension parse tables are built, only one factor is unknown: the exact index at which the block of states $States_{E_i}$ will be situated in the table. This unknown factor could be represented by a bit attribute on shift actions, indicating whether the destination of the action is a host or extension state. There are then two ways to arrange the T_{E_i} columns of the extension parse tables:

1. Keep **a separate array** for each and translate the exact array indices of extension states at runtime. This has the advantage that the extension parse

[2] With a traditional scanning apparatus, this arrangement would cause erroneous parses if a terminal in T_{E_i} was recognized when in a state belonging to another extension ($n \in States_{E_j}$), but this is not a problem when using a context-aware scanner.

tables can be copied byte-for-byte as the host-language parse table is, and the disadvantage of more overhead for a more cluttered "virtual" array. See Figure 4(a) for a diagram of this arrangement.

2. Use an approach in which the host-language table's array and the marking terminal array are separate, but all the extension-symbol columns are in a single array with $\max_{E_i} |T_{E_i}|$ columns, for a total of **three arrays**, translating array indices at compile-time. This has the advantage of being less cluttered, but the disadvantages that the same processing must be carried out at compile-time and that a byte-for-byte copy of the extension table cannot be made. See Figure 4(b) for a diagram of this arrangement.

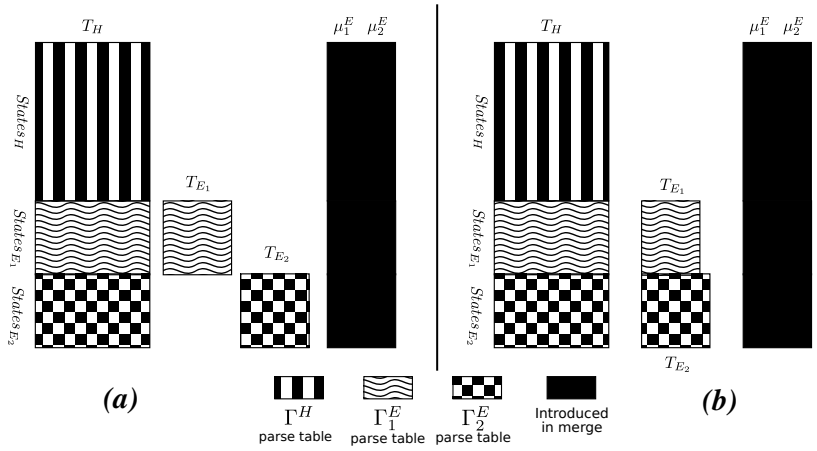

Fig. 4. Ways of partitioning the composed parse table for faster composition: (a): the multiple-array approach; (b): the three-array approach.

5.2 Time complexity analysis.

The bulk of the computation in this process is copying and/or adapting pieces of parse tables. If a parse action takes up A bytes, the multiple-array approach involves copying exactly $A \cdot (|States_H| \cdot |T_H| + \sum_{E_i} (|States_{E_i}| \cdot |T_{E_i}|))$ bytes. In the three-array approach, $A \cdot (|States_H| \cdot |T_H|)$ bytes are copied (the host-language parse table, which likely comprises the bulk of the composed table) and $|States| \cdot \max_{E_i} |T_{E_i}|$ cells in the extension-language array are filled. For each such cell that is filled, if the action that fills it has an extension state as its destination, it must be updated with an exact state number, as stated above. This takes constant time.

In the naïve single-array approach, no part of the table can be copied in; $|States| \cdot \left|T_H \cup \bigcup_{E_i} TE_i\right|$ cells in the single array are filled, using the same process as in the three-array approach.

In all approaches, for each extension, a parse-table column for the extension's marking terminal (*States* cells) must be filled out from the maps *initNTs* and *laSources*. For each cell (n, μ_i^E), one lookup on *initNTs* must be made to determine if a shift action should be inserted in the cell, and one lookup on *laSources* must be made to determine if a reduce action should be inserted in that cell. Each lookup takes constant time.

6 Related work.

This paper is largely a corollary of our previous work on verifiable grammar composition in a deterministic framework, which shares the same motivation: to have a guarantee that the syntax of several eclectic extensions can be merged together seamlessly in a deterministic parsing framework. Context-aware scanning [**?**] ensures that these schemes will work in practice, by taking context as a factor in scanning. Each time the parser calls to a context-aware scanner for a token, the scanner is passed the *valid lookahead set* — those terminals that have a valid parse action (shift, reduce, or accept, but not error) in the given parse state — and scans only for terminals in that set. Using a traditional scanner, every terminal in the composed grammar would be scanned for at every scan, including terminals from all extensions in any extension state, so no writer of any single extension can predict how to handle the scanning. But in a context-aware system, in the states of $States_{E_i}$, only the terminals of $T_H \cup T_{E_i}$ are in the valid lookahead set, so the scanning is made in exactly the same way as if Γ^{E_i} had been composed alone with Γ^H. The modular determinism analysis [**?**] utilizes exactly the same concepts as presented in this paper, but in the context of composing *grammars* rather than parse tables. That paper resolves additional practical issues such as disambiguation by operator precedence and associativity.

Bravenboer and Visser [**?**] 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 compose, 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," 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. Although this has the advantage of being general, it has the disadvantage of being dependent on the nondeterministic GLR approach in which grammar ambiguities could arise in the composition of extensions. Also, the generality means that the approach works essentially on *memoization* — holding metadata to ensure that only the smallest possible parts of the parse table are recompiled — and this raises the possibility that in some cases, large portions or even the entire parse table must be recompiled. This also means that more metadata must be maintained in this approach than in ours.

Tatoo [**?**], which is both a scanner and parser generator, introduces a number of innovations. One is support for rapid composition of extensions without the need for regenerating parse tables; the system can switch between different precompiled 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 [**?**] (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. This approach, however, is underpinned by the idea that host syntax always has precedence — that the parse table of an extension component is entered only if there is no valid host syntax at that point. We make the entry points more explicit, so that it is possible to choose whether host or extension syntax should take precedence at the entry points.

7 Discussion and Future Work

In this section we cover some opportunities for future work in the area of composing parse tables at runtime. We also discuss the importance of having compile-time guarantees of determinism when building extensible languages. We are currently incorporating the ideas presented here into *Copper*, our integrated parser and context-aware scanner generator.

Runtime composition. Our experience to date has been with a model of language extension that is carried out at compile time. The compiler is provided with a file listing the host grammar and all the extensions, which are then compiled into a finished parser prior to any input being parsed. But there is also a possible application for composition of extensions at parse time. Consider the practice in Java of importing library classes through the use of import statements. A parser could be made to read such an import statement for *language extensions* instead, and compose in the extension on the fly. Clearly, compiling a composed *grammar* in such conditions would take far too much time to be practical, but the use of parse table composition in this case could very well be plausible: assemble the necessary parse tables on the fly when reducing on

the production of the import statement. Another approach might be to compose beforehand, but only "turn on" an extension's marking terminal if the extension is imported.

Determinism guarantees. It might be posited that the restrictions imposed by our method of parse table composition are too narrow to be of significant utility, but we argue that it outweighs the moderate loss of expressivity imposed by the restrictions if the writer of an extension is able to perform a static analysis that ensures the extension will compose *deterministically*. Other techniques of parse table composition allow varying ranges of extensions, including some that can compose *arbitrary* extensions; it is apropos to offer, as we have, some comparison of the different approaches.

Another appropriate comparison for all of these approaches is with libraries. We have drawn this parallel above in discussing runtime composition, but in general, libraries are the accepted mechanism for programmers to "extend" their language with new capabilities. Although libraries provide no new syntactic constructs or semantic analysis, the library writer can distribute the code following compilation and type-checking, ensuring that *the programmer can bring together and use any combination of libraries needed to address a particular problem.*

It is this guarantee that our approach provides, a guarantee essential when parsing extensible languages: if they are to become widely used, we need this ability of the extension writer to provide a guarantee of safe composition by any non-expert.