

Parse Table Composition

Separate Compilation and Binary Extensibility of Grammars

Martin Bravenboer¹ and Eelco Visser²

¹ University of Oregon, USA, martin.bravenboer@acm.org

² Delft University of Technology, The Netherlands, visser@acm.org

Abstract. Module systems, separate compilation, deployment of binary components, and dynamic linking have enjoyed wide acceptance in programming languages and systems. In contrast, the syntax of languages is usually defined in a non-modular way, cannot be compiled separately, cannot easily be combined with the syntax of other languages, and cannot be deployed as a component for later composition. Grammar formalisms that do support modules use whole program compilation.

Current extensible compilers focus on source-level extensibility, which requires users to compile the compiler with a specific configuration of extensions. A compound parser needs to be generated for every combination of extensions. The generation of parse tables is expensive, which is a particular problem when the composition configuration is not fixed to enable users to choose language extensions.

In this paper we introduce an algorithm for *parse table composition* to support separate compilation of grammars to *parse table components*. Parse table components can be composed (linked) efficiently at runtime, i.e. just before parsing. While the worst-case time complexity of parse table composition is exponential (like the complexity of parse table generation itself), for realistic language combination scenarios involving grammars for real languages, our parse table composition algorithm is an order of magnitude faster than computation of the parse table for the combined grammars, making online language composition feasible.

1 Introduction

Module systems, separate compilation, deployment of binary components, and dynamic linking have enjoyed wide acceptance in programming languages and systems. In contrast, the syntax of languages is usually defined in a non-modular way, cannot be compiled separately, cannot easily be combined with the syntax of other languages, and cannot be deployed as a component for later composition. Grammar formalisms that do support modules use whole program compilation and deploy a compound parser. In this paper we introduce an algorithm for *parse table composition* to support separate compilation of grammars to *parse table components*.

The lack of methods for deploying the definition and implementation of languages as components is harming programming practices. Languages are combined in an undisciplined and uncontrolled way, for example by using SQL, HQL, Shell commands, XPath, regular expressions, and LDAP in string literals. The compiler of the host language has no knowledge at all of these languages. Hence, the compiler cannot check if the programs are syntactically correct, nor can the compiler help protect the program against security

```
import table person [ id INT, name VARCHAR, age INT ];
connection c = "jdbc:postgresql:mybook";
ResultSet rs = using c query { SELECT name FROM person WHERE age > {limit}};
```

Fig. 1: SQL extension of Java in ableJ (Silver)

```
$name = $_GET['name'];
$q = "SELECT * FROM users WHERE name = '" . $name . "'";
$q = <| SELECT * FROM users WHERE name = ${$name} |>;
system("svn cat \"file name\" -r" . $rev);
system(<| svn cat "file name" -r${$rev} |>);
```

Fig. 2: SQL and Shell extensions of PHP (StringBorg)

```
class FileEditor {
  void handle(Event e) when e@Open { ... }
  void handle(Event e) when e@Save { ... }
  void handle(Event e) {...} }
```

Fig. 3: Predicate dispatch in JPred (Polyglot)

```
Return(ConditionalExpression(e1, e2, e3)) -> If(e1, Return(e2), Return(e3))
[[ return e1 ? e2 : e3; ]] -> [[ if($e1) return $e2; else return $e3; ]]
```

Fig. 4: Transformation with concrete Java syntax (Stratego)

vulnerabilities caused by user input that is not properly escaped. Extensible compilers such as ableJ [1] (based on Silver), JastAddJ [2], and Polyglot [3] support the modular extension of the base language with new language features or embeddings of domain-specific languages. For example, the security vulnerabilities caused by the use of string literals can be avoided by extending the compiler to understand the syntax of the embedded languages. The extended compiler compiles the embedded fragments with the guarantee that user input is properly escaped according to the escaping rules of the embedded language. Figure 1 shows an application of the extensible compiler ableJ. This extension introduces constructs for defining database schemas and executing SQL queries. The implementation of the extension is modular, i.e. the source code of the base Java compiler is not modified. Silver and the ableJ compiler have also been applied to implement extensions for complex numbers, algebraic datatypes, and computational geometry [1]. Similar to the SQL extension of ableJ, the StringBorg syntactic preprocessor [4] supports the embedding of languages in arbitrary base languages to prevent security vulnerabilities. Figure 2 shows applications of embeddings of SQL and Shell commands in PHP. In both cases, the StringBorg compiler guarantees that embedded sentences are syntactically correct and that strings are properly escaped, as opposed to the unhygienic string concatenation on the previous lines. The implementations of these extensions are modular, e.g. the grammar for the embedding of Shell in PHP is a module that imports grammars of PHP and Shell. Finally, the Polyglot [3] compiler has been used for the implementation of many language extensions, for example Jedd's database relations and binary decision diagrams [5] and JPred's predicate dispatch [6]. Figure 3 illustrates the JPred extension.

Similar to the syntax extensions implemented using extensible compilers, several metaprogramming systems feature an extensible syntax. Metaprograms usually manipulate programs in a structured representation, but writing code generators and transformations in the abstract syntax of a language can be very unwieldy. Therefore, several

metaprogramming systems [7–10] support the embedding of an object language syntax in the metalanguage. The embedded code fragments are parsed statically and translated to a structured representation, thus providing a concise, familiar notation to the metaprogrammer. Figure 4 illustrates this with a transformation rule for Java, which lifts conditional expressions from return statements. The first rule is defined using the abstract syntax of Java, the second uses the concrete syntax. Metaprogramming systems often only require a grammar for the object language, i.e. the compilation of the embedded syntax is generically defined [7, 11].

Extensibility and Composition. Current extensible compilers focus on *source-level extensibility*, which requires users to compile the compiler with a specific configuration of extensions. Thus, every extension or combination of extensions results in a different compiler. Some recent extensible compilers support composition of extensions by specifying language extensions as attribute grammars [1, 12] or using new language features for modular, type-safe scalable software composition [13, 14]. In contrast to the extensive research on composition of later compiler phases, the grammar formalisms used by current extensible compilers do not have advanced features for the modular definition of syntax. They do not support separate compilation of grammars and do not feature a method for deploying grammars as components. Indeed, for the parsing phase of the compiler, a compound parser needs to be generated for every combination of extensions using *whole program compilation*.

Similarly, in metaprogramming systems with support for concrete syntax, grammars for a particular combination of object language embeddings are compiled together with the grammar of the metalanguage into a single parse table. Metaprograms often manipulate multiple object languages, which means that parse tables have to be deployed for all these combinations. The implementation of the later compiler phases is often generic in the object language, therefore the monolithic deployment of parse tables is the remaining obstacle to allowing the *user* to select language extensions.

A further complication is that the base language cannot evolve independently of the extensions. The syntax of the base language is deployed as part of the language extension, so the deployed language extension is bound to a specific version of the base language. Language extensions should only depend on an *interface* of the base language, not a particular implementation or revision.

As a result, extensions implemented using current extensible compilers cannot be deployed as a plugin to the extensible compiler, thus not allowing the *user* of the compiler to select of a series of extensions. For example, it is not possible for user to select a series of Java extensions for ableJ (e.g. SQL and algebraic datatypes) or Polyglot (e.g. JMatch and Jedd) without compiling the compiler. Third parties should be able to deploy language extension that do not require the compiler (or programming environment) to be rebuilt. Therefore, methods for deploying languages as *binary components* are necessary to leverage the promise of extensible compilers. We call this *binary extensibility*. One of the challenges in realizing binary extensible compilers is binary extensibility of the syntax of the base language. Most extensible compilers use an LR parser, therefore this requires the introduction of LR parse table components.

Parse Table Composition. In this paper we introduce an algorithm for *parse table composition* to support separate compilation of grammars to *parse table components*. Parse

table components can be composed (linked) efficiently at run-time (i.e. just before parsing) by a minimal reconstruction of the parse table. This algorithm can be the foundation for separate compilation in parser generators, load-time composition of parse table components (cf. dynamic linking), and even run-time extension of parse tables for self-extensible parsers [15]. As illustrated by our AspectJ evaluation, separate compilation of modules can even improve the performance of a whole program parser generator. Using parse table composition extensible compilers can support binary extensibility (at least for syntax) by deploying the parser of the base language as a parse table component. Language extensions can be deployed as binary components plugging into the extensible compiler by providing a parse table component generated for the language extension only.

While the worst-case time complexity of parse table composition is exponential (like the complexity of parse table generation itself), for realistic language combination scenarios involving grammars for real languages, our parse table composition algorithm is an order of magnitude faster than computation of the parse table for the combined grammars, making online language composition feasible. The goal is to make composition fast enough to make the user unaware of the parse table composition. This will allow composition of the components at every invocation of a compiler, similar to dynamic linking of executable programs and libraries.

We have implemented parse table composition in a prototype ³ that generates parse tables for scannerless [16] generalized LR (GLR) [17, 18] parsers. It takes SDF [19] grammars as input. The technical contributions of this work are:

- The idea of parse table composition as symmetric composition of parse tables as opposed to incrementally adding productions, as done in work on incremental parser generation [20–22]
- A formal foundation for parse table modification based on automata
- An efficient algorithm for partial reapplication of NFA to DFA conversion, the key idea of parse table composition

2 Grammars and Parsing

Context-free Grammars. A context-free grammar G is a tuple $\langle \Sigma, N, P \rangle$, with Σ a set of terminal symbols, N a set of nonterminal symbols, and P a set of productions of the form $A \rightarrow \alpha$, where we use the following notation: V for the set of symbols $N \cup \Sigma$; A, B, C for variables ranging over N ; X, Y, Z for variables ranging over V ; a, b for variables ranging over Σ ; v, w, x for variables ranging over Σ^* ; and α, β, γ for variables ranging over V^* . The context-free grammar $G_1 = \langle \Sigma, N, P \rangle$ will be used throughout this paper, where

$$\Sigma = \{+, \mathbb{N}\} \quad N = \{E, T\} \quad P = \{E \rightarrow E + T, E \rightarrow T, T \rightarrow \mathbb{N}\} \quad (G_1)$$

The relation \Rightarrow on V^* defines the derivation of strings by applying productions, thus defining the language of a grammar in a generative way. For a grammar G we say that $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma \in P(G)$. A series of zero or more derivation steps from α to β is denoted by $\alpha \Rightarrow^* \beta$. The relation \Rightarrow_{rm} on V^* defines rightmost derivations, i.e. where only the rightmost nonterminal is replaced. We say that $\alpha A w \Rightarrow_{rm} \alpha \gamma w$ if $A \rightarrow \gamma \in P(G)$. If $A \Rightarrow_{rm}^* \alpha$ then we say that α is a right-sentential form for A .

³ available at <http://www.strategox.org/Stratego/ParseTableComposition>

LR Parsing. We define an LR(0) parse table to be a tuple $\langle Q, \Sigma, N, \text{start}, \text{action}, \text{goto}, \text{accept} \rangle$ with Q a set of states, Σ a set of terminal symbols, N a set of nonterminal symbols, $\text{start} \in Q$, action a function $Q \times \Sigma \rightarrow \text{action}$ where action is either shift q or reduce $A \rightarrow \alpha$, goto a function $Q \times N \rightarrow Q$, and finally $\text{accept} \subseteq Q$, where we use the following additional notation: q for variables ranging over Q ; and S for variables ranging over $\mathcal{P}(Q)$.

An LR parser [23–25] is a transition system with as configuration a stack of states and symbols $q_0 X_1 q_1 X_2 q_2 \dots X_n q_n$ and an input string v of terminals. The next configuration of the parser is determined by reading the next terminal a from the input v and peeking the state q_n at the top of the stack. The entry $\text{action}(q_n, a)$ indicates how to change the configuration. The entries of the action table are shift or reduce actions, which introduce state transitions that are recorded on the stack. A shift action removes the terminal a from the input, which corresponds to a step of one symbol in the right-hand sides of a set of productions that is currently expected. A reduce action of a production $A \rightarrow X_1 \dots X_k$ removes $2k$ elements from the stack resulting in state q_{n-k} being on top of stack. Next, the reduce action pushes A and the new current state on the stack, which is determined by the entry $\text{goto}(q_{n-k}, A)$.

The symbols on the stack of an LR parser are always a prefix of a right-sentential form of a grammar. The set of possible prefixes on the stack is called the *viable prefixes*. We do not discuss the LR parsing algorithm in further detail, since we are only interested in the generation of the action and goto tables.

Generating LR Parse Tables. The action and goto table of an LR parser are based on a deterministic finite automaton (DFA) that recognizes the viable prefixes for a grammar.

The DFA for recognizing the viable prefixes for grammar G_1 is shown in Figure 5. Every state of the DFA is associated to a set of items, where an item $[A \rightarrow \alpha \bullet \beta]$ is a production with a dot (\bullet) at some position of the right-hand side. An item indicates the progress in possibly reaching a configuration where the top of the stack consists of $\alpha\beta$. If the parser is in a state q where $[A \rightarrow \alpha \bullet \beta] \in q$ then the α portion of the item is currently on top of the stack, implying that a string derivable from α has been recognized and a string derivable from β is predicted. For example, the item $[E \rightarrow E + \bullet T]$ represents that the parser has just recognized a string derivable from $E +$. We say that an item $[A \rightarrow \alpha \bullet X\beta]$ *predicts* the symbol X .

To deal with increasingly larger classes of grammars, various types of LR parse tables exist, e.g. LR(0), SLR, LALR, and LR(1). The LR(0), SLR, and LALR parse tables all have the same underlying DFA, but use increasingly more precise conditions on the application of reduce actions. Figure 6 shows the standard algorithm for the generation of LR(0) parse tables. The main function `generate-tbl` first calls `generate-dfa` to construct a DFA. The function `generate-dfa` collects states as sets of items in Q and edges between the

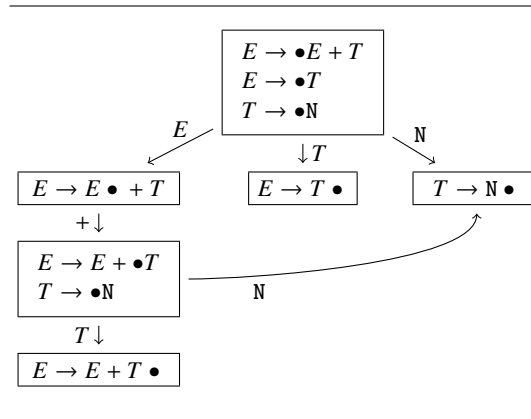


Fig. 5: LR(0) DFA for grammar G_1

```

function generate-tbl( $A, G$ ) =
1  $\langle Q, \delta, \text{start} \rangle := \text{generate-dfa}(A, G)$ 
2 for each  $q \rightarrow_X q' \in \delta$ 
3   if  $X \in \Sigma(G)$  then  $\text{action}(q, X) := \text{shift } q'$ 
4   if  $X \in N(G)$  then  $\text{goto}(q, X) := q'$ 
5 for each  $q \in Q$ 
6   if  $[A \rightarrow \alpha \bullet \text{eof}] \in q$  then
7      $\text{accept} := \text{accept} \cup \{q\}$ 
8   for each  $[A \rightarrow \alpha \bullet] \in q$ 
9     for each  $a \in \Sigma(G)$ 
10       $\text{action}(q, a) := \text{reduce } A \rightarrow \alpha$ 
11 return  $\langle Q, \Sigma(G), N(G), \text{start}, \text{action}, \text{goto}, \text{accept} \rangle$ 

function move( $q, X$ ) =
1 return  $\{ [A \rightarrow \alpha X \bullet \beta] \mid [A \rightarrow \alpha \bullet X \beta] \in q \}$ 

function generate-dfa( $A, G$ ) =
1  $\text{start} := \text{closure}(\{ [S' \rightarrow \bullet A \text{eof}] \}, G)$ 
2  $Q := \{ \text{start} \}, \delta := \emptyset$ 
3 repeat until  $Q$  and  $\delta$  do not change
4   for each  $q \in Q$ 
5     for each  $X \in \{ Y \mid [B \rightarrow \alpha \bullet Y \beta] \in q \}$ 
6        $q' := \text{closure}(\text{move}(q, X), G)$ 
7        $Q := Q \cup \{q'\}$ 
8        $\delta := \delta \cup \{q \rightarrow_X q'\}$ 
9 return  $\langle Q, \delta, \text{start} \rangle$ 

function closure( $q, G$ ) =
1 repeat until  $q$  does not change
2   for each  $[A \rightarrow \alpha \bullet B \beta] \in q$ 
3      $q := q \cup \{ [B \rightarrow \bullet \gamma] \mid B \rightarrow \gamma \in P(G) \}$ 
4 return  $q$ 

```

Fig. 6: LR(0) parse table generation for grammar G

states in δ . The start state is based on an initial item for the start production. For each set of items, the function `generate-dfa` determines the outgoing edges by applying the function `move` to all the predicted symbols of an item set. The function `move` computes the *kernel* of items for the next state q' based on the items of the current state q by shifting the \bullet over the predicted symbol X . Every kernel is extended to a closure using the function `closure`, which adds the initial items of all the predicted symbols to the item set. Because LR(0), SLR, and LALR parse tables have the same DFA the function `generate-dfa` is the same for all these parse tables. The LR(0) specific function `generate-tbl` initializes the action and goto tables based on the set of item sets. Edges labelled with a terminal become shift actions. Edges labelled with a nonterminal are entries of the goto table. Finally, if there is a final item, then for all terminal symbols the action is a reduce of this production.

Parse Table Conflicts. LR(0) parsers require every state to have a deterministic action for every next terminal in the input stream. There are not many languages that can be parsed using an LR(0) parser, yet we focus on LR(0) parse tables for now. The first reason is that the most important solution for avoiding conflicts is restricting the application of reduce actions, e.g. using the SLR algorithm. These methods are orthogonal to the generation and composition of the LR(0) DFA. The second reason is that we target a generalized LR parser [17, 18], which already supports arbitrary context-free grammars by allowing a *set* of actions for every terminal. The alternative actions are performed pseudo-parallel, and the continuation of the parsing process will determine which action was correct. Our parse table composition method can also be applied to target deterministic parsers, but the composition of deterministic parse tables might result in new conflicts, which have to be reported or resolved.

3 LR Parser Generation: A Different Perspective

The LR(0) parse table generation algorithm is very non-modular due to the use of the closure function, which requires all productions of a nonterminal to be known at parse table

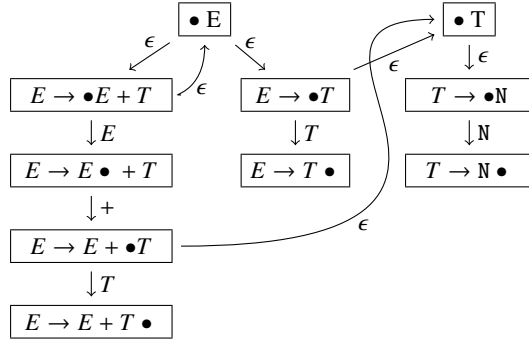


Fig. 7: LR(0) ϵ -NFA for grammar G_1

```

function generate-nfa( $G$ ) =
1   $Q := \{\bullet A \mid A \in N(G)\}$ 
2  for each  $A \rightarrow \alpha \in P(G)$ 
3     $q := \{[A \rightarrow \bullet \alpha]\}$ 
4     $Q := Q \cup \{q\}$ 
5     $\delta := \delta \cup \{\bullet A \rightarrow_\epsilon q\}$ 
6  repeat until  $Q$  and  $\delta$  do not change
7    for each  $q = \{[A \rightarrow \alpha \bullet X\beta]\} \in Q$ 
8       $q' := \{[A \rightarrow \alpha X \bullet \beta]\}$ 
9       $Q := Q \cup \{q'\}$ 
10      $\delta := \delta \cup \{q \rightarrow_X q'\}$ 
11     for each  $q = \{[A \rightarrow \alpha \bullet B\gamma]\} \in Q$ 
12        $\delta := \delta \cup \{q \rightarrow_\epsilon \bullet B\}$ 
13  return  $\langle Q, \Sigma(G) \cup N(G), \delta \rangle$ 

```

Fig. 8: LR(0) ϵ -NFA generation

generation time. As an introduction to the solution for separate compilation of grammars, we discuss a variation of the LR(0) algorithm that first constructs a non-deterministic finite automaton (NFA) with ϵ -transitions (ϵ -NFA) and converts the ϵ -NFA into an LR(0) DFA in a separate step using the subset construction algorithm [26, 27]. The ingredients of this algorithm and the correspondence to the one discussed previously naturally lead to the solution to the modularity problem of LR(0) parse table generation.

Generating LR(0) ϵ -NFA. An ϵ -NFA [28] is an NFA that allows transitions on ϵ , the empty string. Using ϵ -transitions an ϵ -NFA can make a transition without reading an input symbol. An ϵ -NFA A is a tuple $\langle Q, \Sigma, \delta \rangle$ with Q a set of states, Σ a set of symbols, and δ a transition function $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$, where we use the following notation: q for variables ranging over Q ; S for variables ranging over $\mathcal{P}(Q)$, but ranging over Q_D for a DFA D ; X for variables ranging over Σ ; and $q_0 \rightarrow_X q_1$ for $q_1 \in \delta(q_0, X)$.

Figure 7 shows the LR(0) ϵ -NFA for the example grammar G_1 (observe the similarity to a syntax diagram). For every nonterminal A of the grammar, there is a *station state* denoted by $\bullet A$. All other states contain just a single LR item. The station states have ϵ -transitions to all the initial items of their productions. If an item predicts a nonterminal A , then there are two transitions: an ϵ -transition to the station state of A and a transition to the item resulting from shifting the dot over A . For an item that predicts a terminal, there is just a single transition to the next item.

Figure 8 shows the algorithm for generating the LR(0) ϵ -NFA for a grammar G . Note that states are singleton sets of an item or just a dot before a nonterminal (the station states). The ϵ -NFA of a grammar G accepts the same language as the DFA generated by the algorithm of Figure 6, i.e. the language of viable prefixes.

Eliminating ϵ -Transitions. The ϵ -NFA can be turned into a DFA by eliminating the ϵ -transitions using the subset construction algorithm [28, 25], well-known from automata theory and lexical analysis. Figure 9 shows the algorithm for converting an ϵ -NFA to a DFA. The function ϵ -closure extends a given set of states S to include all the states reachable through ϵ -transitions. The function move determines the states reachable from a set of states S through transitions on the argument X . The function labels is a utility function

that returns the symbols (which does not include ϵ) for which there are transitions from the states of S . The main function ϵ -subset-construction drives the construction of the DFA. For every state $S \subseteq Q_E$ it determines the new subsets of states reachable by transitions from states in S .

Applying ϵ -subset-construction to the ϵ -NFA of Figure 7 results in the DFA of Figure 5. This is far from accidental because the algorithm for LR(0) DFA generation of Figure 6 has all the elements of the generation of an ϵ -NFA followed by subset construction. The ϵ -closure function corresponds to the function closure, because ϵ -NFA states whose item predicts a nonterminal have ϵ -transitions to the productions of this nonterminal via the station state of the nonterminal. The first move function constructs the kernel of the next state by moving the dot, whereas the new move function constructs the kernel by following the transitions of the NFA. Incidentally, these transitions exactly correspond to moving the dot, see line 8 of Figure 8. Finally, the main driver function generate-dfa is basically equivalent to the function ϵ -subset-construction. Note that most textbooks call the closure function from the move function, but to emphasize the similarity we moved this call to the callee of move.

```

function  $\epsilon$ -subset-construction( $A, \langle Q_E, \Sigma, \delta_E \rangle$ ) =
1   $Q_D := \{\epsilon\text{-closure}(\{\bullet A\}, \delta_E)\}$ 
2   $\delta_D := \emptyset$ 
3  repeat until  $Q_D$  and  $\delta_D$  do not change
4    for each  $S \in Q_D$ 
5      for each  $X \in \text{labels}(S, \delta_E)$ 
6         $S' := \epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E)$ 
7         $Q_D := Q_D \cup \{S'\}$ 
8         $\delta_D := \delta_D \cup \{S \rightarrow_X S'\}$ 
9  return  $\langle Q_D, \Sigma, \delta_D \rangle$ 

function  $\epsilon$ -closure( $S, \delta$ ) =
1  repeat until  $S$  does not change
2     $S := S \cup \{q_1 \mid q_0 \in S, q_0 \rightarrow_\epsilon q_1 \in \delta\}$ 
3  return  $S$ 

function labels( $S, \delta$ ) =
1  return  $\{X \mid q_0 \in S, q_0 \rightarrow_X q_1 \in \delta\}$ 

function move( $S, X, \delta$ ) =
1  return  $\{q_1 \mid q_0 \in S, q_0 \rightarrow_X q_1 \in \delta\}$ 

```

Fig. 9: Subset construction algorithm from ϵ -NFA E to DFA D

4 Composition of LR(0) Parse Tables

We discussed the ϵ -NFA variation of the LR(0) parse table generation algorithm to introduce the ingredients of parse table composition. LR(0) ϵ -NFA's are much easier to compose than LR(0) DFA's. A naive solution to composing parse tables would be to only construct ϵ -NFA's for every grammar at parse table generation-time and at composition-time merge all the station states of the ϵ -NFA's and run the subset construction algorithm. Unfortunately, this will not be very efficient because subset construction is the expensive part of the LR(0) parse table generation algorithm. The ϵ -NFA's are in fact not much more than a different representation of a grammar, comparable to a syntax diagram.

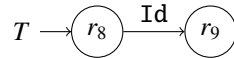
The key to an efficient solution is to apply the DFA conversion to the individual parse table components at generation-time, but also preserve the ϵ -transitions as metadata in the resulting automaton, which we refer to as an ϵ -DFA. The ϵ -transitions of the ϵ -DFA can be ignored as long as the automaton is not modified, hence the name ϵ -DFA, though there is no such thing as a deterministic automaton with ϵ -transitions in automata theory. The ϵ -transitions provide the information necessary for reconstructing a correct DFA using subset construction if states (corresponding to new productions) are added to the ϵ -DFA. In this way the DFA is computed per component, but the subset construction can be rerun

partially where necessary. The amount of subset reconstruction can be reduced by making use of information on the nonterminals that overlap between parse table components. Also, due to the subset construction applied to each component, many states are already part of the set of ϵ -NFA states that corresponds to a DFA state. These states do not have to be added to a subset again.

Figure 10a shows the ϵ -DFA for grammar G_1 , generated from the ϵ -NFA of Figure 7. The E and T arrows indicate the closures of the station states for these nonterminals. The two dashed ϵ -transitions correspond to ϵ -transitions of the ϵ -NFA. The ϵ -DFA does not contain the ϵ -transition that would result in a self-edge on the station state E . Intuitively, an ϵ -transition from q_0 to q_1 expresses that station state q_1 is supposed to be closed in q_0 (i.e. the subset q_0 is a superset of the subset q_1) as long as the automaton is not changed, which makes self-edges useless since every state is closed in itself. The composition algorithm is oblivious to the set of items that resulted in a DFA state, therefore the states of the automaton no longer contain LR item sets.

Figure 10b combines the ϵ -DFA of 10a with a second ϵ -DFA to form an automaton where the ϵ -transitions become relevant, thus being an ϵ -NFA. The parse table component adds variables to the tiny expression language of grammar G_1 based on the following grammar and its ϵ -DFA.

$$\Sigma = \{\text{Id}\} \quad N = \{T\} \quad P = \{T \rightarrow \text{Id}\} \quad (G_2)$$



The combined automaton connects station states of the same nonterminal originating from different parse table components by ϵ -transitions, in this case the two station states q_6 and r_8 for T (bold). Intuitively, these transitions express that the two states should always be part of ϵ -closures (subsets) together. In a combination of the original ϵ -NFA's, the station state T would have ϵ -transitions to all the initial items that now constitute the station states q_6 and r_8 .

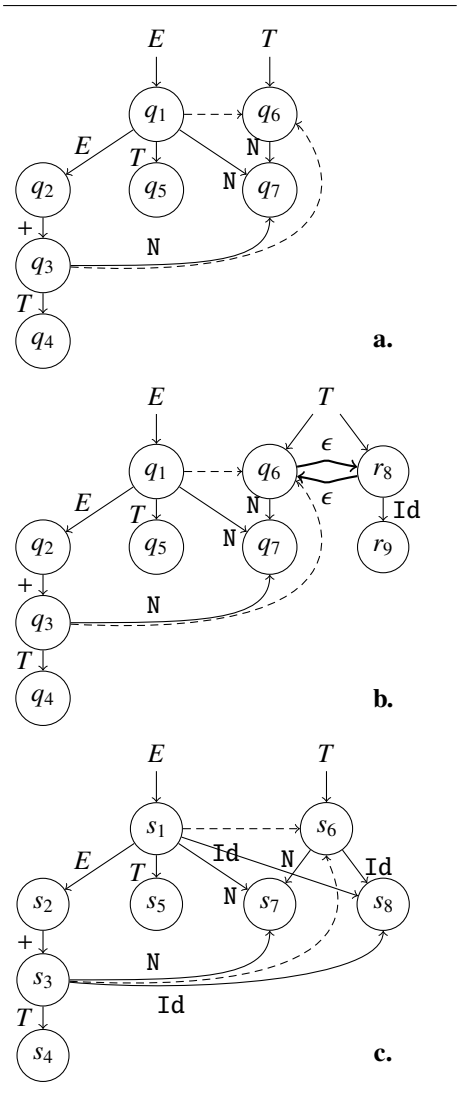


Fig. 10: **a.** LR(0) ϵ -DFA for grammar G_1 **b.** Combination of ϵ -DFA's for grammars G_1 and G_2 **c.** ϵ -DFA after subset construction

Figure 10c is the result of applying subset *reconstruction* to Figure 10b, resulting in a deterministic automaton (ignoring the now irrelevant ϵ -edges). State q_1 is extended to s_1 by including r_8 because there is a new path from q_1 to r_8 over ϵ -transitions, i.e. r_8 enters the ϵ -closure of q_1 . As a result of this extended subset, s_1 now has a transition on Id to s_8 . Similarly, state q_3 is extended to s_3 . Finally, station states q_6 and r_8 are merged into the single state s_6 because of the cycle of ϵ -transitions. Observe that five of the nine states from Figure 10b are not affected because their ϵ -closures have not changed.

4.1 Generating LR(0) Parse Tables Components

The visualizations of automata only show the states, station states and the transitions. However, LR parse tables also have reduce and accept actions and distinguish transitions over terminals (shift actions) from nonterminals (gotos). To completely capture parse tables, we define an LR(0) parse table component T to be a tuple

$$\langle Q, \Sigma, N, \delta, \delta^\epsilon, \text{station}, \text{predict}, \text{reduce}, P, \text{accept} \rangle$$

with Q a set of states, Σ a set of terminal symbols, N a set of nonterminal symbols, δ a transition function $Q \times (\Sigma \cup N) \rightarrow Q$, δ^ϵ a transition function $Q \rightarrow \mathcal{P}(Q)$ (visualized by dashed edges), station the function $N \rightarrow Q$ (visualized using arrows into the automaton labelled with a nonterminal), predict a function $Q \rightarrow \mathcal{P}(N)$, reduce a function $Q \rightarrow \mathcal{P}(P)$, P a set of productions of the form $A \rightarrow \alpha$, and finally $\text{accept} \subseteq Q$.

Note that the δ function of a component returns a single state for a symbol, hence it corresponds to a deterministic automaton. The ϵ -transitions are captured in a separate function δ^ϵ . For notational convenience we do not explicitly restrict the range of the δ^ϵ function to station states in this definition.

Parse table components do not have a specific start nonterminal, instead the station function is used to map all nonterminals to a station state. Given the start nonterminal, the station function returns the start state. We say that a station state q is *closed in* a state q' if $q' \rightarrow_\epsilon q$.

Figure 11 shows the algorithm for generating a parse table component, which is very similar to the subset construction algorithm of Figure 9. First, an ϵ -NFA is generated¹ for the grammar G . For every nonterminal A the ϵ -closure (defined in Figure 9) of its station state is added to the set of states Q_D ² of the ϵ -DFA, thus capturing the closure of the initial items of all productions of A . Next, for every state, the nonterminals³ predicted by the items of this subset are determined. Note that the predicted nonterminals of a state correspond to the ϵ -transitions to station states, or equivalently the transitions

```

function generate-xtbl( $G$ ) =
1   $\langle Q_E, \Sigma, \delta_E \rangle := \text{generate-nfa}(G)$ 
2  for each  $A \in N(G)$ 
3     $S := \epsilon\text{-closure}(\{\bullet A\}, \delta_E)$ 
4     $Q_D := Q_D \cup \{S\}$ 
5     $\text{station}(A) := S$ 
6  repeat until  $Q_D$  and  $\delta_D$  do not change
7  for each  $S \in Q$ 
8     $\text{predict}(S) := \{A \mid q \in S, q \rightarrow_\epsilon \{\bullet A\} \in \delta_E\}$ 
9     $\delta_D^\epsilon(S) := \{\text{station}(A) \mid A \in \text{predict}(S)\} - \{S\}$ 
10   for each  $X \in \text{labels}(S, \delta_E)$ 
11      $S' := \epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E)$ 
12      $Q_D := Q_D \cup \{S'\}$ 
13      $\delta_D := \delta_D \cup \{S \rightarrow_X S'\}$ 
14    $\text{reduce}(S) := \{A \rightarrow \alpha \mid [A \rightarrow \alpha \bullet] \in S\}$ 
15   if  $[A \rightarrow \alpha \bullet \text{eof}] \in S$  then
16      $\text{accept} := \text{accept} \cup \{S\}$ 
17 return  $\langle Q_D, \Sigma(G), N(G), \delta_D, \delta_D^\epsilon,$ 
18    $\text{station}, \text{predict}, \text{reduce}, P(G), \text{accept} \rangle$ 

```

Fig. 11: LR(0) parse table component generation

```

function compose-xtbl( $T_0, \dots, T_k$ ) =
1  $\langle N_E, \delta_E, \delta_E^{\epsilon+}, \text{stations}_E, \text{predict}_E, \text{reduce}_E, \text{accept}_E \rangle := \text{combine-xtbl}(T_0, \dots, T_k)$ 
2 for each  $A \in N_E$ 
3    $S := \epsilon\text{-closure}(\text{stations}_E(A), \delta_E^{\epsilon+})$ 
4    $Q_D := Q_D \cup \{S\}$ 
5    $\text{station}(A) := S$ 
6 repeat until  $Q_D$  and  $\delta_D$  do not change
7   for each  $S \in Q_D$ 
8      $\text{predict}(S) := \bigcup \{\text{predict}_E(q) \mid q \in S\}$ 
9      $\delta_D^{\epsilon}(S) := \{\text{station}(A) \mid A \in \text{predict}(S)\} - \{S\}$ 
10    for each  $X \in \text{labels}(S, \delta_E)$ 
11       $S' := \epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E^{\epsilon+})$ 
12       $Q_D := Q_D \cup \{S'\}$ 
13       $\delta_D := \delta_D \cup \{S \rightarrow_X S'\}$ 
14       $\text{reduce}(S) := \bigcup \{\text{reduce}_E(q) \mid q \in S\}$ 
15      if  $(\text{accept}_E \cap S) \neq \emptyset$  then  $\text{accept} := \text{accept} \cup \{S\}$ 
16 return  $\langle Q_D, \bigcup_{i=0}^k \Sigma_i, N_E, \delta_D, \delta_D^{\epsilon}, \text{station}, \text{predict}, \text{reduce}, \bigcup_{i=0}^k P_i, \text{accept} \rangle$ 

function combine-xtbl( $T_0, \dots, T_k$ ) =
1  $N_E := \bigcup_{i=0}^k N(T_i), \quad \delta_E^{\epsilon+} := \bigcup_{i=0}^k \delta^{\epsilon+}(T_i), \quad \delta_E := \bigcup_{i=0}^k \delta(T_i)$ 
2 for each  $A \in N_E$ 
3   for  $0 \leq i \leq k$ 
4     for  $0 \leq j \leq k, j \neq i$ 
5       if  $A \in N(T_i) \wedge A \in N(T_j)$ 
6          $\delta_E^{\epsilon+} := \delta_E^{\epsilon+} \cup \{\text{station}(T_i, A) \rightarrow \text{station}(T_j, A)\}$ 
7  $\text{stations}_E := \bigcup_{i=0}^k \text{station}(T_i), \quad \text{predict}_E := \bigcup_{i=0}^k \text{predict}(T_i)$ 
8  $\text{reduce}_E := \bigcup_{i=0}^k \text{reduce}(T_i), \quad \text{accept}_E := \bigcup_{i=0}^k \text{accept}(T_i)$ 
9 return  $\langle N_E, \delta_E, \delta_E^{\epsilon+}, \text{stations}_E, \text{predict}_E, \text{reduce}_E, \text{accept}_E \rangle$ 

```

Fig. 12: LR(0) parse table component composition

on nonterminal symbols from this state. The set of predicted symbols of a state (`predict`) is not necessary for a naive implementation of composition, but it will help to improve the performance as we will discuss later. The ϵ -transitions⁹ of a state are determined based on the predicted nonterminals, but it could also be based on δ_E . The self-edges are removed by subtracting the state itself. To drive the construction of the complete DFA, the next states¹⁰ are determined by following the transitions of the ϵ -NFA using the move function for all labels of this subset (see Figure 9). Finally, the reduce actions¹⁴ of a state are all the productions for which there is an item with the dot at the last position. If there is an item that predicts the special `eof` terminal¹⁵, then the state becomes an accepting state. Note that this definition requires the items of a subset to be known to determine accepting states and reduce actions, but this can easily be avoided by extending the ϵ -NFA with reduce actions and accepting states.

4.2 Composing LR(0) Parse Table Components

We first present a high-level version of the composition algorithm that does not take much advantage of the subset construction that has been applied to the individual parse table components. The algorithm is not intended to be implemented in this way, similar to the

algorithms for parse table generation. In all cases the fixpoint approach is very inefficient and needs to be replaced by a worklist algorithm. Also, efficient data structures need to be chosen to represent subsets and transitions. Figure 12 shows the high-level algorithm for parse table composition. Again, the algorithm is a variation of subset construction. First, the `combine-xtbl` function is invoked to combine the components (resulting in Figure 10b of the example). The δ^ϵ functions of the individual components are collected into a transition function $\delta_E^{\epsilon+}$ ¹. To merge the station states $\delta_E^{\epsilon+}$ is extended to connect the station states of the same nonterminal in different components². Finally, the relations `station`, `predict`, and `reduce`, and the set `accept` are combined. The domain of the relations `predict` and `reduce` are states, which are unique in the combined automaton. Thus, the combined functions `predictE` and `reduceE` have the same type signature as the functions of the individual components. However, the domain of `station` functions for individual components might overlap, hence the new function `stationsE` is a function of $N \rightarrow \mathcal{P}(Q)$.

Back to the `compose-xtbl` function, we now initialize the subset reconstruction by creating the station states² of the new parse table. The new station states are determined for every nonterminal by taking the ϵ -closure of the station states of all the components (`stationsE`) over $\delta_E^{\epsilon+}$. The creation of the station states initializes the fixpoint operation on Q_D and δ_D^ϵ . The fixpoint loop is very similar to the fixpoint loop of parse table component generation (Figure 11). If the table is going to be subject to further composition, then the `predict8` and δ_D^{ϵ} ⁹ functions can be computed similar to the generation of components. For final parse tables this is not necessary. Next, the transitions to other states are determined using the `move` function¹¹ and the transition function δ_E . Similar to plain LR(0) parse table generation the result of the `move` function is called a *kernel* (but it is a set of states, not a set of items). The kernel is turned into an ϵ -closure using the extended set of ϵ -transitions, i.e. $\delta_E^{\epsilon+}$. Finally, the `reduce` actions are simply collected¹⁴ and if any of the involved states is an `accept` state, then the composed state will be an `accept` state¹⁵.

This algorithm performs complete subset reconstruction, since it does not take into account that many station states are already closed in subsets. Also, it does not use the set of predicted nonterminals in any way. The correctness of the algorithm is easy to see by comparison to the ϵ -NFA approach to LR(0) parser generation. Subset construction can be applied partially to an automaton, so extending a deterministic automaton with new states and transitions and applying subset construction subsequently is not different from applying subset construction to an extension of the original ϵ -NFA.

4.3 Optimization

In worst case, subset construction applied to a NFA can result in an exponential number of states in the resulting DFA. There is nothing that can be done about the number of states that have to be created in subset reconstruction, except for creating these states as efficiently as possible. As stated by research on subset construction [29, 30], it is important to choose the appropriate algorithms and data structures. For example, the fixpoint iteration should be replaced, checking for the existence of a subset of states in Q must be efficient (we use uniquely represented treaps [31]), and the kernel for a transition on a symbol X from a subset must be determined efficiently. In our implementation we have applied some of the basic optimizations, but have focused on optimizations specific to parse table composition. The performance of parse table composition mostly depends on (1) the number of ϵ -closure invocations and (2) the cardinality of the resulting ϵ -closures.

Avoiding Closure Calls. In the plain subset construction algorithm ϵ -closure calls are inevitable for every subset. However, subset construction has already been applied to the parse table components. If we know in advance that for a given kernel a closure call will not add any station states to the closure that are not already closed in the states of the kernel, then we can omit the ϵ -closure call. For the kernel $\text{move}(S, X, \delta_E)$ ¹¹ it is not necessary to compute the ϵ -closure if none of the states in the kernel predict a nonterminal that occurs in more than one parse table component, called an *overlapping* nonterminal. If predicted nonterminals are not overlapping, then the ϵ -transitions from the states in this kernel only refer to station states that have no new ϵ -transitions added by combine-xtbl ². Hence, the ϵ -closure of the kernel would only add station states that are already closed in this kernel. Note that new station states cannot be found indirectly through ϵ -transitions either, because $\forall q_0 \rightarrow q_1 \in \delta^\epsilon(T_i) : \text{predict}(q_0) \supseteq \text{predict}(q_1)$. Thus, the kernel would have predicted the nonterminal of this station state as well.

To reduce the number of overlapping nonterminals it is useful to avoid unintentional overlap of nonterminals by supporting external and internal symbols.

Reduce State Rewriting. If a closure $\epsilon\text{-closure}(\text{move}(S, X, \delta_E), \delta_E^{\epsilon+})$ is a singleton set $\{q_0\}$, then q_0 can be added directly to the states of the composed parse table without any updating of its actions and transitions. That is, not only does the closure $\{q_0\}$ have the same actions and transitions as q_0 , but the transitions can also use the same names for the states they refer to. However, for this we need to choose the names of new states strategically. If there is a transition $q_0 \rightarrow_X q_1$ in the component of q_0 , then $\text{move}(\{q_0\}, X, \delta_E)$ will always be the single-state kernel $\{q_1\}$, but $\epsilon\text{-closure}(\{q_1\}, \delta_E^{\epsilon+})$ is not necessarily the single-state closure $\{q_1\}$. Hence, it is not straightforward that the transition $q_0 \rightarrow_X q_1$ can be included as is in the composed parse table. To avoid the need for updating transitions from single-state closures, we choose the names for the ϵ -closure of a single-state kernel $\{q\}$ strategically as the name of q .

This optimization makes it very useful to restrict the number of states in a closure aggressively. If the closure is restricted to a single state, then the compose algorithm only needs to traverse the transitions to continue composing states.

Reduce Closure Cardinality. Even if a kernel predicts one or more overlapping symbols (thus requiring an ϵ -closure call) it is not necessarily the case that any station states will be added to the kernel to form a closure. For example, if two components T_0 and T_1 having an overlapping symbol E are composed and two states $q_0 \in Q(T_0)$ and $q_1 \in Q(T_1)$ predicting E are both in a kernel, then the two station states for E are already closed in this kernel. To get an efficient ϵ -closure implementation the closures could be pre-computed *per state* using a transitive closure algorithm, but this example illustrates that the *subset* in which a state occurs will make many station states unnecessary. Note that for a state q in table T not all station states of T are irrelevant. Station states of T might become reachable through ϵ -transitions by a path through a different component.

A first approximation of the station states that need to be added to a kernel S to form a closure is the set of states that enter the ϵ -closure because of the ϵ -transitions added by combine-xtbl ²

$$S^{\text{approx}} = \epsilon\text{-closure}(S, \delta_E^{\epsilon+}) - \epsilon\text{-closure}(S, \cup \delta_i^\epsilon)$$

However, another complication compared to an ordinary transitive closure is that the station states have been transitively closed already inside their components. Therefore, we are not interested in *all* states that enter the ϵ -closure, since many station states in S^{approx} are already closed in other station states of S^{approx} . Thus, the minimal set of states that need to be added to a kernel to form a closure is the set of station states that (1) are not closed in the states of the kernel (S^{approx}) and are not already closed by another state in S^{approx} :

$$S^{min} = \{q_0 \mid q_0 \in S^{approx}, \nexists q_1 \in S^{approx} : q_1 \rightarrow q_0 \in \bigcup \delta_i^\epsilon\}$$

The essence of the problem is expressed by the *predict graph*, which is a subgraph of the combined graph shown in Figure 10b restricted to station states and ϵ -transitions between them. Every δ_i^ϵ transition function induces an acyclic, transitively closed graph, but these two properties are destroyed in the graph induced by $\delta_E^{\epsilon+}$. We distinguish the existing ϵ -transitions and the new transitions introduced by `combine-xtbl`² in *intra-component* and *inter-component* edges, respectively.

Figure 13 shows the optimized ϵ -closure algorithm, which is based on a traversal of the predict graph. For a given kernel S , the procedure `mark` first marks the states that are already closed in the kernel. If a state q_1 in the subset is a station state, then the station state itself³ and all the other reachable station states in the same component are marked⁴. Note that the graph induced by ϵ -transitions of a component is transitively closed, thus there are direct edges to all the reachable station states (intra-edges). For a state q_1 of the subset S that is not a station state, we mark all the station states that are closed in this non-station state q_1 ⁵. Note that q_1 itself is not present in the predict graph. The station states are marked with a source state, rather than a color, to indicate which state is responsible for collecting states in this part of the graph. If the later traversal of the predict graph encounters a state with a source different from the one that initiated the current traversal, then the traversal is stopped. The source marker is just used to make the intuition of the algorithm more clear, i.e. a coloring scheme could be used as well.

Next, the function `ϵ -closure` initiates traversals of the predict graph by invoking the `visit` function for all the involved station states. The `visit` function traverses the predict

```

procedure visit( $v, src$ )
1  color[ $v$ ] := BLACK
2  result[ $v$ ] :=  $\emptyset$ 
3  for each  $w \in$  intra-edges[ $v$ ]
4    if (source[ $w$ ] =  $src \vee$  source[ $w$ ] = NULL)  $\wedge$  color[ $w$ ] = WHITE
5      visit( $w, src$ )
6      result[ $v$ ] := result[ $v$ ]  $\cup$  result[ $w$ ]
7  for each  $w \in$  inter-edges[ $v$ ]
8    if source[ $w$ ] = NULL  $\wedge$  color[ $w$ ] = WHITE
9      visit( $w, src$ )
10   result[ $v$ ] := { $w$ }  $\cup$  result[ $v$ ]  $\cup$  result[ $w$ ]
procedure mark( $S, \delta^\epsilon$ )
1  for each  $q_1 \in S$ 
2    if  $q_1$  is station state
3      source[ $q_1$ ] :=  $q_1$ 
4      for each  $q_2 \in$  intra-edges[ $q_1$ ] do source[ $q_2$ ] :=  $q_1$ 
5    else
6      for each  $q_2 \in \delta^\epsilon(q_1)$ 
7        source[ $q_2$ ] :=  $q_2$ 
8        for each  $q_3 \in$  intra-edges[ $q_2$ ] do source[ $q_3$ ] :=  $q_2$ 
function  $\epsilon$ -closure( $S, \delta^\epsilon$ ) =
1  color[*] := WHITE, result[*] := NULL, source[*] := NULL
2  result :=  $S$ 
3  mark( $S, \delta^\epsilon$ )
4  for each  $q \in S$ 
5    if  $q$  is station state then maybe-visit( $q$ )
6    else for each  $q_A \in \delta^\epsilon(q)$  do maybe-visit( $q_A$ )
7  return result
8  local procedure maybe-visit( $q$ ) =
9    if source[ $q$ ] =  $q \wedge$  color[ $q$ ] = WHITE
10   visit( $q, q$ )
11   result := result  $\cup$  result[ $q$ ]

```

Fig. 13: Optimized ϵ -closure implementation

graph, collecting ¹⁰ only the states reached *directly* through inter-component edges, thus avoiding station states that are already closed in an earlier discovered station state. For every state, the intra-component edges are visited first, and mark states `BLACK` as already visited. The visit function stops traversing the graph if it encounters a node with a different source (but continues if the source is `NULL`), thus avoiding states that are already closed in other station states.

5 Extensions

SLR. For many languages, the LR(0) parse table generation algorithm results in states where the parser can perform a shift as well as a reduce action. If the parser generator targets a deterministic parser, then the parser generation fails at this point, or applies heuristics to resolve the conflict. To support a bigger class of languages, the SLR (Simple LR) algorithm [32] extends LR(0) by guarding the application of reduce actions by examining the next terminal in the input stream. An SLR parse table generator determines the set of terminals that can follow the nonterminal A and a reduce action for $A \rightarrow \alpha$ is only applied if the next terminal in the input stream is in this set. If using a deterministic parser, then the main reason for restricting the application of reduce actions is to support a larger class of grammars. For a GLR parser this is not necessary: a GLR parser can perform both actions of a shift/reduce conflict and the continuation of the parsing process will determine which action was correct. However, for GLR it is still useful to reduce the number of conflicts to improve performance [33].

The SLR algorithm determines the follow set of nonterminals by analyzing the productions of the grammar. Unfortunately, the follow set of a nonterminal can (and usually will) change if new productions are added to a grammar. Thus, the calculation of the follow set of a nonterminal requires a global analysis of the grammar. For this reason, reduce actions of parse table components are guarded by a symbolic reference to the follow set of a nonterminal. The actual follow sets are calculated at composition-time. Parse table components store their contribution to the global nullable, first, and follow relations. These relations induce first and follow graphs, which are traversed using the efficient Digraph [34] algorithm to calculate follow sets. The digraph algorithm is based on Tarjan's strongly connected components algorithm [35, 36]. We also apply a series of optimizations for strongly connected component transitive closure algorithms [37]. Section 6 shows that the time spent on calculating follow sets at composition-time is minimal. The full algorithm is presented in [38].

Lexical Analysis. Before syntax analysis, a lexical analyzer splits the input stream of characters into a sequence of tokens that correspond to the terminals of a context-free grammar. Until now we have ignored the composition of the lexical syntax definition, but any solution for extensible syntax needs to consider the lexical analysis phase as well. If different languages are used together in the same source file, then the finite automata-based implementation techniques for lexical syntax get more problematic because finite automata are oblivious to context. This is usually solved using a stateful lexical analyzer, which recognizes a different set of tokens in every context. The transitions between the lexical states are based on a careful analysis of the *complete* language [39]. If new tokens

are added by composition with other components, then the transitions might no longer be correct.

A scannerless parser [16] directly applies syntax analysis to the characters of the input stream. Instead of a separate specification of lexical and context-free syntax, a single grammar is used that defines all aspects of the language. Scannerless parsers elegantly deal with lexical context issues in syntax embeddings and extensions. Rather than parsing a lexical entity in isolation, as is done with regular expressions, the parsing context acts naturally as lexical state [39]. Therefore, the target parser of our prototype is a scannerless parser. Scannerless GLR parsers add a few disambiguation techniques to GLR parsing to define typical restrictions and disambiguations on the lexical syntax of a language. These restrictions are easy to support in parse table composition, since they can be applied to individual components. The scannerless extensions are discussed in more detail in [38].

6 Evaluation

In the worst case scenario an LR(0) automaton can change drastically if it is combined with another LR(0) automaton. The composition with an automaton for just a *single* production can introduce an exponential number of new states [20]. Parse table composition cannot circumvent this worst case. Fortunately, grammars of typical programming languages do not exhibit this behaviour. To evaluate our method, we measured how parse table composition performs in *typical* applications. Parse table components usually correspond to languages that form a natural sublanguage. These languages do not change the structure of the language invasively but hook into the base language at some points.

We compare the performance of our prototype ⁴ to the SDF parser generator that targets the same scannerless GLR parser (`sdf2-bundle-2.4pre212034`). SDF grammars are compiled by first normalizing the high-level features to a core language and next applying the parser generator. Our parser generator accepts the same core language as input. The prototype consists of two main tools: one for generating parse table components and one for composing them. As opposed to the SDF parser generator, the generator of our prototype has not been heavily optimized because its performance is not relevant to the performance of runtime parse table composition. We have implemented a generator and composer for *scannerless* GLR SLR parse tables. This affects the performance of the composer: grammars have more productions, more symbols, and layout occurs between many symbols. Note that the composed parse tables are identical to parse tables generated using the SDF parser generator. Therefore, the performance of the parsers is equivalent.

Figure 14 presents the results for a series of metaprogramming concrete syntax extensions using Stratego [9], StringBorg [4] extensions, and AspectJ. For Stratego and StringBorg, the number of overlapping symbols is very limited and there are many single-state closures. Depending on the application, different comparisons of the timings are useful. For runtime parse table composition, we need to compare the performance to generation of the full automaton. The total composition time (col. 16) is only about 2% to 16% of the SDF parse table generation time (col. 4). The performance benefit increases more if we include the required normalization (col. 3) (SDF does not support separate normalization of modules). For larger grammars (e.g. involving Java) the performance benefit is bigger.

⁴ available at <http://www.strategox.org/Stratego/ParseTableComposition>

	productions symbols		sdf pgen		parse table composition											
			normalization (ms)	table generation (ms)	overlapping symbols	\sum states	composed states	% single state	\sum normalization (ms)	\sum generate-xtbl (ms)	compose-xtbl (ms)	nullables (ms)	first sets (ms)	follow sets (ms)	follow sets total (ms)	compose total (ms)
Stratego+XML	782	436	430	230	4	2983	2127	85	160	580	27	0	0	10	10	37
Stratego+Java	1766	836	3330	1790	3	6513	6612	93	2110	4250	67	0	0	10	20	80
Stratego+Stratego	1115	536	780	600	4	4822	4085	89	410	1530	37	0	0	7	7	47
Stratego+Stratego+XML	1420	745	1530	830	5	5752	4807	89	440	1600	60	0	3	10	13	77
Stratego+Stratego+Java	2404	1145	6180	2710	4	10405	9295	93	2390	5780	127	0	3	20	23	150
Java+SQL	1391	750	2800	1300	3	5175	3698	92	1350	2440	63	0	0	10	10	73
Java+XPath	1084	554	1560	780	3	4158	2848	90	1150	2010	60	0	0	3	3	63
Java+LDAP	1024	545	1550	760	3	3831	2467	89	1140	1940	43	0	0	3	3	53
Java+XPath+SQL	1580	853	3560	1290	3	5784	4272	93	1390	2530	63	0	0	10	13	83
Java+XPath+LDAP	1213	648	1910	1050	3	4440	3041	91	1170	2030	57	0	3	3	10	63
AspectJ + 5x Java	3388	2426	48759	10261	62	19332	8305	51	1460	1990	477	0	3	30	33	520

Fig. 14: Benchmark of parse table composition compared to the SDF parser generator. (1) number of reachable productions and (2) symbols, (3) time for normalizing the full grammar and (4) generating a parse table using SDF pgen, (5) number of overlapping symbols, (6) total number of states in the components, (7) number of states after composition, (8) percentage of single-state closures, (9) total time for normalizing the individual components and (10) generating parse table components, (11) time for reconstructing the LR(0) automaton, (12, 13, 14, 15) time for various aspects of follow sets, (16) total composition time (11 + 15). We measure time using the clock function, which reports time in units of 10ms on our machine. We measure total time separately, which combined with the accuracy of clock explains why some numbers do not sum up exactly. All results are the average of three runs.

The AspectJ composition is clearly different from the other embeddings. The AspectJ grammar [39] uses grammar mixins to reuse the Java grammar in 5 different contexts. All contexts customize their instance of Java, for example reserved keywords differ per context. Without separate compilation this results in 5 copies of the Java grammar, which all need to be compiled, thus exercising the SDF parser generator heavily. With separate compilation, the Java grammar can be compiled once, thus avoiding a lot of redundant work. This composition is not intended to be used at runtime, but serves as an example that separate compilation of grammars is a major benefit if multiple instance of the same grammar occur in a composition. The total time to generate a parse table from source using parse table composition (col. 10 + 11 + 16) is only 7% of the time used by the SDF parser generator.

7 Related Work

Modular Grammar Formalisms. There are a number of parser generators that support splitting a grammar into multiple files, e.g. Rats! [40], JTS [8], PPG [3], and SDF [19]. They vary in the expressiveness of their modularity features, their support for the extension of lexical syntax, and the parsing algorithm that is employed. Many tools ignore the intricate aspects of syntax extensions, whereas some apply scannerless parsing or

context-aware scanning [41]. However, except for a few research prototypes discussed next, these parser generators all generate a parser by first collecting all the sources, essentially resulting in whole-program compilation.

Extensible Parsing Algorithms. For almost every single parsing algorithm extensible variants have already been proposed. What distinguishes our work from all the existing work is the idea of separately compiled parse table components and a solid foundation on finite automata for combining these parse table components. The close relation of our principles to the LR parser generation algorithm makes our method easy to comprehend and optimize. All other methods focus on adding productions to an existing parse table, motivated by applications such as interactive grammar development. However, for the application in extensible compilers we do not need incremental but *compositional* parser generation. In this way, a language extension can be compiled, checked, and deployed independently of the base language in which it will be used. Next, we discuss a few of the related approaches.

Horspool's [20] method for incremental generation of LR parsers is most related to our parse table composition method. Horspool presents methods for adding and deleting productions from LR(0), SLR, as well as LALR(1) parse tables. The work is motivated by the need for efficient grammar debugging and interactive grammar development, where it is natural to focus on addition and deletion of productions instead of parse table components. Interactive grammar development requires the (possibly incomplete) grammar to be able to parse inputs all the time, which somewhat complicates the method. For SLR follow sets Horspool uses an incremental transitive closure algorithm based on a matrix representation of the first and follow relations. In our experience, the matrix is very sparse, therefore we use Digraph. This could be done incrementally as well, but due to the very limited amount of time spend on the follow sets, it is hard to make a substantial difference.

IPG [21] is a lazy and incremental parser generator targeting a GLR parser using LR(0) parse tables. This work was motivated by interactive metaprogramming environments. The parse table is generated by need during the parsing process. IPG can deal with modifications of the grammar as a result of the addition or deletion of rules by resetting states so that they will be reconstructed using the lazy parser generator. Rekers [18] also proposed a method for generating a single parse table for a set of languages and restricting the parse table for parsing specific languages. This method is not applicable to our applications, since the syntax extensions are not a fixed set and typically provided by other parties.

Dypgen [15] is a GLR self-extensible parser generator focusing on scoped modification of the grammar from its semantic actions. On modification of the grammar it generates a new LR(0) automaton. Dypgen is currently being extended by its developers to incorporate our algorithm for runtime extensibility. *Earley* [42] parsers work directly on the productions of a context-free grammar at parse-time. Because of this the Earley algorithm is relatively easy to extend to an extensible parser [43, 44]. Due to the lack of a generation phase, Earley parsers are less efficient than GLR parsers for programming languages that are close to LR. *Maya* [45] uses LALR for providing extensible syntax but regenerates the automaton from scratch for every extension. *Cardelli's* [22] extensible syntax uses an extensible LL(1) parser. Camlp4 [46] is a preprocessor for OCaml using an extensible top down recursive descent parser.

Automata Theory and Applications. The egrep pattern matching tool uses a DFA for efficient matching in combination with lazy state construction to avoid the initial overhead of constructing a DFA. egrep determines the transitions of the DFA only when they are actually needed at runtime. Conceptually, this is related to lazy parse table construction in IPG. It might be an interesting experiment to apply our subset reconstruction in such a lazy way. Essentially, parse table composition is a DFA maintenance problem. Surprisingly, while there has been a lot of work in the maintenance of transitive closures, we have not been able to find existing work on DFA maintenance.

References

1. van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute grammar-based language extensions for Java. In: ECOOP'07. LNCS, Springer (July 2007)
2. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA '07, ACM Press (2007) 1–18
3. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for Java. In: CC'03. Volume 2622 of LNCS., Springer (April 2003) 138–152
4. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embedding – a host and guest language independent approach. [47]
5. Lhoták, O., Hendren, L.: Jedd: A BDD-based relational extension of Java. In: PLDI 2004, ACM Press (2004)
6. Millstein, T.: Practical predicate dispatch. In: OOPSLA '04, ACM (2004) 345–364
7. Arnoldus, B.J., Bijpost, J.W., van den Brand, M.G.J.: Repleo: A syntax-safe template engine. [47]
8. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: tools for implementing domain-specific languages. In: ICSR'98, IEEE Computer Society Press (June 1998) 143–153
9. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.16. Components for transformation systems. In: PEPM'06, ACM (January 2006)
10. : ASF+SDF MetaEnvironment website. <http://www.meta-environment.org>
11. Visser, E.: Meta-programming with concrete object syntax. In Batory, D., Consel, C., Taha, W., eds.: GPCE'02. Volume 2487 of LNCS., Springer (October 2002) 299–315
12. van Wyk, E., Bodin, D., Huntington, P.: Adding syntax and static analysis to libraries via extensible compilers and language extensions. In: LCSD'06. (2006)
13. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA '06, ACM (2006) 21–36
14. Odersky, M., Zenger, M.: Scalable component abstractions. In: OOPSLA '05, ACM (2005) 41–57
15. Onzon, E.: Dypgen: Self-extensible parsers for ocaml. <http://dypgen.free.fr> (2007)
16. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) parsing of programming languages. In: PLDI '89, ACM Press (1989) 170–178
17. Tomita, M.: Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1985)
18. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, Univ. of Amsterdam (1992)
19. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, Univ. of Amsterdam (September 1997)
20. Horspool, R.N.: Incremental generation of LR parsers. *Computer Languages* **15**(4) (1990) 205–223
21. Heering, J., Klint, P., Rekers, J.: Incremental generation of parsers. *IEEE Transactions on Software Engineering* **16**(12) (1990) 1344–1351

22. Cardelli, L., Matthes, F., Abadi, M.: Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, Palo Alto, California (February 1994)
23. Knuth, D.E.: On the translation of languages from left to right. *Information and Control* **8**(6) (December 1965) 607–639
24. Aho, A.V., Johnson, S.C.: LR parsing. *ACM Computing Surveys* **6**(2) (1974) 99–124
25. Aho, A.V., Sethi, R., Ullman, J.: *Compilers: Principles, techniques, and tools*. Addison Wesley, Reading, Massachusetts (1986)
26. Grune, D., Jacobs, C.J.H.: *Parsing Techniques - A Practical Guide*. Ellis Horwood, Upper Saddle River, NJ, USA (1990)
27. Johnstone, A., Scott, E.: Generalised reduction modified LR parsing for domain specific language prototyping. In: 35th Annual Hawaii International Conference on System Sciences (HICSS'02), Washington, DC, USA, IEEE Computer Society Press (2002) 282
28. Hopcroft, J.E., Motwani, R., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation* (3rd Edition). Addison-Wesley, Boston, MA, USA (2006)
29. Leslie, T.: Efficient approaches to subset construction. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada (1995)
30. van Noord, G.: Treatment of epsilon moves in subset construction. *Computational Linguistics* **26**(1) (2000) 61–76
31. Seidel, R., Aragon, C.R.: Randomized search trees. *Algorithmica* **16**(4/5) (1996) 464–497
32. DeRemer, F.: Simple LR(k) grammars. *Communications of the ACM* **14**(7) (1971) 453–460
33. Johnstone, A., Scott, E., Economopoulos, G.: Evaluating GLR parsing algorithms. *Science of Computer Programming* **61**(3) (2006) 228–244
34. DeRemer, F., Pennello, T.J.: Efficient computation of LALR(1) look-ahead sets. In: CC '79, New York, NY, USA, ACM Press (1979) 176–187
35. Tarjan, R.E.: Depth-first search and linear graph algorithms. *SIAM Journal on Computing* **1**(2) (1972) 146–160
36. Eve, J., Kurki-Suonio, R.: On computing the transitive closure of a relation. *Acta Informatica* **8**(4) (1966) 303–314
37. Nuutila, E.: Efficient Transitive Closure Computation in Large Digraphs. PhD thesis, Helsinki University of Technology (1995)
38. Bravenboer, M.: Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates. PhD thesis, Utrecht University, Utrecht, The Netherlands (Jan. 2008)
39. Bravenboer, M., Tanter, E., Visser, E.: Declarative, formal, and extensible syntax definition for AspectJ – A case for scannerless generalized-LR parsing. In: OOPSLA'06, ACM Press (2006)
40. Grimm, R.: Better extensibility through modular syntax. In Cook, W.R., ed.: PLDI'06, ACM Press (June 2006)
41. van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. [47]
42. Earley, J.: An efficient context-free parsing algorithm. *Communications of the ACM* **13**(2) (1970) 94–102
43. Tratt, L.: The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London (February 2005)
44. Kolbly, D.M.: Extensible Language Implementation. PhD thesis, University of Texas at Austin (December 2002)
45. Baker, J., Hsieh, W.: Maya: multiple-dispatch syntax extension in java. In: PLDI '02, ACM Press (2002) 270–281
46. de Rauglaudre, D.: *Camlp4 Reference Manual*. (September 2003)
47. Lawall, J., ed.: *Generative Programming and Component Engineering*, Sixth International Conference, GPCE 2007, New York, NY, USA, ACM Press (October 2007)