Parse Table Composition

6

ABSTRACT

Composition of context-free languages is useful in a wide range of scenarios, including program transformation and generation with concrete object syntax, language extension, domain-specific language embedding, and the definition of the syntax of language conglomerates such as AspectJ. The generation of parse tables for language combinations is expensive, which is a particular problem when the composition configuration is not fixed, as is for example the case in the instantiation of a template engine for a new target language. In this chapter we introduce an algorithm for parse table composition to support separate compilation of grammars. While the worst-case time complexity of parse table composition is exponential (like the complexity of parse table generation itself), for realistic language combination scenarios involving grammars for real languages, our parse table composition algorithm is an order of magnitude faster than computation of the parse table for the combined grammars, making online language composition feasible.

6.1 INTRODUCTION

Consider the following task: 'In Java, write a method implementing a web service that fetches data from a database, processes it in some way, and then sends the data as XML to the client.' This task involves three languages, i.e. Java, SQL, and XML. However, the SQL and XML code is not treated as code by a typical mainstream development environment. Instead, SQL and XML code fragments are generally composed using string composition with all the associated problems, such as a lack of syntactic and other static checks at compile-time, and vulnerabilities to injection attacks. This situation is not restricted to this particular combination, but holds for any combination of general-purpose (host) language (e.g. Java, C#, C, C++, Haskell) and special-purpose (guest) language (e.g. SQL, HQL, Shell, regular expressions, XPath, XQuery, LDAP).

The situation can be improved by making the host language syntactically and semantically aware of the guest languages. This type of *language composition* has many applications including program transformation and generation with concrete object syntax, language extension, domain-specific language embedding, and the definition of the syntax of language conglomerates such as AspectJ (Section 6.2 provides an extensive motivation). Given the combinatorial problem of possible language combinations, and the fact that new (special-purpose) languages are introduced regularly, it is not feasible to ex-

pect language developers (vendors) to build in support for each and every special purpose language.

Extensible compilers promise to make languages and their tools open to these types of extensions. The state-of-the-art in extensible compilers is represented by tools and frameworks such as Polyglot [Nystrom et al. 2003], MetaBorg [Bravenboer & Visser 2004 (Chapter 2)], Silver [van Wyk et al. 2007], and JastAdd [Ekman & Hedin 2004]. These tools focus on *source-level extensibility*, i.e. creating a language extension by extending the source code of the base compiler and compiling the source code of the extension together with the source code of the base compiler to build a binary compiler that supports the base language with this particular extension. Despite the advances in code reuse techniques applied or introduced by these extensible compiler frameworks, language extension is still a fairly heavyweight process and has not reached the state where language extensions can be deployed as separate plugins that can be added to the programming environment by an end user.

One of the challenges in realizing *binary extensible* compilers is binary extensibility of the syntax of the host language. Even if an extensible compiler framework supports the implementation of *independently extensible* [Szyperski 1996, Odersky & Zenger 2005] language extensions for later phases of the compiler (e.g. type-checking), then still a compound parser needs to be generated for every particular combination of language extensions. The generation of parse tables for language combinations is expensive, which is a particular problem when the composition configuration (i.e. the set of language extensions) is not fixed, and a parser needs to be generated at run-time (of the compiler).

In this chapter we introduce an algorithm for parse table composition to support separate compilation of grammars. As a result, an extensible compiler can be deployed using a parse table component for the base language. Plugins for language extensions provide a parse table component generated for the language extension only. A parser for a particular combination of languages is generated from the parse table components on the fly. While the worst-time complexity of parse table composition is exponential (like the complexity of parse table generation itself), for realistic language combination scenarios involving grammars for real languages, our parse table composition algorithm is an order of magnitude faster than computation of the parse table for the combined grammars, making online language composition feasible.

Contributions The technical contributions of this work are:

- The idea of parse table composition as symmetric composition of parse tables as opposed to incrementally adding productions, as done in work on incremental parser generation [Horspool 1990, Heering et al. 1990, Cardelli et al. 1994]
- A formal foundation for parse table modification based on automata
- An efficient algorithm for partial reapplication of NFA to DFA conversion, the key idea of parse table composition

- An efficient algorithm for SLR follow sets based on Digraph extended with optimizations for the characteristics of first and follow sets

6.2 MOTIVATION

In this section we motivate the need for parse table composition by considering a number of typical usage scenarios of language combination and analyzing their implementation requirements. These scenarios have in common that they require a combination of languages and that this combination is *not fixed*.

6.2.1 *Program Transformation and Generation*

The first type of application where language combination plays a prominent role is metaprogramming. Metaprograms are programs that manipulate programs as data, to transform existing programs or generate new ones. It is good practice to apply such manipulations on a *structured* representation of a program. For example, the Stratego program transformation language [Bravenboer et al. 2008] uses terms for the representation of abstract syntax trees and term rewriting for program transformation. The following Stratego rewrite rule, which lifts conditional expressions from return statements, uses terms to represent fragments of Java programs:

```
LiftConditionalFromReturn :
  Return(Some(Cond(e1, e2, e3))) ->
  If(e1, Return(Some(e2)), Return(Some(e3)))
```

The downside of the use of terms (and similar structured representations) is that the program fragments become very hard to read and write when they become larger. The mental gap between the abstract representation of program fragments and the object language syntax can be reduced by using the *concrete syntax* of the object language [Visser 2002]. The following Stratego rewrite rule uses concrete syntax to implement the exact same transformation as the rule above:

```
LiftConditionalFromReturn :
  |[ return e1 ? e2 : e3; ]| -> |[ if(e1) return e2; else return e3; ]|
```

While the program fragments are written in concrete syntax (text), the rule is interpreted as (translated to) a rewrite rule operating on terms, i.e. the abstract representation.

Template engines

Concrete syntax for program fragments is used heavily in template engines such as Velocity and StringTemplate, which are popular tools for program generation. Code generation with these tools is text-based, that is, program fragments are considered as character strings, rather than (abstract) syntax trees. As a consequence, templates are not checked syntactically, and there is no structured representation of the generated program to apply further transformations to. Exceptions include MetaAspectJ [Zook et al. 2004], which

extends Java with syntactically checked templates for generation of AspectJ code, and Repleo [Arnoldus et al. 2007], which uses the grammar of the target language to create a grammar of a template language.

Analysis

Mainstream template engines are text-based because the implementation of an engine supporting concrete syntax *and* a structured representation requires a parser for the template language and each target language. That is, the implementation of concrete object syntax requires the extension of the meta-language (e.g. Stratego, Velocity) with the syntax of the object language (e.g. Java, XML) for arbitrary combinations of object languages. Often it is necessary to support multiple languages in a single application of the metalanguage. For example, in the case of a code generator for a DSL, the grammars of the DSL itself and the grammars of the target languages (e.g. Java, SQL and XML) need to be available to produce the parser for this combination of languages. Fixing the meta-programming language to a particular set of languages reduces its scope.

Stratego, ASF+SDF [van den Brand et al. 2001] and Repleo provide a solution to parsing combinations of a metalanguage and object languages generic in the object language, based on the modular syntax definition formalism SDF [Visser 1997b]. Grammars for a particular combination of object language embeddings are compiled together with the grammar of the metalanguage into a parse table. The modularity features of SDF naturally allow the combination of arbitrary context-free languages in this way. Unfortunately, creating a parser for a different (but maybe overlapping) combination of embeddings requires *(1)* creating a new SDF module, *(2)* importing the involved extensions, and *(3)* generating a completely new parse table, i.e. without reusing the parse tables for the separate embedded languages. This entails that the instantiation of the metalanguage for a specific object language cannot be deployed as a separately compiled plugin. Rather, the parse tables of all the necessary combinations of the embeddings need to be deployed.

Deploying support for a specific object language as a single parse table for parsing the specific combination of this object language and the meta-language not only introduces a composition problem, but also hinders the evolution of the metalanguage. For example, it frequently leads to problems with the evolution of Stratego if extensions for a particular embedding are not updated when the Stratego language evolves. Indeed, the language embeddings should only depend on an *interface* of the metalanguage, not a particular implementation or revision.

6.2.2 *Language Extension and Embedding*

Another class of language combinations is that of language extension and embedding. The basic language extension scenario is the addition of new language constructs that provide some syntactic abstraction not previously available in the language, at least not with the same syntactic conciseness. A

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

Figure 6.1 Java variable declaration with initialization expression in Swing User interface Language (SWUL) to construct UI component

typical example is the addition of the for-each loop to Java 1.4, an extension that was added to the base language in Java 5. Characteristic of this type of extension is that the extension is relatively small in comparison to the base language (a few productions added) and that the lexical syntax of the base language is adopted in the extension.

Domain-specific Languages Embedding

A more disruptive type of language extension is the embedding of domain-specific languages in general-purpose languages, as implemented in approaches such as MetaBorg [Bravenboer & Visser 2004 (Chapter 2)] and Silver [van Wyk et al. 2007]. As an example consider the MetaBorg example program fragment in Figure 6.1. The fragment is a Java variable declaration with as initializer an expression in the Swing User interface Language (SWUL). The language provides a better notation than the usual series of statements needed to compose a user interface with Swing components, by following the hierarchical structure of components (e.g. create a panel consisting of label and a textarea). The embedding is realized by a syntactic extension of Java, along with an *assimilation* transformation mapping the extension to the base language.

Query and Script Embedding

A crossover between program generation and DSL embedding is the embedding of query and script languages in general-purpose languages. Many domain-specific languages are implemented by an interpreter (query/script engine) accessed through an API. Internally, the interpreter may use a structured representation for the interpreted programs, but the API often just provides a string based interface. This may be for convenience, to reduce the mental gap between concrete and abstract syntax, or the engine may run in a different process or even a different machine, requiring serialization of the query. Typical examples are SQL database queries, shell scripts, regular expressions, and XPath queries. The following Java fragment illustrates how a

query is composed from user input and passed to a database engine.

```
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0) ...
```

The generation of queries using string composition has several problems. First of all, the syntactic correctness of the query cannot be determined at compile-time and is only detected at run-time, during testing in the best case, or in production in the worst-case. Secondly, meta-characters of the host language (e.g. double quotes) need to be escaped, which can become particularly convoluted if the query languages has similar escape characters (e.g. regular expressions in Java string literals). But most importantly, the approach is vulnerable to injection attacks; user input that subverts the intended meaning of the query to gain access to restricted data. For example, passing the string `' OR 'x' = 'x` as `userName` to the query above turns the condition into a tautology, resulting in unintended access. Thus, user input should be rewritten to escape meta-characters such as the quote above. While database access APIs (e.g. JDBC) and object-relational mapping frameworks (e.g. Hibernate) provide a mechanism for passing parameters to SQL (HQL) queries that is safe for injection attacks, the use of this mechanism is not enforced, and most other query/script engines do not provide such mechanisms.

In the StringBorg approach [Bravenboer et al. 2007 (Chapter 4)] these problems are solved by embedding the syntax of the query language in the syntax of the host language and automating the escaping of user input strings. For example, the query above is written as follows with StringBorg:

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

For this example, the host language Java is extended with the syntax of SQL with queries as Java expressions of type `SQL`. A query is quoted between `<|` and `|>` and may escape to the host language using the `${...}` anti-quotation. The quotation mechanism ensures that only syntactically correct queries can be constructed, no character escaping is needed, and that string values passed as parameters via anti-quotation are properly escaped. A simple analysis then suffices to determine that only properly constructed query strings are passed to the query engine. The StringBorg method does not just work for SQL in Java, but can be applied to any combination of host language and guest languages.

Analysis

Embedding of domain-specific languages can be applied to any combinations of host and guest languages; it is not unreasonable for a compilation unit to use several DSLs. An important requirement in this type of extension is that the syntax of the embedded language is accurately described. In particular, the lexical syntax of the embedded language is often different from the lexical

syntax of the host language, which requires a technique that supports composition of lexical syntax. The state-of-the-art for realization of embeddings of complete languages (i.e. modular syntax definition and scannerless generalized LR parsing), relies on offline computation of a parse table for each combination of languages. This technology supports extensible compilers, in the sense that a compiler for the extended language can be defined without touching the implementation of the base compiler. However, it does not support binary extensibility, where a compiler can be extended by plugging in a separately developed and deployed language component. Hence, while the implementation of StringBorg is entirely generic in the object language and support for an object language is not specific to a particular metalanguage, it is still necessary to build a parse table for each combination of host and guest languages. Making the StringBorg approach really effective requires the separate deployment of syntax embedding plugins such that query engine vendors can provide a language extension with their product that does not require the compiler (or programming environment) to be rebuilt, and that allows users to combine components from different vendors. The perspective is that of DSL embeddings separately deployed as plugins to an open compiler. In order to use such embeddings, users should not have to recompile/rebuild the compiler for each combination of embeddings. Realization of this perspective requires binary extensibility of syntax and assimilation. The parse table composition mechanism introduced in this chapter provides the former.

6.2.3 *Language Conglomerates*

The language combinations in the previous scenarios are rather asymmetric; one language is the host (or meta) language, the other languages are the guest (or object) languages embedded in the former. *Language conglomerates* are languages that consist of a combination of several more or less 'equal' languages. An example is AspectJ, the aspect-oriented extension of Java, which introduces the concepts of aspect, pointcut, and advice to the language. Unlike the extensions discussed above, the extensions are not about domain-specific notation, and unlike more traditional language extension, the new concepts (especially pointcuts) syntactically diverge from the 'host' language. For example, the following pointcut definition uses a pointcut expression to indicate a set of join points.

```
pointcut cached(int value):
  execution(* Calc+.get*(int)) && args(value);
```

The syntactic structure and interpretation of pointcut expressions is completely different from regular Java expressions and symbols such as `*` and `+` have different roles. Also, depending on the context, different sets of identifiers should be considered as keywords (e.g. `execution` in a pointcut expression, but not in regular Java code).

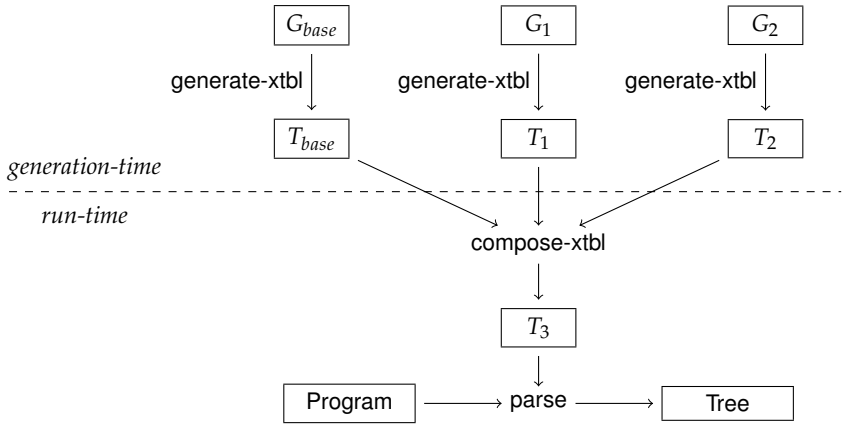


Figure 6.2 Architecture of parse table composition for a base language G_{base} with extensions/embeddings G_1 and G_2

Analysis

In [Bravenboer et al. 2006 (Chapter 5)] we describe the challenges AspectJ poses to traditional parsing techniques and how *grammar mixins* and scannerless generalized LR parsing can solve these challenges. In particular, we describe how the languages that make up AspectJ can be described as separate language components, composed into the full language using a mixin-like mechanism [Bracha & Cook 1990], which makes it possible to make multiple instantiations of a language component, for example, to make different sets of identifiers into keywords. Again, the composition is applied to the grammars for the language components, not to their parse tables. In the case of AspectJ that uses 5 instantiations of the base Java grammar, this results in a considerable cost for the parser generator (excessive memory consumption and one minute in the benchmark of Section 6.8). Separately compiling the Java grammar and then combining the AspectJ parse table using different instantiations of this single Java parse table component turns out to be much cheaper.

6.2.4 *Requirements*

In this section we have discussed several scenarios for combining languages. There is a considerable body of work on extensible compilers, addressing the implementation of language combination (e.g. MetaBorg [Bravenboer & Visser 2004 (Chapter 2)], Polyglot [Nystrom et al. 2003], Silver [van Wyk et al. 2007], JastAdd [Ekman & Hedin 2004]). The focus of this work is on optimizing the work of the compiler developer through source-level extensibility; that is, to create an implementation of an extended compiler by touching as little as possible the code of the base compiler. However, each extension (or combination of extensions) results in a different compiler. In this

work, syntactic extensibility is mostly based on the monolithic scenario discussed so far, where often less sophisticated parsing technologies are used than SDF/SGLR [Bravenboer et al. 2006 (Chapter 5)]. To support scenarios which require more dynamic configurations of language combinations, it is desirable to support *binary* extensibility. That is, extensibility that does not require rebuilding the compiler. This would make it possible to deploy a single version of the compiler, which users can then extend by plugging in separately deployed language components, where potentially each compilation unit uses a different combination of extensions.

To solve the syntactic aspect of this goal we have developed *parse table composition*, a mechanism for combining parse tables, rather than grammars. Figure 6.2 illustrates the workflow. At generation-time, grammars are compiled separately into *parse table components*. At run-time of a compiler, the parse table for the base language T_{base} is combined with a series of parse tables based on a user-selection of the desired extensions, e.g both T_1 and T_2. This results in a single parse table T_3 that is used by the parser to parse a source program. The main contribution is that composing a series of parse table components can be performed *very* efficiently, making the user of the compiler unaware of the parse table composition. Soon, the user will be familiar with the idea of a parser that parses a source file according to a series of parse table components, rather than a single one. After the parser generator has been applied to the individual components, linking the components typically just requires a minimal reconstruction of the parse table. This is a classical partial evaluation argument: as opposed to applying the full parser generation to the grammars G_{base}, G_1, and G_2, the parser generation has already been partially applied.

6.3 GRAMMARS AND PARSING

In this section we define the notions and notations for context-free grammars. Also, we review the basic concepts of the LR parsing algorithm and the generation of LR(0) and GLR parse tables.

6.3.1 *Context-free Grammars*

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

$$\begin{array}{lcl} \Sigma & = & \{+, \mathtt{N}\} \\ N & = & \{E, T\} \\ P & = & \{E \to E + T,\ E \to T,\ T \to \mathtt{N}\} \end{array} \qquad (G_1)$$

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

6.3.2 *LR Parsing*

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

An LR parser [Knuth 1965, Aho & Johnson 1974, Aho et al. 1986] is a transition system with as configuration a stack of states and symbols $q_o X_1 q_1 X_2 q_2 \ldots X_n q_n$ and an input string v of terminals. The next configuration of the parser is determined by reading the next terminal a from the input v and peeking the state q_n at the top of the stack. The entry $\mathsf{action}(q_n, a)$ indicates how to change the configuration. The entries of the action table are shift or reduce actions, which introduce state transitions that are recorded on the stack. A shift action removes the terminal a from the input, which corresponds to a step of one symbol in the right-hand sides of a set of productions that is currently expected. A reduce action of a production $A \to X_1 \ldots X_k$ removes $2k$ elements from the stack resulting in state q_{n-k} being on top of stack. Next, the reduce action pushes A and the new current state on the stack, which is determined by the entry $\mathsf{goto}(q_{n-k}, A)$.

Informally, a *handle* is the location in a string where a production can be applied in the reverse of a rightmost derivation, i.e. reducing a number of symbols to a nonterminal. Formally, a handle of a right-sentential form $\alpha\gamma w$ for S (i.e. $S \Rightarrow^*_{rm} \alpha\gamma w$) is a production $A \to \gamma$ in the position following α such that $S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\gamma w$. In the configuration of an LR parser, handles always appear on top of the stack. The symbols on the stack of an LR parser are always a prefix of a right-sentential form of a grammar. The set of possible prefixes on the stack is called the *viable prefixes*. A viable prefix is a prefix of a right-sentential form, where the prefix does not include any symbols after the rightmost handle of the right-sentential form. We do not discuss the LR parsing algorithm in further detail, since we are only interested in the generation of the action and goto tables.

6.3.3 *Generating LR Parse Tables*

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

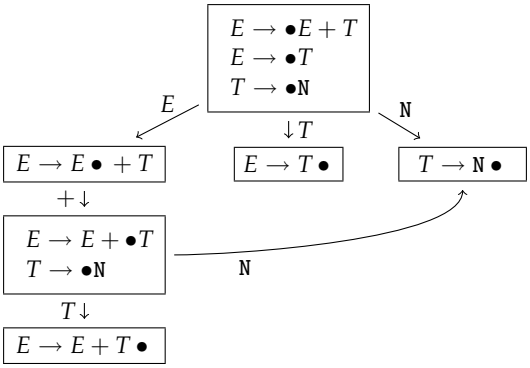


Figure 6.3 LR(0) DFA for grammar G_1

for recognizing the viable prefixes for grammar G_1 is shown in Figure 6.3. Every state of the DFA is associated to a set of items, where an item $[A \to \alpha \bullet \beta]$ is a production with a dot ($\bullet$) at some position of the right-hand side of a production. An item indicates the progress in possibly reaching a configuration where the top of the stack consists of $\alpha\beta$. If the parser is in a state q where $[A \to \alpha \bullet \beta] \in q$ then the α portion of the item is currently on top of the stack, implying that a string derivable from α has been recognized and a string derivable from β is predicted. For example, the item $[E \to E + \bullet T]$ represents that the parser has just recognized a string derivable from $E +$. We say that an item $[A \to \alpha \bullet X\beta]$ *predicts* the symbol X.

To deal with increasingly larger classes of grammars, various types of LR parse tables exist, e.g. LR(0), SLR, LALR, and LR(1). The LR(0), SLR, and LALR parse tables all have the same underlying DFA, but use increasingly more precise conditions on the application of reduce actions. We will return to SLR and LALR in Section 6.6. Figure 6.4 shows the standard algorithm for the generation of LR(0) parse tables. Because LR(0), SLR, and LALR parse tables have the same DFA the functions closure, move, and generate-dfa are the same for all these parse tables. The function generate-tbl is specific to LR(0). The main function generate-tbl first calls generate-dfa to construct a DFA. The function generate-dfa collects states as sets of items in Q and edges between the states in δ. The start state is based on an initial item for the start production. For each set of items, the function generate-dfa determines the outgoing edges by applying the function move to all the predicted symbols of an item set. The function move computes the *kernel* of items for the next state q' based on the items of the current state q by shifting the $\bullet$ over the predicted symbol X. Every kernel is extended to a closure using the function closure, which adds the initial items of all the predicted symbols to the item set.

The LR(0) specific generate-tbl procedure initializes the action and goto tables based on the set of item sets. Edges labelled with a terminal become shift actions. Edges labelled with a nonterminal are entries of the goto table.

```
function generate-tbl(A, G) =
 1  ⟨Q, δ, start⟩ := generate-dfa(A, G)
 2  for each q →_X q' ∈ δ
 3     if X ∈ Σ(G) then action(q, X) := shift q'
 4     if X ∈ N(G) then goto(q, X) := q'
 5  for each q ∈ Q
 6     if [A → α • eof] ∈ q then accept := accept ∪ {q}
 7     for each [A → α • ] ∈ q
 8        for each a ∈ Σ(G)
 9           action(q, a) := reduce A → α
10  return ⟨Q, Σ(G), N(G), start, action, goto, accept⟩

function generate-dfa(A, G) =
 1  start := closure({[S' → •A eof]})
 2  Q := {start}
 3  δ := ∅
 4  repeat until Q and δ do not change
 5     for each q ∈ Q
 6        for each X ∈ {Y | [B → α • Yβ] ∈ q}
 7           q' := closure(move(q, X))
 8           Q := Q ∪ {q'}
 9           δ := δ ∪ {q →_X q'}
10  return ⟨Q, δ, start⟩

function move(q, X) =
 1  return { [A → αX • β] | [A → α • Xβ] ∈ q}

function closure(q) =
 1  repeat until q does not change
 2     for each [A → α • Bβ] ∈ q
 3        q := q ∪ { [B → •γ] | B → γ ∈ P(G)}
 4  return q
```

Figure 6.4 LR(0) parse table generation for grammar G

Finally, if there is a final item, then for all terminal symbols the action is a reduce of this production.

Dealing with Parse Table Conflicts

LR(0) parsers require every state to have a deterministic action for every next terminal in the input stream. There are not many languages that can be parsed using an LR(0) parser, yet we focus on LR(0) parse tables for now. The first reason is that the most important solution for avoiding conflicts is restricting the application of reduce actions, e.g. using the SLR algorithm. These methods are orthogonal to the generation and composition of the LR(0) DFA. We discuss SLR tables in Section 6.6. The second reason is that we target a generalized LR parser [Tomita 1985, Rekers 1992], which already supports ar-

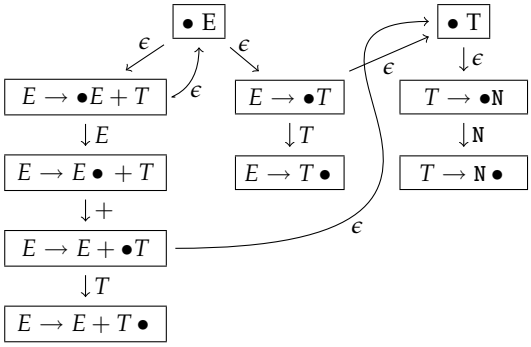


Figure 6.5 LR(0) ϵ-NFA for grammar G_1

bitrary context-free grammars by allowing a *set* of actions for every terminal. The alternative actions are performed pseudo-parallel, and the continuation of the parsing process will determine which action was correct. Our parse table composition method can also be applied to target deterministic parsers, but the composition of deterministic parse tables might result in new conflicts, which have to be reported or resolved.

6.4 LR PARSER GENERATION: A DIFFERENT PERSPECTIVE

The LR(0) parse table generation algorithm is very non-modular due to use of the closure function, which requires all productions of a nonterminal to be known at parse table generation time. If only a subset of the full grammar is known, then there is not much the algorithm of Figure 6.4 can do. To see a glimpse of a possible solution for separate compilation of grammars, we discuss a less common variation of the LR(0) algorithm in this section. This variation first constructs a non-deterministic finite automaton (NFA) with ϵ-transitions (ϵ-NFA) and converts the ϵ-NFA into an LR(0) DFA in a separate step using the standard subset construction algorithm [Grune & Jacobs 1990, Johnstone & Scott 2002]. The ingredients of this algorithm and the correspondence to the one discussed previously naturally lead to the solution to the modularity problem of LR(0) parse table generation.

6.4.1 *Generating LR(0) ϵ-NFA*

An ϵ-NFA [Hopcroft et al. 2006] is an NFA that allows transitions on ϵ, the empty string. Using ϵ-transitions an ϵ-NFA can make a transition without reading an input symbol. An ϵ-NFA A is a tuple $\langle Q, \Sigma, \delta \rangle$ with Q a set of states, Σ a set of symbols, and δ a transition function $Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$, where we use the following notation: q for variables ranging over Q; S for variables ranging over $\mathcal{P}(Q)$, but ranging over Q_D for a different automaton D; X for variables ranging over Σ; and $q_0 \to_X q_1$ for $q_1 \in \delta(q_0, X)$.

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

Figure 6.6 LR(0) ϵ-NFA generation for grammar G

Figure 6.5 shows the LR(0) ϵ-NFA for the example grammar G_1 (observe the similarity to a syntax diagram). For every nonterminal A of the grammar, there is a *station state* denoted by $\bullet A$. All other states contain just a single LR item. The station states have ϵ-transitions to all the initial items of their productions. If an item predicts a nonterminal A, then there are two transitions: an ϵ-transition to the station state of A and a transition to the item resulting from shifting the dot over A. For an item that predicts a terminal, there is just a single transition to the next item.

Figure 6.6 shows the algorithm for generating the LR(0) ϵ-NFA for a grammar G. Note that states are singleton sets of an item or just a dot before a nonterminal (the station states). The ϵ-NFA of a grammar G accepts the same language as the DFA generated by the algorithm of Figure 6.4, i.e. the language of viable prefixes.

6.4.2 *Eliminating ϵ-Transitions*

The ϵ-NFA can be turned into a deterministic automaton by eliminating the ϵ-transitions using the subset construction algorithm [Hopcroft et al. 2006, Aho et al. 1986], well-known from automata theory and lexical analysis. Figure 6.7 shows the algorithm for converting an ϵ-NFA to a DFA. The function ϵ-closure extends a given set of states S to include all the states reachable through ϵ-transitions. The function move determines the states reachable from a set of states S through transitions on the argument X. The function labels is a utility function that returns the symbols (which does not include ϵ) for which there are transitions from the states of S. The main function ϵ-subset-construction drives the construction of the DFA by considering the current DFA states and for every state $S \subseteq Q_E$ determine the new subsets of states reachable by transitions from states in S.

Applying ϵ-subset-construction to the ϵ-NFA of Figure 6.5 results in the DFA

```
function ε-subset-construction(A, ⟨Q_E, Σ, δ_E⟩) =
1  Q_D := {ε-closure({•A}, δ_E)}
2  δ_D := ∅
3  repeat until Q_D and δ_D do not change
4    for each S ∈ Q_D
5      for each X ∈ labels(S, δ_E)
6        S' := ε-closure(move(S, X, δ_E), δ_E)
7        Q_D := Q_D ∪ {S'}
8        δ_D := δ_D ∪ {S →_X S'}
9  return ⟨Q_D, Σ, δ_D⟩

function ε-closure(S, δ) =
1  repeat until S does not change
2    S := S ∪ {q_1 | q_0 ∈ S, q_0 →_ε q_1 ∈ δ}
3  return S

function labels(S, δ) =
1  return {X | q_0 ∈ S, q_0 →_X q_1 ∈ δ}

function move(S, X, δ) =
1  return {q_1 | q_0 ∈ S, q_0 →_X q_1 ∈ δ}
```

Figure 6.7 Subset construction algorithm from ϵ-NFA E to DFA D

of Figure 6.3. This is far from accidental because the algorithm for LR(0) DFA generation of Figure 6.4 has all the elements of the generation of an ϵ-NFA followed by subset construction. The ϵ-closure function corresponds to the function closure, because ϵ-NFA states whose item predicts a nonterminal have ϵ-transitions to the productions of this nonterminal via the station state of the nonterminal. The first move function constructs the kernel of the next state by moving the dot, whereas the new move function constructs the kernel by following the transitions of the NFA. Incidentally, these transitions exactly correspond to moving the dot, see line 8 of Figure 6.6. Finally, the main driver function generate-dfa is basically equivalent to the function ϵ-subset-construction. Note that most textbooks call the closure function from the move function, but to emphasize the similarity we moved this call to the callee of move.

6.5 COMPOSITION OF LR(0) PARSE TABLES

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

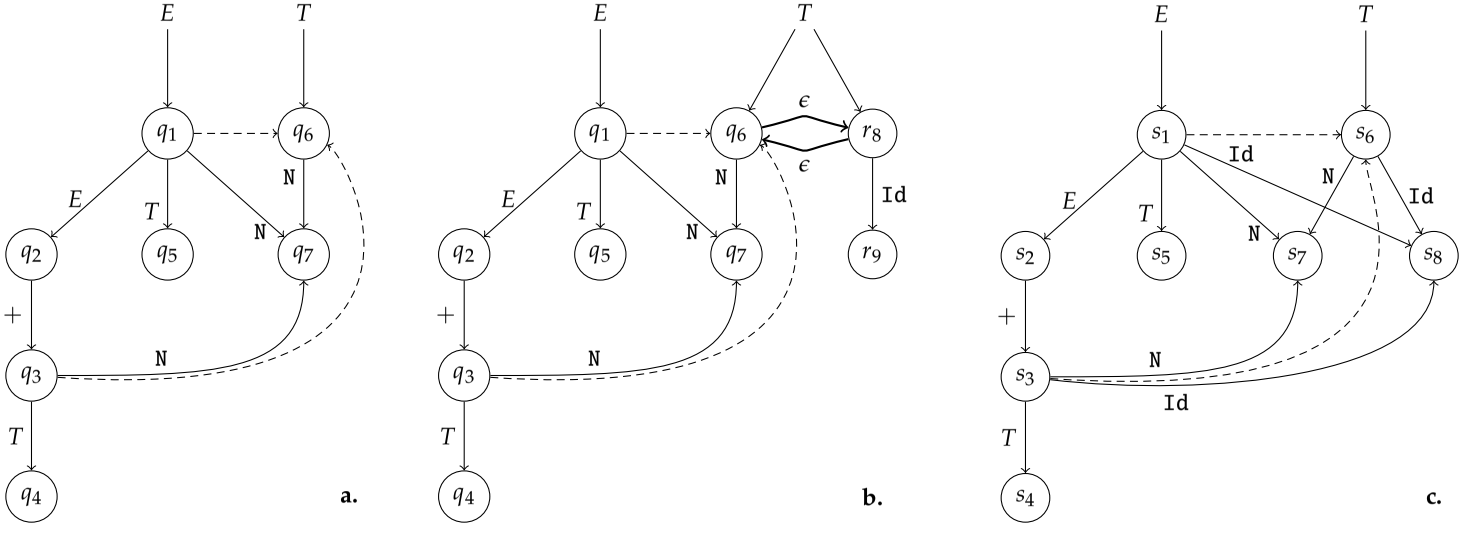


Figure 6.8 **a.** LR(0) ϵ-DFA for grammar G_1 **b.** Combination of ϵ-DFA's for grammars G_1 and G_2 **c.** ϵ-DFA after subset construction

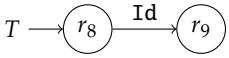
The ϵ-NFA's are in fact not much more than a different representation of a grammar, comparable to a syntax diagram.

The key to an efficient solution is to apply the DFA conversion to the individual parse table components at generation-time, but also preserve the ϵ-transitions as metadata in the resulting automaton, which we refer to as an ϵ-DFA. The ϵ-transitions of the ϵ-DFA can be ignored as long as the automaton is not modified, hence the name ϵ-DFA, though there is no such thing as a deterministic automaton with ϵ-transitions in automata theory. The ϵ-transitions provide the information necessary for reconstructing a correct DFA using subset construction if states (corresponding to new productions) are added to the ϵ-DFA. In this way the DFA is computed per component, but the subset construction can be rerun partially where necessary. The amount of subset reconstruction can be reduced by making use of information on the nonterminals that overlap between parse table components. Also, due to the subset construction applied to each component, many states are already part of the set of ϵ-NFA states that corresponds to a DFA state. These states do not have to be added to a subset again.

Figure 6.8a shows the ϵ-DFA for grammar G_1, generated from the ϵ-NFA of Figure 6.5. The composition algorithm is oblivious to the set of items (a subset) that resulted in a DFA state, therefore the states of the automaton no longer contain LR item sets. The E and T arrows indicate the closures of the station states for these nonterminals. The two dashed ϵ-transitions correspond to ϵ-transitions of the ϵ-NFA. The ϵ-DFA does not contain the ϵ-transition that would result in a self-edge on the station state E. Intuitively, an ϵ-transition from q_0 to q_1 expresses that station state q_1 is supposed to be closed in q_0 (i.e. the subset of q_0 is a superset of the subset of q_1) as long as the automaton is not changed, which makes self-edges useless since every state is closed in itself.

Figure 6.8b combines the ϵ-DFA of 6.8a with a second ϵ-DFA to form an automaton where the ϵ-transitions become relevant, thus being an ϵ-NFA. The parse table component adds variables to the tiny expression language of grammar G_1 based on the following grammar and its automaton.

$$\Sigma = \{\texttt{Id}\} \quad N = \{T\} \quad P = \{T \rightarrow \texttt{Id}\} \tag{G_2}$$



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

Figure 6.8c is the result of applying the subset *reconstruction* to Figure 6.8b, resulting in a deterministic automaton (ignoring the irrelevant ϵ-edges). State

q_1 is extended to s_1 by including r_8 because there is a new path from q_1 to r_8 over ϵ-transitions, i.e. r_8 enters the ϵ-closure of q_1. As a result of this extended subset, s_1 now has a transition on `Id` to s_8. Similarly, state q_3 is extended to s_3. Finally, station states q_6 and r_8 are merged into the single state s_6 because of the cycle of ϵ-transitions. Observe that five of the nine states from Figure 6.8b are not affected because their ϵ-closures have not changed.

6.5.1 *Generating LR(0) Parse Tables Components*

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

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

with Q a set of states, Σ a set of terminal symbols, N a set of nonterminal symbols, δ a transition function $Q \times (\Sigma \cup N) \to Q$, δ^ϵ a transition function $Q \to \mathcal{P}(Q)$ (visualized by dashed edges), station the function $N \to Q$ (visualized using arrows into the automaton labelled with a nonterminal), predict a function $Q \to \mathcal{P}(N)$, reduce a function $Q \to \mathcal{P}(P)$, P a set of productions of the form $A \to \alpha$, and finally accept $\subseteq Q$. Note that the δ function of a component returns a single state for a symbol, hence it corresponds to a deterministic automaton. The ϵ-transitions are captured in a separate function δ^ϵ. For notational convenience we do not explicitly restrict the range of the δ^ϵ function to station states in this definition. Parse table components do not have a specific start nonterminal, instead the station function is used to map all nonterminals to a station state. For the special start nonterminal, the station function will return the start state. We say that a station state q is *closed in* a state q' if the subset that was used to the construct q' includes q.

Figure 6.9 shows the algorithm for generating a parse table component, which is very similar to the subset construction algorithm of Figure 6.7. First, an ϵ-NFA is generated [1] for the grammar G. For every nonterminal A the ϵ-closure (defined in Figure 6.7) of its station state is added to the set of states Q_D [2] of the ϵ-DFA, thus capturing the closure of the initial items of all productions of A. Next, for every state, the nonterminals [8] predicted by the items of this subset are determined. Note that the predicted nonterminals of a state correspond to the ϵ-transitions to station states, or equivalently the transitions on nonterminal symbols from this state. The implementation could also preserve the full set of items, which can be used as another source of the predicted symbols. The set of predicted symbols of a state (predict) is not necessary for a naive implementation of composition, but it will help to improve the performance as we will discuss later. The ϵ-transitions [9] of a state are determined based on the predicted nonterminals, but it could also be based on δ_E. The self-edges are removed by subtracting the state itself. To drive the construction of the complete DFA, the next states [10] are determined by follow-

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

Figure 6.9 LR(0) parse table component generation for grammar G

ing the transitions of the ϵ-NFA using the move function for all labels of this subset (see Figure 6.7). Finally, the reduce actions[14] of a state are all the productions for which there is an item with the dot at the last position. If there is an item that predicts the special `eof` terminal[15] that is part of an augmented production, then the state becomes an accepting state. Note that this definition requires the items of a subset to be known to determine accepting states and reduce actions, but this can easily be avoided by extending the ϵ-NFA with reduce actions and accepting states.

6.5.2 *Composing LR(0) Parse Table Components*

We first present a high-level version of the composition algorithm that does not take much advantage of the subset construction that has been applied to the individual parse table components. The algorithm is not intended to be implemented in this way, similar to the algorithms for parse table generation. In all cases the fixpoint approach is very inefficient and needs to be replaced by a worklist algorithm. Also, efficient data structures need to be chosen to represent subsets and transitions. Figure 6.10 shows the high-level algorithm for parse table composition. Again, the algorithm is a variation of subset construction. First, the combine-xtbl function is invoked to combine the components (resulting in Figure 6.8b of the example). The δ^ϵ functions of the individual components are collected into a transition function $\delta_E^{\epsilon+}$ [2]. To merge the station states this transition function $\delta_E^{\epsilon+}$ is extended to connect the station states of the same nonterminal in different components[4]. Finally, the relations station, predict, and reduce, and the set accept are combined. The domain

function compose-xtbl$(T_0, \ldots, T_k) =$

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

function combine-xtbl$(T_0, \ldots, T_k) =$

1 $N_E \coloneqq \bigcup_{i=0}^{k} N(T_i)$
2 $\delta_E^{\epsilon+} \coloneqq \bigcup_{i=0}^{k} \delta^{\epsilon}(T_i)$
3 $\delta_E \coloneqq \bigcup_{i=0}^{k} \delta(T_i)$
4 **for** each $A \in N_E$
5 $\quad$ **for** $0 \leq i \leq k$
6 $\quad\quad$ **for** $0 \leq j \leq k, j \neq i$
7 $\quad\quad\quad$ **if** $A \in N(T_i) \wedge A \in N(T_j)$
8 $\quad\quad\quad\quad \delta_E^{\epsilon+} \coloneqq \delta_E^{\epsilon+} \cup \{\mathsf{station}(T_i, A) \rightarrow \mathsf{station}(T_j, A)\}$
9 $\mathsf{stations}_E \coloneqq \bigcup_{i=0}^{k} \mathsf{station}(T_i)$
10 $\mathsf{predict}_E \coloneqq \bigcup_{i=0}^{k} \mathsf{predict}(T_i)$
11 $\mathsf{reduce}_E \coloneqq \bigcup_{i=0}^{k} \mathsf{reduce}(T_i)$
12 $\mathsf{accept}_E \coloneqq \bigcup_{i=0}^{k} \mathsf{accept}(T_i)$
13 **return** $\langle N_E, \delta_E, \delta_E^{\epsilon+}, \mathsf{stations}_E, \mathsf{predict}_E, \mathsf{reduce}_E, \mathsf{accept}_E \rangle$

Figure 6.10 LR(0) parse table component composition

of the relations predict and reduce are states, which are unique in the combined automaton. Thus, the combined functions $\mathsf{predict}_E$ and reduce_E have the same type signature as the functions of the individual components. However, the domain of station functions for individual components might overlap, hence the new function $\mathsf{stations}_E$ is a function of $N \rightarrow \mathcal{P}(Q)$.

Back to the compose-xtbl function, we now initialize the subset reconstruction by creating the station states[2] of the new parse table. The new station states are determined for every nonterminal by taking the ϵ-closure of the sta-

tion states of all the components ($\mathsf{stations}_E$) over $\delta_E^{\epsilon+}$. The creation of the station states initializes the fixpoint operation on Q_D and δ_D [6]. The fixpoint loop is very similar to the fixpoint loop of parse table component generation (Figure 6.9). If the table is going to be subject to further composition, then the $\mathsf{predict}$ [8] and δ_D^{ϵ} [9] functions can be computed similar to the generation of components. For final parse tables this is not necessary. Next, the transitions to other states are determined using the move function [11] and the transition function δ_E. Similar to plain LR(0) parse table generation the result of the move function is called a *kernel* (but it is a set of states, not a set of items). The kernel is turned into an ϵ-$\mathsf{closure}$ using the extended set of ϵ-transitions, i.e. $\delta_E^{\epsilon+}$. Finally, the reduce actions are simply collected [14] and if any of the involved states is an accept state, then the composed state will be an accept state [15].

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

6.5.3 *Optimization*

In worst case, subset construction applied to a NFA can result in an exponential number of states in the resulting DFA. There is nothing that can be done about the number of states that have to be created in subset reconstruction, except for creating these states as efficiently as possible. As stated by research on subset construction [Leslie 1995, van Noord 2000], it is important to choose the appropriate algorithms and data structures. For example, the fixpoint iteration should be replaced, checking for the existence of a subset of states in Q must be efficient (we use uniquely represented treaps for subsets), and the kernel for a transition on a symbol X from a subset must be determined efficiently. In our implementation we have applied some of the basic optimizations, but have focused on optimizations specific to parse table composition. The performance of parse table composition mostly depends on *(1)* the number of ϵ-closure invocations and *(2)* the cardinality of the resulting ϵ-closures.

Avoiding Closure Calls

In the plain subset construction algorithm closure calls are inevitable for every subset. However, subset construction has already been applied to the parse table components. If we know in advance that for a given kernel a closure call will not add any station states to the closure that are not already closed in the states of the kernel, then we can omit the ϵ-closure call. For a kernel $\mathsf{move}(S, X, \delta_E)$ it is not necessary to compute the ϵ-closure if none of the states in the kernel predict a nonterminal that occurs in more than one parse table

component, called an *overlapping* nonterminal. If predicted nonterminals are not overlapping, then the ϵ-transitions from the states in this kernel only refer to station states that have no new ϵ-transitions added by combine-xtbl [4]. Hence, the ϵ-closure of the kernel would only add station states that are already closed in this kernel. Note that new station states cannot be found indirectly through ϵ-transitions either, because $\forall q_0 \rightarrow q_1 \in \delta^{\epsilon}(T_i) : \mathsf{predict}(q_0) \supseteq \mathsf{predict}(q_1)$. Thus, the kernel would have predicted the nonterminal of this station state as well.

To reduce the number of overlapping nonterminals it is useful to avoid unintentional overlap of nonterminals by supporting external and internal symbols. This is discussed in Section 6.7.2.

Reduce State Rewriting

If a closure ϵ-$\mathsf{closure}(\mathsf{move}(S, X, \delta_E), \delta_E^{\epsilon+})$ is a single state q, then q can be added directly to the states of the composed parse table without any updating of its actions and transitions. This is clear for the reduce and accept actions of a state, but the transitions from q also involve other states, which might not be single-state closures. Therefore, we need to choose the names of new states strategically. The state resulting from the closure of any single-state kernel (i.e. ϵ-$\mathsf{closure}(\{q\})$ is always given the same name as q. In this way, there is no need for updating transitions from single-state closures. This optimization makes it very useful to restrict the number of states in a closure aggressively. If the closure is restricted to a single state, then the compose algorithm only needs to traverse the transitions to continue composing states.

Reduce Closure Cardinality

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

A first approximation of the station states that need to be added to a kernel S to form a closure is the set of states that enter the ϵ-closure because of the ϵ-transitions added by combine-xtbl [4]

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

However, another complication compared to an ordinary transitive closure is that the station states have been transitively closed already inside their components. Therefore, we are not interested in *all* states that enter the ϵ-closure, since many station states in S^{approx} are already closed in other station

states of S^{approx}. Thus, the minimal set of states that need to be added to a kernel to form a closure is the set of station states that *(1)* are not closed in the states of the kernel (S^{approx}) and are not already closed by another state in S^{approx}:

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

The essence of the problem is expressed by the *predict graph*, which is a subgraph of the combined graph shown in Figure 6.8b restricted to the ϵ-transitions and station states. Every δ_i^{ϵ} transition function induces an acyclic, transitively closed graph, but these two properties are destroyed in the graph induced by $\delta_E^{\epsilon+}$. We distinguish the existing ϵ-transitions and the new transitions introduced by combine-xtbl [4] in intra-component and inter-component edges, respectively.

Figure 6.11 shows the optimized ϵ-closure algorithm, which is based on a traversal of the predict graph. For a given kernel S, the procedure mark first marks the states that are already closed in the kernel. If a state q_1 in the subset is a station state, then the station state itself [3] and all the other reachable station states in the same component are marked [4]. Note that the graph induced by ϵ-transitions of a component is transitively closed, thus there are direct edges to all the reachable station states (intra-edges). For a state q_1 of the subset S that is not a station state, we mark all the station states that are closed in this non-station state q_1 [7]. Note that q_1 itself is not present in the predict graph. The station states are marked with a source state, rather than a color, to indicate which state is responsible for collecting states in this part of the graph. If the later traversal of the predict graph encounters a state with a source different from the one that initiated the current traversal, then the traversal is stopped. The source marker is just used to make the intuition of the algorithm more clear, i.e. a coloring scheme could be used as well.

Next, the function ϵ-closure initiates traversals of the predict graph by invoking the visit function for all the involved station states. The visit function traverses the predict graph, collecting [10] only the states reached *directly* through inter-component edges, thus avoiding station states that are already closed in an earlier discovered station state. For every state, the intra-component edges are visited first, and mark states BLACK as already visited. The visit function stops traversing the graph if it encounters a node with a different source (but continues if the source is NULL), thus avoiding states that are already closed in other station states.

6.6 EXTENSION TO SLR

For many languages, the LR(0) parse table generation algorithm results in states where the parser can perform a shift as well as a reduce action. If the parser generator targets a deterministic parser, then the parser generation fails at this point, or applies heuristics to resolve the conflict and issues a warning. To support a bigger class of languages, the SLR (Simple LR) algorithm [DeRemer 1971] extends LR(0) by guarding the application of a reduce action for a production $A \rightarrow \alpha$ by examining the next terminal in the input stream. An

```
procedure visit(v, src)
 1  color[v] := BLACK
 2  result[v] := ∅
 3  for each w ∈ intra–edges[v]
 4     if (source[w] = src ∨ source[w] = NULL) ∧ color[w] = WHITE
 5        visit(w, src)
 6        result[v] := result[v] ∪ result[w]
 7  for each w ∈ inter–edges[v]
 8     if source[w] = NULL ∧ color[w] = WHITE
 9        visit(w, src)
10        result[v] := {w} ∪ result[v] ∪ result[w]

procedure mark(S, δ^ε)
 1  for each q1 ∈ S
 2     if q1 is station state
 3        source[q1] := q1
 4        for each q2 ∈ intra-edges[q1]
 5           source[q2] := q1
 6     else
 7        for each q2 ∈ δ^ε(q1)
 8           source[q2] := q2
 9           for each q3 ∈ intra-edges[q2]
10              source[q3] := q2

function ε-closure(S, δ^ε) =
 1  color[*] := WHITE
 2  result[*] := NULL
 3  source[*] := NULL
 4  result := S
 5  mark(S, δ^ε)
 6  for each q ∈ S
 7     if q is station state
 8        maybe-visit(q)
 9     else
10        for each qA ∈ δ^ε(q)
11           maybe-visit(qA)
12  return result

13  local procedure maybe-visit(q) =
14     if source[q] = q ∧ color[q] = WHITE
15        visit(q, q)
16        result := result ∪ result[q]
```

Figure 6.11 Optimized ϵ-closure implementation

$$\mathsf{nullable}(X_1 \dots X_n) = \{X_1, \dots, X_n\} \subseteq \mathsf{nullable} \tag{R_3}$$

$$\frac{A \to \alpha \in P(G),\ \mathsf{nullable}(\alpha)}{A \in \mathsf{nullable}} \tag{R_4}$$

$$\mathsf{first}(t) = \{t\} \tag{R_5}$$

$$\frac{A \to \alpha X \beta \in P(G),\ \mathsf{nullable}(\alpha)}{\mathsf{first}(A) \supseteq \mathsf{first}(X)} \tag{R_6}$$

$$\frac{A \to \alpha B \beta X \gamma \in P(G),\ \mathsf{nullable}(\beta)}{\mathsf{follow}(B) \supseteq \mathsf{first}(X)} \tag{R_7}$$

$$\frac{A \to \alpha B \beta \in P(G),\ \mathsf{nullable}(\beta)}{\mathsf{follow}(A) \supseteq \mathsf{follow}(B)} \tag{R_8}$$

Figure 6.12 Rules for computing nullable nonterminals, the first set of a symbol, and the follow set of a nonterminal.

SLR parse table generator determines the set of terminals that can follow the nonterminal A and a reduce action is only applied if the next terminal in the input stream is in this set. If this is not the case, then there is no derivation possible with $A \to \alpha$ applied at this point of the input, so performing the reduce action is useless.

If using a deterministic parser, then the main reason for restricting the application of reduce actions (e.g. using SLR or LALR) is to support a larger class of grammars. For a GLR parser this is not necessary: a GLR parser can perform both actions of a shift/reduce conflict and the continuation of the parsing process will determine which action was correct. Thus, the GLR algorithm applied to (possibly non-deterministic) LR(0) parse tables already supports the full class of context-free grammars. However, for GLR it is still useful to reduce the number of conflicts to improve performance of the parser by avoiding the unnecessary application of reduce actions [Johnstone et al. 2006].

The SLR algorithm determines the follow set of nonterminals by analyzing the productions of the grammar. Unfortunately, the follow set of a nonterminal can (and usually will) change if new productions are added to a grammar. Thus, the calculation of the follow set of a nonterminal requires a global analysis of the grammar. Even if a nonterminal occurs in only a single parse table component, its follow set cannot be calculated before composition-time. For this reason, we do not guard reduce actions in parse table components by their actual follow set, but by a symbolic reference to the follow set of a nonterminal. The actual follow sets are calculated at composition-time.

The rules of Figure 6.12 define the set $\mathsf{follow}(A)$ of terminals that can directly follow A. The follow set of a nonterminal is determined using two other

properties of symbols, namely the set of terminals $\mathsf{first}(X)$ that can begin a string derived from X, and the set $\mathsf{nullable}$ of nonterminals that can derive the empty string. The first, follow, and nullable sets all require a global analysis of the grammar.

6.6.1 *Nullable Nonterminals*

The set $\mathsf{nullable}$ plays a significant role in the calculation of the first and the follow sets because it determines the dependencies between the first and follow sets of symbols. For example, Equation R_8 adds a dependency of the $\mathsf{follow}(A)$ on $\mathsf{follow}(B)$ on the condition that all symbols of β are nullable. It would be useful if the dependencies between the first and follow sets would be known at generation-time, but this would impose the unacceptable restriction that the nullable property of a nonterminal cannot be different across parse table components. Hence, we cannot avoid recalculating the nullable nonterminals at composition-time and change the dependencies between the first and follow sets accordingly.

The set $\mathsf{nullable}$ is usually calculated using a fixpoint algorithm, where productions are examined until the set no longer changes in an iteration. Such a fixpoint algorithm performs unnecessary computations, but whereas we can avoid a fixpoint algorithm for first and follow sets, the nullable relation is not a graph but a hypergraph. The time spent on calculating the nullable nonterminals at composition-time is negligible (see Figure 6.14), so we decided to avoid the complexity of an efficient algorithm on hypergraphs and stick to the plain fixpoint algorithm.

The set of productions that have to be considered at composition-time can be reduced substantially at generation-time. If a nonterminal is nullable in some component, then it will always be nullable in a composition. Therefore, the parser generator can split the nonterminals into two sets, known and not known to be nullable. To examine if nonterminals not known to be nullable have become nullable in a composition, we store a nullable relation in the parse table component. For every production $A \to \alpha$ where A is not known to be nullable and α does not contain a terminal, we store the relation $\beta \to A$, where $\beta = \{B \mid B \in \alpha, B \notin \mathsf{nullable}\}$. As an additional optimization, the relation could be stored in quasi-topological order to improve the performance of the composition-time fixpoint algorithm.

6.6.2 *First and Follow Sets*

The specification of first and follow sets of Figure 6.12 translates directly into a fixpoint algorithm, as is suggested by most compiler textbooks. However, benchmarks have shown that these algorithms are too slow for use at composition-time. To find a more efficient algorithm, we observe that the calculation of first and follow sets can be expressed as a transitive closure problem. The specification of Figure 6.12 introduces a superset relation over the first and follow sets of symbols. For the first set of symbols, Equation R_6

defines that $\mathsf{first}(A)$ has a superset relation with $\mathsf{first}(X)$. This relation induces a graph, which we call the *first graph*. Similarly, Equation R_8 introduces a superset relation over the follow sets of nonterminals, thus defining a *follow graph*. The first and follow graphs are typically sparse and can contain cycles.

Since the relations introduced by the specification of Figure 6.12 are all local to a production, the parser generator can determine the edges of the first and follow graphs for every parse table component. However, the nullable conditions of Equations R_6, R_7, and R_8 are not all known to be valid or not until composition-time. Therefore, the parse table components use two kinds of edges for the first and follow graphs: edges that are known at generation-time and *possible* edges. The possible edges are only used in the composed first and follow graphs if a set of symbols attached to the possible edge is nullable in the composition. For example, Equation R_8 will introduce an edge $A \rightarrow B$ if all symbols of β are nullable in the composition. If β contains a terminal, then β can never be nullable, so the generator does not produce possible edges in this case. Also, nonterminals that are already known to be nullable can be removed from the condition.

6.6.3 *Algorithm*

The main complication in the calculation of first and follow sets is the presence of cycles in the first and follow graphs. If these were acyclic graphs, then we could just traverse the graph, calculating the first or follow sets for every node by unifying the sets of their successors. Unfortunately, applying the same method during the depth-first traversal of a cyclic graphs will result in incorrect results for the strongly connected components. All the nodes of a strongly connected component should have the same first and follow sets, which will not be the case in a straightforward depth-first traversal. Alternatively, we could apply a transitive closure graph algorithm to the first and follow graphs and calculate the first and follow sets based on the resulting transitively closed edges (successor sets), but it is more efficient to combine the traversal of the graph with set computations to calculate the first and follow sets directly. This is the essence of the Digraph algorithm [DeRemer & Pennello 1979], which was originally used for the calculation of LALR(1) lookahead sets. Digraph modifies a transitive closure algorithm based on Tarjan's strongly connected components algorithm [Tarjan 1972, Eve & Kurki-Suonio 1966] to compute lookahead sets instead of successor sets. The use of an efficient transitive closure algorithm results in a great reduction of set computations compared to a naive fixpoint algorithm for first and follow sets.

The Digraph algorithm deals with cycles in dependency graphs by identifying the strongly connected components introduced by cycles, and calculating the lookahead sets over the collapsed *acyclic* graph induced by the strongly connected components and their edges. However, it does not actually construct the collapsed graph, rather it directly applies the set computations during the discovery of components. In our work, we have adapted the Digraph algorithm to be more efficient for typical first and follow graphs. Figure 6.13

```
procedure closure(G) =
 1  for v ∈ V(G)
 2     color[v] := WHITE
 3  time := 1
 4  stack := empty
 5  for v ∈ V(G)
 6     if color[v] = WHITE
 7        visit(v)

procedure visit(v) =
 1  var originalStack := stack
 2  color[v] := GRAY
 3  d[v] := time
 4  root[v] := d[v]
 5  time := time + 1
 6  result[v] := init[v]
 7  for w ∈ Adj(v)
 8     if color[w] = WHITE
 9        visit(w)
10        result[v] := result[v] ∪ result[w]
11     else
12        unless backward(v, w) ∨ forward(v, w)
13           result[v] := result[v] ∪ result[w]
14     if color[w] ≠ BLUE
15        if root[w] < root[v]
16           root[v] := root[w]
17
18  if root[v] = d[v]
19     color[v] := BLUE
20     while stack ≠ originalStack
21        w := pop(stack)
22        result[w] := result[v]
23        color[w] := BLUE
24  else
25     color[v] := BLACK
26     push(stack, v)

function backward(v, w) =
 1  return color[w] = GRAY

function forward(v, w) =
 1  return color[w] ∈ {BLUE, BLACK} ∧ d[v] < d[w]
```

Figure 6.13 Optimized transitive closure algorithm for computing first and follow sets, based on Tarjan's strongly connected component algorithm and the Digraph LALR lookahead set algorithm

shows the adapted transitive closure algorithm we use for calculating first and follow sets.

The basis of the Tarjan's strongly connected component algorithm (and hence the Digraph algorithm) is a depth-first traversal. During the traversal, the algorithm determines if the current node v is the *root* of a strongly connected component, i.e. if it is the first node of a component discovered during the traversal of the graph. This root node is important because the computed set (fist, follow, lookahead, successor, etc.) is correct for this root node, while the result for the other nodes of the component might be incorrect. To update these possibly incorrect results, the traversal collects the nodes that constitute a strongly component on a stack. Once the root of a component has been identified, the nodes of the component are popped from the stack and their results are updated to the value of the root node.

We observed that first and follow graphs typically contain only a few cycles, i.e. there are many trivial components and just a few non-trivial components. Therefore, it is very useful to optimize the algorithm for trivial components. In the original strongly connected components and Digraph algorithms the trivial components still use stack operations for collecting the nodes that constitute a component. However, because we always know that the node that is currently being visited is a member of the component, we can reduce the stack operations by not pushing the current node on the stack until we know that it is a member of a *non-trivial* component. In this way, the stack is only used if there actually is a non-trivial component.

Line by Line

To make the algorithm more clear, we use a different presentation than usual by applying an explicit coloring scheme. All nodes are initially white and become gray if they are being visited. The color blue is used to indicate a node that is part of a component that has completely been visited. The color black indicates a node has been visited, but the component of the node has not yet been visited completely.

The main function closure initializes the color of all the nodes to WHITE [2] and sets the stack for collecting nodes of a component to empty [4]. Next, closure visits every node of the graph [7], if the node has not been visited already. The function visit first records the original stack [1] to remember which nodes to pop from the stack if the component will be finished after visiting this node. The color of v is initialized to GRAY [2], and the discovery time of the node v is set to the current time [3]. For now, the visit function assumes that the root of the component of the node v is v itself. To identify the root, we use the discovery time rather than the node itself to avoid an unnecessary indirection if the discovery time of two candidate roots needs to be compared. To set the initial values of the special character class nodes (for computing first sets) and first set nodes (for computing follow sets), the result value of the current node is initialized to a specified value init$[v]$. For most nodes this will be the empty set. Next, we visit the adjacent nodes [9] if they have not been visited already (i.e. their color is still WHITE). After visiting the node w, the result is added to

current node v [10]. Note that the edge from v to w corresponds to a *tree edge* in depth-first traversal terminology.

If node w was already visited [11], then this is either a back, forward, or cross edge. We apply an optimization that there is no need for set computations for forward and backward edges [12]. An edge is a backward edge if the color of the node w is GRAY. An edge is a forward edge if the color of w is BLUE or BLACK and the visit time of v is smaller than the visit time of w. Forward edges can be ignored because there always exists a different path to w through which the result of w was already included in the result of v. Backward edges can be ignored because including the backward edge will not add anything to the result of the root of the component of v. The result of the other nodes of the component are irrelevant because they will be updated later with the result of the root.

Next, we reconsider if v is indeed a root of the current component. If w is not in a finished component [14], and the candidate root of w has been discovered before the current candidate root of v [15], then the root of w is a better candidate root. Therefore, we update the root of v to the root of w [16].

After visiting the adjacent nodes, we check if the root of v is still v itself [18]. If this is the case, then we have discovered the root of a component and we can finish the entire component. Usually, this will just be a trivial component, i.e. consisting solely of v. In this case, the original stack will immediately be equal to the current stack. However, if this is a non-trivial component. then the function visit unwinds the stack to its original value [20] and updates the results of all the nodes that are popped from the stack to the result of v [22]. Also, all nodes are marked BLUE to indicate that this component has been visited completely [23].

On the other hand, if the root of v is not v itself, then there is a different node that is currently being visited (i.e. its color is GRAY) that will turn out to be the root. Therefore, we mark v BLACK [25] and push v on the stack [26] so that its result can be updated later. Thus, only nodes that are part of a non-trivial component are pushed on the stack.

RESOURCES Both the stack operation optimization for trivial components and the optimization for avoiding set operations for backward and forward edges are based on Nuutila's series of optimizations for strongly connected component transitive closure algorithms [Nuutila 1995]. Several more optimizations are described in Nuutila's work, but these mostly optimize the set computations for non-trivial components, which is not that useful for first and follow graphs. Proofs of the correctness of Tarjan's strongly connected component algorithm, the Digraph algorithm, and its optimizations can be found in the cited literature.

6.7 EXTENSIONS FOR LEXICAL ANALYSIS

Before syntax analysis, a lexical analyzer splits the input stream of characters into a sequence of tokens that correspond to the terminals of a context-free grammar. The reason for the division of work is the use of different techniques

for the implementation of lexical analysis and syntax analysis. The lexical syntax of a language is usually specified using a set of regular expressions, which can be recognized by a deterministic finite automaton, whereas syntax analysis requires a pushdown automaton.

Until now we have ignored the composition of the lexical syntax definition, but any solution for extensible syntax needs to consider the lexical analysis phase as well. Unfortunately, if different languages are used together in the same source file, then the finite automata-based implementation techniques for lexical syntax get more problematic because finite automata are oblivious to context. The lexical syntax of the involved languages is usually different, i.e. they have a different set of tokens. For example, the languages can have different reserved keywords, literals, operators, and use different layout (whitespace and comments). In this way, the lexical syntax depends on the context in the source file and cannot be split into a single sequence of tokens without considering this context.

A possible solution is to ignore this problem and just unify the lexical syntax of the involved languages, resulting in a lexical analyzer that recognizes all tokens in every context. This will result in lexical ambiguities if no additional precedence rules are considered for tokens. On the other hand, if such precedence rules are applied, then the composition will often not have the desired effect, for example by reserving keywords globally while they were intended to be specific to one of the involved languages (see [Bravenboer & Visser 2004 (Chapter 2)] for an extensive discussion). Another solution is to require all the involved languages to use some subset of the same lexical syntax. This is rather restrictive, but for some applications it is still acceptable, for example in languages that want to allow users to extend the syntax with some minor sugar only.

Similar context problems already occur in lexical analyzers for 'single' programming languages, for example to use a different set of tokens in the context of string literals, *here documents*, or regular expressions. These language constructs are in fact different sublanguages of the programming language, leading to a language conglomerate. This is usually solved using a stateful lexical analyzer, which recognizes a different set of tokens in every context and switches the state of analyzer if it encounters certain tokens. As discussed extensively in [Bravenboer et al. 2006 (Chapter 5)], the transitions between the lexical states are based on a careful analysis of the *complete* language. If new tokens are added by composition with other components, then the transitions might no longer be correct.

6.7.1 *Scannerless Parsing*

A scannerless parser [Salomon & Cormack 1989] directly applies syntax analysis to the characters of the input stream, i.e. there is no separate lexical analyzer. Instead of a separate specification of lexical and context-free syntax, a single grammar is used that defines all aspects of the language. The terminals of these grammars are character classes. As argued in [Bravenboer & Visser

2004 (Chapter 2), Bravenboer et al. 2006 (Chapter 5)], scannerless parsers elegantly deal with lexical context issues in syntax embeddings and extensions. Rather than parsing a lexical entity in isolation, as is done with regular expressions, the parsing context acts naturally as lexical state. Therefore, the target parser of our prototype is a scannerless parser. Scannerless parsing has been integrated with GLR parsing in the implementation of SDF [Visser 1997b], adding a few disambiguation techniques to GLR parsing to define typical restrictions and disambiguations on the lexical syntax of a language. Therefore, SDF grammars and the scannerless GLR parser have a few unusual features, which we need to support in our parse table components.

Follow(1) Restrictions

To define a preference for the longest match of what would normally constitute a token in lexical syntax, follow restrictions can be defined for nonterminals. For example, nonterminals for identifiers typically have the restriction that they cannot be followed by a character that is allowed as an identifier. A follow restriction $A \not\succ cc$ specifies that there can be no derivation where the application of a production $A \rightarrow \alpha$ is directly followed by a character from the character class cc. In an SLR parse table, follow restrictions are trivial to implement by subtracting the character class of the restriction from the follow set that guards the application of a reduce action.

In parse table components the follow sets of nonterminals are not yet known, so they cannot be subtracted. Therefore, we store this information separately and subtract the restriction after computing the follow set at composition-time. In this way, follow restrictions can also be applied to the reduce actions of different parse table components, though it might also be useful to apply the restrictions only internally.

Follow(1+) Restrictions

The scannerless parser we target also supports follow restrictions of more than a single character lookahead. Because follow(1+) restrictions involve multiple characters, they cannot be applied to the follow sets of reduce actions. Therefore, follow(1+) restrictions are an attribute of the reduce action and are checked separately by the parser. Unfortunately, the application to the reduce actions is somewhat involved: a reduce action of the production $A \rightarrow \alpha$ with follow set cc_0 that is runtime restricted by $A \succ cc_1 \succ cc_2 \succ \ldots \not\succ cc_n$ needs to be split in two reduce actions: one with follow set $cc_0 - cc_1$ and one with the follow set cc_1 and the runtime restriction $cc_2 \succ \ldots \not\succ cc_n$.

The splitting of reduce actions is the only reason a closure of a single state needs to be modified by the composer. Follow(1+) restrictions are rather uncommon, so it might be useful to avoid this overhead by changing the way reduce action with follow(1+) restrictions are represented in the parse table.

Reserved Keywords

The scannerless parser allows the definition of reserved keywords for specific nonterminals using *reject productions* [Visser 1997b]. A reject production

$A \rightarrow \alpha$ {reject} declares that $A \Rightarrow^* \beta$ is invalid if also $\alpha \Rightarrow^* \beta$. Reject production are just normal productions with a special status; therefore the parse table composition algorithm automatically supports composition of parse table components involving reject productions. Hence, parse table composition also supports component specific reserved keywords.

6.7.2 *Context-Specific Layout*

Grammars for scannerless parsers define the syntax of a language completely down to the character level, including whitespace and comments. To keep the production rules concise the SDF grammar language (which targets the same scannerless parser we target) automatically inserts the nonterminal for optional layout between the symbols of a production. Unfortunately, the layout nonterminal that is used is a global nonterminal, which makes it difficult to have a different syntax for layout in different contexts. However, for many syntax embedding applications this feature is essential. For example, in a concrete object syntax embedding of Java in Stratego we want the whitespace and comments of Java in the quotations of Java code, but the layout of Stratego outside the quotations. The layout of one language can even conflict with the syntax of a different language. For example, Java uses // as end-of-line comments, but in XPath // is part of a path expression. Clearly, in embedding of XPath in Java, the Java end-of-line comment should not be layout in the context of an XPath expression.

For this reason, our parser generator supports making nonterminals internal to a parse table component. The set of productions for this nonterminal is not extended and other parse table components cannot refer to the nonterminal. Thus, the layout of the Java concrete object syntax embedding can be internal, avoiding overlap with the layout of the host language Stratego. Incidentally, parse table components that use this feature can also be composed much more efficiently. The runtime of the composer mostly depends on the number of overlapping symbols and their number of occurrences. Layout is used everywhere in a grammar, so if the layout nonterminals overlap between two parse table components, more LR states will need to be extended and result in closures of more than a single state.

6.8 EVALUATION

In the worst case scenario an LR(0) automaton can change drastically if it is combined with another LR(0) automaton. The composition with an automaton for just a *single* production can introduce an exponential number of new states [Horspool 1990]. Parse table composition cannot cirumvent this worst case. Fortunately, grammars of typical programming languages do not exhibit this behaviour. To evaluate our method, we measured how parse table composition performs in *typical* applications. We have applied an extensive benchmark to the applications that motivated this work. For these applications, parse table components correspond to languages that form a natural

sublanguage. These languages do not change the structure of the language invasively but hook into the base language at some points.

We compare the performance of our prototype implementation[1] to the SDF parser generator (sdf2-bundle-2.4pre212034) that targets the same scannerless GLR parser. SDF grammars are compiled by first normalizing the high-level features to a core language and next applying the parser generator. Our parser generator accepts the same core language as input. The prototype consists of two main tools: one for generating parse table components and one for composing them. As opposed to the SDF parser generator, the generator of our prototype has not been heavily optimized because its performance is not relevant to the performance of runtime parse table composition. We have implemented a generator and composer for *scannerless* generalized LR SLR parse tables. This affects the performance of the composer: grammars have more productions, more symbols, and layout occurs between many symbols. Also, handling of reduce actions is more complicated due to follow restrictions.

Figure 6.14 presents the results for a series of Stratego concrete object syntax embeddings, StringBorg grammars, and AspectJ. For Stratego and StringBorg, the number of overlapping symbols is very limited and there are many single-state closures. Depending on the application, different comparisons of the timings are useful. For runtime parse table composition, we need to compare the performance to generation of the full automaton. The total composition time (col. *16)* is only about 2% to 16% of the SDF parse table generation time (col. *4*). The performance benefit increases even more if we include the required normalization (col. *3*) (SDF does not support separate normalization). For larger grammars (e.g. involving Java) the performance benefit is bigger.

The AspectJ composition is clearly different from the other embeddings. This composition is not intended to be used at runtime, but serves as an example that separate compilation of grammars is a major benefit if multiple instance of the same grammar occur in a composition. The total time to generate a parse table from source using parse table composition (col. $10+11+16$) is only 7% of the time used by the SDF parser generator.

6.9 RELATED WORK

Modular Grammar Formalism

There are a number of parser generators that support splitting a grammar into multiple files, e.g. Rats! [Grimm 2006], JTS [Batory et al. 1998], PPG [Nystrom et al. 2003], and SDF [Visser 1997b]. They vary in the expressiveness of their modularity features, their support for the extension of lexical syntax, and the parsing algorithm that is employed. Many tools ignore the intricate lexical aspects of syntax extensions, whereas some apply scannerless parsing or context-aware scanning [van Wyk & Schwerdfeger 2007]. However, except for a few research prototypes discussed next, these parser generators

[1] available at www.strategoxt.org/ParseTableComposition

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

Figure 6.14 Benchmark of parse table composition compared to the SDF parser generator. *(1)* number of reachable productions and *(2)* symbols, *(3)* time for normalizing the full grammar and *(4)* generating a parse table using SDF pgen, *(5)* number of overlapping symbols, *(6)* total number of states in the components, *(7)* number of states after composition, *(8)* percentage of single-state closures, *(9)* total time for normalizing the individual components and *(10)* generating parse table components, *(11)* time for reconstructing the LR(0) automaton, *(12, 13, 14, 15)* time for various aspects of follow sets, *(16)* total composition time ($11 + 15$). We measure time using the clock function, which reports time in units of 10ms on our machine. We measure total time separately, which combined with the accuracy of clock explains why some numbers do not sum up exactly. All results are the average of three runs.

all generate a parser by first collecting all the sources, essentially resulting in whole-program compilation.

Extensible Parsing Algorithms

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

Horspool's [Horspool 1990] method for incremental generation of LR parsers is most related to our parse table composition method. Horspool presents methods for adding and deleting productions from LR(0), SLR, as well as LALR(1) parse tables. The work is motivated by the need for efficient grammar debugging and interactive grammar development, where it is natural to focus on addition and deletion of productions instead of parse table components. Interactive grammar development requires the (possibly incomplete) grammar to be able to parse inputs all the time, which somewhat complicates the method. For SLR follow sets Horspool uses an incremental transitive closure algorithm based on a matrix representation of the first and follow relations. In our experience, the matrix is very sparse, therefore we use Digraph. This could be done incrementally as well, but due to the very limited amount of time spend on the follow sets, it is hard to make a substantial difference.

IPG [Heering et al. 1990] is a lazy and incremental parser generator targeting a GLR parser using LR(0) parse tables. This work was motivated by interactive meta programming environments. The parse table is generated by need during the parsing process. IPG can deal with modifications of the grammar as a result of the addition or deletion of rules by resetting states so that they will be reconstructed using the lazy parser generator. Rekers [Rekers 1992] also proposed a method for generating a single parse table for a set of languages and restricting the parse table for parsing specific languages. This method is not applicable to our applications, since the syntax extensions are not a fixed set and typically provided by other parties.

Dypgen [Onzon 2007] is a GLR parser generator focusing on scoped modification of the grammar from its semantic actions. On modification of the grammar it generates a new LR(0) automaton.

Earley [Earley 1970] parsers work directly on the productions of a context-free grammar at parse-time. Because of this the Earley algorithm is relatively easy to extend to an extensible parser [Tratt 2005, Kolbly 2002]. Due to the

lack of a generation phase, Earley parsers are less efficient than GLR parsers for programming languages that are close to LR.

Maya [Baker & Hsieh 2002] uses LALR for providing extensible syntax but regenerates the automaton from scratch for every extension.

Cardelli's [Cardelli et al. 1994] extensible syntax uses an extensible LL(1) parser. Camlp4 [de Rauglaudre 2003] is a preprocessor for OCaml for the implementation of syntax extensions using an extensible top down recursive descent parser.

Automata Theory and Applications

The egrep pattern matching tool uses a DFA for efficient matching in combination with lazy state construction to avoid the initial overhead of constructing a DFA. egrep determines the transitions of the DFA only when they are actually needed at runtime. Conceptually, this is related to lazy parse table construction in IPG. It might be an interesting experiment to apply our subset reconstruction in such a lazy way.

Essentially, parse table composition is a DFA maintenance problem. While there has been a lot of work in the maintenance of transitive closures, we have have not been able to find existing work on DFA maintenance.

ACKNOWLEDGMENTS

We thank Emmanuel Onzon (of Dypgen) for his valuable comments on earlier renderings of the algorithms. We thank Arthur van Dam, Rob Vermaas and Shan Shan Huang for their comments on an earlier draft of this chapter. We thank Todd Veldhuizen for a useful discussion on the use of the treap data-structure in combination with ATerms.