Context-Aware Scanning and Determinism-Preserving Grammar Composition, in Theory and Practice

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

August Schwerdfeger

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

July, 2010

Acknowledgments

I would like to thank all my colleagues in the MELT group for all their input and assistance through the course of the project, and especially for their help in providing ready-made tests and applications for my software. Thanks specifically to Derek Bodin and Ted Kaminski for their efforts in integrating the Copper parser generator into MELT's attribute grammar tools; to Lijesh Krishnan for developing the ableJ system of Java extensions and assisting me extensively with its further development; and to Yogesh Mali for implementing and testing the Promela grammar with Copper.

I also thank my advisor, Eric Van Wyk, for his continuous guidance throughout my time as a student, and also for going well above and beyond the call of duty in helping me to edit and prepare this thesis for presentation.

I thank the members of my thesis committee, Mats Heimdahl, Gopalan Nadathur, and Wayne Richter, for their time and efforts in serving and reviewing. I especially wish to thank Prof. Nadathur for his many detailed questions and helpful suggestions for improving this thesis.

Work on this thesis has been partially funded by National Science Foundation Grants 0347860 and 0905581. Support has also been received from funds provided by the Institute of Technology (soon to be renamed the College of Science and Engineering) and the Department of Computer Science and Engineering at the University of Minnesota. Opinions, findings, and conclusions or recommendations expressed in this

thesis should be understood as mine, and not reflective of the views of the University of Minnesota or of the National Science Foundation.

Dedication

Wounded I hung on a wind-swept gallows
For nine long nights,
Pierced by a spear, pledged to Odin,
Offered, myself to myself:
The wisest know not from whence spring
The roots of that ancient rood.

They gave me no bread,
They gave me no mead,
I looked down;
with a loud cry
I took up [wisdom];
from that tree I fell.

— *Poetic Edda* (W.H. Auden, trans.)

To all the Minnesotans who came before me
and to all the scientists who will come after me
I humbly dedicate this volume.

ABSTRACT

This thesis documents several new developments in the theory of parsing, and also the practical value of their implementation in the Copper parser generator.

The most widely-used apparatus for syntactic analysis of programming languages consists of a scanner based on deterministic finite automata, built from a set of regular expressions called the lexical syntax, and a separate parser, operating on the output of this scanner, built from a context-free grammar and utilizing the LALR(1) algorithm.

While the LALR(1) algorithm has the advantage of guaranteeing a non-ambiguous parse, and the approach of keeping the scanner and parser separate make the compilation process more clean and modular, it is also a brittle approach. The class of grammars that can be parsed with an LALR(1) parser is not closed under grammar composition, and small changes to an LALR(1) grammar can remove the grammar from the LALR(1) class. Also, the separation of scanner and parser prevent the use, in any organized way, of parser context to resolve ambiguities in the lexical syntax.

One area in which both of these drawbacks pose a particular problem is that of parsing embedded and extensible languages. In particular, it forms one of the last major impediments to the development of an extensible compiler in which language extensions are imported and composed by the end user (programmer) in an analogous manner to the way libraries are presently imported. This is due not only to the problem of the LALR(1) grammar class not being closed under composition, but to the very real possibility that the lexical syntax of two different extensions will clash, making it impossible to construct a scanner without extensive manual resolution of ambiguities, if at all.

This thesis presents three innovations that are a large step towards eliminating parsing as an Achilles' heel in this application. Firstly, it describes a new algorithm of scanning called context-aware scanning, in which the scanner at each scan is made aware of what sorts of tokens are recognized as valid by the parser at that point. By allowing the use of parser context in the scanner to disambiguate, context-aware scanning makes the specification of embedded languages much simpler — instead of specifying a scanner that must explicitly switch "modes" to scan on the different embedded languages, one simply compiles a context-aware scanner from a single lexical specification, which has implicit "modes" to scan properly on each embedded language. Similarly, in an extensible language, this puts a degree of separation between the lexical syntax of different language extensions, so that any clashes of this sort will not be an issue.

Secondly, the thesis describes an analysis that operates on grammar extensions of a certain form, and can recognize their membership in a class of extensions that can be composed with each other and still produce a deterministic parser — enabling end-users to compose extensions written by different language developers with this guarantee of determinism. The analysis is made practical by context-aware scanning, which ensures a lack of lexical issues to go along with the lack of syntactic nondeterminism. It is this analysis — the first of its kind — that is the largest step toward realizing the sort of extensible compiler described above, as the writer of each extension can test it independently using the analysis and thus resolve any lexical or syntactic issues with the extension before the end user ever sees it.

Thirdly, the thesis describes a corollary of this analysis, which allows extensions that have passed the analysis to be distributed in parse table form and then composed on-the-fly by the end users, with the same guarantee of determinism. Besides expediting the operation of composition, this also enables the use of the analysis in situations

where the writer of a language or language extension does not want to release its grammar to the public.

Finally, the thesis discusses how these approaches have been implemented and made practical in Copper, including runtime tests, implementations and analyses of real-world grammars and extensions.

Contents

List of Tables

List of Figures

List of Definitions

List of Theorems and Lemmas

Chapter 1

Introduction.

1.1 The parsing question.

Two of the older problems in computer science are *scanning* and *parsing*, which put together form the first phase of program compilation. Traditionally, scanning has referred to the process of breaking a string of characters into a sequence of *tokens*, which correspond roughly to words in a natural language. Then, once the scanner has finished, the parser will take this sequence of tokens and, using a set of grammatical rules, build a tree representing the program, called a *parse tree*, which describes the structure of the string of tokens.

For programming languages, there are algorithms for both scanning and parsing that have seen many years of use and are perhaps more popular than any alternatives. Scanners, if not completely *ad hoc* hand-coded programs, are usually built by compiling a series of regular expressions (regexes, for short) into one or more deterministic finite automata (DFAs). Simpler as well as more complex alternatives have been suggested [Kan04, For04], but the DFA-based model remains the standard, used in popular scanner generators such as Lex [LMB92] (and its derivatives, such as JFlex [Kle09]).

Parsing of programming languages, also, has had its *de facto* standard since 1965, when Don Knuth published a seminal paper on the subject of parsing [Knu65], introducing the LR parsing algorithm. Today, this algorithm is still very widely used in the parsing of programming languages; popular parser generators such as Yacc [LMB92] (and its derivatives, such as CUP [HFA+06], and SableCC [Gag98]) use its LALR(1) variant as their primary or sole algorithm of parsing, and the official grammars of major languages such as ANSI C [KR02] and Java [GJS96] are specifically designed with LALR(1) compatibility in mind. The typical parsing framework consists of such an LALR(1) parser coupled with a DFA-based scanner operating separately from the parser.

LR's continued dominance would lead some to ask the question: *Why is parsing not a dead field?* The tenacity of the LR algorithm is not at all due to a lack of alternatives. The well-established LL parsing algorithm and associated mature tools such as the LL(*) parser generator ANTLR [PQ95] certainly provides a viable alternative. From LL, to the nondeterministic extensions to LR such as GLR [Tom86] and SGLR [Vis97, vdBSVV02], up to the very recent development of practical packrat parsers [WDM08], new and improved parsing algorithms have continuously been introduced. Nor is it due to the LR algorithm being particularly intuitive or "mathematically beautiful," or to LR parsers being easy to specify. To have a deterministic parser built for a grammar within the traditional LALR(1) framework, the grammar must be in the LALR(1) class of grammars — a somewhat restricted body. Writers of LR grammars often must go to some trouble reworking the grammars to resolve *parse-table conflicts* and thus ensure that the grammars are in the LALR(1) class. An LR parser traverses the input in a strictly left-to-right order and builds its parse trees in a bottom-up order, holding the children of subtrees still to be built in a *parse stack*; a parse-table conflict occurs when the grammar is ambiguous, or is structured in such a way that the LR parser would need

to look more than one token beyond the current position to determine what operation to take on this stack.

For example, an LALR(1) parser for a grammar that defines an arithmetic expression either as a number or recursively as two expressions joined by a '+' symbol ($E \rightarrow n \mid E + E$) would contain a parse table conflict, because the parser parsing the expression $n+n+n$, having pushed $n+n$ onto its parse stack, and seeing the second $+$, cannot tell whether to build the stack contents into an expression, eventually yielding $(n+n)+n$, or push the $+$ onto the stack and continue, eventually yielding $n+(n+n)$. Although this is an ambiguous grammar that must be rewritten to eliminate the conflict, there are also many unambiguous grammars that have the same problem.

Additionally, the separate DFA-based scanners traditionally used in conjunction with LALR(1) parsers can be difficult to construct, as the grammar writers must avoid *lexical ambiguities* (*a.k.a.*, *lexical conflicts*), which occur when a part of the input could possibly be interpreted as two different sorts of tokens; the most familiar example of a resolved lexical conflict is the reservation of a keyword, such as `while` or `int` in C, against the alphanumeric identifier token.

Despite these shortcomings, the traditional LALR(1) framework is actually a good fit for most kinds of programming languages. Its guarantee that a grammar is unambiguous — a problem that is undecidable in the general case — is good to have, and the consequent requirement that there be no parse-table conflicts is not unreasonably restrictive in these cases. There is also a guarantee that one will end up with the same parse tree for a given input no matter what priority one gives to the productions of the grammar — unlike with parsing expression grammars (PEGs).

However, there are some languages for which the framework is of limited utility. One reason for this is that there is no feedback from the parser to the scanner in the traditional framework. Another is that the framework is "brittle": seemingly innocuous

changes to a grammar in the LALR(1) class can place it outside the LALR(1) class. This particular shortcoming, as we discuss in section 1.2.1, is especially a problem with respect to the development of extensible languages.

1.2 A vision for extensible compilers.

The motivation for the work presented in this thesis derives from a broader vision of extensible languages and compilers, in which extensions to a given *host* language may be developed by several independent extension writers, and in which the programmer — the end-user of the extended language, who may not be an expert in scanning and parsing, and who may not know the inner workings of the compiler — is the one who selects the language extensions to be composed with the host language.

This vision of extensible compilers, in which the programmer chooses the extensions to use, may be analogized to another technique of "language extension": *libraries*. When programmers need a language to be extended with new functionality that requires no change in syntax or semantics, they will generally import libraries of code or objects providing that functionality. More importantly here, the library writer can compile and debug the code of his/her library before it is distributed, thus *providing a guarantee to the programmer that s/he can use any combination of libraries in a program without the worry that they will conflict with each other*. We seek a similar guarantee for language extensions that may also provide new syntax and semantics; our approach provides this guarantee with respect to syntax, while the other considered approaches do not.

There is also the issue of *where* expertise and knowledge of programming language development may be needed in the process of constructing the scanner and parser that can parse the extended language consisting of the host composed with several extensions. Traditionally, language extensions have been composed with the host language

by experts who have a profound understanding of the language and of parsing technology. In some cases, this is intentional, such as in the case of AspectJ (discussed below) or of the extensions within the JastAdd extensible compiler framework for Java [EH07].

On the other hand, we envision a system in which there is no expert to perform the final composition, in which all necessary tests on each extension are performed independently by the extension writer in the same way that libraries are presently tested, in which the end user can choose the combination of extensions s/he needs and have a guarantee that they will all compose together automatically.

It is this vision of extensible languages and compilers, in which extensions function analogously to libraries, that would have far broader applications. If extensible languages are to become more widely adopted, analyses are needed that allow the individual extension writer to provide a *guarantee* that his/her language extension can be *safely and automatically* composed by any non-expert programmer, rather than simply leaving it to be cobbled together by an expert, as the practice has been.

1.2.1 Parsing as an Achilles' heel.

In the realization of this vision for extensible compilers, the problem of parsing has long been an "Achilles' heel." One reason for this is that the LALR(1) framework's "brittleness" significantly limits its utility in parsing extensible languages, as language extensions are rarely drastic alterations to the grammar they extend but often alter the grammar they extend in just the minor way that causes an entirely unexpected conflict — and even if two extensions can each be added *separately* without taking a grammar out of the LALR(1) class, adding them *together* might still take it out. However, it is not only the LALR(1) framework that exhibits this problem of non-modularity.

If it is assumed that the person composing the language is an expert, one may choose

to use a GLR-based framework, because the expert will be able to make the extensive tests and manual analyses needed to ensure that composing two unrelated extensions did not cause any grammatical ambiguities. One could also use a PEG framework, and in the situation — unlikely, though not unreasonable — that two extensions introduce the same syntax, the expert could determine the order in which the extensions are added, thus determining which of the overlapping constructs is recognized.

But if, as is the case with the vision of extensible compilers discussed above, it is *not* assumed that an expert is available at the point of composition — that any programmer could, with no expertise in parsing or the inner workings of compilers, pick and choose extensions from a variety of sources and put them together into an extended compiler — then GLR- and PEG-based frameworks, which will ultimately require the person who composes the grammars to be an expert, cannot be properly employed.

The non-modularity of these parsing approaches also does not sit well with the nature of the semantic analysis that, in the compilation process, follows the phase of parsing or syntactic analysis. Most of the semantic analysis and functionality specified in language extensions must be set in a *modular* framework: it is impermissible for the analysis performed by one extension to alter the analysis performed in another extension, as then the extensions are not functioning in the intended way.

On the semantic level, most semantic analyses work over *abstract syntax trees*, which are highly structured and allow for easy differentiation between host and extension constructs (usually, in an abstract syntax tree, each extension construct will be a subtree unto itself, and one need only look at the root node to differentiate).

It is important to note that current methods of semantic analysis in this area do not provide all that may be desired, either. There is not yet a way to provide *any* guarantee that the semantic analysis of two extensions will not conflict, much less a guarantee based on tests run by the extension writer; this is a problem to which a solution is

actively being sought. However, due partially to the fact that the input to the semantic analysis phase is highly structured, there is considerably less risk that such a conflict will occur.

On the other hand, the input to a parsing algorithm is a string of characters rather than an abstract syntax tree. This can pose more of a problem for a modular system, as a string of characters, or of tokens, is a highly unstructured piece of data. It is difficult to tell with any ease or certainty what part of such a string belongs to an extension construct, and what part belongs to a host construct; one cannot even tell, without parsing it first, what part of a string represents such constructs as the body of a function or a class declaration. This makes the process of parsing much more *brittle* than that of semantic analysis; a single syntactic ambiguity or parse table conflict can prevent a parser from being built, depriving a compiler of an essential part of its front end.

Previous attempts to address this problem in practice — such as the `abc` compiler for AspectJ considered as a means of extending Java — have been *ad hoc* in nature and focused on the scanner, thus both restricting the kinds of extension constructs that can be used, and not permitting much modularity in the process of incorporating extensions. Furthermore, efficient parsers are, by and large, monolithic programs that do not allow straightforward modification *in situ*.

We now briefly discuss two example grammars demonstrating the Achilles' heel status of parsing, as they are not good fits with the traditional LALR(1) framework, and furthermore illustrate the benefits of the modifications to that framework presented in this thesis.

1.2.2 ableJ.

ableJ [VWKSB07] is an extensible compiler framework for Java 1.4, allowing extensions to Java (such as AspectJ, discussed in the next section) to be specified easily and

declaratively through the use of attribute grammars. ableJ is meant to be a model of our vision for extensible compilers, with any number of grammar writers able to write extensions to Java within the framework, and that the framework's end user (the programmer) may then choose whatever combination of these extensions is necessary to his/her task and compose them automatically with the host Java language. One obstacle to this goal is that an extension whose semantics can be added easily may not always fit so seamlessly into Java's concrete syntax.

More information about ableJ's corpus of extensions appears in chapter 9. This thesis focuses on the problem of building one parser that will parse a host language (Java, in this case) with all desired extensions — some of which may be written by different people who are not in communication, and composed by the end-user of the composed language, a programmer who is not necessarily a grammars expert. Since the class of grammars that can be parsed with the LALR(1) algorithm is not closed under composition, adding two such extensions to a host language may render it incapable of being parsed in an LALR(1) framework, even if each of the extensions can be added separately to Java and those languages are all LALR(1). Also, the usual scanner's lack of ability to consider context will create problems with keywords that may not be foreseeable by the writers of the extensions; *e.g.*, if a keyword is reserved against identifiers in one extension, that keyword cannot be matched as an identifier even in the context of another extension.

Consider the example in Figure 1.1 on the following page, demonstrating three of the extensions implemented with it: condition tables, SQL embedding, and generics (introduced as a standard feature in Java 1.5). There are three particular ways in which this code is interesting from the scanning and parsing standpoints; the points of interest are marked by underscoring.

Firstly, `SELECT` is used in two different ways: as a variable name or identifier in

```
 1. class Demo {
 2.   int demoMethod () {
 3.       List<List<Integer>> dlist ;
 4.       int SELECT ;
 5.       int T ;
 6.       connection c "jdbc:derby:/home/testdb"
 7.         with table person [ person_id INTEGER,
 8.                             first_name VARCHAR,
 9.                             last_name VARCHAR ] ,
10.              table details [ person_id INTEGER,
11.                              age INTEGER,
12.                              gender VARCHAR ] ;
13.       Integer limit ;
14.       limit = 18 ;
15.       ResultSet rs ;
16.       rs = using c query
17.         { SELECT age, gender, last_name
18.           FROM person , details
19.           WHERE person.person_id = details.person_id
20.           AND phonebook.age > limit
21.         } ;
22.       Integer age = rs.getInteger("age");
23.       String gender = rs.getString("gender");
24.       boolean b ;
25.       b = table ( age > 40      :  T * ,
26.                   gender == "M" :  T F ) ;
27.   }
28. }
```

Figure 1.1: Example of ableJ code with three extensions. Adapted from [VWS07].

Java code (line 4) and as a reserved keyword in the SQL extension (line 17). The typical way of specifying reserved keywords for a traditional scanner is to give the regular expression for the reserved keyword higher precedence, meaning that in all cases where the reserved keyword can be matched instead of the identifier, it will be. Although this is how reserved keywords are supposed to work, in this case it means that a nondeclarative work-around, turning the "reservation" on or off as required, would need to be specified.

Secondly, `table` is used as a keyword both in the SQL extension (lines 7 and 10) and the tables extension (line 25), and a traditional scanner will not know which one to match if it sees the string `table`. Furthermore, if the SQL and tables extensions had been developed separately, neither the developer of the SQL extension nor the developer of the tables extension could have anticipated this lexical conflict.

Thirdly, line 3 is a parameterized variable declaration that contains two closing angle-brackets next to each other. A straightforward, intuitive grammar for this sort of type expression might be

- $Type \rightarrow Id \mid Id$ ‘<’ $Type$ ‘>’

In this grammar, the parser sees each > as a separate token. However, with a traditional scanner, the Java operator for the rightward bit shift, >>, gets in the way, even though the scanner's returning >> at that point would cause a syntax error. The traditional scanner works on the principle of "maximal munch," which stipulates that if the scanner can match several strings of different lengths, it should return the one of greatest length, which in this case is >>. This requires that a more convoluted grammar be constructed:

- $Type \rightarrow$

  $BaseType$

| *BaseType* ‘<’ *BaseType* ‘>’

| *BaseType* ‘<’*BaseType* ‘<’ *Type* ‘>>’

- *BaseType* → *Id*

But while this workaround is functional in Java, it does not work in C++, which uses a similar construct for declaring parameterized types. In C++, the programmer must separate the right angle brackets with spaces to avoid having pairs of them scanned as >>.

In this thesis we present methods of parsing and analyses of language extensions that solve the problems caused by context-insensitive application of maximal munch.

1.2.3 AspectJ.

AspectJ [KHH+01] is a language that introduces *aspect constructs* to Java, which is interesting from the scanning and parsing perspectives. Although it is not one of the modular extensions for which ableJ was designed, it is an extension to Java that illustrates some of the same parsing issues.

Aspect constructs form a new way of arranging programs known as *aspect-oriented software development* (AOSD) [FEC04], which aims to give a way to put all parts of any programming “concern” in one place or module, as visitor patterns in Java allow for implementing, in one file, a new algorithm across several classes. In AspectJ, this manifests itself in the form of these aspect constructs, which allow, for example, a new method to be added to a Java class by specifying it in an aspect in a different file.

A traditional separate DFA-based scanner for AspectJ is impossible to specify, as AspectJ has several new keywords that cannot be used as identifiers in the context of an aspect construct, but can be used as identifiers in other contexts; see Figure 1.3.1 on page 13 for an example with the AspectJ keyword `after`, used as the name of a

method in the class `X`, but in its keyword capacity in the aspect `MoveTracker`. There are also instances, discussed in section 2.4.6, where the same string must be separated into different numbers of tokens based on the context. But the usual scanner, being entirely separate from the parser, cannot take context into account to resolve either of these issues.

```
 1. class X
 2. {
 3.     boolean after() { return false; }
 4. }
 5. aspect MoveTracker
 6. {
 7.     ...
 8.     after():  moves() { flag = true; }
 9.     ...
10. }
```

Figure 1.2: AspectJ code example, adapted from [KHH+01].

This problem can be remedied by using a custom-coded scanner or a scannerless generalized LR (SGLR) parser (see section 2.3.5). However, the custom-coded scanner is not a declarative solution, and the SGLR parser, though it is declarative, is also non-deterministic. Our parser for AspectJ, on the other hand, is both deterministic and declarative.

1.3 The contributions of this thesis.

In this thesis is presented a new framework for parsing and scanning that is based on the LALR(1) algorithm used in conjunction with a modified type of DFA-based scanner called a *context-aware* scanner. Being LALR(1)-based, it retains the determinism

verification of the traditional LALR(1) framework without introducing the problems of the universal determinism of PEGs; it also retains the $O(n)$ runtime of a traditional LALR(1) parser, though at the expense of a slightly larger memory footprint, and offers gains in flexibility and declarativeness over the traditional LALR(1) framework. Although the SGLR framework offers still more flexibility, this comes at the cost of the determinism verification.

Following is a summary of the new framework and its capabilities.

1.3.1 Context-aware scanning.

In any state of a parser, there is a set of terminals deemed to be syntactically valid, which may vary as the state varies. There are also terminals in each state that are deemed invalid and if the scanner returns one, it is counted as an error. This set of terminals deemed valid is called the "valid lookahead set" for the parser state.

For example, in many programming languages, in a state reached when the parser is at the start of an expression construct, terminals that are in the valid lookahead set include numbers, identifiers, prefix unary operators (such as +, -, ++, *etc.*), left parentheses, and anything else that is valid at the start of an expression. Not in the valid lookahead set are symbols that cannot occur at the start of an expression, such as infix binary operators (`/`, `&`, `|`, *etc.*) and right parentheses.

In a traditional framework, the scanner does not know the state the parser is in when making any scan, or what terminals are valid syntax at that point; it simply returns a token based on the characters it reads, and the parser continues or fails accordingly. This is promoted as a principle of clean, modular compiler design in Aho *et al.*'s seminal text *Compilers: Principles, Techniques, and Tools* [ASU86], and keeping the two entirely separate is certainly very desirable and beneficial, when it can be done.

However, having a separate scanner simply does not work on a class of reasonable

languages, including the examples cited above. The reason for this is that at every scan the scanner must be prepared to match any terminal in the language, and if these terminals have regular expressions with overlapping languages, a scheme must be worked out for resolving lexical ambiguities without recourse to any context.

A context-aware scanner takes advantage of the notion of the valid lookahead set by, at each scan, looking only for those terminals that are in the valid lookahead set for the present parse state. This allows the same string to be read as different terminals depending on the parse state, thus resolving such problems.

Consider the above examples. In Figure 1.1, when scanning the generic expression, the scanner will subordinate the principle of "maximal munch" to staying within the valid lookahead set, and instead of matching >> (not in the valid lookahead set) it will match > (in the valid lookahead set). In Figure 1.2, when parsing the first occurrence of `after`, the scanner will be called from a parser state used for parsing Java constructs in which the AspectJ keyword *after* is not in the valid lookahead set and the string `after` will be matched as an identifier. The second occurrence is within an aspect construct, where the keyword *is* in the valid lookahead set, and the string will be matched as such. In Figure 1.1, the same goes for the occurrences of `SELECT` and `table`.

Context-aware scanning can, in theory, be used with any parsing framework in which it is possible to run the parser and scanner in lock-step; in this thesis, we discuss the abstract idea of context-aware scanning independent of a specific framework, as well as its particular application to the LR framework.

In the LR framework, a convenient and automatic method for determining the valid lookahead set in a state is by examining the state's valid parse actions. Corresponding to each state are actions telling the parser what to do; different actions for when the scanner returns different terminals. On a terminal there might be a *shift* action telling the parser to consume the terminal, or a *reduce* action telling the parser to pop some

number of elements off the stack and push on a parse subtree built from those elements, or an *accept* action indicating that the parse is finished. The valid lookahead set for the state can then be simply the set of terminals with parser actions.

The specifics of the context-aware scanning algorithm are discussed in chapter 3.

1.3.2 Modular determinism analysis.

Although context-aware scanning by itself allows for more expressivity in grammars, it does not tackle other issues in the abovementioned ableJ framework. Namely, when importing more than one of its characteristic extensions, one would like to have a guarantee that, if several extensions are compatible semantically, and each one composes with the "host" grammar (the Java grammar in the ableJ case) and generates a conflict-free parser and a scanner with no lexical ambiguities, then the composition of those *several* extensions with the Java grammar will also generate a conflict-free parser and a scanner with no lexical ambiguities.

This is difficult, because the class of grammars that produce conflict-free LALR(1) parsers is not closed under composition. However, this guarantee can be made by adding a set of restrictions on the extension grammars, which are presented here. These restrictions allow this guarantee to be made, yet are not so restrictive as to exclude many useful extensions, such as those that had already been written for ableJ. These restrictions are discussed in detail in chapter 6; broadly, they take the *LR DFA* — a finite state machine built by the parser generator from the context-free grammar representing the composition of host and extension — and ensure that this LR DFA is partitioned so that when all the extensions are composed and the resulting LR DFA is compiled into a parser, there will be no conflicts. The partitions are:

1. A partition of states used to parse only host-language constructs, which were not

introduced as a result of an extension being added. An example of this sort of state is the start state of the LR DFA. This partition is to correspond exactly to the LR DFA built for the host language alone, with the exception of new items and lookahead involving extension marking terminals (terminals that must come at the beginning of all extension constructs; see the list of further restrictions immediately below).

2. A partition of states used to parse only host-language constructs, but which were introduced by one or more extensions. An example of this sort of state is the one the parser is in after consuming the colon on line 8 of Figure 1.2 on page 12 and is parsing Java code again (the function name `moves`) but is still within the aspect construct.

3. For each extension, a partition of states used to parse only constructs for that extension. An example of this sort of state is the one the parser is in after consuming the keyword `connection` on line 6 in Figure 1.1 on page 9.

Further restrictions are as follows:

- A formal restriction on the grammar: each extension is required to have exactly one production with a nonterminal for the host language on the left-hand side, and this one production's right-hand side is required to start with a unique terminal μ_E, the *marking terminal*. In other words, this marking terminal must prefix all constructs in the extension language. This stops the extension partitions from overlapping.

  Note that no similar terminal is required at the *end* of a construct in the extension language, as is the case in simpler approaches to language extension such as island grammars [SCD03].

- A restriction on the *follow* sets of the grammar. A nonterminal's follow set is the set of terminals that can validly follow some construct derived from the nonterminal. The restriction is that no extension can introduce any new terminals, except for its marking terminal, to the follow set of a nonterminal belonging to the host grammar. This stops extensions from following an embedded host construct with different — possibly conflicting — extension constructs.

These restrictions have the aim of ensuring that those states falling into the several extension partitions are identical (except for strictly circumscribed sorts of changes) to those in the DFA created when that extension is composed alone with the host, so that if the host and extension composed by themselves generate a deterministic parser, composition with several extensions *preserves* that determinism.

This guarantee ensures that one can freely compose any number of extensions thus "certified" without fear of parse-table or lexical conflicts. Also, not only can this composition be carried out at the grammar level (discussed in chapter 6), but the partitioned nature of the composed parse table also allows it to be created by composing parse tables from the individual extensions (discussed in chapter 7).

1.4 Thesis outline.

The outline of the rest of this thesis is as follows. Chapter 2 presents background and related work. Chapter 3 presents its first primary theoretical contribution, context-aware scanning. Chapter 4 discusses a number of other modifications to the framework needed to make context-aware scanning practical and best utilize it. Chapter 5 discusses two ways to implement a context-aware scanner and the practical modifications. Chapter 6 presents the second primary theoretical contribution, an analysis that improves the

flexibility of the context-aware framework by addressing the extension-merging problem of languages such as ableJ. Chapter 7 discusses a way to utilize this analysis in aid of rapid merging of these grammars. Chapter 8 gives a time complexity analysis of each implementation as well as charts with showing runtime comparisons. Chapter 9 discusses applications for the context-aware framework, two of which were also discussed in this chapter, and chapter 10 concludes. Appendix A contains full grammars of example applications.

Chapter 2

Background and related work.

This chapter contains a discussion of the background of the parsing problem, including a presentation of several axes or criteria by which to measure *frameworks* for scanning and parsing. The framework consists of both the formalism used to specify the parser (*e.g.*, a grammar in Backus-Naur form, or *BNF* for short, and a list of regular expressions) and the algorithms on which the parser is based (*e.g.*, LALR(1) and scanner DFAs). The chapter also contains some illustrative examples, as well as a discussion of background on traditional scanning and LR parsing, and of related work.

2.1 Evaluation criteria.

2.1.1 Evaluating parsers and frameworks.

As demonstrated in section 1.2, there are desirable languages that cannot be parsed in the traditional LALR(1) framework; however, there are a wide range of alternatives, including the age-old approach of coding a scanner and parser by hand. Therefore,

criteria are needed by which to compare scanners, parsers, and frameworks for parsing.

It is *essential* that a grammar for a programming language be *unambiguous*, *i.e.*, for each string in the corresponding language, a parser built for it must return exactly one parse tree, obtained via a clear, intentional process that has been specified entirely at parser compile time by the parser's author. This removes from consideration not only ambiguous parsers, but parsers that use probabilistic methods [NS06] to remove ambiguities. Note that it is all right if a parser is built using a nondeterministic algorithm such as the scannerless generalized LR (SGLR) algorithm (see section 2.3.5), as long as the parser always returns an unambiguous parse in the form of either a single parse tree or a parse error.

It is desirable that a parsing framework incorporate *verifiability*, *i.e.*, that there be a guarantee that a scanner and parser will function correctly on all inputs. For example, if an LALR(1) parser has a conflict-free parse table there are guarantees that it will not produce an ambiguous parse and that all unambiguous parses are correct, and a traditional DFA-based scanner will always match the longest possible lexeme (maximal munch) to the highest-precedence regular expression. On the other hand, a GLR-based parser does not have a guarantee that it will produce an unambiguous parse, and a packrat parser does not have a guarantee that all unambiguous parses are "correct," in that altering the rule order can change a grammar in undecidable ways.

It is desirable that a parser be *efficient*, in the senses of both time and space — fast with a small memory footprint.

It is desirable that a parsing framework be *expressive* — able to build parsers for a broad range of grammars. See Figure 2.16 on page 65 for a Venn diagram showing the classes of grammars expressible in various parsing frameworks. Each of these frameworks is described more fully below.

It is desirable that a parsing framework be *flexible*, *i.e.*, the framework be able to

accommodate changes to the grammars for which parsers are built within it. More precisely, if a larger number of changes can be made to a grammar whose parser fits in the framework, while still keeping the resulting parser in the framework, that framework is more flexible. As an example, one complaint commonly levied against the traditional LALR(1) approach is that it is inflexible, because even if a grammar compiles without any parse table conflicts, small changes to the grammar can introduce a conflict.

It is desirable that a parsing framework be *declarative*, *i.e.*, a greater proportion of scanners/parsers can be specified within the framework without using extraformal means of any sort. For example, in tools such as *Lex*, this is compensated for by having a block of code follow each regular expression, which is run if that regular expression is matched, that generates the necessary information about the terminal to return in that case. Often this code will be a simple return statement identifying the terminal; if it is not, the scanner has not been specified in an altogether *declarative* manner. An example is the typename/identifier ambiguity commonly found in LALR(1) grammars for ANSI C. In C, both typenames and identifiers have the same regular expression (`[A-Za-z_$][A-Za-z0-9_$]*`); the scanner must decide upon matching this regular expression whether it should be matched as a typename or identifier, which is done by a block of code that reads a list of typenames previously defined by use of `typedef` and matches a typename only if the matching string is in the list.

Of these criteria, flexibility, expressivity and declarativeness are more subjective measurements, while non-ambiguity, efficiency and verifiability are objective and easily quantified. Some of these measurements are presented in chapter 8.

Most of the criteria, while desirable, are not essential in the general case. The only one that is required is non-ambiguity: one can do without declarativeness (for example, the use of scanner modes in Lex, which requires that the mode be changed by user-written code, makes a specification non-declarative) and sacrifice efficiency

and verifiability in some cases, but when parsing programming languages, a parser that does not return an unambiguous parse every time is completely useless. For example, grammars written in the GLR-based franeworks, which does not feature verifiable non-ambiguity, must go through an extensive manual debugging process [Vin07] to ensure that they are not ambiguous. A single ambiguity is treated as a bug to be removed rather than a shortcoming to be tolerated.

The traditional LALR(1) framework has held on because it is verifiable, providing a guarantee of determinism — in the LALR(1) framework, no parser is ambiguous — and measures up reasonably well with regard to the other four criteria.

2.1.2 Blurring the parser-scanner dividing line.

Traditionally, the processes of scanning and parsing have been kept altogether separate and disjoint, with the input string being run through the scanner, and the parser then being called on the output of the scanner. The tools that generate the scanner and parser are also separate: Lex and Yacc for C, JFlex and CUP for Java, and many other scanning and parsing tools, while generally used in conjunction, are developed separately.

Although this rule of separation is easily broken in practice via global data structures accessible by both scanner and parser, and while several tools have in the past brought parser and scanner under one roof as a practical optimization measure, they did not, for the most part, deviate from pre-existing frameworks. Examples of integrated parser-scanner generators are CoCo/R [MÖ6] for the LL(*) framework and, more recently, the Styx scanner and parser generator [DM07] for the LALR(1) framework. Recent developments in the field of parsing, however, have broken the mold, blurring the line between parser and scanner.

Parsing expression grammars, or *PEGs* [For04, WDM08], discussed in detail in

section 2.3.4 on page 53, are a new sort of grammar, unrelated to the Chomsky hierarchy or the familiar context-free and regular formalisms; they are specified in a single EBNF-like formalism from the character level on up, and parsed by scannerless *packrat parsers.*

Scannerless generalized-LR parsing, or *SGLR* [Vis97, vdBSVV02], discussed in detail in section 2.3.5 on page 55, uses a "generalized" variant on the LR algorithm that can parse any context-free grammar because it handles parse table conflicts by parsing *nondeterministically*. A nondeterministic parser, upon encountering a parse table conflict or ambiguity, will nondeterministically take all possible parse actions; if this results in an ambiguous parse, it will return a *parse forest* representing parse trees generated from all possible parses of the input.

In the SGLR framework, grammars are specified in a formalism largely resembling that of the traditional LALR(1) framework, but in compilation the entire specification is translated into a character-level context-free grammar, which is compiled into a scannerless SGLR parser.

2.2 Background.

In this section, we give a brief description of the algorithms used in LR parsers and the disjoint scanners traditionally used with them, as well as the process used to build them. Those already familiar with the traditional LR parsing framework may skip ahead to section 2.3, beginning on page 50.

For a complete and rigorous treatment of the process, see Aho *et al.*'s *Compilers: Principles, Techniques, and Tools* [ASU86] and Grune *et al.*'s *Modern Compiler Design* [GBJL02]. Although these processes have been developed over a period of

decades, Knuth's original paper [Knu65] provides the fundamentals on the LR algorithm. The algorithms in this section (scanner and LR DFA construction, first- and follow-set derivation) are adapted from, but not identical to, those presented in [Sip06] and [AP02].

We will discuss several constructs and concepts here (*e.g.*, context-free grammars, scanner DFAs) for which precise definitions are provided in later chapters. It is important to note that those definitions do not necessarily apply to this section, as they were made to accommodate the new scanning and parsing algorithms presented in this thesis and may differ from the familiar definitions.

2.2.1 Overview.

As with any parsing procedure, the problem is to take an input string and interpret it according to a series of rules specifying the language being parsed. If the input string is not in the language, the procedure should fail; otherwise, it should convert the input string into a parse tree according to the given rules.

In the traditional parsing framework, the syntax of a language is split into two parts: context-free and lexical. Likewise the program used to parse it is split into two parts, the parser and the scanner.

The parser is an implementation of the context-free syntax, which as the name implies is based around a context-free language. The context-free syntax consists of, at minimum, a context-free grammar, with terminals, nonterminals, productions, and one of the nonterminals designated as a start symbol.

The scanner is an implementation of the lexical syntax, which is based around regular languages. The lexical syntax always consists of a list of regular expressions, which correspond (roughly) to the terminals of the context-free grammar. The scanner uses the lexical syntax to break the input string into a series of tokens, which is then read in

by the parser and built into a parse tree.

2.2.2 Traditional scanners.

2.2.2.1 How a scanner DFA is constructed.

Scanners in this framework are based around deterministic finite automata (DFAs), as regular expressions are equivalent in power to DFAs; which is to say, that there is a DFA recognizing a language L iff there is a regular expression specifying it. Hence, for any one of the regular expressions of the lexical syntax, there will be a DFA recognizing it. Furthermore, there are simple procedures for converting regular expressions to DFAs.

The type of DFA generally used, as defined over an alphabet Σ, consists of a 4-tuple $\langle Q, s, \delta, F \rangle$, where Q is a finite set of DFA states, $s \in Q$ is the *start state*, $\delta : Q \times \Sigma \to Q$ is the *transition function*, and $F \subseteq Q$ is a set of *final* or *accept states*. Given a string w as input, a DFA will run according to the algorithm in Figure 2.2 on the following page; the DFA starts at the start state and makes transitions according to the transition function and the characters of w.

If the transition function is undefined at any point, $w \notin L$.[1] If all the transitions are defined, the DFA may or may not end up in a state that is a member of the set F of final states. If it does, then $w \in L$ and the algorithm returns `true`; if not, $w \notin L$ and the algorithm returns `false`.

See Figure 2.1 on the next page for an example of a DFA diagram. The DFA recognizes the language $\{a, aba, abaa, aaba, aabaa, aabaaba \ldots\}$ — sequences of at least one `a` punctuated by single `b`s.

The process of converting a regular expression to a DFA consists of first converting

[1] In many theoretical treatments of DFAs, the transition function is defined for all state-alphabet symbol pairs, and an implicit "trap state" is added as a destination for all transitions not appearing explicitly in the DFA's state diagram.

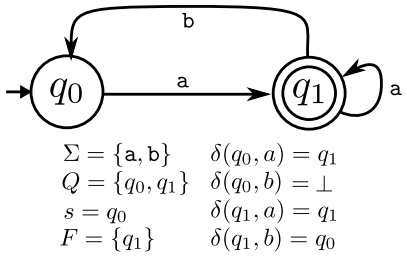

Figure 2.1: Example of a DFA diagram.

function *runDFA*(w) : $\Sigma^\star \to \{\mathtt{true}, \mathtt{false}\}$

1. $q_0 = s$
2. for $i = 1$ to $|w|$ do
   (a) if $\delta(q_{i-1}, w_i) \neq \perp$ then $q_i = \delta(q_{i-1}, w_i)$
   (b) else return `false`
3. return `true` iff $q_{|w|} \in F$

Figure 2.2: Algorithm for running a DFA.

the regular expression to a nondeterministic finite automaton (NFA), then converting that to a DFA.

An NFA is similar to a DFA in its composition. Like a DFA, it consists of a 4-tuple $\langle Q, s, \delta, F \rangle$. Like with a DFA, Q is a set of states, $s \in Q$ is the start state, and $F \subseteq Q$ is a set of final states. But the transition functions are different; an NFA is nondeterministic, meaning that it can have several transitions from the same state on the same alphabet symbol. It can also have *ε-transitions*; if there is an ε-transition between states q_1 and q_2, the NFA can transition from q_1 into q_2 without consuming any input. An NFA's nondeterminism means that there are many paths through the NFA following transitions marked with that input. If just one such path ends in a final state, the NFA accepts the input. This contrasts with a DFA where there is only one path for any input.

To accommodate this nondeterminism, while a DFA transition function will take a state and an alphabet symbol, and return a state, an NFA transition function will take a state and either an alphabet symbol or the empty string, and return a *set* of states, each representing a possible transition for the NFA. So an NFA's transition function is $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathscr{P}(Q)$.

In discussing how regular expressions are converted to NFAs, we will first consider the composition of a regular expression. Structurally, it is like a well-formed logical formula, consisting of symbols (which can be either alphabet symbols or the empty string) joined together by various connectives. In practical applications there are a great number of such connectives: bracketed sets such as `[a-z]`, repetition operators such as + and `?`, and shorthand operators such as `a{4}` to mean `aaaa`.

However, any given regular expression made with these connectives has an equivalent that uses only three connectives: concatenation, choice (|), and Kleene star ($\star$, zero or more repetitions). Hence, an NFA may be constructed from a regular expression using only five rules, for which representative state diagrams can be found on the following page: two basic NFAs matching an alphabet symbol and the empty string, and three connective rules, one for each of the concatenation, choice, and Kleene star connectives.

- The NFA for an alphabet symbol σ, recognizing the language $\{\sigma\}$ (Figure 2.3) contains two states — a start state and a final state — and one transition, marked σ. Thus it will end in a final state iff its input is that single symbol.

- The NFA for the empty string, recognizing the language $\{\varepsilon\}$ (Figure 2.4) contains only one (final) state and no transitions.

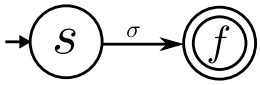

Figure 2.3: NFA representing regular expression σ.

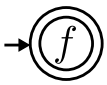

Figure 2.4: NFA representing regular expression ε.

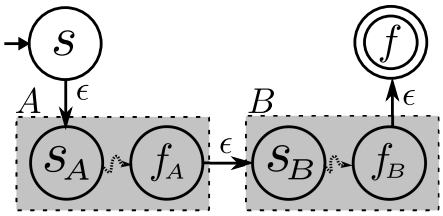

Figure 2.5: NFA representing regular expression AB, the concatenation of regular expressions A and B.

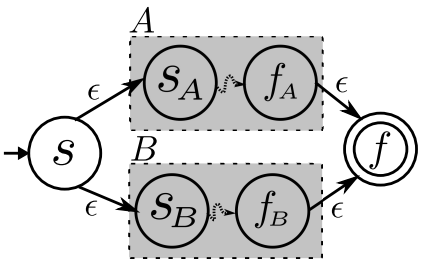

Figure 2.6: NFA representing regular expression $A|B$, the choice of regular expressions A and B.

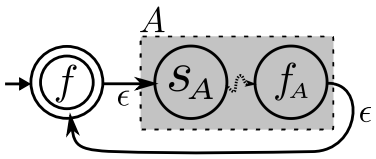

Figure 2.7: NFA representing regular expression $A^\star$, the Kleene star of regular expression A.

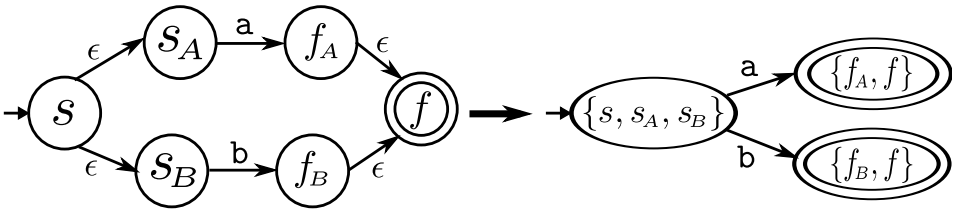

Figure 2.8: Converting the NFA for regular expression `a|b` into a DFA.

- The NFA recognizing the concatenation AB of regular expressions A and B (Figure 2.5) is constructed as follows. Suppose machine $M_A = \langle Q_A, s_A, \delta_A, \{f_A\}\rangle$ recognizes A, and $M_B = \langle Q_B, s_B, \delta_B, \{f_B\}\rangle$ recognizes B. The machine recognizing AB will put together all the states of the two DFAs, along with a new start state s and new final state f: $M_{AB} = \langle Q_A \cup Q_B \cup \{s, f\}, s, \delta_{AB}, \{f\}\rangle$. δ_{AB} combines the transitions of δ_A and δ_B, but also adds three ε-transitions: between the new start state s and s_A, between f_A and s_B, and between f_B and the new final state f.

- The NFA for choice (Figure 2.6) works much the same way, setting it up so a matching path through *either* M_A or M_B will lead to the new final state f.

  An example of an actual NFA constructed from a choice regular expression $(a|b)$ can be seen on the left-hand side of Figure 2.8.

- The Kleene star NFA (Figure 2.7) also works similarly, setting up a loop so that zero or more matching paths through M_A can be followed in succession with the NFA finishing in the new final state f.

The process of converting an NFA $M = \langle Q, s, \delta, F\rangle$, built on the alphabet Σ, into an equivalent DFA consists of, in effect, running M on all possible inputs and recording all the states reached for each input. The new DFA will consist of states labeled with members of $\mathscr{P}(Q)$: sets of M's states, indicating that the DFA state represents a nondeterministic run in which the NFA could possibly be in any state in that set.

Formally, we will write the the DFA equivalent to M as $M' = \langle Q', S, \delta', F' \rangle$. Q', S, and δ' — the DFA's state set, start state, and transition function — will be defined by the conversion process enumerated below. Here, we define the DFA's final state set F' to be $\{S \in Q' : F \cap S \neq \emptyset\}$ — any state that is labeled with a final state of the NFA, indicating that in a run of the NFA, one of the nondeterministic paths has led to a final state.

Before going into detail about the conversion process, we must also define the ε-closure of an NFA state. $\varepsilon^\star(q)$ consists of all the states that can be reached from q along paths consisting only of ε-transitions. For example, in the NFA in Figure 2.7, $\varepsilon^\star(f_A)$ contains at least f and s_A, since both those states are reachable along ε-transitions from f_A.

Formally, there are three steps to NFA-to-DFA conversion:

1. Start with a state marked with the set $S = \varepsilon^\star(s)$. This is the new DFA's start state. Enqueue it in a queue of states $\mathbb{Q}_{new}$.

2. Dequeue a state from $\mathbb{Q}_{new}$; call the set of states it is marked with Q_{new}. Add Q_{new} to the set of DFA states Q'.

   For each $\sigma \in \Sigma$, set $\delta'(Q_{new}, \sigma) = \{q \in Q : \exists q' \in Q_{new}. [q \in \varepsilon^\star(\delta(q', \sigma))]\}$; that is to say, if Q_{new} represents the set of states the NFA could possibly be in, $\delta'(Q_{new}, \sigma)$ represents the set of states it could then possibly be in after consuming the alphabet symbol σ and transitioning over any number of ε-transitions.

   If $\delta'(S, \sigma) \notin Q'$ and it is not in $\mathbb{Q}_{new}$, add it to $\mathbb{Q}_{new}$. Repeat this process on each newly added state until $\mathbb{Q}_{new}$ is empty, indicating that all states in the new DFA have been processed.

3. Remove the state marked $\emptyset$ and all transitions leading to it.[2]

[2] In the abovementioned formalisms for DFAs that require transitions for all state-symbol pairs and

See Figure 2.8 on page 29 for an example of NFA to DFA conversion. In the NFA in that figure, there are six states, s, s_A, s_B, f_A, f_B, and f. We will now run through the process of converting that NFA to the DFA also shown in the figure.

- In the first step, $\varepsilon^\star(s)$ is added to $\mathbb{Q}_{new}$. s_A and s_B are reachable by ε-transition from s, so $S = \varepsilon^\star(s) = \{s, s_A, s_B\}$.

- Progressing to the second step, we dequeue S again, add it to Q', and calculate $\delta'(S, \mathtt{a})$ and $\delta'(S, \mathtt{b})$.

  - The only state in S having a transition on `a` is s_A, and this transition points to f_A. f_A contains an ε-transition to f, so $\varepsilon^\star(f_A) = \{f_A, f\}$. Therefore, $\delta'(S, \mathtt{a}) = \{f_A, f\}$, which is added to $\mathbb{Q}_{new}$. As f, the NFA's final state, is included in this state, $\{f_A, f\}$ will become a final state of the DFA.

  - Similarly, the only state in S having a transition on `b` is s_B; $\delta'(S, \mathtt{b}) = \{f_B, f\}$, which is also added to $\mathbb{Q}_{new}$ and also becomes a final state of the DFA.

  - Now $\{f_A, f\}$ is dequeued from $\mathbb{Q}_{new}$ and added to Q', which then becomes $\{S, \{f_A, f\}\}$.

  - As neither f_A nor f have any non-ε transitions out of them, all the transitions out of $\{f_A, f\}$ will go to the state marked $\emptyset$, which is added to $\mathbb{Q}_{new}$.

  - Now $\{f_B, f\}$ is dequeued from $\mathbb{Q}_{new}$ and added to Q'. Its processing is similar to that of $\{f_A, f\}$: there are no non-ε transitions, so all transitions out of it go to $\emptyset$. However, $\emptyset$ is already in $\mathbb{Q}_{new}$, so it is not enqueued.

  - Now $\emptyset$ is dequeued and added to Q'. All transitions out of $\emptyset$ will go back to $\emptyset$, so no new states are added to $\mathbb{Q}_{new}$.

maintain an implicit "trap state," the state $\emptyset$ is not removed, but instead becomes the trap state.

  - $\mathbb{Q}_{new}$ is now empty; this ends the second step.

- In the third step, $\emptyset$ and the two transitions leading to it are removed, leaving the DFA shown in Figure 2.8.

2.2.2.2 How scanner DFAs are made into scanners.

Once a DFA has been built as described in the previous section, it must be fitted for use in a scanner.

As mentioned above, the lexical syntax of a language consists of, at minimum, a list of regular expressions, $R_1, R_2, \ldots, R_n$, coupled with some means of assigning terminals to strings that match the regular expressions. When building a scanner, one must build DFAs to recognize these regular expressions. However, the typical scanner generator does not build a separate DFA for each regular expression. It instead builds a single DFA for the regular expression $R_1|R_2|\cdots|R_n$, which will recognize the union of the languages of all the regular expressions in the list. This DFA will then be fed into a scanning algorithm that uses it in creating the stream of tokens that is the scanner's output.

However, as may be plainly seen, if this union DFA were then employed in the usual way, it would be useless: one could not determine *which* of the regular expressions had been matched. Hence, the DFA must be annotated so that each final state contains a record of the regular expression it matches.

This is a fairly straightforward task. Recall from Figure 2.6 that in building an NFA from a choice regular expression $R_1|R_2|\cdots R_n$, ε-transitions are added to the final state f from the final states $f_1, f_2, \ldots, f_n$ of the NFAs recognizing the constituent regular expressions. Recall also that when converting an NFA to a DFA, a state in the DFA is a final state iff one of the NFA states it represents was a final state (in this case, f).

But since f is only reachable by the ε-transitions from $f_1, f_2, \ldots, f_n$, each DFA state representing f will also represent one of $f_1, f_2, \ldots, f_n$. For example, in the DFA in Figure 2.8 on page 29, which was built from a choice regular expression `a|b`, each of the two states representing f also represents either f_A, the final state indicating a match of `a`, or f_B, indicating a match of `b`.

Hence, one need only take note of exactly *which* f_i each final state represents, and one can map a regular expression to each final state; R_1 if the state is f_1, *etc*.

If the languages of two of the regular expressions R_i and R_j overlap, then both f_i and f_j will be represented by one of the DFA's final states. This is called a *lexical ambiguity*, and it is customary to resolve it by preferring the regular expression that is highest on the original list (*i.e.*, if $i < j$, prefer R_i). This forms a lexical *precedence relation*; precedence relations are discussed further in section 4.2.

See Figure 2.9 on the following page for the algorithm used by the scanner to match and return a single token from the input. It starts at the beginning of the input string and the start state of the DFA. Then it will run the DFA until it reaches either the end of the input, or a state in which there is no transition defined for the given input. At that point it will take the longest prefix of the input for which the chain of transitions led to a final state (according to the principle of "maximal munch") and return a token built from that prefix. In Figure 2.9, this token is built from two parameters: $r(\mathit{lastMatchState})$, representing the regular expression or terminal matched in the last state where there was a match, and the prefix itself, $w_{1..\mathit{lastMatchPos}}$.

The scanner will generally call this function repeatedly, consuming the part of the input that was matched, until the end of the input is reached.

Given DFA $M = \langle Q, s, \delta, F \rangle$, function $r : F \to \{R_1, \ldots, R_n\}$ for matching regular expressions to final states, and function *buildToken* for building tokens from regular expressions and lexemes.

function $readToken(w) : \Sigma^\star \to Token$

1. $i = 1$
2. $q = s$
3. if $q \in F$ then
   (a) $lastMatchPos = 0$
   (b) $lastMatchState = q$
4. else
   (a) $lastMatchPos = -1$
   (b) $lastMatchState = \perp$
5. while $i \leq |w| \wedge \delta(q, x_i) \neq \perp$ do
   (a) $q = \delta(q, x_i)$
   (b) if $q \in F$ then
      i. $lastMatchPos = i$
      ii. $lastMatchState = q$
   (c) $i = i + 1$
6. if $lastMatchPos \neq -1$ then
   (a) return $buildToken(r(lastMatchState), w_{1..lastMatchPos})$
7. else `error`

Figure 2.9: Algorithm for scanning one token.

2.2.3 LR parsers.

In this section we discuss LR parsers. Firstly, we explain the general principles on which they are built (context-free grammars, and the formal machines to recognize context-free languages, and grammar derivations). Secondly, we discuss how to construct LR parsers; thirdly, we discuss the specifics of the LR parsing algorithm.

2.2.3.1 Context-free grammars, pushdown automata, derivations.

The class of regular languages are distinct in that they can be recognized by DFAs, which take $\Theta(1)$ space to run (they only have to maintain the current state of the DFA). However, regular languages are too restricted a class to describe most programming languages; most are context-free languages and are specified using context-free grammars.

In the theoretical sense, a context-free language can be recognized using a *pushdown automaton* (PDA), which is very much like a DFA: it has a set of states, one of which is a start state and some of which are final states, and a transition function. But while a DFA's transitions are based only on the input, a pushdown automaton also maintains a stack, and its transitions take into account not only the input, but the state number on the top of the stack.

In this section, we will discuss the construction and execution processes of LR parsers, and in particular for a specific kind of pushdown automaton, the LR DFA, from which an LR parser may be built. It is not strictly a DFA, but it is called that because it uses *only* the stack in maintaining its state (no separate "state") and its transitions are based only on the parser's input.

A context-free grammar $\Gamma = \langle T, NT, P, s \in NT \rangle$, as mentioned above, contains three finite sets — a set of *terminal* symbols T, a set of *nonterminal* symbols NT, a set of *productions* P — and one of the nonterminals s designated as a start symbol.

Productions are rewrite rules, consisting of one nonterminal on the left-hand side and zero or more terminals and nonterminals on the right (*e.g.*, $nt \rightarrow t_1 t_2 nt_2$). The nonterminals are thus those symbols that can be rewritten using the productions; the terminals are those symbols that cannot be rewritten.

In the abstract sense, the task of any parser is to produce, given a sequence of terminals, a *derivation* of that terminal sequence from the start nonterminal. One sequence of symbols *yields* another (written $X \Rightarrow Y$) if a nonterminal A in the sequence X has a production $A \rightarrow \alpha$ (where α is a sequence of grammar symbols, $\alpha \in (T \cup NT)^\star$) such that X can be transformed into Y by replacing one occurrence of A with α. For example, if there is a production $A \rightarrow b$, then for any sequences α, β of grammar symbols, $\alpha A \beta \Rightarrow \alpha b \beta$. Note that only *one* occurrence of A may be replaced (*e.g.*, $\alpha AA\beta \Rightarrow \alpha bA\beta$ and $\alpha AA\beta \Rightarrow \alpha Ab\beta$, but $\alpha AA\beta \not\Rightarrow \alpha bb\beta$).

Terminals:

- *y* with regular expression `y`
- *x* with regular expression `x+`
- $+$ with regular expression `\+`
- *ws* with regular expression `[ ]*`

Start nonterminal: **E**
Productions:

- **E** $\rightarrow$ **T**
  | **T** *y*
  | **T** $+$ **E**
- **T** $\rightarrow$ *x*

Figure 2.10: Example grammar from appendix A.4.

Such a statement constitutes a step in a derivation. For example, suppose that one

wished to derive the string $x + xy$ based on the small grammar in Figure 2.10. One derivation of this string would be $\mathbf{E} \overset{\mathbf{E}\to\mathbf{T}+\mathbf{E}}{\Longrightarrow} \mathbf{T} + \mathbf{E} \overset{\mathbf{T}\to x}{\Longrightarrow} x + \mathbf{E} \overset{\mathbf{E}\to\mathbf{T}y}{\Longrightarrow} x + \mathbf{T}y \overset{\mathbf{T}\to x}{\Longrightarrow} x + xy$.

For any given sequence of terminals, there are generally several different derivations of it, depending on which order the productions are applied in. Hence, a parsing framework will usually utilize a particular systematic sort of derivation. The example derivation above is a *leftmost* derivation — the leftmost nonterminal in the sequence is always processed first — which is used by LL ("Left-to-right, Leftmost derivation") frameworks.

LR ("Left-to-right, Rightmost derivation") frameworks use rightmost derivations instead (*e.g.*, $\mathbf{E} \overset{\mathbf{E}\to\mathbf{T}+\mathbf{E}}{\Longrightarrow} \mathbf{T} + \mathbf{E} \overset{\mathbf{E}\to\mathbf{T}y}{\Longrightarrow} \mathbf{T} + \mathbf{T}y \overset{\mathbf{T}\to x}{\Longrightarrow} \mathbf{T} + xy \overset{\mathbf{T}\to x}{\Longrightarrow} x + xy$). Furthermore, the LR parser works from the bottom up — starts with the final string of terminals and produces the derivation backwards via a series of *shift* and *reduce* actions: the shift actions consuming input by pushing terminals onto the parse stack, the reduce actions performing a reverse step in the derivation by popping the right-hand side of a production off the stack and pushing the left-hand side in its place. This is discussed further in section 2.2.3.3.

2.2.3.2 Construction of LR parsers.

An LR parser is constructed by the following process. Firstly, from the context-free grammar, three groups of "context sets" are built, to obtain further information about the terminals and nonterminals of the grammar. Secondly, from the grammar and these sets, an LR DFA is built. Thirdly, the LR DFA is converted into a more functional form, the parse table, which is then coupled to the generic LR algorithm to create the LR parser.

Building the LR DFA is the most complex part of this operation. The basic idea of an LR DFA is this: Each LR DFA state *n* contains a set of *LR items*, of the form

$A \rightarrow \alpha \bullet \beta$, where α and β are sequences of grammar symbols. These indicate that if the LR parser is in state n, it is possible that it is parsing a construct derived from the nonterminal A and that it is at the point represented by the bullet (*i.e.*, the sequence of α is on the top of the stack, and β derives some prefix of the input that has not been consumed). As the parser transitions further, the range of possibilities will grow smaller until, by the time a reduce action is needed, there is only one possibility.

In the context of the above example parsing $x + xy$, the parser would be in a state containing an item $\mathbf{E} \rightarrow \mathbf{T} \bullet y$ after it consumes the second x in the input string and reduces it to **T**, but before it consumes the y.

Before one can construct the LR DFA, one must add some constructs to the grammar to permit the parser to recognize and accommodate the end of the input: a new start symbol, $^\wedge$; a new terminal, \$, representing end-of-input; and a new production, $^\wedge \rightarrow s\$$. The shift action that consumes \$ is called an *accept* action, as it ends the parse (provided that the parse has not terminated prematurely with a syntax error).

Next, one must gather some information about the grammar in the form of a series of "context sets" — two for each nonterminal and one for the grammar. The sets are $\mathit{nullable} \subseteq NT$, $\mathit{first} : NT \rightarrow \mathscr{P}(T)$, and $\mathit{follow} : NT \rightarrow \mathscr{P}(T)$.

- A nonterminal is *nullable* if it can derive the empty string.
- A nonterminal's *first* set is the set of all terminals that can occur at the beginning of a terminal sequence derived from the nonterminal.

  For example, in the grammar in Figure 2.10, **T** derives x, thus $x \in \mathit{first}(\mathbf{T})$. **E** derives xy, thus $x \in \mathit{first}(\mathbf{E})$.

  For convenience, we will also refer to the first sets of terminals. These are simply the terminals themselves, *i.e.*, $\forall t \in T.\, [\mathit{first}(t) = \{t\}]$.

- A nonterminal's *follow* set is the set of all terminals that can validly occur immediately after the nonterminal.

  For example, **E** derives both **T**y and **T** + **E**. Therefore, y and $+$ are both in *follow*(**T**).

1. Set all first and follow sets, and *nullable*, to $\emptyset$
2. for $t \in T$ do $first(t) = t$
3. do
   (a) for $(A \rightarrow A_1 \cdots A_n) \in P$ do
      i. if $\{A_1, \ldots, A_n\} \subseteq nullable$ then $nullable = nullable \cup \{A\}$
      ii. for $i = 1$ to n , $j = i+1$ to n do
         A. if $\{A_1, \ldots, A_{i-1}\} \subseteq nullable$ then $first(A) = first(A) \cup first(A_i)$
         B. if $\{A_{i+1}, \ldots, A_n\} \subseteq nullable$ then $follow(A_i) = follow(A_i) \cup follow(A)$
         C. if $\{A_{i+1}, \ldots, A_{j-1}\} \subseteq nullable$ then $follow(A_i) = follow(A_i) \cup first(A_j)$

   while *first*, *follow*, or *nullable* changed in the current iteration

Figure 2.11: Procedure for deriving *first*, *follow*, and *nullable*.

The exact procedures for deriving these sets are shown in Figure 2.11. These sets are not needed when building some sorts of LR parsers, which are discussed below; they are, however, needed for building LALR(1) parsers.

As we have mentioned, each LR DFA state consists of a set of LR items; to build the LR DFA is to build these sets. The process of building an LR DFA roughly parallels the process detailed in the previous section for converting a lexical NFA to a DFA: start with one state, build transitions to new states, and repeat until there are no new states. It is the process of building transitions to new states that is different.

We now define two operations on sets of LR DFA states: *Closure* and *Goto*. *Goto* builds a transition by determining an initial set of items to be placed in the transition's target state — items from the transition's source state with the bullet moved past the symbol with which the transition is labeled (*e.g.*, if $A \rightarrow \bullet xy$ was in the original state, $A \rightarrow x \bullet y$ will be in the *Goto* set constructed for a transition x). *Closure* expands that initial set to the full set to create the state in its final form. The processes for building *Closure* and *Goto* are defined in Figure 2.12 and Figure 2.13 respectively.

function $Closure(S) \ : \ \mathscr{P}(Items) \rightarrow \mathscr{P}(Items)$

1. do
   (a) for $A \rightarrow \alpha \bullet B\beta$ in S (where $\alpha, \beta \in (T \cup NT)^\star$) do
      i. $S = S \cup \{B \rightarrow \bullet\gamma \ : \ (B \rightarrow \gamma) \in P\}$ (where $\gamma \in (T \cup NT)^\star$)

   while S changed in the current iteration
2. return S

Figure 2.12: Closure routine. (*Items* is the set of all LR items; see Definition 4.1.1 on page 91).

function $Goto(S,a) \ : \ \mathscr{P}(Items) \times (T \cup NT) \rightarrow \mathscr{P}(Items)$

1. return $\{A \rightarrow \alpha a \bullet \beta \ : \ (A \rightarrow \alpha \bullet a\beta) \in S\}$

Figure 2.13: Goto routine. (*Items* is the set of all LR items.)

The LR DFA construction process, then, is as follows:

1. Start with one state, having the item set $Closure(\{^\wedge \rightarrow \bullet s\$\})$. This is the LR DFA's start state. Enqueue it to be processed.

2. Dequeue an unprocessed state n. It will have a set of LR items, call it S. For each symbol $g \in (T \cup NT)$ such that $Goto(S,g) \neq \emptyset$:

(a) If a state n' with an item set $Closure(Goto(S,g))$ is not already in the LR DFA, add such a state to the LR DFA and enqueue it to be processed.

(b) Let $\delta(n,g) = n'$.

3. Repeat step 2 until the process queue is empty.

We will now run through an example of LR DFA construction. Take a grammar with one terminal x, one nonterminal $\mathbf{E}$ (by default, the start nonterminal); and two productions, $\mathbf{E} \to \mathbf{E}x$ and $\mathbf{E} \to \varepsilon$.

Firstly, we will add the production to recognize end of input, which is in this case $\wedge \to \mathbf{E}\$$; then we will construct the context sets. With such a small grammar it is simpler to work from first principles rather than via the algorithm in Figure 2.11.

- $\mathbf{E} \overset{\mathbf{E}\to\varepsilon}{\Longrightarrow} \varepsilon$. Thus, $\mathbf{E}$ is nullable.
- $\wedge$ is never nullable ($ is treated as a part of the input sequence).
- $first(x) = \{x\}$ and $first(\$) = \{\$\}$.
- $\wedge \overset{\wedge\to\mathbf{E}\$}{\Longrightarrow} \mathbf{E}\$ \overset{\mathbf{E}\to\mathbf{E}x}{\Longrightarrow} \mathbf{E}x\$ \overset{\mathbf{E}\to\varepsilon}{\Longrightarrow} x\$$. Thus, $x \in first(\mathbf{E})$ and $x \in first(\wedge)$.
- $\wedge \overset{\wedge\to\mathbf{E}\$}{\Longrightarrow} \mathbf{E}\$ \overset{\mathbf{E}\to\varepsilon}{\Longrightarrow} \$$. Thus, $\$ \in first(\wedge)$.
- $\wedge \overset{\wedge\to\mathbf{E}\$}{\Longrightarrow} \mathbf{E}\$ \overset{\mathbf{E}\to\mathbf{E}x}{\Longrightarrow} \mathbf{E}x\$$. Thus, $x \in follow(\mathbf{E})$.
- $\wedge \overset{\wedge\to\mathbf{E}\$}{\Longrightarrow} \mathbf{E}\$$. Thus, $\$ \in follow(\mathbf{E})$.
- The follow set of $\wedge$ never comes into consideration.
- Finally $nullable = \{\mathbf{E}\}$, $first(\mathbf{E}) = \{x\}$, $first(\wedge) = \{x,\$\}$, and $follow(\mathbf{E}) = \{x,\$\}$.

The LR DFA wil be constructed as follows:

- Start with state 1, containing item $^\wedge \rightarrow \bullet\mathbf{E}\$$.
- One iteration of the *Closure* routine adds two more items to this state, $\mathbf{E} \rightarrow \bullet\mathbf{E}x$ and $\mathbf{E} \rightarrow \bullet$.
- The next iteration adds one more item, $\mathbf{E} \rightarrow \bullet\mathbf{E}x$, but as this is already in the set, it does not change. The *Closure* routine is finished.
- The only symbol with a non-empty *Goto* set for state 1 is **E**. There are two items in state 1 with **E** immediately following the bullet point, $^\wedge \rightarrow \bullet\mathbf{E}\$$ and $\mathbf{E} \rightarrow \bullet\mathbf{E}x$; the *Goto* set will thus contain items $^\wedge \rightarrow \mathbf{E} \bullet \$$ and $\mathbf{E} \rightarrow \mathbf{E} \bullet x$.
- Running the *Closure* routine on this set will not alter it as there are no productions therein with a bullet immediately preceding a nonterminal.
- This state is not in the LR DFA and hence is entered as state 2, with a transition marked **E** between states 1 and 2.
- The only symbol with a non-empty *Goto* set for state 2 is x (there are no transitions placed in the DFA on the symbol \$). In state 2 there is one item with x immediately following the bullet point, $\mathbf{E} \rightarrow \mathbf{E} \bullet x$. Hence, the *Goto* set will contain only the item $\mathbf{E} \rightarrow \mathbf{E}x\bullet$, which will not be changed by the *Closure* routine.
- This state is not in the LR DFA and hence is entered as state 3, with a transition marked x between states 2 and 3.
- There are no symbols with a non-empty *Goto* set for state 3. The LR DFA construction is completed.

See Figure 2.14 on the following page for the finished LR DFA.

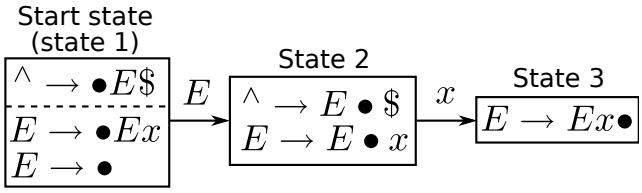

Figure 2.14: State diagram for example LR DFA.

An LR DFA is not immediately useful to a parser, however; following its construction it will be distilled into a more useful form, the parse table. This can be done in several ways; we will first discuss the simplest, the building of an LR(0) parse table, which is fairly straightforward:

- The parse table contains one row for each LR DFA state and one column for each grammar symbol (terminal or nonterminal).
- For each transition from state n to state n' on terminal t, an action marked $\mathit{shift}(n')$ will be added to the cell at row n, column t of the parse table.
- For each transition from state n to state n' on nonterminal A, an action marked $\mathit{goto}(n')$ will be added to the cell at row n, column A of the parse table.
- If there is an item in state n of the form $A \to \alpha\bullet$ (with $\alpha \in (T \cup NT)^\star$), several actions marked $\mathit{reduce}(A \to \alpha)$ are placed in row n of the parse table, one in each column marked with a terminal.
- If there is an item $\wedge \to s \bullet \$$ (s being the grammar's start nonterminal) in state n, an action marked *accept*, signifying the end of the parse, is added to the cell at row n, column \$ of the parse table.
- If no actions are placed in some cell, that cell is said to contain an "error action."

- If more than one action is placed in any cell, this is a parse-table conflict and the parser construction has failed. In practice, some parse-table conflicts can be resolved to a limited degree, but this is not an integral part of LR parsing; it is discussed further in section 4.6.

  See Table 2.1 for the parse table derived from the example LR DFA in Figure 2.14.

	$	x	E
1	$reduce(\mathbf{E} \rightarrow \varepsilon)$	$reduce(\mathbf{E} \rightarrow \varepsilon)$	$goto(2)$
2	$accept$	$shift(3)$	
3	$reduce(\mathbf{E} \rightarrow \mathbf{E}x)$	$reduce(\mathbf{E} \rightarrow \mathbf{E}x)$	

Table 2.1: Parse table for example LR DFA in Figure 2.14.

The class of grammars that can be parsed with an LR(0) parser is fairly restricted, as the LR(0) parser cannot look at any input before consuming it: it must decide whether to shift or reduce based on the stack only. This is why the reduce actions are placed on every column in a row — the parser has to know that it can reduce no matter what the next token in the input is. Hence, any shift actions in a state that can be reduced from will cause a parse table conflict.

On the other hand, if the parser is able to "look ahead" at the next token of input, reduce actions can be placed more selectively. There are several algorithms for parse table construction that can achieve this.

The simplest is SLR (Simple LR), which instead of placing reduce actions in *all* columns when encountering an item $A \rightarrow \alpha\bullet$, will only place them in the columns of terminals in $follow(A)$. However, even this might not be enough, and since different uses of nonterminals can be in vastly different contexts within a grammar, there is more opportunity for trimming reduce actions. To take a simple example, in C, expressions

can occur within function calls — followed by a comma or right parenthesis — or in statements — followed by a semicolon.

Unnecessary reduce actions are trimmed by assigning to each item a set of *lookahead* — essentially a follow set for the item, enumerating which terminals may be at the head of the input when that item is ready to be reduced.

Lookahead is used in two LR-based algorithms. In the first, LR(1), lookahead is made an integral part of items, and the *Closure* and *Goto* routines are modified:

- In the *Goto* routine, the item $A \rightarrow \alpha a \bullet \beta$ will have the same lookahead as the item $A \rightarrow \alpha \bullet a\beta$.
- In the *Closure* routine, recall that if a state contains the item $A \rightarrow \alpha \bullet B\beta$, and there is a production $B \rightarrow \gamma$, an item $B \rightarrow \bullet\gamma$ will also be placed in the state. In the LR(1) algorithm, for every symbol ϕ in the lookahead of the item $A \rightarrow \alpha \bullet B\beta$, then the item $B \rightarrow \bullet\gamma$ will have $\mathit{first}(\beta\phi)$ in its lookahead.

However, incorporating lookahead in this way significantly increases the number of states in the LR DFA. Thus, we come to the most commonly used of the LR variants, LALR(1) (LookAhead LR), which combines the manageably-sized DFA of LR(0) with the benefits of lookahead. An LALR(1) DFA for a grammar is identical to the grammar's LR(0)/SLR DFA, but has lookahead added as a post-process. An LALR(1) DFA can also be made by building an LR(1) DFA and then merging all states that only differ in their lookahead sets, but this is obviously much slower than the post-process approach.

Figure 2.15 on the following page shows the example DFA from above annotated with lookahead. Here, we will run through the process of adding it, which parallels the process of building the LR DFA.

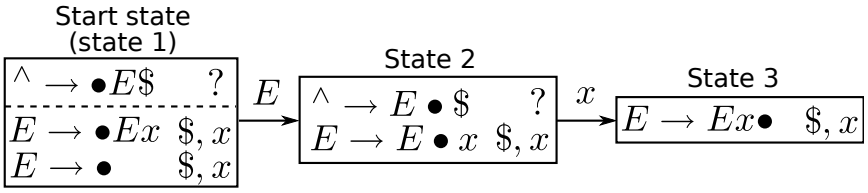

Figure 2.15: State diagram for example LR DFA, with lookahead.

- Start in state 1. The lookahead of the initial item $\wedge \to \bullet\mathbf{E}\$$ is undefined, as representing input "past the end," so we write ? to represent it.

- The position of **E** in the initial item causes $\mathit{first}(\$) = \{\$\}$ to be added to the lookahead of the two items in state 1 with **E** on the left-hand side, $\mathbf{E} \to \bullet$ and $\mathbf{E} \to \bullet\mathbf{E}x$.

- The position of **E** in $\mathbf{E} \to \bullet\mathbf{E}x$ causes $\mathit{first}(x) = \{x\}$ to be added to the lookahead of the same two items. The lookahead sets of these two items are now $\{\$, x\}$, as shown.

- This means that $\{\$, x\}$ is added to the lookahead of the item $\mathbf{E} \to \mathbf{E} \bullet x$ in state 2, which changes the lookahead sets in that state, so state 2 must now be processed.

- No further modifications are needed to the lookahead in state 2, but $\{\$, x\}$ is added to the lookahead of the item $\mathbf{E} \to \mathbf{E}x\bullet$ in state 3. Thus state 3 must be processed.

- No further modifications are needed to the lookahead in state 3, and there are no transitions out of that state, so the lookahead additions are complete.

The LR(0) parse table for this example grammar, as laid out in Table 2.1, is identical to the grammar's SLR, LALR(1), and LR(1) parse tables. In the case of SLR, this is because $\mathit{follow}(\mathbf{E}) = \{\$, x\}$, so reduce actions will be placed in those two columns

for the reductions in states 1 and 3. In the case of LALR(1) and LR(1), similarly, the lookahead sets for the items $\mathbf{E} \rightarrow \bullet$ and $\mathbf{E} \rightarrow \mathbf{E}x\bullet$ in states 1 and 3 respectively are $\{\$, x\}$.

When construction of the parse table is completed and it is found to contain no conflicts, the construction of the LR parser is complete.

2.2.3.3 Execution of LR parsers.

An LR parser in the traditional framework does not attempt to analyze, or make any assumptions about, the character of the terminals given to it. For most languages, these terminals are given to the parser in the form of tokens that have already been processed by the scanner.

Running an LR parser is fairly simple compared to constructing it. It consists of running the actions in the parse table over the *parse stack* — the construct used by the LR parser to maintain the parser's precise status. The parse stack elements we use here are ordered pairs $\langle n, CST \rangle$, where n is a parse state (corresponding to a parse table row) and *CST* is a fragment of a parse tree. n represents the state of the parser, while *CST* represents the portion of the parse tree that was built during the transition into that state; whether it was a terminal brought in by a shift action or a full subtree built by a reduce action.

Although different parse tables are produced for SLR, LALR(1), and LR(1) parsers, all three approaches use the same algorithm of parsing, which is this:

1. Start at the beginning of the given input with an empty parse stack. A parse stack element must contain, at minimum, the identifier of a state.

2. Push the parser's start state onto the parse stack.

3. Peek at the parse stack to ascertain the current state; call it n.

4. Peek at the next token in the input to obtain its terminal name; call it t.

5. Look up the action in the cell at row n, column t of the parse table, and act as follows:

   (a) If there is no action in the cell, exit with an error.

   (b) If the action is *accept*, the parse is completed; exit successfully.

   (c) If the action is $shift(n')$, consume the token t and push $\langle n', t \rangle$ onto the stack. This new stack element represents the terminal just consumed.

   (d) If the action is $reduce(A \rightarrow \alpha_1 \cdots \alpha_k)$, where each α_i is a single grammar symbol, do not consume any input; instead:

      i. Pop k elements off the stack. These elements represent the k right-hand side symbols.

      ii. Peek at the stack to ascertain the new state; call it n'.

      iii. Look up the action in row n', column A of the parse table. It will be an action $goto(n'')$ for some parse state n''.

      iv. Push $\langle n'', A \rangle$ onto the stack. This element represents a parse subtree from the production that was just reduced upon, headed by the nonterminal A.

6. Repeat steps 3 through 5 until an error or *accept* action terminates the loop.

The LR(0) algorithm is similar: step 4 is eliminated, while the decision to consume input or not must be made before the token of input is seen.

It is fairly straightforward to show how this algorithm performs the reverse-order, rightmost derivation alluded to above. The pushes performed by shift actions effectively move a marker of progress along the input, while the pops and pushes performed by

Stack	Remainder of input	Action(s) taken
1	`xxx`	*reduce*($\mathbf{E} \to \varepsilon$); *goto*(2)
1, $\langle 2, \mathbf{E} \rangle / \varepsilon$	`xxx`	*shift*(3)
1, $\langle 2, \mathbf{E} \rangle / \varepsilon$, $\langle 3, x \rangle$ /`x`	`xx`	*reduce*($\mathbf{E} \to \mathbf{E}x$); *goto*(2)
1, $\langle 2, \mathbf{E} \rangle$ /`x`	`xx`	*shift*(3)
1, $\langle 2, \mathbf{E} \rangle$ /`x`, $\langle 3, x \rangle$ /`x`	`x`	*reduce*($\mathbf{E} \to \mathbf{E}x$); *goto*(2)
1, $\langle 2, \mathbf{E} \rangle$ /`xx`	`x`	*shift*(3)
1, $\langle 2, \mathbf{E} \rangle$ /`xx`, $\langle 3, x \rangle$ /`x`	$ (none)	*reduce*($\mathbf{E} \to \mathbf{E}x$); *goto*(2)
1, $\langle 2, \mathbf{E} \rangle$ /`xxx`	$ (none)	*accept*

Table 2.2: Example parse of `xxx` using Table 2.1.

reduce actions represent the reverse derivation step, the replacement of a production's right hand side (the popped stack elements) with its left hand side (the one pushed element).

To show this more clearly, we enumerate an example parse of the string *xxx* using Table 2.1. The parse stacks shown are in the form $\langle n, S \rangle / \ell$, where n is the state number, S is the symbol, and ℓ is the string represented by that symbol. For example, in several steps, element $\langle 3, x \rangle$ /`x` is pushed on the stack; meaning that the parser has transitioned into state 3, consuming a token x with lexeme `x`. In the penultimate step, two elements, $\langle 2, \mathbf{E} \rangle$ /`xx` and $\langle 3, x \rangle$ /`x`, are popped off the stack and replaced by one, $\langle 2, \mathbf{E} \rangle$ /`xxx`, representing the reverse of the derivation $\mathbf{E} \Rightarrow \mathbf{E}x$.

2.3 Related work.

2.3.1 Nawrocki: Left context.

An independent development of context-aware scanning, made particularly to accommodate exceptions to the principle of maximal munch, is Jerzy Nawrocki's concept of "left context" [Naw91]. We were unaware of this work until quite recently (April 2010).

In his paper, Nawrocki presents two methods of lexical disambiguation, left context and "extended left context." Lexical disambiguation by left context is what we call context-aware scanning. The left context of a terminal $LC(Y)$ is defined as the set of parse states in which Y has a non-error action; hence, the set of terminals X for which a state n is in $LC(X)$ is what we call n's valid lookahead set. Extended left context is more complex, taking into account not only the current parse state, but also the set of strings that can validly precede an ambiguously-matched terminal in disambiguation.

However, Nawrocki's left context was developed with specific reference to the accommodation of exceptions to maximal munch, and was not fleshed out with the necessary practical modifications discussed in chapter 4. Nawrocki mentions implementation only by citing a source [Gro89] indicating that left context can be implemented within the scanner generator *Rex*, the primary innovation of which was the provision of scanner modes (there called "start states"), such as those used to parse AspectJ in `abc`. While scanner modes can be used in this capacity, Nawrocki does not provide a method to implement left context in an automatic manner; on the other hand, we discuss two such methods in chapter 5. The first method (section 5.2) builds a single scanner DFA for the entire grammar and annotates it with metadata so it can scan on any valid lookahead set. The second (section 5.3) automates Nawrocki's implied scanner-modes approach by building several non-annotated scanner DFAs, one for each valid lookahead set that

needs to be scanned on.

However, with regard to the scanner-modes approach, we have found that when automatically generating context-aware scanners using these modes, the scanners become impractically large for the bigger grammars if optimizations are not employed (see sections 5.3.3 and 10.2.1 for further discussion of this point).

2.3.2 The "Tatoo" parser generator.

Cervelle *et al.* [CFR06] have implemented the *Tatoo* parser generator, which is aimed primarily at resolving some practical problems with established parser generators; it offers optimized support for several different scanner alphabets and a pluggable interface for error recovery mechanisms, as well as a mechanism for several versions of a language to be specified in one specification in a simple manner.

In aid of implementing these innovations, Tatoo utilizes a scanning apparatus that, at each scan, checks each regular expression in the language to see if it matches. As an optimization to this, Tatoo provides a feature called a "lookahead activator," which at each scan limits the set of checked regular expressions to those that are valid syntax at that point.

This is, essentially, a context-aware scanner. However, like Nawrocki's left context, the Tatoo lookahead activator is presented in the light of a very specific problem — in this case, optimizing the parser — without so much emphasis on the gain in expressivity; the many practical modifications are not present. It is nevertheless worthy of mention as another independent development of the context-aware scanning idea — the developers of Tatoo were apparently also unaware of Nawrocki's work as they did not cite his paper.

2.3.3 LL(*) parser generators, *e.g.*, ANTLR.

The classic LL parsing algorithm, which has always been popular as the heart of simple LL(1) or "recursive descent" parsers (the designers of which often criticize LR for its counterintuitive nature), was given new life with the advent of ANTLR [PQ95] (the name officially stands for "ANother Tool for Language Recognition," but is rumored to be also a contraction of "anti-LR").

This popular parser generator is based on the LL(*) approach: if a grammar is LL(k), for any k whatever, ANTLR is able to build an LL(k) parser for it. Like an LR(k) parser, an LL(k) parser *might* be of exponential size; however, the optimizations of ANTLR have shown that LL(k) parsers are, like scanner DFAs and LALR(1) DFAs, usually of reasonable size in practice.

As with the traditional LALR(1) framework, grammars can be specified for the LL(*) framework and not be able to be compiled into an LL(k) parser; there are also conflicts possible in the LL(*) framework, and some of these do not yield to increasing the amount of lookahead [PQ95].

One significant problem with LL(k) parsers is that they cannot parse left-recursive grammars, which are often the most convenient way of expressing a language. The language of four-function arithmetic is a textbook example of this: each of the arithmetic functions is parsed in the left-associative manner, meaning that the most natural way of specifying them is a left-recursive grammar (*e.g.*, $E \rightarrow E + T$, $T \rightarrow T \times F$).

Although any left-recursive grammar can be converted into one that is not left-recursive, this grammar may not be LL(k); LR-based parsers can handle these in a much more "natural" and reliable way.

2.3.4 PEGs and packrat parsers.

Parsing Expression Grammars (PEGs, for short) [For04, Gri06] were developed as an alternative to the hybrid approach of BNF/EBNF and regular expressions that are used to specify grammars for most frameworks.

Like a BNF grammar, a PEG is a series of rules with nonterminals on the left-hand side. But PEGs are defined on the character level rather than the token level, eliminating the hybrid nature of BNF-based grammars. Also, while the BNF grammars allow only concatenation and unordered choice to assemble the terminals and nonterminals on the right-hand side, PEGs have a much richer collection of connectives, including most of the regular-expression syntax.

Notably absent is BNF's unordered choice operator $|$, which has been replaced with an ordered choice operator, $/$. This is done for the following reason: If a BNF grammar contains the rule $A \rightarrow B \mid C$, and there is a certain string w such that there are derivations of w from both B and C (written $B \Rightarrow^{\star} w$ and $C \Rightarrow^{\star} w$), the parser will not know what to do if w is input, because the grammar is ambiguous. But if the BNF grammar is converted to a PEG and contains the rule $A \rightarrow B/C$, then the parser will know to parse B in that case.

PEGs are generally parsed using scannerless *packrat parsers*, which are slightly larger than an analogous LR parser but run in $O(n)$ time like an LALR(1) parser. The packrat parsers originally did not support left-recursive grammars, but with more recent developments [WDM08] left-recursion is now supported.

The replacement of unordered choice with ordered choice makes every PEG completely deterministic: one cannot specify a PEG that no packrat parser can parse. This means that the class of PEGs is closed under composition, a fact that has been leveraged to the advantage of extensible language researchers; PEGs have been designed for extensible versions of C and Java [Gri06].

Any LR(k) language also has a PEG (although possibly a very complicated or counterintuitive one). PEGs can express some non-context-free languages, and, possibly, all context-free languages, although there is no formal proof of the latter as yet [For04].

The primary problem with PEGs has the same origin as their primary benefit: the framework's universal determinism. As Ford [For04] notes, switching the order of constituents in an ordered choice may change the language, and it is an undecidable problem to determine whether it does. By replacing unordered choice with ordered choice, PEGs force the grammar writer to make many choices of disambiguation before the grammar is even compiled, and determining whether these are the right choices is an undecidable problem.

In the LALR(1) framework, on the other hand, if a deterministic parser can be compiled from a grammar, it is a guarantee that the order of the choice does not matter. The PEG framework takes this objective guarantee and pushes it into the subjective area of whether or not the grammar is "correct," a question that can only be answered in the affirmative after considerable debugging and testing.

Now this might not seem like a problem *prima facie*; the subjective tests for correctness will need to be performed no matter what, and adding to it this small bit is not going to complicate matters more than they are simplified by PEGs' universal determinism. But in an extensible-languages situation, where a language is written in several modular parts, each by a different writer, which are ultimately put together by a non-expert programmer, this means that the non-expert will need to make some of the choice orderings, and if the wrong choices are made, the introduction of one part of the language might affect another part in ways that none of the separate writers could have foreseen. Therefore, some of the subjective testing must be done by the non-expert programmer in this case, which is undesirable.

By contrast, in deterministic BNF-based frameworks such as LALR(1) and LL(k),

the non-expert programmer would not have to make any choices at all: the language could simply be composed and run through the objective verification, and if this verification succeeds, each part of the language is guaranteed to behave exactly as its individual writer intended.

Our approach improves on this another step, by allowing the objective verification to take place *before* the language is composed (see chapter 6).

2.3.5 SGLR.

The generalized-LR (GLR) algorithm [Tom86] was developed in 1986, as an algorithm for natural languages, by Masaru Tomita. The idea of it is, essentially, fairly simple: it can parse on an LR parse table that contains conflicts, by nondeterministically taking all conflicting actions; instead of a parse tree it builds a *parse forest* representing several parse trees.

The original algorithm had some flaws and inefficiencies; it was corrected and improved by Farshi [Far91] and Rekers [Rek92] to produce a version that was standard for over a decade. This version ran in $O(n^k)$ time, k being the maximum number of symbols on the right-hand side of a production in the grammar, with a few corner cases requiring exponential time. (It should be noted, however, that all GLR parsers with no conflicts in their parse tables run in linear time.)

The first version changed only the parsing algorithm, using the traditional LALR(1) scanning arrangement unaltered; this version has been incorporated into several parser generators meant for general use, including GNU Bison and the optimized Elkhound framework [MN04].

In 1997, Visser [Vis97] presented a modification that, while retaining the traditional LALR(1) formalism of BNF combined with regular expressions, eliminates the scanner and converts each regular expression to context-free grammar rules, creating a single

character-level context-free grammar. This grammar can then be compiled into a GLR parser. This is called Scannerless GLR (SGLR).

The GLR algorithm later underwent further improvements. Right Nulled GLR (RNGLR) [JSE04] is an $O(n^k)$ algorithm that eliminates the exponential-time corner cases of the Farshi algorithm. The more experimental Binary RNGLR (BRNGLR) [SJE07] is a further improvement that brings the runtime down to $O(n^3)$.

Visser's SGLR framework, which underpins such tools as the Stratego program transformation language [Vis01], continues to use the Farshi/Rekers algorithm, however. This is due to the fact that a number of "disambiguation filters" [vdBSVV02] must be used within an SGLR parser to enable the sort of scanning behavior exhibited by a traditional scanner. *Follow restrictions*, for instance, indicating that certain characters cannot follow an occurrence of a certain regular expression, are used in the place of the "maximal munch" idea found in traditional scanners; for example, when specifying an identifier with the regular expression `[A-Za-z]+`, a follow restriction would be added indicating that no letter could directly follow such an identifier.

These filters were made specifically for the Farshi/Rekers version of GLR and could not trivially be applied to either of the improved versions. It is only very recently that the filters have been found applicable to RNGLR [EKV09]. They have not yet been found applicable to BRNGLR.

Since a GLR framework can create parsers even for ambiguous grammars, it is much more expressive than a deterministic framework. It is also more flexible, since the class of context-free grammars is closed under composition, unlike the class of LALR(1) grammars. Also, SGLR is more flexible than traditional GLR because its "scanning" is context-aware: each "terminal" or lexical symbol in the specification is represented by a grammar nonterminal deriving a generated series of productions, which means that the parsing of each terminal construct is concluded by taking a reduce

action with this lexical symbol on the left hand side. Such a reduce action will simply not be present in contexts where the lexical symbol is invalid.

At first glance, it would appear that the high time-complexity of SGLR is a problem, but in practice this is not the case. Experiments with the Elkhound parser generator show that GLR parsers for reasonable programming languages run deterministically (hence in linear time) 70% of the time [MN04]. Wagner and Graham have found [WG97] that when ambiguous parses are made on programming languages, only 5% of parse forest nodes are ambiguous, and ambiguities tend to be clustered near the bottom of the parse forest. It should be noted, however, that these observations were made on the traditional GLR framework, and the scannerless version parses much more nondeterministically.

A more significant problem with GLR is its nondeterminism. A GLR parser offers no guarantee of non-ambiguity, and to remove ambiguities a GLR grammar must be taken through a subjective "debugging" process [Vin07] that is never guaranteed to have eliminated all of them. As with PEGs, this subjective process can be lumped in with the correctness testing; as with PEGs, this can cause problems.

Also, many of SGLR's advantages, especially in the area of extensible languages, are also exhibited by our approach: insofar as the term applies to this system, the "scanning" of SGLR is implicitly context-aware (as mentioned above), since only those "terminals" that are valid in a specific context are ever "scanned" for. This is a strength of the system, allowing, *e.g.*, AspectJ to be parsed easily with it, by allowing implicit scanner modes in the place of the explicit ones such as `abc`'s.

2.3.5.1 Why not deterministic *and* scannerless?

The approach of converting BNF/regular-expression grammars to character-level grammars from which scannerless parsers are generated generalizes to all frameworks using

BNF. So a fair question to ask is, why not combine the scannerless approach with a deterministic parsing algorithm to get the benefits of both?

The problem with this is that the regular sub-grammars to which the regular expressions are converted are not at all LALR(1) or LL(k), and may require much lookahead in order to parse deterministically [Vis97]. The DFAs in a traditional scanner are not necessarily deterministic in the strict sense: they have been built from NFAs that have been built from the regular expressions, and are analogous to GLR parsers for regular languages.

Scannerless deterministic approaches, such as noncanonical SLR(1) [SC89, Sal90] have been put forth, but not widely adopted, due to their reduced expressivity.

2.3.6 Schrödinger's Token.

Another approach, striking a middle ground between the complete nondeterminism of the GLR frameworks and the complete determinism of the traditional LALR(1) framework, is Aycock and Horspool's use of the Schrödinger's Token [AH04].

A Schrödinger's Token is a lexical token consisting of *several* terminal/lexeme pairs instead of one, named in analogy to the scenario of "Schrödinger's Cat," wherein a cat is locked in a box with a single atom of radioactive material and a vat of deadly acid that is rigged to spill if the radioactive atom decays, and the fate of the cat is "undefined" until the box is opened. If several different tokens can be recognized at a certain point (*e.g.*, a keyword and an identifier) a Schrödinger's Token containing all the choices is made and sent back to the parser. If there is an ambiguity between one token and two (for example, if the string `forth` could be read as one identifier or as the keyword `for` followed by the identifier `th`), then a *null* token with an empty lexeme is used as padding (in the `forth` example, the first Schrödinger's Token would contain the tokens (FOR,‘‘`for`’’) and (ID,‘‘`forth`’’), while the second would contain (ID,‘‘`th`’’)

and $(null, \text{“”})$).

This approach allows a limited amount of nondeterminism by tolerating a lexical ambiguity until further parsing can reveal which of the several terminals actually match, while not allowing any parse table conflicts. Because of this nondeterminism it is necessary to use the GLR algorithm, or some other nondeterministic algorithm, to parse in this framework; this makes it subject to the same lack of a determinism guarantee as any other nondeterministic framework — although, due to the lack of parse-table conflicts, it is both less likely to lead to an ambiguity and less likely to run in significantly superlinear time.

It is also worthy of note that this system of scanning is, like SGLR, implicitly context-aware, since taking the nondeterministic choice of a token that is not valid in a certain context will lead to the failure of that choice. It is used in this capacity to resolve the issue of angle-brackets in C++ generics. However, Aycock and Horspool present the system primarily as a method for dealing with non-reserved keywords, such as in the PL/I language.

2.3.7 Lexer-based context-aware scanning.

As there is more than one kind of context, so there is more than one kind of context-aware scanning. While the approach presented in this thesis uses a context based on the parse state, Rus and Halverson [RH98] present an approach that introduces context-awareness entirely within the scanner, allowing the maintenance of the scanner-parser separation of the traditional LALR(1) framework. This is done simply by allowing the scanner at each scan to take into account the results of a certain number of previous scans. This number is fixed, and must be set at scanner compilation time.

Rus and Halverson's approach is simpler than the one presented here, and easier to integrate into existing frameworks, but it is less expressive in that our approach

can take into account, through receipt of the parse state, *all* the previous scans. Also, the approach is presented as a solution to the problem of scanning in an entirely new framework that can perform partial parsing through pattern recognition on parts of the parser's input, rather than as a general scanning solution.

2.3.8 Component-based LR.

Component LR-parsing (CLR) [WBGM10] is an approach to extensible language parsing in which each language extension is represented as its own "component parser." Component parsers are then switched between while parsing to handle the various extension constructs.

To determine when to switch between component parsers, two new parse actions, *switch* and *return*, are introduced. These new actions are attributes of parse states rather than parse table cells (similar to reduce actions in an LR(0) parser). When a component-based parser encounters an *error* action, it will see if the current state contains a *switch* or *return* action, at which point it will either *switch* to another component parser, *return* to the previous component parser, or backtrack to the point where the last *switch* occurred (if a component parser failed and another one needs to be tried).

The component-based approach, however, is disadvantaged by the requirement that "switches" and "returns" must only occur when the parser encounters an *error* action. Our approach, while making use of a similar notion of "components," integrates the switches and returns more fully into the parsing process; this makes the process more flexible, so it is possible to choose whether host or extension syntax should take precedence at the entry points.

2.3.9 Pseudo-scannerless LR(1) parsing.

Denny [Den10] also tackles the problem of language composition in his doctoral thesis. He explicitly addresses not only extensible languages but a broader range of composite languages, including "regular sub-languages" such as those representing escape characters in string constants and "subtle sub-languages," which we would call contexts with different valid lookahead sets. He attacks the problem with three innovations: pseudo-scanners, minimal LR(1) parse tables, and an extensive reformation of lexical precedence.

Denny's idea of a pseudo-scanner is identical to our idea of a context-aware scanner, though Denny does not address the problem of implementation to the depth that we do. Minimal LR(1) parse tables are implemented as part of the IELR(1) parsing algorithm [DM08]. Like LALR(1), they reduce the number of parse states needed by merging together LR(1) states with identical item sets; unlike LALR(1), it is only a partial merge, merging only those states that will not cause parse table conflicts when merged.

Denny's modifications to lexical precedence are the most noticeable to the end-user of his parsing framework. It includes such innovations as the ability to control the method of disambiguation by token length (*i.e.*, a user-controlled override of the use of maximal munch); however, the most significant innovation is the separation of "lexical precedence" from "lexical tying." For Denny, lexical precedence resolves the problem of the same string being matched to two terminals at a particular scan, while lexical tying resolves the problem of one terminal i being matched to a string, when another terminal k should be matched to it but is not valid in that context. One application of lexical tying is keyword reservation.

In the related work section, Denny addresses our paper *Context-Aware Scanning for*

Parsing Extensible Languages [VWS07]. He points out that that paper focuses on extensible languages as opposed to the wider group of languages more handily parsed in conjunction with a context-aware scanner, and that our alterations to lexical precedence (see section 4.2) are not as thorough or expressive as those he presents — in particular, the distinction between lexical precedence and lexical tying, which could be used to solve the problems discussed in section 10.2.3 with lexical precedence vis-a-vis modular grammar extensions. Denny also states that our modified parsing algorithm may run differently from a traditional parser in two instances.

Firstly, he notes that the parser may match different lookahead tokens during multiple scans of the same point in the input, unlike a traditional parser, which operates on a fixed stream of tokens and hence always has the same token at the same input position. It is true that, in a limited number of cases, multiple scans in the same input position may be on entirely disjoint valid lookahead sets. However, this is not a problem, as all the parses for which this occurs are on syntactically invalid sentences. We discuss this further, with proofs, in sections 3.3 and 4.7.2.

Secondly, he states that parsers utilizing disambiguation functions (see section 4.3) may behave differently on the same grammar and input based on whether they are used with an LR(1) or an LALR(1) parse table. This is also true. However, in characterizing this as a flaw or error in the scanning algorithm, Denny rather misses the point that disambiguation functions are designed to handle specific ambiguities that have been reported to the grammar writer through the process described in section 4.7.3; they are not an integral part of the grammar, but a method of disambiguation applied in post process. Since it is not expected that disambiguation functions will be written before the grammar writers are informed of the existence of the ambiguities they resolve, grammar writers will be able to tell which disambiguation functions are needed whether they are using LALR(1) or LR(1).

2.3.10 PetitParser.

A very recent development in this area, making note of our work on parse table composition (see chapter 7), is Renggli *et al.*'s *PetitParser* [RDGN10]. A PEG-based tool producing parsers with comparable runtimes to their LALR(1) counterparts, PetitParser addresses the problem with ordered choice discussed in section 2.3.4. It does this by allowing unordered choice on the "merge points" that knit together grammars in a composition, such as a *bridge production* for an extension (see Definition 6.1.1 on page 148), and then parsing both choices instead of parsing only until one choice matches, as is usually done with PEGs.

Although this does ensure that ambiguities between extensions do not go unnoticed, it has some drawbacks. Firstly, the new unordered choice is an "exclusive-or," unlike BNF's inclusive-or, so it fails when parsing any strings that match both extensions. Use of this crude method of disambiguation entails that any ambiguities between extensions will go undetected at parser compile time and may later, at parse-time, cause syntax errors on perfectly valid extension syntax. Secondly, the parsing method used to handle unordered-choice operators is effectively non-deterministic, and according to Renggli *et al.* entails exponential parse times.

By contrast, with our approach to this problem, one can, through the use of transparent prefixes (discussed in section 4.5) parse and properly disambiguate — in the usual linear time — extensions whose languages overlap.

2.4 How current alternatives measure up.

In this section, we compare various parsing frameworks, including LALR(1), SGLR, and PEGs, using the evaluation criteria laid out in section 2.1.

2.4.1 Non-ambiguity/verifiability.

Grammars specified for the LALR(1) framework can be ambiguous, but no parser based on an ambiguous grammar will compile into a deterministic LALR(1) parser, so the LALR(1) framework is able to *verify* non-ambiguity — although many unambiguous grammars are also rejected by the verification.

A GLR or SGLR parser can be generated for any context-free language, including ambiguous ones, and there is no test or verification that can determine whether a parser is unambiguous. When an ambiguity does occur, it is regarded as a bug in the grammar; these grammars must then be "debugged" by tweaking the grammar to remove the ambiguity. Like any debugging process, this is not an "exact science," but it has met with some success [Vin07].

PEGs are universally unambiguous, so users of packrat parsers get this by default. However, there are certain problems that arise from this, which are discussed in section 2.4.4 on page 66.

2.4.2 Efficiency.

LALR(1) parsers, scanners, and packrat parsers all run in $O(n)$ time.

The GLR algorithm runs in $O(n^3)$ time at best, using the experimental Binary Right Nulled GLR (BRNGLR) modifications that implicitly convert a grammar to Chomsky Normal Form by breaking reductions into multiple stages [SJE07]. Scannerless GLR requires some modifications to the algorithm; the most efficient algorithm for *scannerless* GLR is Scannerless Right Nulled GLR (SRNGLR) [EKV09], which runs in $O(n^k)$ time, k being the maximum number of symbols on the right-hand side of a production in the grammar.

The memory footprint of all these approaches is reasonable, and roughly comparable; an SGLR parser takes up slightly more due to its capacity for nondeterminism and ambiguity, but this is negligible when parsing reasonable languages.

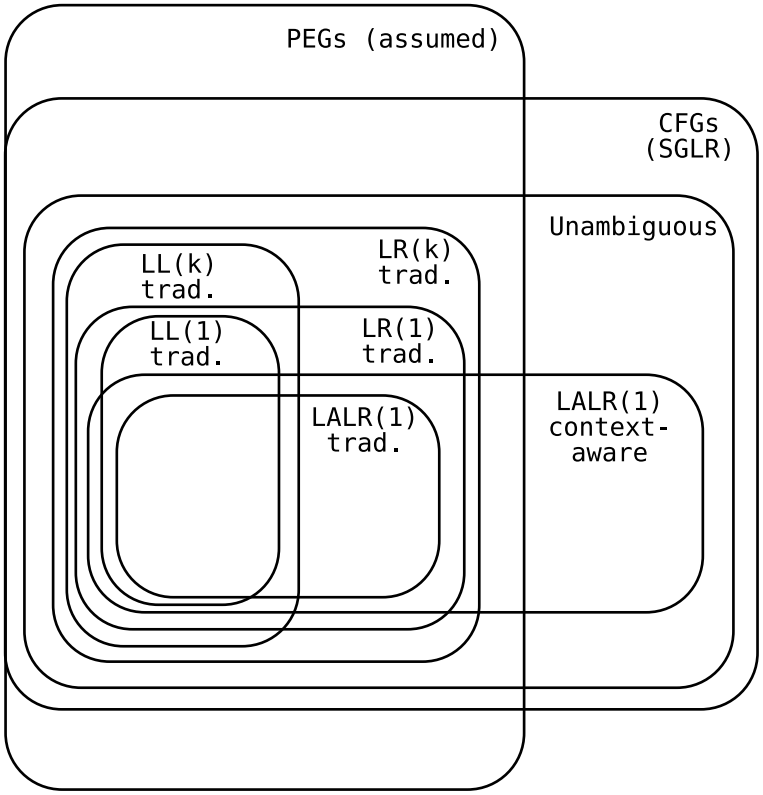

Figure 2.16: Expressivity of various parsing frameworks.

2.4.3 Expressivity.

Again, see Figure 2.16 for a Venn diagram comparing the expressivity of various frameworks. The LALR(1) framework admits only a restricted class of grammars, but this restricted class covers a large proportion of programming languages, as evidenced by

LALR(1)'s popularity. SGLR admits any context-free grammar, while PEGs seem to cut across that field, admitting some non-context-free languages and possibly excluding some context-free ones. As Ford says of PEGs:

> PEGs can express all deterministic LR(k) languages and many others, including some non-context-free languages ... a formal proof that there are context-free languages not expressible via PEGs appears surprisingly elusive. [For04]

It must be noted, however, that this refers to *languages*, not to grammars, so what has a very reasonable and straightfoward LR(k) grammar might have a highly convoluted PEG.

Also, a discrepancy must be mentioned concerning the traditional LALR(1) scanner formalisms vis-a-vis their scanner generators. The *Lex* scanner generator takes its specifications mostly in code (one block for each regular expression). It is possible, in theory, to implement a context-aware scanner in *Lex* using code, along with any other sort of hand-coded scanner. However, the closer one adheres to the *intended* use of *Lex* (minimal deviation from the pattern of a regular expression coupled with a return statement), the less expressivity one may achieve with it.

2.4.4 Flexibility.

Although it is not *very* flexible, one can alter a grammar within the traditional LALR(1) framework very easily as long as the altered grammar still compiles with no parse table conflicts. On the character level, however, the grammar is much more difficult to change due to the separation between scanner and parser: new terminals often cannot be introduced without disrupting the language of the parser in unwanted ways (*e.g.*, addition of new keywords automatically "reserves" them).

SGLR can admit any grammar and is therefore a quintessentially flexible framework.

PEGs are universally deterministic, so it would appear, topically, that the PEG framework is as flexible as SGLR. But this is not the case: the PEG framework guarantees determinism by replacing the "unordered choice" operator used in BNF with an "ordered choice" operator, `/`. This forces the grammar writer to order *every* set of choices before compilation of the parser, possibly making incorrect choices, and according to Ford, the problem of determining this correctness is undecidable:

> In place of having to determine whether two possible alternatives in a CFG are ambiguous, PEGs present language designers with the analogous challenge of determining whether two alternatives in a '`/`' expression can be reordered without affecting the language. This question is often obvious, but sometimes is not, and is undecidable in general. [For04]

Also, both packrat parsers and SGLR parsers are scannerless, so they lack that difficulty of traditional LALR(1) parsers.

2.4.5 Declarativeness.

In terms of declarativeness, LALR(1) is much better than custom approaches where the entire parser is coded up by hand. However, traditional LALR(1) frameworks allow much more custom code within the framework than SGLR or PEGs, so it is less declarative than both those approaches.

2.4.6 Expressivity example: AspectJ.

The AspectJ problem primarily illustrates the *expressivity* shortcomings of the traditional LALR(1) framework.

Both the traditional LALR(1) framework [HdMC04, KHH+01] and the SGLR framework [BETV06] have been successfully used to parse AspectJ, although not without drawbacks. Our approach offers a less problematic solution.

As exemplified by Figure 1.2 on page 12, AspectJ requires a *context-aware* scanner, with several keyword-matching lexemes needing to be matched differently based on the context in which they occur. There are also places where the same string might, depending on the context, need to be separated into a differing number of tokens. For example, in an aspect construct, the string `get*` would be matched as a single token describing a pattern to match. In a regular Java construct, `get*` would be matched as two tokens, an identifier `get` and a multiplication operator `*`.

The reference implementation, `ajc`, uses a piecemeal approach where different parsers are used to parse different parts of an AspectJ source file. The AspectBench Compiler (`abc`) [HdMC04] uses a single LALR(1) grammar, but makes use of a custom scanner with several "modes" that are changed by the custom code.

SGLR allows a declarative solution to the problem, but AspectJ cannot be parsed by the SGLR framework as originally implemented. As noted by Bravenboer *et al.*:

> We have just a single identifier non-terminal[3] ... but an identifier can occur in every context, and for every context we need to reserve a different set of keywords. Since we cannot refer to the identifier in a specific context, it is impossible to define reserved keywords for it. [BETV06]

The SGLR implementation of AspectJ relies on a novel construct called "grammar mix-ins," which *parameterize* a certain grammar or nonterminal so that it will parse differently in different contexts.

[3] In a framework utilizing a traditional scanner, this "non-terminal" would be called a terminal, but in SGLR only characters are properly "terminals."

On the other hand, since our approach [Sch09] retains the use of terminals, the identifier is still a terminal and it is simple to specify two different identifier terminals, one for the Java context and one for the aspect context. Hence, we can parse AspectJ both declaratively and deterministically.

2.4.7 Flexibility example: ableJ.

The problems with implementing ableJ in a traditional LALR(1) framework are enumerated above, and are primarily related to the *flexibility* shortcomings.

SGLR is somewhat better with the problem of combining extensions in ableJ, since the class of languages it parses is closed under composition. However, there is still no guarantee that a *deterministic* parse occurs, even if a deterministic parser is produced when Java is composed alone with each extension.

Use of PEG-based grammars solves the determinism problem, but introduces another problem, the problem of ordering choices, which brings down the verifiability of the approach. If several extensions are written separately, ordering the choices of the combined grammar must be done by the person who composes the extensions, and that person may not be a parsing expert. It is undecidable whether the order of choices affects the language represented by the grammar; hence, there is no guarantee that the choices the possible non-expert makes are correct ones, and s/he might not even want to consider the problem.

2.4.8 Summary.

In general, SGLR sacrifices efficiency and verifiability for gains in expressivity, flexibility and declarativeness compared to LALR(1). PEGs sacrifice expressivity and verifiability for universal determinism and a modest gain in flexibility.

The context-aware framework presented in this thesis retains the hard-and-fast determinism guarantee of LALR(1) without sacrificing the objective correctness verification that PEGs lack. The expressivity and flexibility are both significantly increased with respect to LALR(1), as evidenced by the examples cited above, although obviously not to the levels of GLR; the framework also does away with some of the "brittleness" of the LALR(1) algorithm, due to the fact that the scanner being aware of context can help with the necessary disambiguation. The declarativeness is also increased, particularly due to the modifications listed in chapter 4.

The efficiency of the framework depends on the implementation, both in the sense of time/space complexity and practical runtime and memory requirements; this is discussed in detail in chapter 5. The efficiency appears to fall very close to, but not exactly on, that of the traditional LALR(1) framework; all increases in time and space complexity over the traditional framework are known to be constant with respect to the size of the input to the parser — the usual measurement. One implementation maintains roughly identical space requirements to the traditional approach while increasing the runtime in a proportion linear to the number of terminals in the grammar (although the constant multiplier is very low, so the increase is negligible for most reasonable grammars). Another maintains identical time complexity but at the cost of a significantly larger memory footprint, which is nevertheless not unreasonable for several languages, such as C. This is discussed further in chapters 8 and 10.1.1.

For a fuller documentation of how the context-aware LALR(1) framework measures up against these criteria (and against the other frameworks), many more measurements on non-trivial grammars will be required.

Chapter 3

Context-aware scanning.

In this chapter, a formal specification is given of the algorithm for context-aware scanning, based on work presented in our paper *Context-Aware Scanning for Parsing Extensible Languages* [VWS07]. Specifications are made both for the abstract idea of context-aware scanning, which can be employed in any parsing framework where it is possible to run the scanner and parser in lock-step, and for the specific meaning of context-aware scanning within the deterministic LR framework. In the LR-specific discussion is given the algorithm of a parser utilizing a context-aware scanner in pseudocode and an example illustrating its operation.

3.1 Traditional and context-aware scanners.

In this section, we define the *valid lookahead set* — a concept central to context-aware scanning — and then give a formal definition of the requirements of a scanner in the traditional framework and of a *context-aware* scanner.

Concerning regular expressions, we will use the label *Regex* for the set of all regular expressions over some alphabet Σ, and define the function $L : \mathit{Regex} \to \mathscr{P}(\Sigma^\star)$ to map

each regular expression to its language. Thus, for example, $L(\mathtt{a}^\star) = \{\varepsilon, \mathtt{a}, \mathtt{aa}, \ldots\}$. When we discuss a mapping of a set of terminals T to regular expressions, we will use a function $regex : T \rightarrow Regex$ to signify it.

We next give precise definitions of the familiar terms *token* and *lexeme*. We do not assign any alternate meaning to them; a *token* is a pair consisting of a terminal and a string that matches its regular expression, $(t \in T, w \in L(regex(t)))$, while this string w is called a *lexeme*. As an example, in most programming languages there is an identifier terminal, call it *Id*, with a regular expression matching alphanumeric strings. Any specific identifier (say, `foo`) that is matched is the lexeme, and its token would be $(Id, \mathtt{foo})$. The formalism of the lexeme can be easily extended to support returning additional data, such as line and column numbers, but we omit this for simplicity.

A context-aware scanner must by definition have some means of being aware of context; this context is represented by the *valid lookahead set*, defined immediately below. In this high-level definition, we are deliberately nonspecific about what is meant by "valid" so as to enable this idea to be used with a wide range of parsing frameworks. In Definition 3.2.1 on page 78, we provide a useful definition for use with the LR framework, which has the advantage of being able to be determined automatically for each parse state, with no special input from the grammar writer.

Definition 3.1.1. *Valid lookahead set.*

> A *valid lookahead set* corresponding to a given point in a parse is defined as the set of all terminals deemed to be valid at that point in the parse, *i.e.*, the set of terminals that the scanner is allowed to return.

We now define the minimum requirements for both traditional and context-aware scanners. A traditional scanner must be a function defined over input strings, while a context-aware scanner must be a function defined over triples of parse states, terminal

sets, and input strings. Any function matching the criteria laid out in the definitions qualifies as a traditional or context-aware scanner for the given grammar. Note in particular that the definitions are not tied to any particular implementation or even any particular algorithm; as there are many different algorithms, and variations on algorithms, used for traditional scanning, there are also at least two algorithms (as discussed further in chapter 5) that can be used for context-aware scanning.

Note that these definitions make no allowances for *layout* such as whitespace and comments — terminals that are matched but not returned by the scanner. Layout is not discussed here for reasons of simplicity; accommodation of layout in a context-aware scanner is discussed in detail in section 4.4. They also make no allowance for returning the unconsumed part of the input, as is often done, since we are making the simplifying assumption that the scanner returns exactly the part of the input that will be consumed. Adding the capacity for this is straightforward.

Definition 3.1.2. *Requirements of a traditional scanner.*

A traditional or disjoint *scanner* for a terminal set T with associated regular expression mapping *regex* is a function that takes as input a string over a certain alphabet Σ and returns a token:

$$scan_{trad} : \Sigma^\star \to ((T \cup \{\bot\}) \times \Sigma^\star)$$

If $scan_{trad}(w) = (t, x)$, the following criteria must be satisfied:

- x must be a prefix of w and in the language of $regex(t)$ (or if $t = \bot$, indicating "no match," $x = \varepsilon$).
- There must be no longer prefix of w in the language of any regular expression in the grammar, $\bigcup_{t \in T} L(regex(t))$ (the principle of "maximal munch").

Definition 3.1.3. *Requirements of a context-aware scanner.*

A context-aware scanner for a terminal set T with associated regular expression mapping *regex* is a function that takes as input a valid lookahead set, and a string over a certain alphabet Σ and returns a token:

$$scan_{ca} \ : \ \mathscr{P}(T) \times \Sigma^\star \to ((T \cup \{\bot\}) \times \Sigma^\star)$$

If $scan_{ca}(validLA, w) = (t, x)$, the following criteria must be satisfied:

- t must be in $validLA \cup \{\bot\}$.
- x must be a prefix of w and in the language of $regex(t)$ (or if $t = \bot$, indicating "no match," $x = \varepsilon$).
- There must be no longer prefix of w in the language of any regular expression in the given valid lookahead set, $\bigcup_{u \in validLA} L(regex(u))$. This is, again, the principle of maximal munch, but now *subordinated* to the specific context.

As noted, the primary *functional* difference between a traditional and a context-aware scanner is that in a traditional scanner, the principle of disambiguation by maximal munch is paramount: a longer lexeme will be matched even if the terminal it matches is not valid (*i.e.*, it will result in a parse error if returned). In a context-aware scanner, by contrast, the principle of maximal munch is subordinated to that of context, so matching a shorter string to a valid terminal is preferred over matching a longer string to an invalid terminal.

To repeat the example of generic type expressions, if there are two operators > and >>, and a context in which > is valid and >> is not, the traditional scanner will follow maximal munch and match >>, while the context-aware scanner will follow context and

match >. The lack of contextual information in the traditional scanner gives rise to such requirements as are discussed in section 1.3.

Note that this exclusion of invalid terminals from consideration in the scanner also means that on syntactically invalid input, a context-aware scanner will return $\perp$ instead of returning an invalid token t. This means that the parser would not have enough information at that point to give an error message such as "Unexpected token t." To obtain this information, if the scanner returns $\perp$, it must be rerun with a valid lookahead set consisting of all the terminals in the grammar, which will allow the parser to match the unexpected token and output the error message. This is discussed further in section 4.7.4 on page 125.

3.2 Context-aware scanning in the LR framework.

In the previous section, we outlined an abstract idea of context-aware scanning that can be applied to a broad range of parsing algorithms using any arbitrary criterion for what constitutes "valid" lookahead in any given context. In this section, we present an application specifically to the LR parsing algorithm, which uses the parse state as the context for building valid lookahead sets, meaning that all valid lookahead sets can be automatically generated with no explicit input from the grammar writer.

Firstly, we give some definitions relating to grammatical constructs within the standard LALR(1) scanning and parsing framework; defining precisely what a context-free grammar and a parse table are, and the criteria for a parse table being conflict-free. We then give more precise definitions of valid lookahead sets and context-aware scanners within the framework, present a lightly modified LR parsing algorithm making use of context-aware scanning, and step through an example of this algorithm in operation.

3.2.1 Preliminary definitions.

A context-free grammar is defined in the usual way, with a finite set T of terminals, a finite set NT of nonterminals, a finite set P of productions (of the form $NT \rightarrow (T \cup NT)^\star$), and a start nonterminal s. But we also add, as part of the grammar, the mapping *regex* from terminals to regular expressions defined above. Regular expressions are not usually included as part of context-free grammars; however, we include them because to build a context-aware scanner, one must have access to the context-free syntax as well as the lexical syntax of a language, and thus the formalisms for lexical and context-free syntax must be in some way combined.

Formally, a context-free grammar then forms a 5-tuple:

$$\Gamma = \langle T, NT, P, s \in NT, regex \ : \ T \rightarrow Regex \rangle$$

We also define parse tables precisely — these being the constructs generated from context-free grammars by the LR parser compilation process that are used to direct the execution of the LR parsing algorithm. We will use the label *States* to denote the set of all rows in all parse tables; *i.e.*, any parse table PT will have a set of states, usually labeled $States_{PT}$, which is a subset of *States*. Distinct parse tables will have disjoint subsets of states.

Formally, a parse table comprises a 4-tuple

$$PT = \langle \Gamma_{PT}, States_{PT}, s_{PT} \in States_{PT}, \pi_{PT} \ : \ States \times (T \cup NT) \rightarrow \mathscr{P}(Actions) \rangle$$

The components are also defined in the usual manner, specifying the context-free grammar Γ_{PT} (with terminal set T_{PT}, nonterminal set NT_{PT}, and production set P_{PT}) from which it was constructed by the familiar process, a state set $States_{PT} \subset States$ (one state of which is designated as a start state, s_{PT}), and a function comprising the literal table. Note that the presence of the grammar Γ_{PT} is not necessary for the operation

of the table, but we include it in the formal definition as many other definitions in this chapter and in chapter 4 rely on its constituent parts.

The function π_{PT} maps pairs of states and grammar symbols (members of the set $States_{PT} \times (T_{PT} \cup NT_{PT})$) to zero or more LR parse actions (members of the set *Actions*). These parse actions are what direct the parser how to proceed; they can be *shift* actions (signifying the consumption of a token of input), *reduce* actions (signifying the construction of a part of the final parse tree), *accept* actions (signifying the end of the parse); and *goto* actions (performed immediately after a reduce action). Formally, the range of the table function is $Actions = \{accept\} \cup \{reduce(p) \ : \ p \in P_{PT}\} \cup \{shift(x) \ : \ x \in States\} \cup \{goto(x) \ : \ x \in States\}$. Note that in this definition, while the reduce actions are limited to productions that belong to Γ_{PT}, shift and goto actions are permitted to have for a destination any state in any parse table. This is necessary in consideration of parse tables that are assembled separately but are meant to be used in conjunction, and hence will contain references to each other. We discuss parse tables of this sort in chapter 7.

We also define several other terms relating to parse tables.

- *Parse table row.* For any $n \in States_{PT}$, the mapping of actions $\{(t, \pi(n,t)) \ : \ t \in (T \cup NT)\}$ is called a *row* of the parse table.

- *Parse table cell.* A particular mapping $\pi_{PT}(n,t)$ is called a *cell* in the table.

- *Error action.* If a parse table cell is empty — that is, if for some state and terminal (n,t), $\pi_{PT}(n,t) = \emptyset$ — this cell is said to contain an *error action*.

- *Parse table conflict.* If a parse table cell contains more than one action — that is, if for some state and terminal (n,t), $|\pi_{PT}(n,t)| \geq 2$ — this cell is said to have a *parse-table conflict*. If there is a shift action among the table's actions, it is called a *shift-reduce* conflict; otherwise, a *reduce-reduce* conflict.

A parse table is *conflict-free* if it contains no parse table conflicts.

3.2.2 Valid lookahead sets and scanner requirements.

The main question to be settled in this application of context-aware scanning to the LR framework is how to determine the valid lookahead sets — which terminals are deemed valid in which context. There is more than one way to do this, but we have settled on a method that is both useful and automatic — *i.e.*, the grammar writer does not have to supply any explicit context information to help build them, as with custom scanners like the AspectJ scanner discussed in section 9.1.

The context represented by the valid lookahead set is the parse state; which terminals are valid in a parse state is able to be told from the parse actions in that state. A parse table row has some cells with actions in them, and other cells that are empty. The empty cells represent terminals that are not valid syntax at that parse state; if the scanner were to match such a terminal, and return it, the parser would fail with a syntax error such as "Unexpected terminal 'xyz'." Hence, there is no reason for the scanner to match such a terminal, and indeed doing so will often cause unnecessary lexical conflicts. Thus, for the LR framework we define the valid lookahead set as those terminals that have valid parse actions and hence will not cause a syntax error.

Definition 3.2.1. *Valid lookahead set in the LR framework.*

> The *valid lookahead set* for a given LR parse state n is defined as all those terminals t that have non-error states in the parse table row corresponding to n. Formally, $validLA_{PT} : States_{PT} \rightarrow \mathscr{P}(T_{PT})$ and
>
> $$validLA_{PT}(n) = \{t \in T_{PT} : \pi_{PT}(n,t) \neq \emptyset\}$$

We next define the requirements of context-aware scanners with respect to the LR framework. In being made specific to LR, these requirements incorporate the parse

state as the context to be passed to the scanner; this expands on the signature of the context-aware scanner function to include the current parse state.

Definition 3.2.2. *Requirements of a context-aware scanner in the LR framework.*

A context-aware scanner pertaining to a parse table PT is a function that takes as input a parse state, a valid lookahead set, and a string over a certain alphabet Σ and returns a token:

$$scan_{PT} \ : \ States_{PT} \times \mathscr{P}(T_{PT}) \times \Sigma^{\star} \rightarrow ((T_{PT} \cup \{\perp\}) \times \Sigma^{\star})$$

If $scan_{PT}(n, validLA, w) = (t, x)$, the following criteria must be matched:

- t must be in $validLA \cup \{\perp\}$.
- x must be a prefix of w and in the language of $regex_{PT}(t)$ (or if $t = \perp$, indicating "no match," $x = \varepsilon$).
- There must be no longer prefix of w in the language of any regular expression in the given valid lookahead set, $\bigcup_{u \in validLA_{PT}(n)} L(regex_{PT}(u))$.
- It is permitted for $validLA$ not to be equal to $validLA_{PT}(n)$. If this is the case, then further rules may be set for cases in which $t \notin validLA_{PT}(n)$. This is used when keyword reservation or other lexical precedence settings make it necessary to scan for other terminals besides the actual valid lookahead; lexical precedence is discussed in detail in section 4.2.

Prima facie, it might seem redundant for the parser to take the valid lookahead set explicitly as input if it also takes the current parse state, since the valid lookahead set for that state, as set out in Definition 3.2.1, can be easily *inferred* from the parse table. However, we have made several modifications to the context-aware scanning algorithm

to meet practical challenges, which are discussed at length in chapter 4. Some of these involve scanning for different sets of terminals than $validLA_{PT}(n)$ (although still using that set as a reference); see section 4.2.3 on page 96 for an example of when a different set of terminals is passed to the scanner.

3.2.3 Parsing algorithm.

For the purposes of this chapter, only the abstract definition of a context-aware scanner is used, as there are several ways to implement one (see chapter 5) and embellish one (see chapter 4).

There are not many modifications needed to the standard LR parsing algorithm to make it work with context-aware scanners; see Figure 3.1 on the following page for pseudocode describing the modified LR algorithm. For the most part, it is unmodified and does not require any particular explanations; it is passed a string as input and maintains the parser's position in this string by shaving off pieces of it so that the next lexeme to be scanned is always at the front of the string. While parsing it maintains a parse stack of ordered pairs as defined in section 2.2.3.3, the first element being a state number n and the second a token or concrete syntax subtree CST.

The most significant departure occurs in lines 6b and 6c; instead of simply calling out to the scanner for the next token based only on its position in the input, it calls to a context-aware scanning apparatus. Line 6b calls to a function $getValidLA_{PT}$, which takes a state number and returns a valid lookahead set for that state number; line 6c calls to a function $runScan_{PT}$, which calls the $scan_{PT}$ function based on the state number, the valid lookahead set from line 6b, and the input, and returns the result of that function.

In the simplest version of the algorithm, these two auxiliary functions are as they appear in Figure 3.2 on page 82, with $getValidLA_{PT}$ acting as a wrapper for $validLA_{PT}$ and $runScan_{PT}$ acting as a wrapper for $scan_{PT}$. Both these functions must be altered for

function parse(w) : $\Sigma^\star \rightarrow$ *ConcreteSyntaxTree*

1. *startState* = s_{PT}
2. *done* = `false`
3. *pos* = 0
4. *tok*= `nil` **// Current lookahead token**
5. $push(\langle startState, \texttt{nil}\rangle)$
6. while *done* = `false` do
   (a) $(ps, \_) = peek()$ **// Read the parse state on the top of parse stack**
   (b) $vLA = getValidLA_{PT}(ps)$ **// Might be the function shown in Figure 3.2 that simply calls** $validLA_{PT}$ **from Def. 3.2.1; might be a modified function. See section 4.2**
   (c) $(t_{LA}, lexeme) = runScan_{PT}(ps, vLA, w)$ **// Might be** *scan* **from Definition 3.2.2 or a modified function; see chapter 4**
   (d) if $t_{LA} = \perp$ then exit with syntax error
   (e) $action = \pi_{PT}(ps, t_{LA})$
   (f) switch *action*
      i. case $shift(ps')$:
         A. **// Perform semantic actions for** t_{LA}
         B. $push(\langle ps', t_{LA}\rangle)$
         C. Remove the prefix *lexeme* from the front of w **// Consume token**
      ii. case $reduce(p : A \rightarrow \alpha)$:
         A. $children = multipop(|\alpha|)$
         B. $tree = p(children)$ **// Build concrete syntax subtree**
         C. **// Perform semantic actions for** p
         D. $ps' = \pi_{PT}(ps, A)$ **// Look up in goto table**
         E. $push(\langle ps', tree\rangle)$
      iii. case *accept*:
         A. if $w = \varepsilon$ then *done* = `true`
         B. else **// Report error and exit**
7. end while
8. let $\langle \_, CST\rangle = pop()$ in return *CST*

Figure 3.1: Modified LR parsing algorithm for context-aware scanning.

- function $getValidLA_{PT}(n)$: $States_{PT} \to \mathscr{P}(T_{PT})$

  1. return $validLA_{PT}(n)$

- function $runScan_{PT}(n, validLA, w)$: $States_{PT} \times \mathscr{P}(T_{PT}) \times \Sigma^\star \to ((T_{PT} \cup \{\bot\}) \times \Sigma^\star)$

  1. return $scan_{PT}(n, validLA, w)$

Figure 3.2: Auxiliary functions $getValidLA_{PT}$ and $runScan_{PT}$ (unembellished).

some of the practical modifications listed in chapter 4, including keyword reservation and lexical precedence.

3.2.4 Example of operation.

- **Nonterminals:** E (start), N
- **Terminals:**
  - *op* with regular expression `-|+`
  - *neg* with regular expression `-`
  - *digit* with regular expression `0|([1-9][0-9]*)`
  - \$, the pseudo-terminal representing the end of input
- **Productions:**
  - $E \to N\ op\ E \mid N$
  - $N \to digit \mid neg\ digit$

Figure 3.3: Simple arithmetic grammar.

Consider the grammar in Figure 3.3. Note that this grammar contains a lexical construct that requires a context-aware scanner: both the terminals *op* and *neg* match the lexeme `-`; hence, in the traditional framework one would have a *lexical ambiguity*

	$	*op*	*neg*	*digit*	*E*	*N*
1			s5	s3	g2	g4
2	a	s6				
3	r($N \rightarrow digit$)	r($N \rightarrow digit$)				
4	r($E \rightarrow N$)	r($E \rightarrow N$)				
5				s7		
6			s5	s3		g8
7	r($N \rightarrow neg\ digit$)	r($N \rightarrow neg\ digit$)				
8	r($E \rightarrow E\ op\ N$)	r($E \rightarrow E\ op\ N$)				

Table 3.1: Parse table for simple arithmetic grammar.

(a set of terminals matching the same string) between those two terminals even though they never occur in the same context, and a traditional scanner is of limited utility in this case. On the other hand, with a context-aware scanner no such ambiguity would occur.

See Table 3.1 for the parse table compiled from this grammar, a simplified arithmetic grammar specifying a language of sums and differences over positive and negative numbers.

Now consider parsing the expression `3-2+-1` using this parser. Below is explicitly enumerated the parsing process for this string; each bullet point in the list represents a particular state of the parser, *i.e.*, a particular configuration of the parse stack. Those points marked in **bold face** indicate states in which the behavior of the context-aware scanner differs from that of a traditional scanner.

As may be seen, when the parser is in a state where only the terminal *op* is valid lookahead, `-` is matched as *op*, and where only *neg* is in the valid lookahead set, `-` is matched as *neg*.

1. The parser starts in state 1. The *CST* element in this stack element will never be read, so it is set to a "nil" value, as shown in line 5 of Figure 3.1.

- Stack = $[\langle 1, \texttt{nil} \rangle]$, $w = \texttt{3} - \texttt{2} + -\texttt{1}$.

2. The valid lookahead set in state 1 is $\{neg, digit\}$. The call to $scan_{PT}$ returns the token $(digit, \texttt{3})$. $\pi(1, digit) = shift(3)$, so 3 is consumed; a parse-tree leaf node is built from the token and pushed on the stack.

   - Stack = $[\langle 3, digit(\texttt{3}) \rangle, \langle 1, \texttt{nil} \rangle]$, $w = -\texttt{2} + -\texttt{1}$.

3. **The valid lookahead set in state 3 is $\{\$, op\}$. The call to $scan_{PT}$ returns the token $(op, -)$, neg not being in the valid lookahead set.**

   $\pi(3, op) = reduce(N \rightarrow digit)$**.**

   Had a traditional scanner been used, it would have at this step had to find some way of resolving a lexical ambiguity between *op* and *neg*.

   - **Stack =** $[\langle 4, N(\texttt{3}) \rangle, \langle 1, \texttt{nil} \rangle]$**,** $w = -\texttt{2} + -\texttt{1}$**.**

4. The valid lookahead set in state 4 is $\{\$, op\}$. $(op, -)$ is again returned; a reduction is performed on the production $E \rightarrow N$.

   - Stack = $[\langle 2, E(\texttt{3}) \rangle, \langle 1, \texttt{nil} \rangle]$, $w = -\texttt{2} + -\texttt{1}$.

5. The valid lookahead set in state 2 is $\{\$, op\}$. $(op, -)$ is once again returned; - is consumed and a shift is performed to state 6.

   - Stack = $[\langle 6, op(-) \rangle, \langle 2, E(\texttt{3}) \rangle, \langle 1, \texttt{nil} \rangle]$, $w = \texttt{2} + -\texttt{1}$.

6. The valid lookahead set in state 6 is $\{neg, digit\}$. The call to $scan_{PT}$ returns the token $(digit, \texttt{2})$; 2 is consumed and a shift is performed to state 3.

   - Stack = $[\langle 3, digit(\texttt{2}) \rangle, \langle 6, op(-) \rangle, \langle 2, E(\texttt{3}) \rangle, \langle 1, \texttt{nil} \rangle]$, $w = + - \texttt{1}$.

7. The valid lookahead set in state 3 is $\{\$, op\}$. The call to $scan_{PT}$ returns the token $(op, +)$. $\pi(3, op) = reduce(N \rightarrow digit)$.

   - Stack = $[\langle 8, N(2)\rangle, \langle 6, op(-)\rangle, \langle 2, E(3)\rangle, \langle 1, \texttt{nil}\rangle]$, $w = +-\texttt{1}$.

8. The valid lookahead set in state 8 is $\{\$, op\}$. $(op, +)$ is again returned; a reduction is performed on the production $E \rightarrow E\ op\ N$.

   - Stack = $[\langle 2, E(\texttt{3}-\texttt{2})\rangle, \langle 1, \texttt{nil}\rangle]$, $w = +-\texttt{1}$.

9. The valid lookahead set in state 2 is $\{\$, op\}$. $(op, +)$ is once again returned; + is consumed and a shift is performed to state 6.

   - Stack = $[\langle 6, op(+)\rangle, \langle 2, E(\texttt{3}-\texttt{2})\rangle, \langle 1, \texttt{nil}\rangle]$, $w = -\texttt{1}$.

10. **The valid lookahead set in state 6 is $\{neg, digit\}$. The call to $scan_{PT}$ returns the token $(neg, -)$, op not being in the valid lookahead set. - is consumed and a shift is performed to state 5.**

    - **Stack =** $[\langle 5, neg(-)\rangle, \langle 6, op(+)\rangle, \langle 2, E(\texttt{3}-\texttt{2})\rangle, \langle 1, \texttt{nil}\rangle]$**,** $w = \texttt{1}$**.**

11. The valid lookahead set in state 5 is $\{digit\}$. The call to $scan_{PT}$ returns the token $(digit, \texttt{1})$. 1 is consumed and a shift is performed to state 7.

    - Stack = $[\langle 7, digit(\texttt{1})\rangle, \langle 5, neg(-)\rangle, \langle 6, op(+)\rangle, \langle 2, E(\texttt{3}-\texttt{2})\rangle, \langle 1, \texttt{nil}\rangle]$, $w = \varepsilon$.

    For the remainder of the parse, $scan_{PT}$ will return the token $(\$, \varepsilon)$, signifying "end of input."

12. From state 7, a reduction is made on production $N \rightarrow neg\ digit$.

- Stack = $[\langle 8, N(-1)\rangle, \langle 6, op(+)\rangle, \langle 2, E(3-2)\rangle, \langle 1, \texttt{nil}\rangle]$, $w = \varepsilon$.

13. From state 8, a reduction is made on production $E \rightarrow E\ op\ N$.

    - Stack $[\langle 2, E\,(3-2+-1)\rangle, \langle 1, \texttt{nil}\rangle]$, $w = \varepsilon$.

14. From state 2, the *accept* action is taken and parsing completes.

3.3 Discussion.

In this chapter, we have enumerated the minimal set of characteristics of a context-aware scanner, and described an LALR(1) parsing algorithm that may be used in conjunction with it. In this section we provide a brief further discussion of the process.

As partially shown in the example of operation in the previous section, context-aware scanning provides an increase in the expressivity of the LALR(1) framework over that using a traditional scanner. In steps 3 and 10 are demonstrated the process of disambiguation by context, which allows the existence of two or more terminals with the same regular expression, or overlapping regular expressions. Although that particular grammar could, in theory, be restructured to use a single terminal for `-`, this would increase the complexity of the grammar by requiring new productions (one production each for `+` and `-` as opposed to a single one for binary operations). And, of course, there are those grammars that cannot be thus restructured; most significantly, the extensible languages that are developed by different people and may have to be composed by programmers who know nothing about restructuring of grammars.

Tailing this increase in expressivity, however, is what might appear to be a drawback: while the traditional LALR(1) algorithm operates, in theory, on a pre-processed stream of tokens, and the disjoint scanner has been over each of these tokens only once, the modified algorithm calls the scanner (on line 6c of Figure 3.1 on page 81) *every* time

lookahead is needed, which may include several calls from the same position before a token of input is consumed; see steps 3 through 5 of the example.

The obvious solution, to avoid needless repetition in scanning, is to memoize the results of the scan at any given position in the input, so when called again at that position, the scanner can simply return the previous result. However, there is one issue to be resolved first: a reduce action, although it does not consume any input, does put the parser in a different state, which means the scanner will be scanning on a different valid lookahead set. We must therefore have some means of ensuring that the scanner will not match a different terminal in the new state.

This can be verified in most instances, simply because on syntactically valid inputs, an LR parser will not recognize a terminal t as lookahead if it will not eventually consume it in a shift action. This is true on account of the LR(1) versions of the *Closure* and *Goto* routines from Figure 2.12 on page 40 and Figure 2.13 on page 40. Recall from page 45 that *Goto* maintains identical lookahead between items when moving the dot further to the right. A reduce action on production $A \to \alpha$ consists of popping $|\alpha|$ elements from the stack, taking the parser from a state with an item $I = (A \to \alpha\bullet)$ to a state with an item $I' = (A \to \bullet\alpha)$. The lookahead of these two items is exactly identical in an LR(1) DFA. In an LALR(1) DFA, where the state with I may have been merged with others and new lookahead introduced, the lookahead of I' is only known to be a subset of that of I. But if we are assuming a syntactically valid input, t must be in both lookahead sets (otherwise the parse would fail).

Now according to the *Closure* routine, the presence of I' with t in the lookahead set implies the presence of another item $B \to \beta \bullet A\gamma$ such that either $t \in \mathit{first}(\gamma)$ or t is in the item's lookahead set. Following the reduce action, the parser will take a goto action on nonterminal A, bringing it into a state where there is an item $B \to \beta A \bullet \gamma$. According to *Goto*, this item also has t in its lookahead set if $t \notin \mathit{first}(\gamma)$, so t still has a valid parse

action.

Therefore, if t is matched as lookahead before a reduce action, we know that t will be in the valid lookahead sets of all subsequent states in which the parser calls for lookahead, until t is shifted. In the above example, once *op* is matched for a reduce action it is consistently in the valid lookahead sets from steps 3 through 5, it being shifted in the latter step.

If we assume no lexical ambiguities within valid lookahead sets, *i.e.*,

$$\forall n \in States_{PT}.\,[\forall s,t \in validLA_{PT}(n).\,[s \neq t \Rightarrow L(regex_{PT}(s)) \cap L(regex_{PT}(t)) = \emptyset]]$$

this means that t would be matched in all those states and the scanner does not have to be run again. However, this assumption often does not hold and other methods are used to resolve lexical ambiguities; with these methods there will be a small number of cases where a rescan may be required, which are discussed further in section 4.7.

Chapter 4

Making context-aware scanning more useful in Copper.

This chapter discusses four modifications to the context-aware scanning algorithm presented above, which are part of the algorithm's implementation in Copper and serve to make it useful. These modifications are unique to context-aware scanning and some were originally presented in [VWS07] along with the algorithm.

The first is generalized precedence relations (section 4.2). In traditional scanners, lexical ambiguities (terminals with regular expressions matching the same string) must, in all cases except those where two terminals share the exact same regular expression, be resolved by the order in which the regular expressions appear in the specification. This is a strict total order, in which for any pair of terminals in the grammar, one takes precedence over the other. However, a context-aware scanner requires a more nuanced relation to handle the cases of terminals with identical regular expressions that do not cause lexical ambiguities because they occur in different contexts (such as the two `table` keywords in Figure 1.1 on page 9). Hence, context-aware scanning generalizes its notion of precedence to accommodate any sort of precedence relation.

The second is disambiguation functions (section 4.3). There are often cases in which a lexical ambiguity cannot be resolved by lexical precedence; either because a particular terminal in the ambiguity is not always to be preferred, or because the disambiguation must depend on lexical context, which disambiguation by lexical precedence does not. Disambiguation functions provide a more general, non-reserving method of disambiguation in this case.

The third is handling of layout (whitespace and comments) both per grammar and per production (section 4.4). In traditional scanners, layout is usually handled by means of regular expressions or terminals that are marked "ignore," so that when they are matched, nothing is returned to the parser. Grammar layout simulates these "ignore" terminals, while layout per production solves the problem of parsing languages that require different layout in different contexts (*e.g.*, HTML with JavaScript).

The fourth is transparent prefixes (section 4.5). Transparent prefixes are another technique of lexical disambiguation, allowing terminals such as the two `table` keywords to be disambiguated by prefixes placed before them in the fashion of Java fully-qualified names.

There is also an explanation (section 4.6) of the familiar concept of disambiguation by precedence and associativity on infix binary operators, which is employed in Copper with no modification whatsoever; and finally some issues to be resolved when implementing all these modifications in the same framework (section 4.7).

Except for two — the familiar disambiguation by operator precedence and associativity common to LR parser generators, and the new layout per production — these modifications are not specific to any parsing framework, and we discuss them in an abstract manner as well as their application in the LR framework.

4.1 Preliminary definitions.

For the LR-specific parts of the discussion we must first make formal definitions of LR DFAs, and of their constituent parts, states, LR items, and lookahead sets. These were discussed informally in section 2.2.3.

We first define an *LR item*. An LR item represents a particular point in a particular LR parse; each state in an LR DFA is built from several LR items, each representing a point that the parser could be in the parse. It is this packaging of LR items into LR DFA states that enables the LR parser to make a rightmost derivation of what it is parsing during a left-to-right parse: unlike an LL parser, which must decide exactly what grammatical construct it is parsing before it even starts to parse it, these states allow the LR parser to defer this decision until it is absolutely necessary (*i.e.*, when the parser is to perform a reduce action, finishing the parsing of the construct in question).

An LR item takes the form of a production with a bullet marker to indicate the position. To take a simple example, suppose there are two terminals t and u. If the parser has just consumed (shifted) t and is about to shift u, it will be in a state containing an item $A \rightarrow t \bullet u$ for some nonterminal A.

Definition 4.1.1. *LR item.*

> Formally, an *LR item* i consists of a pair $(p,n) \in P \times \mathbb{N}$, the p signifying the production being parsed, the n being the present position of the parse.

The item is usually written in the form of the production p with a dot or bullet following the first n symbols on p's right hand side. For example, the item $(A \rightarrow \alpha\beta\gamma, 2)$ would be written $A \rightarrow \alpha\beta \bullet \gamma$, while the item $(A \rightarrow \alpha, 0)$ would be written $A \rightarrow \bullet\alpha$. Let *Items* signify the set of all items.

We next define the *LR DFA state* and the LR DFA itself. As there is a direct correspondence between LR DFA states and the parse states generated from them, *States*

may also be used to indicate the set of all states in all LR DFAs. Functionally, an LR DFA state is identical to any other DFA state; the LR parser keeps track of its position in the parse by maintaining a *parse stack* of states, and transitions from state to state based on the input. Those transitions, however, are defined by a set of LR items, which as explained above represent all the different parses the parser could possibly be making when passing through the containing state. Also like other DFAs, an LR DFA is defined as the sum of its states and transitions.

Definition 4.1.2. *LR DFA state.*

An *LR DFA state* consists of a set of LR items. Let the function $items : States \rightarrow \mathscr{P}(P \times \mathbb{N})$ represent the item sets of an LR DFA state.

Definition 4.1.3. *LR DFA.*

An LR DFA for a grammar Γ_L consists of a 4-tuple

$$M_L = \langle \Gamma_L, States_L \subseteq States, s_L \in States_L, \delta_L \rangle$$

where $States_L$ is the DFA's state set, s_L is its start state, and $\delta_L : States \times (T \cup NT) \rightarrow States$ is its transition function.

We next define *lookahead*. In parsing, "lookahead" refers to a terminal that the parser "looks at" prior to consuming it, in order to decide what action to take. LL(k) parsers can read ahead up to k tokens past their current position in the input, but LR(1) and LALR(1) parsers read only one token ahead. The LR algorithms use lookahead to decide whether to shift, reduce, or exit with an error message, based on the lookahead terminal's parse table cell. The first time a parser will need a particular token from the scanner is as lookahead (since tokens are always considered as lookahead before they are consumed), which gives the valid lookahead set its name.

Definition 4.1.4. *Lookahead, item with lookahead.*

In LALR(1) and LR(1) DFAs, there will also be a set of *lookahead* associated with each item. Let the function $la_n : items(n) \rightarrow \mathscr{P}(T)$ represent the lookahead sets for the items of a certain state n. An item $i = A \rightarrow \alpha \bullet \beta$ with $la_n(i) = z$ is written $A \rightarrow \alpha \bullet \beta,\ z$.

4.2 Generalized precedence relations.

Here is discussed the general notion of lexical precedence relations, with a comparison of the kind used in traditional scanners with the kind we have developed for context-aware scanning. We also present an alternative implementation of the function $getValidLA_{PT}$ from Figure 3.2, implementing lexical precedence for context-aware scanning in the LR framework.

4.2.1 About lexical precedence — summary and definitions.

The regular expressions provided in the map *regex* are often not *exact* descriptions of the languages intended for them. The regular expressions for keyword terminals, which usually only match one string (*e.g.*, `int`), are mostly exact descriptions, but regular expressions with broader languages, such as those for identifiers (C, for example, uses the regular expression `[A-Za-z_$][A-Za-z0-9_$]*` for identifiers) are often not. Identifier terminals are usually given regular expressions matching all alphanumeric strings, but they are rarely intended to match *every* alphanumeric string — the keywords of the language, for example, are often not meant to be matched as identifiers, though they are certainly alphanumeric strings.

For the traditional scanning apparatus to guarantee that there are no lexical ambiguities, the language matched by each terminal must be disjoint from the others. For a

context-aware scanner, this is supposed to be true at least of the terminals within each valid lookahead set. However, in reality, the languages of the regular expressions provided for the terminals often overlap, especially keywords and identifiers. It is for such cases that lexical precedence is employed.

Definition 4.2.1. *Precedence relation.*

> A *precedence relation* $\prec : T \times T$ is a relation on the terminals of a grammar that places additional restrictions upon the output of the functions *scan* from Definitions 3.1.2 and 3.1.3. Specifically, if the token (t,x) has been returned from the scanner, then there can be no terminal with precedence over the terminal t also matching the lexeme x:
>
> $$scan(validLA, w) = (t,x) \Rightarrow \neg\exists u \in T.\,[t \prec u \wedge x \in L(regex(u))]$$
>
> In the LR framework, when speaking of a precedence relation associated with a parse table, the subscript PT may be added to $\prec$ and to the associated T, *regex*, *etc.*

4.2.2 Traditional and generalized precedence relations.

In most traditional scanners, the precedence relation $\prec$ is a strict total order (asymmetric, transitive, and total) that is inferred from the order in which regular expressions are placed in the scanner specification. Those that are nearer the top of the file have the higher precedence, so that if t is specified before u in the file, then $t \prec u$, and when the regular expressions of t and u overlap — *i.e.*, $x \in L(regex(t)) \cap L(regex(u))$ for some string x — a scanner run on $w = x \cdot y$ will return (u,x).

This does not work for context-aware scanning, since the primary advantage of the context-aware scanner is to be able to handle situations where the regular expressions

of two terminals t and u overlap on some lexeme x, but t and u never appear in the same valid lookahead set. Having a strict total order for precedence would negate most of this advantage, since if $t \prec u$, t could never match x even if u was not in the valid lookahead set. More importantly, since the strict total order is transitive and total, it is almost impossible *not* to have such a relation for any two terminals.

For example, two of our extensions to Java in ableJ contain keywords matching the lexeme `table`; both keywords are used in Figure 1.1 on page 9. Within the ableJ grammar, they never occur in the same valid lookahead set. In a strict linear precedence order, one of these keywords would have to take precedence over another, which would mean that one of them could never be matched, despite the fact that they are never valid in the same place.

One might ask why the valid lookahead set could not be used as a boundary to "contain" a strict total order of precedence, by simply matching the highest-precedence terminal within the given valid lookahead set. But this makes it impossible to use precedence to implement its most common use: *keyword reservation*, wherein a terminal k with a regular expression matching one string (a keyword) and a terminal i with a regular expression matching any alphanumeric string (an identifier) are set with $i \prec k$ so that the identifier does not match the keyword's string under any circumstances, even in contexts where the keyword is invalid. For example, in C, the statement `while(while == 0);` is syntactically invalid, since `while` is a reserved keyword and cannot be used as an identifier.

Thus, a global, context-independent precedence is a necessity. But it does not at all need to be a strict total order; indeed, its primary application, keyword reservation, usually requires only a series of precedence relations between several keyword terminals and just one or two identifier terminals, with neither transitivity or totality a necessity. See Figure 4.1 on the next page for an example of the precedence relations needed

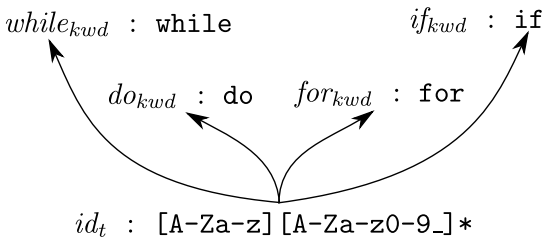

Figure 4.1: Topography of a precedence relation on Java's lexical syntax ($id_t \prec while_{kwd}$, *etc.*).

between the flow-control keywords of Java and Java identifiers.

Therefore, in context-aware scanners, the concept of precedence remains exactly the same as put forth in Definition 4.2.1 on page 94, but is no longer required to be a strict total order. Precedences are given explicitly, in a form that essentially says, "Terminal t has precedence over terminal u."

It might be asked whether this is *too* explicit and generalized, requiring too many individual ordered pairs to be written out in specifications of lexical syntax. In Copper's implementation, however, the implementation is significantly simplified by the introduction of *lexer classes*. These are simply sets of terminals that can be used as shorthand in specifying lexical precedence; for example, one could specify that the terminals k_1, k_2, ..., k_n are all part of a lexer class K. Then one could specify that lexer class K takes precedence over terminal i, which is shorthand for $i \prec k_1 \wedge i \prec k_2 \wedge \ldots i \prec k_n$.

4.2.3 Implementing precedence in a context-aware scanner.

We now formally define the minimal requirements for a context-aware scanner that incorporates lexical precedence, modifying Definition 3.1.3 on page 74.

Definition 4.2.2. *Context-aware scanner with lexical precedence.*

A context-aware scanner incorporating lexical precedence has an additional

restrictions imposed on its output (t,x), *viz.*, there must be no terminal of higher precedence matching x:

$$(t,x) = scan(validLA, w) \Rightarrow \neg\exists u \in T_{PT}\,.\,[t \prec u \wedge x \in L(regex(u))]$$

function $getValidLA_{PT}(n) : States_{PT} \rightarrow \mathscr{P}(T_{PT})$

1. $validLA = validLA_{PT}(n)$
2. $validLA = validLA \cup \bigcup_{t \in validLA} \{u : t \prec u\}$
3. return $validLA$

Figure 4.2: Auxiliary function $getValidLA_{PT}$, embellished with global precedence relations.

Note that the aim of this global precedence relation (for example, $t \prec u$) is not so much to ensure that u is matched, but that t is *not* matched even when u would not be deemed valid. Thus, the *validLA* used to call the scanner must be modified to add those terminals that take precedence over some valid terminal.

This is done in the LR framework by means of modifying $getValidLA_{PT}$; Figure 4.2, showing the modified function, illustrates the required changes. The function $runScan_{PT}$ remains the same. Essentially, the embellished $getValidLA_{PT}$ adds to the original valid lookahead set (this is $validLA_{PT}(n)$ in Figure 4.2, but in the abstract is simply the set of terminals deemed valid in that context) all terminals that take precedence over the terminals in the set. For example, if $t \prec u$, $u \prec v$ and $u \prec w$, and $w \prec x$, a valid lookahead set of $\{t\}$ would be expanded to $\{t,u\}$, a valid lookahead set of $\{u\}$ would be expanded to $\{u,v,w\}$, and a valid lookahead set of $\{x\}$ would remain the same. Hence, for example, any lexeme matching both t and u would be sure to be matched to u even if only t is in the original valid lookahead set.

If *validLA* is expanded to accommodate terminals of higher precedence, it will potentially be expanded beyond the set $validLA_{PT}(n)$ of terminals with valid parse actions. This is the situation provided for in Definition 3.2.2 on page 79 by allowing $validLA \neq validLA_{PT}(n)$; additional requirements are then needed to verify that when scanning lexemes matching u but not t, the scanner behaves as expected.

Definition 4.2.3. *Context-aware scanner with lexical precedence — additional requirement for LR.*

> In the LR framework, if the function $scan_{PT}$ is called from a parse state n with $validLA \neq validLA_{PT}(n)$, then if a certain terminal t has no parse action, the scanner can return a token (t,x) only when t takes precedence over all those terminals with parse actions that match x:
>
> $$(t,x) = scan_{PT}(n, validLA, w) \wedge t \in validLA_{PT}(n) \Rightarrow$$
> $$\forall u \in T_{PT}.\,[x \in L(regex_{PT}(u)) \Rightarrow u \prec t].$$

4.3 Disambiguation functions.

Global precedence relations cannot handle situations where the terminal that should match a certain lexeme *does* vary with context, with terminal t matching lexeme x when t is in the valid lookahead set, and u matching when u is in the valid lookahead set.

An arrangement of this sort is handled automatically by a context-aware scanner so long as t and u never appear in the same valid lookahead set. But if t and u do appear in the same valid lookahead set, then some mechanism is needed to disambiguate between them *only* when matching lexemes $x \in L(regex(t)) \cap L(regex(u))$, letting the context disambiguate in other cases.

For example, in our grammar specification for AspectJ (see section 9.1), we have two identifier terminals: one for Java identifiers, one for AspectJ identifiers. The AspectJ keywords are reserved, by lexical precedence, against the AspectJ identifiers. But there are five keywords (`after`, `around`, `before`, `declare`, and `proceed`) that are also valid in the same context as Java identifiers. As these cannot be Java reserved keywords — this would prevent any Java identifiers from being named `after`, `around`, `before`, `declare`, or `proceed`.

4.3.1 Formal definitions.

We first define a formal construct, similar to Aycock and Horspool's "Schrödinger's token" [AH04], for carrying ambiguous terminals out of the scanner. Essentially, the parser generator will enumerate all unresolved ambiguities and, for each unresolved ambiguity, generate a new terminal, called a *Schrödinger terminal*, to represent the set of terminals that is the ambiguity. Then a post-process to the scanner, receiving a Schrödinger terminal, can disambiguate further. The Schrödinger terminal is a "virtual" terminal, hence does not have any precedences set on it and is not used in the scanner except to be returned when the ambiguity it represents is matched.

To return to the AspectJ example, for states where the keyword `after` is in the same valid lookahead set as the Java identifier *Id*, a Schrödinger terminal a will be generated for the ambiguity $\{\texttt{after}, Id\}$. When the scanner matches the string `after`, it will return a token $(a, \texttt{after})$, containing the Schrödinger terminal a, to the post-process to be disambiguated before control is returned to the parser.

Definition 4.3.1. *Schrödinger terminal, T^{schr}, ambiguity.*

> A *Schrödinger terminal* is a special terminal (call it a) used to represent a particular lexical ambiguity $A \subseteq T$, T being the set of terminals for a

particular grammar. Recall that a lexical ambiguity is a set of terminals whose regular expressions overlap, so that there are one or more strings matching all the terminals. The Schrödinger terminal is intended to be used when all the terminals in its ambiguity, and no others, are matched, *i.e.*, for strings in $\bigcap_{t \in A} L(regex(t)) \setminus \bigcup_{t \in T \setminus A} L(regex(t))$.

Let $T^{schr} \subset T$ represent the set of all such terminals for T, and $ambiguity : T^{schr} \to \mathscr{P}\left(T \setminus T^{schr}\right)$ represent the ambiguity represented by each Schrödinger terminal; in the above example, $ambiguity(a) = A$.

We next make a formal definition of the disambiguation function itself — a function built with respect to a certain lexical ambiguity (represented by a set of terminals) that takes in a lexeme and returns one of the terminals in the set, thus resolving the ambiguity. We also define the *disambiguation group*, a special case of the disambiguation function that does not take in the lexeme.

Definition 4.3.2. *Disambiguation function.*

A *disambiguation function* $df_A : \Sigma^\star \to A$ is a function intended to resolve a particular lexical ambiguity ($A \subseteq T$).

If $t = df_A(x)$, it is required that $x \in \bigcap_{t \in A} L(regex(t)) \setminus \bigcup_{t \in T \setminus A} L(regex(t))$ and $t \in A$ (*i.e.*, if the function takes a set of terminals A, along with a lexeme x, x is required to match exactly every terminal in A, and the terminal returned must be in A).

Definition 4.3.3. *Disambiguation group.*

A *disambiguation group* is a disambiguation function that does not take the lexeme into account in its disambiguation.

Returning again to the AspectJ example, to disambiguate those keywords, a disambiguation group is provided for each of the five ambiguities mentioned, which simply returns the keyword, allowing the Java identifier to be matched in contexts where the keyword is not in the valid lookahead set [Sch09].

It is also possible that a parser framework incorporates *parser attributes*, which are discussed further below. In this case, a disambiguation function may access and modify these parser attributes as a side effect.

4.3.2 Implementing disambiguation functions in a context-aware scanner.

We now, as in Definition 4.2.2, modify Definition 3.2.2 on page 79 by formally defining the minimal requirements for a context-aware scanner incorporating disambiguation functions.

Definition 4.3.4. *Context-aware scanner with disambiguation functions.*

> There will be a function $df : \mathscr{P}(T) \times \Sigma^\star \to T$ implementing all the disambiguation functions associated with the grammar for which T is a terminal set (*i.e.*, $df(A,x) = df_A(x)$). Each disambiguation function df_A will be represented in a context-aware scanner by a "virtual" Schrödinger terminal t_A^{schr}.
>
> If $(t,x) = scan_{ca}(n, validLA, w)$, and if the set of valid lookahead matching x is exactly the ambiguity resolved by df_A — *i.e.*, $x \in L(t_A^{schr})$ and $\{t : x \in L(t)\} \cap validLA = ambiguity(t_A^{schr})$ — then the scanner must return t_A^{schr}.

See Figure 4.3 on the next page for the embellished function $runScan_{PT}$ used when running a context-aware scanner with disambiguation functions in the LR framework;

function $runScan_{PT}(n, validLA, w)$: $States_{PT} \times \mathscr{P}(T_{PT}) \times \Sigma^\star \rightarrow ((T_{PT} \cup \{\bot\}) \times \Sigma^\star)$

1. $(t,x) = scan_{PT}(n, validLA, w)$
2. if $t \in T_{PT}^{schr}$ then
   (a) return $(df(ambiguity_{PT}(t),x),x)$
3. else
   (a) return (t,x)

Figure 4.3: Auxiliary function $runScan_{PT}$ (embellished to handle disambiguation functions).

$getValidLA_{PT}$ remains the same. $runScan_{PT}$ simply checks to see if the scanner has returned a Schrödinger terminal, and runs the proper disambiguation function if it has.

Note that the use of disambiguation functions partially invalidates the argument presented in section 3.3, that the results of scans can be memoized and reused until the scanned input is consumed. With an ambiguous match from the scanner DFA, memoization is not possible in a limited set of cases; this is discussed further in section 4.7.2.

4.3.3 Example application: the C typename/identifier ambiguity.

The canonical example of a situation where a disambiguation function is needed is the typename/identifier ambiguity in many LALR(1) grammars for ANSI C. In C, `[A-Za-z_$][A-Za-z0-9_$]*` is the regular expression for both typenames and identifiers. Furthermore, the restrictions of the LALR(1) framework require that these two language constructs have separate terminals (using the same terminal for both, as is done in Java, would cause parse table conflicts). Finally, both of them are valid in some contexts, so disambiguation by context alone cannot resolve the ambiguity.

In ANSI C, the traditional way to disambiguate between typenames and identifiers is to maintain a list of typenames declared by `typedef` and only match a typename if the lexeme is in this list; see Figure 9.3 on page 219 for pseudocode of the function used. It is not possible to use a disambiguation function for this if the function can only operate on the matched lexeme.

In order to allow a disambiguation function to take other information into account besides the matched lexeme, we provide the formalism of *parser attributes*. Parser attributes are essentially a formalization of custom fields or variables that can be accessed and modified from semantic actions on terminals and productions. They are specified as part of a grammar, at which point they are given some initial value; their values can also be read on the following other occasions:

- Any time a shift action is performed, parser attributes may be modified based on their values prior to the action being called and on the token being consumed.
- Any time a reduce action is performed, parser attributes may be modified based on their values prior to the action being called and on the data being popped off the parse stack.
- Disambiguation functions may modify parser attributes and use them when deciding what terminal to return.

A disambiguation function can do this if it has access to such a list stored in a parser attribute. See section 9.2 for an example.

4.4 Layout: grammar and per production.

Van den Brand *et al.* [vdBSVV02] define *layout* to be terminals that are defined in the lexical syntax of a language, but not in the context-free syntax: terminals whose

regular expressions match lexemes that are thrown away by the scanner, and never seen by the parser. Layout commonly seen in programming languages includes whitespace, comments, and annotations for documentation in the style of Java's *Javadoc* [Kra99].

In this section we describe adaptations of context-aware scanning to handle two different kinds of layout: grammar layout, which is a close functional match for the layout in traditional scanners, and layout per production, a more nuanced LR-specific approach with particular benefits for parsing embedded constructs.

4.4.1 How layout is handled traditionally.

In a traditional scanner, layout is handled in an implicit or *ad hoc* manner. The traditional scanning apparatus is generally comprised of a list of regular expressions, with each one followed by a block of code to be executed if it is matched. For each regular expression corresponding to a context-free terminal, then, this code will specify that a token be returned corresponding to that terminal. In the case of regular expressions corresponding to layout, however, the block of code will be empty, indicating that nothing is to be returned.

4.4.2 Grammar layout in context-aware scanners.

The traditional approach, as usually implemented, is unsuitable for a context-aware scanner, due to, among other things, the way in which the scanner is called: it is run in a particular place to scan for a particular set of terminals, so in order for the scanner to match any layout, it must explicitly be run with that layout as valid lookahead.

The simplest solution is to adapt the traditional approach so that a context-aware scanner can scan explicitly on a grammar-wide set of layout between each pair of non-layout terminals, as a traditional scanner does implicitly. Most applications do not

require any other sort of layout; we call this sort *grammar layout*.

Definition 4.4.1. *Grammar layout, T^{GL}.*

> With respect to a grammar with terminal set T and production set P, let a *grammar layout* terminal $\ell \in T$ be a terminal that is designated as layout, but that does not appear on the right-hand side of any production in P. Let the set $T^{GL} \subset T$ represent the set of terminals with this designation.

We must modify the requirements of a context-aware scanner to accommodate layout. With layout accommodated, when $scan_{ca}(validLA, w) = (t, x)$, then under limited conditions, x is not required to be a prefix of w. If x is not a prefix of w, then an unbroken series of grammar layout must precede the non-layout terminal t: there must exist a sequence of strings $\ell_1, \ell_2, \ldots, \ell_m$ such that $\forall i \in [1,m] \,.\, \left[\ell_i \in \bigcup_{t \in T_{PT}^{GL}} L(regex(t))\right]$ and $\ell_1 \ell_2 \cdots \ell_m x$ is a prefix of w.

function $runScan_{PT}(n, validLA, w)$: $States_{PT} \times \mathscr{P}(T_{PT}) \times \Sigma^\star \rightarrow ((T_{PT} \cup \{\bot\}) \times \Sigma^\star)$

1. do
   (a) $(t,x) = scan_{PT}(n, T^{GL}, w)$
   (b) if $t \neq \bot$ then remove the prefix x from the front of w

   while $t \neq \bot$ and $x \neq \varepsilon$
2. $(t,x) = scan_{PT}(n, validLA, w)$
3. return (t,x)

Figure 4.4: Auxiliary function $runScan_{PT}$, embellished to handle whole-grammar layout.

See Figure 4.4 for the embellished function $runScan_{PT}$ used when running a context-aware scanner with grammar layout in the LR framework. Essentially, the function

runs the scanner repeatedly using the set of grammar layout until no layout terminal can be matched, or until an empty string of layout is matched (the latter condition to prevent an infinite loop). After that, it makes the regular non-layout scan.

It is important to note that grammar layout in a context-aware scanner is not altogether identical to the layout of a traditional scanner. The primary point of difference concerns the empty string. In a traditional scanner, layout terminals are scanned for at the same time as non-layout terminals; if a layout terminal is matched it is ignored, and if a non-layout terminal is matched it is returned to the parser. On the other hand, in a context-aware scanner, separate scans, with different valid lookahead sets, are used to match layout and non-layout terminals. If there is no layout terminal in a given point in the input, the scan for layout must match the empty string before the scan for non-layout can take place.

This need to match layout introduces the possibility of *requiring* the appearance of layout between terminals; if one does not want to impose this requirement, the regular expressions of the layout terminals must be adjusted accordingly. For example, a common regular expression for whitespace layout in a traditional scanner is `[ ]+`. The regular expression matches one or more spaces and means, in that framework, that one could optionally have one or more spaces between non-layout terminals.

If this were to be used in the context-aware framework, it would *require* there to be at least one space between non-layout terminals. If one did not want this, one would instead use the regular expression `[ ]*`, zero or more spaces, enabling the empty string to be matched as layout and making the spaces optional.

4.4.3 About layout per production.

Grammar layout is context-insensitive, and while it is suitable for most applications, there are several for which it is not. An example is a grammar containing an embedded

language, where the whitespace or comments allowed in the embedded language is different from that of the "host" language in which it is embedded, such as with JavaScript or CSS within HTML documents and Java code within JSP scripts.

Another example is in the parsing of regular expressions as an integral part of the language's grammar; for example, the parsing of the scripting language Perl, which embeds regular expressions within its syntax to support string search-and-replace operations. This poses two lexical problems:

1. There is no whitespace allowed in regular expressions. This means that spaces within a regular expression will be ignored unless escaped; however, most standard languages for specifying regular expressions do not require escaped spaces.

2. Strings of alphanumeric characters in regular expressions are intended to be scanned one symbol at a time, instead of as one "identifier" terminal. However, with a traditional scanner, maximal munch means that the identifier terminal will take precedence; instead of a regular expression `abcde` being scanned as five separate character tokens `a`, `b`, `c`, `d`, `e`, it will be scanned as a single token (*Identifier*, `abcde`).

It is for this reason that regular expressions are often scanned as one string, like a string literal, and parsed by a secondary parser. Context-aware scanning automatically addresses the second of these issues; however, if grammar layout is used, the language's whitespace will be valid as layout even inside regular expressions and the first issue will remain unresolved.

Layout issues have been dealt with in the past by either tolerating the different layout or by using a different scanner and parser for each language, complete with separate grammar specifications. A context-aware scanner, however, can use valid lookahead sets to vary the valid layout from state to state.

4.4.3.1 Visser's solution.

The most useful pre-existing template on which to base a context-sensitive system of layout is Visser's system of scannerless parsing [Vis97, vdBSVV02]. A scannerless parser must incorporate layout into the grammar itself prior to its compilation into the scannerless parser; Visser does this by inserting an implicit layout nonterminal symbol between every two symbols on the right-hand side of a production; for example, the production $A \rightarrow \alpha\ \beta\ \gamma$ would become $A \rightarrow \alpha\ \mathit{LAYOUT}\ \beta\ \mathit{LAYOUT}\ \gamma$. However, there are three shortcomings in Visser's handling of layout:

- *Layout does not vary by production*. The *same* layout is implicitly inserted into all productions. However, this is a topical issue, easily resolved.
- *Nondeterminism is needed for disambiguation by precedence and associativity*. The GLR algorithm being the basis for Visser's framework, he uses a nondeterministic method of disambiguating. Thus, these methods of handling layout cannot be used unaltered in any deterministic framework.
- *Some superfluous reductions occur*. While the standard method removes a conflicting reduce action from the table, Visser's instead removes the goto actions that are taken immediately after such reductions are completed. This leaves a shift-reduce conflict in the table, which means that when the goto action has been removed, a superfluous reduction will be carried out (nondeterministically) and fail, wasting computation time.

4.4.3.2 Summary of Copper's solution.

Copper implements Visser's notion of placing layout between each two right hand side symbols in such a way that the layout is not visible to the parser.

Copper allows a set of layout to be specified for each production. When compiling the parser, the parser generator will keep track of which layout terminals are valid in each parse state, according to a set of rules spelled out precisely below. It will then include a valid layout set along with the valid lookahead set. At parser runtime, the scanner will run using this valid layout set before it runs using the valid lookahead set for the state.

Having different layout for each production is not necessary in most applications, but assigning layout to productions specifically, rather than to the grammar as a whole, allows for much more flexibility — including the ability to implement a variety of middle-ground approaches that, while not putting different layout on each production, vary it in different portions of the grammar.

The first example is grammar layout itself, which can be implemented as a special instance of layout per production, with the set of layout on each production defined to be the set of grammar layout. This also allows for a default set of layout that can be assigned to productions unless another set is explicitly specified. Another example is the *extension grammars* defined in Definition 6.1.1 on page 148. These are extensions to a given grammar (*e.g.*, the abovementioned example of embedded languages) that may require a different set of layout. In such a case, the productions of the extension grammar will have this different layout set.

4.4.4 Formal specification of layout per production.

In this section we provide precise rules for the placement of layout. Note that these rules are LR-specific.

The rules will ensure that each production's layout is placed in the valid layout set of any parse state in which a parser is in the middle of parsing a construct derived from that production (for example, if a whitespace terminal *ws* is specified as the layout of

production $E \rightarrow E + E$, *ws* will be valid layout in any state containing one or both of the items $E \rightarrow E \bullet + E$ and $E \rightarrow E + \bullet E$), and that the layout is properly mapped to the terminals that may follow it (a slightly more complicated process).

We first must make some definitions of nomenclature. Layout per production is specified as a set of terminals attached to a production; we will represent this by the function $layout : P \rightarrow \mathscr{P}(T)$ shall be a function mapping productions to their defined layout tokens.

The final outcome of the process of layout placement will be a *layout map*, which maps various layout terminals to the non-layout terminals that may follow them in a state. We will represent these by the function $layoutMap : States \times T \rightarrow \mathscr{P}(T)$.

We will also need an inverse mapping from non-layout to layout terminals. This we will represent by the function $layoutBefore : States \times T \rightarrow \mathscr{P}(T)$, defined formally with $layoutBefore(n,t) = \{\ell : t \in layoutMap(n,\ell)\}$.

We may now define the rules for building *layoutMap* given *layout*, which parallels the construction process of the LR DFA.

Definition 4.4.2. *ClosureDerives.*

$[A \rightarrow \alpha \bullet B\gamma,\ z_1] \vdash_C [B \rightarrow \bullet\beta,\ z_2]$ if $\{A,B\} \subseteq NT$, $\{\alpha,\beta,\gamma\} \subseteq (T \cup NT)^\star$, $\{z_1,z_2\} \subseteq \mathscr{P}(T)$ (lookahead sets), and $\bigcup_{t \in z_1} first(\gamma t) \subseteq z_2$.

This encapsulates the standard "closure" rule for constructing LR DFA states. $i_1 \vdash_C i_2$ means that if item i_1 is in a state, item i_2 will also be in that state, being derived from the closure.

Definition 4.4.3. *GotoDerives.*

$[A \rightarrow \alpha \bullet \beta\gamma,\ z_1] \vdash_G [A \rightarrow \alpha\beta \bullet \gamma,\ z_2]$ if $A \in NT$, $\beta \in (T \cup NT)$, $\{\alpha,\gamma\} \subseteq (T \cup NT)^\star$, and $\{z_1,z_2\} \subseteq \mathscr{P}(T)$.

This is the standard "goto" rule for constructing LR DFA states. $i_1 \vdash_G i_2$ means that if item i_1 is in a state, then item i_2 will be in another state reachable from that state over a transition β.

Definition 4.4.4. *Prod.*

$prod([A \rightarrow \alpha \bullet \beta,\ z_1]) = A \rightarrow \alpha\beta$ if $A \in NT$, $\{\alpha, \beta\} \subseteq (T \cup NT)^\star$, and $z_1 \subseteq T$.

This is a straightforward mapping of items to their productions.

Definition 4.4.5. *Beginning and Reducible.*

$beginning([A \rightarrow \bullet\alpha,\ z_1])$ and $reducible([A \rightarrow \alpha\bullet,\ z_1])$ if $A \in NT$, $\alpha \in (T \cup NT)^\star$, and $z_1 \subseteq T$.

These two rules identify items *i* with the bullet at the beginning of the right hand side (meaning that there is some item i' in the state with $i' \vdash_C i$) and at the end (indicating that the parser has finished parsing it and a reduction may be performed on it).

Definition 4.4.6. *Encapsulated.*

$encapsulated(I)$ if $\neg beginning(I)$ and $\neg reducible(I)$.

The opposite of *Beginning* and *Reducible*: signifies those items with the bullet in the middle, between two right hand side symbols.

Definition 4.4.7. *Beginning layout.*

Beginning layout refers to that layout that can validly appear at the *beginning* of a construct derived from a certain production: for example, an item $A \rightarrow \bullet\alpha$ would have for its beginning layout the set of layout that can precede a construct derivable from A in the relevant parse state. The map

$beginningLayout : States \times Items \rightarrow \mathscr{P}(T)$ holds this layout set for each item in each state.

Definition 4.4.8. *BeginningStart.*

$beginningLayout(n, [\hat{} \rightarrow \bullet S\$,\ ?]) = layout(\hat{} \rightarrow S\$)$ if $n \in States$ and $[\hat{} \rightarrow \bullet S\$,\ ?] \in items(n)$.

This defines the layout that can occur at the beginning of the parser's input: the layout assigned to the start production $\hat{} \rightarrow S\$$. In Copper, since this production is implicit, this layout set is specified on the start symbol.

Definition 4.4.9. *BeginningBase.*

$layout(prod(I_1)) \subseteq beginningLayout(n, I_2)$ if $n \in States$, $\{I_1, I_2\} \subseteq items(n)$, $I_1 \vdash_C I_2$, and $encapsulated(I_1)$.

This rule stipulates that if an item i_1 with its bullet in the middle (for example, $A \rightarrow \alpha \bullet B\gamma$) closure-derives another item i_2 (for example, $B \rightarrow \bullet\beta$), then the layout of $prod(i_1)$ can appear before the start of the expression derived from B (for example, α and β).

Definition 4.4.10. *BeginningInheritance.*

$beginningLayout(n, I_1) \subseteq beginningLayout(n, I_2)$ if $n \in States$, $\{I_1, I_2\} \subseteq items(n)$, $I_1 \vdash_C I_2$, and $beginning(I_1)$.

This is a propagation rule, carrying beginning layout down a chain of closure-derivations until it reaches a production with a terminal as the first symbol on the right hand side: for example, if there is a series of items $A \rightarrow \bullet B$, $B \rightarrow \bullet C$, and $C \rightarrow \bullet\gamma$, any layout that can occur at the beginning of an expression derived from A in that state can by construction also occur at the beginning of an expression derived from B or C.

Definition 4.4.11. *BeginningGoto.*

> $beginningLayout(n_1, I_1) \subseteq beginningLayout(n_2, I_2)$ if $\{n_1, n_2\} \subseteq \mathscr{P}(States)$, $I_1 \vdash_G I_2$ and the two items are of the required form for this, and $\delta(n_1, \beta) = n_2$.

Beginning layout is preserved across state transitions: $A \rightarrow \bullet\alpha$ has the same beginning layout as $A \rightarrow \alpha\bullet$.

Definition 4.4.12. *Lookahead layout.*

> *Lookahead layout* refers to that layout that can validly appear at the *end* of an expression derived from a certain production; *i.e.*, it would appear before the lookahead telling the parser to reduce on that production.
>
> To represent this layout we define the mapping $lookaheadLayout : States \times Items \times T \rightarrow \mathscr{P}(T)$, mapping triples of states, items and layout terminals to the sets of lookahead symbols that can follow the layout terminals. It is important to note that the lookahead layout is part of the *domain* rather than the range of this mapping.

Definition 4.4.13. *LookaheadInside.*

> $first(\gamma) \subseteq lookaheadLayout(n, I_2, \ell)$ if $\{\alpha, \gamma\} \subseteq (T \cup NT)^\star$, $\beta \in (T \cup NT)$, $n \in States$, $I_1 \vdash_C I_2$ and the two items are of the required form for this, and $\ell \in layout(prod(I_1))$.

This indicates that all items closure-derived from I_1 (*i.e.*, indicating that the parser is beginning the parse of an expression derived from β), which will have the first set of γ for lookahead, may have the layout of I_1 preceding said lookahead.

Definition 4.4.14. *LookaheadEnd.*

$t \in lookaheadLayout(n, I_2, \ell)$ if $\{\alpha, \beta\} \subseteq (T \cup NT)^\star$, $n \in States$, $I_1 \vdash_C I_2$ and the two items are of the required form for this, $\gamma = \varepsilon$, $\beta \in nullable^\star$ (every symbol in β is nullable), and $t \in lookaheadLayout(n, I_1, \ell)$.

This rule is analogous to *LookaheadInside*, but it passes along layout on the lookahead of I_1 in the event that said lookahead is also passed along on account of nullability.

Definition 4.4.15. *LookaheadGoto.*

$lookaheadLayout(n_1, I_1, \ell) \subseteq lookaheadLayout(n_2, I_2, \ell)$ if $I_1 \vdash_G I_2$ and the two items are of the required form for this, $\{n_1, n_2\} \subseteq \mathscr{P}(States)$ $I_1 \in items(n_1)$, $I_2 \in items(n_2)$, $\ell \in T$, and $\delta(n_1, \beta) = n_2$.

This passes layout along with lookahead across a goto derivation.

Definition 4.4.16. *ReducibleTableLayout.*

$t \in layoutMap(n, \ell)$ if $\{\alpha, \beta\} \in (T \cup NT)^\star$, $n \in States$, $I = [A \to \alpha \bullet \beta, z_1] \in items(n)$, $\beta \in nullable^\star$, and $t \in lookaheadLayout(n, I, \ell)$.

If layout ℓ maps to terminal t in the lookahead layout, it does so in the final layout map as well.

Definition 4.4.17. *BeginningTableLayout.*

$\alpha \in layoutMap(n, \ell)$ if $\alpha \in T$, $\beta \in (T \cup NT)^\star$, $n \in States$, $I = [A \to \bullet \alpha\beta, z_1] \in items(n)$, $z_1 \subseteq T$, and $\ell \in beginningLayout(n, I)$.

If layout ℓ is in the beginning layout of an item, it is mapped to any terminals in the first set of the corresponding production in the layout map.

Definition 4.4.18. *EncapsulatedTableLayout.*

$t \in layoutMap(n, I, \ell)$ if $A \in NT$, $\alpha \in (T \cup NT)^+$, $\beta \in T$, $\gamma \in (T \cup NT)^\star$,
$n \in States$, $I = [A \rightarrow \alpha \bullet t\gamma,\ z_1] \in items(n)$, $z_1 \subseteq T$, $\ell \in layout(prod(I))$.

A production's layout is mapped to all terminals in the first set of one of the symbols on its right hand side, in the context where such appear. (Note that this rule covers all productions, not just those with a terminal in front of the bullet, since if there is an item $A \rightarrow \alpha \bullet B\gamma$ in a state, the *Closure* rule provides that for every $t \in first(B)$, there will be a corresponding item in the state with the bullet before t.

4.4.5 Implementation of layout per production.

function $runScan_{PT}(n, validLA, w)$: $States_{PT} \times \mathscr{P}(T_{PT}) \times \Sigma^\star \rightarrow ((T_{PT} \cup \{\bot\}) \times \Sigma^\star)$

1. $validLayout = \bigcup_{t \in validLA_{PT}(n)} layoutBefore(t)$
2. do
   (a) $(t, x) = scan_{PT}(n, validLayout, w)$
   (b) if $t \neq \bot$ then remove the prefix x from the front of w

   while $t \neq \bot$ and $x \neq \varepsilon$
3. $(t, x) = scan_{PT}(n, validLA, w)$
4. return (t, x)

Figure 4.5: Auxiliary function $runScan_{PT}$, embellished to handle layout per production.

See Figure 4.5 for the embellished function $runScan_{PT}$ used when running a context-aware scanner with layout per production. As can be seen, the layout map built by the specifications in the previous section is a part of the parser; the alterations to the functions themselves are minimal, only substituting operations on the layout map for the set T^{GL}.

4.5 Transparent prefixes.

Disambiguation functions are intended to handle lexical ambiguities that cannot be resolved via the use of lexical precedence. Similarly, *transparent prefixes* are intended as a stopgap to handle certain ambiguities that cannot be resolved by disambiguation functions.

In Copper any terminal is allowed to specify another terminal as its transparent prefix. If Copper's scanner encounters a transparent prefix t, it will throw it away and scan again with the valid lookahead set reduced to only those terminals that have specified t as their transparent prefix.

Transparent prefixes must have no lexical conflicts with non-transparent-prefix terminals that are valid at the same location. The best way to ensure this is to have the transparent prefix begin with a unique character not allowed at the beginning of these other terminals.

The application for which transparent prefixes were specifically intended was in resolving the conflicts between the modular language extensions that were mentioned in chapter 1 and are discussed further in chapter 9. We have devised a modular analysis, set out in chapter 6, that allows the writers of such extensions to guarantee that when composed with other like extensions the resulting parser will be free of parse table conflicts. One of the restrictions imposed by this analysis is that each extension construct begin with a unique *marking terminal* (see Definition 6.1.1 on page 148). The analysis, besides guaranteeing a lack of parse table conflicts in the composed parser, also guarantees that the composition will not give rise to any new lexical conflicts — except for conflicts involving one or more of these marking terminals. The solution is to provide each extension with a unique name, akin to a Java package name, and then to use this name as a transparent prefix to the marking terminal in question.

An example, already alluded to, is that of the two `table` keywords in our extensible version of Java, ableJ, example uses of which can both be seen in Figure 1.1. The first keyword, belonging to the extension for embedding SQL in Java, is used on lines 7 and 10 in that figure and is not a marking terminal (it occurs in the middle of an extension construct). The second keyword, belonging to the extension for simplified expression of Boolean tables, is used on line 25 and is that extension's marking terminal, used to begin the extension construct.

Now, since the SQL extension does not embed Java expressions in the context where the SQL `table` keyword occurs, these two keywords are never in the same valid lookahead set. However, if — as is very possible — Java expressions were able to be embedded at the same context as the type of SQL table declaration beginning on line 7 in the figure, this would cause a conflict between the two keywords.

The solution employed would be to give the keywords each their own transparent prefix. The SQL extension has the grammar name `edu:umn:cs:melt:ableJ14:exts:sql`, while the tables extension has the grammar name `edu:umn:cs:melt:ableJ14:exts:tables`. Hence, in a context where either keyword could match, instead of merely using `table`, one would use `:edu:umn:cs:melt:ableJ14:exts:sql:table` to indicate the SQL `table` keyword, and `:edu:umn:cs:melt:ableJ14:exts:tables:table` to indicate the other keyword.

Note that the same effect could be obtained by altering the grammar: if transparent prefix p was assigned to terminal t, the grammar could be modified so that each production $A \rightarrow \alpha\ t\ \beta$ is altered to two, $A \rightarrow \alpha\ t\ \beta \mid \alpha\ p\ t\ \beta$. Although this would avoid an alteration of the formalism, it would potentially cause significant increases both in the number of productions in the grammar, and in the size of the compiled parser. On the other hand, the scanner-based approach outlined in Figure 4.6 on the next page incurs no penalty of time beyond that needed to scan the transparent prefixes, and no space

penalty beyond the extra states needed for the scanner to recognize the transparent prefixes.

4.5.1 Formal specification of transparent prefixes.

We now give formal definitions of the transparent prefix mappings. Let $prefix : T \rightarrow (T \cup \{\bot\})$ be a mapping such that if $p = prefix(t)$, p is t's transparent prefix, and if $prefix(t) = \bot$, then t has no transparent prefix; then let $followsPrefix : T \rightarrow \mathscr{P}(T)$ be the inverse mapping: $followsPrefix(p) = \{t : prefix(p) = t\}$.

See Figure 4.6 for the embellished function $runScan_{PT}$ used when running a context-aware scanner with transparent prefixes in the LR framework. This function will, before running the scanner on the valid lookahead set itself, run it on the set of transparent prefixes pertaining to terminals in the valid lookahead set. If a transparent prefix p is matched, it will restrict the valid lookahead set to $followsPrefix(p)$, thereby resolving some lexical ambiguities.

function $runScan_{PT}(n, validLA, w) : States_{PT} \times \mathscr{P}(T_{PT}) \times \Sigma^\star \rightarrow ((T_{PT} \cup \{\bot\}) \times \Sigma^\star)$

1. $validPrefixes = \bigcup_{t \in validLA_{PT}(n)} prefix(t)$
2. $(t, x) = scan_{PT}(n, validPrefixes, w)$
3. if $t \neq \bot$ then
   (a) remove the prefix x from the front of w
   (b) $validLA = followsPrefix(t)$
4. $(t, x) = scan_{PT}(n, validLA, w)$
5. return (t, x)

Figure 4.6: Auxiliary function $runScan_{PT}$, embellished to handle transparent prefixes.

4.6 Disambiguation by precedence and associativity.

Copper uses the traditional system of disambiguation by precedence and associativity, with no modification. Those familiar with this system may skip ahead to section 4.7.

4.6.1 Introduction.

Disambiguation by operator precedence and associativity is a staple modification to LALR(1) parser generators. We will sometimes call such precedence *operator precedence* to distinguish it from the *lexical precedence* discussed in section 4.2.

It is primarily used to enable the implementation of binary operations according to certain rules of associativity and an order of operations (*e.g.*, the standard order of parentheses, exponents, multiplication, division, addition and subtraction) without having to employ a hierarchy of nonterminals.

For example, to implement the standard order without disambiguation by precedence and associativity, the following productions would be used:

- $E \rightarrow E + T \quad | \quad E - T \quad | \quad T$
- $T \rightarrow T \star F \quad | \quad T \div F \quad | \quad F$
- $F \rightarrow F \wedge B \quad | \quad B$
- $B \rightarrow \mathit{operand} \quad | \quad (\, E \,)$

However, if the operators are explicitly assigned the intended precedence and associativity, all the operations could be defined on a single nonterminal, simplifying the syntax of a language:

- $E \rightarrow E + E \quad | \quad E - E \quad | \quad E \star E \quad | \quad E \div E \quad | \quad E \wedge E \quad | \quad (\, E \,) \quad | \quad \mathit{operand}$

We will now formally define operator precedence, operator associativity, and production precedence. The "operator" assigned to a certain production is defined as the rightmost terminal on the right hand side of that production: for instance, in the production $E \to E - E + E$, the operator is $+$. Many parser generators, including Copper, include an option to substitute a custom operator in the place of the rightmost terminal. We will represent the assignment of operators to productions by the map $op : P \to T$.

Operators are each given a number signifying their *precedence*. More than one operator may have the same number. These precedences are represented by the mapping $oprec : T \to \mathbb{N}$. They are also given an *associativity*, which can be *left*, *right*, *nonassociative*, or *none*. This is represented by the map $assoc : T \to \{Left, Right, Nonassoc, None\}$.

Productions are also given a precedence number in their own right represented by $pprec : P \to \mathbb{N}$. This is done when one production derives a suffix of another and there is confusion as to which production should be reduced upon. For example, if there were two productions $A \to a\ x$ and $B \to x$, and if the parser had just consumed a and x, it might be unclear at that point whether x should be derived from A or from B. One can set production precedence to specify this: $pprec(A \to a\ x) > pprec(B \to x)$ entails that it should be derived from A, vice versa for B.

4.6.2 Disambiguation process.

In section 3.2, we defined shift-reduce and reduce-reduce conflicts. There are no shift-shift conflicts — the LR DFA construction process never places two shift actions in the same state — so any cell with a parse table conflict will have at most one shift action and at least one reduce action. Here, we define precisely the processes commonly used for resolving such conflicts by precedence and associativity.

If for parse table cell (n,t), $\pi(n,t) = \{shift(n'), reduce(p_1), \ldots, reduce(p_n)\}$:

- All reduce actions on productions of lower precedence — *i.e.*, actions $reduce(p_i)$ such that $pprec(p_i) \neq \max_{j=1}^{n} pprec(p_j)$ — are removed. If this leaves more than one reduce action remaining, the conflict cannot be resolved. Afterwards, $\pi(n,t) = \{shift(n'), reduce(p)\}$.

- If $t \neq op(p)$:

  - if $oprec(t) = \perp$ or $oprec(t) = oprec(op(p))$, the conflict cannot be resolved.
  - If $oprec(t) < oprec(op(p))$, the shift action is removed.
  - If $oprec(t) > oprec(op(p))$, the reduce action is removed.

- If $t = op(p)$:

  - If $assoc(t) = Left$, the shift action is removed.
  - If $assoc(t) = Right$, the reduce action is removed.
  - If $assoc(t) = Nonassoc$, both actions are removed, leaving an error action.
  - If $assoc(t) = None$, the conflict cannot be resolved.

4.7 Concluding discussion.

In this section we discuss some minor issues with using more than one of the above modifications in conjunction (section 4.7.1), as well as the issues of memoizing scanner results to prevent repeated scans in the same input position (section 4.7.2), making sure all lexical ambiguities in a context-aware scanner are detected (section 4.7.3), and dealing with errors in scanning when using a context-aware scanner (section 4.7.4).

4.7.1 Putting all the modifications together.

Hitherto the practical modifications have been discussed in isolation, and are best considered in that light for purposes of simplicity. However, there are a few minor points that must be addressed when combining them.

- Disambiguation functions cannot be used to disambiguate layout or transparent prefixes. Lexical precedence relations between layout terminals or between layout and non-layout terminals function identically to relations between non-layout terminals.

- When transparent prefixes are implemented alongside layout per production, the layout is scanned for first, the transparent prefix second (*i.e.*, there can be no layout between a transparent prefix and the terminal it prefixes).

See Figure 4.7 on the next page for the embellished functions $getValidLA_{PT}$ and $runScan_{PT}$ incorporating all the modifications (global precedence, disambiguation functions, layout per production, and transparent prefixes). Only precedence requires a modification of $getValidLA_{PT}$, while in $runScan_{PT}$ it is essentially a matter of executing all the individual modifications in order — first scanning for layout, then scanning for a prefix, then scanning on the actual valid lookahead set, then performing the post-process disambiguation by disambiguation function.

4.7.2 Memoizing scanner results.

We must also consider the problem discussed in section 3.3 of memoizing scanner results to avoid repeating scans after reduce actions. We stated that if it could be assumed that for each valid lookahead set, the regular expressions of all terminals in it were disjoint, this memoization could be safely performed. Two of the modifications presented

- function $getValidLA_{PT}(n) : States_{PT} \rightarrow \mathscr{P}(T_{PT})$

  1. $validLA = validLA_{PT}(n)$
  2. $validLA = validLA \cup \bigcup_{t \in validLA} \{u : t \prec u\}$
  3. return $validLA$

- function $runScan_{PT}(n, validLA, w) : States_{PT} \times \mathscr{P}(T_{PT}) \times \Sigma^{\star} \rightarrow ((T_{PT} \cup \{\bot\}) \times \Sigma^{\star})$

  1. $validLayout = \bigcup_{t \in validLA_{PT}(n)} layoutBefore(t)$
  2. do
     (a) $(t, x) = scan_{PT}(n, validLayout, w)$
     (b) if $t \neq \bot$ then remove the prefix x from the front of w

     while $t \neq \bot$ and $x \neq \varepsilon$
  3. $validPrefixes = \bigcup_{t \in validLA_{PT}(n)} prefix(t)$
  4. $(t, x) = scan_{PT}(n, validPrefixes, w)$
  5. if $t \neq \bot$ then
     (a) remove the prefix x from the front of w
     (b) $validLA = followsPrefix(t)$
  6. $(t, x) = scan_{PT}(n, validLA, w)$
  7. if $t \in T_{PT}^{schr}$ then
     (a) return $(df(ambiguity_{PT}(t), x), x)$
  8. else
     (a) return (t, x)

Figure 4.7: Functions $getValidLA_{PT}$ and $runScan_{PT}$ embellished to handle global precedence, disambiguation functions, layout per production, and transparent prefixes.

in this chapter — lexical precedence relations and disambiguation functions — allow no such assumption, and indeed rescanning is required in a small number of cases.

If only lexical precedence is used for disambiguation, all scans may still be safely memoized. Suppose that in a certain state n, the token (t,x) was matched during the scan. Then, the only way that lexical precedence could cause a different token to be matched at the same input location in a different state n' is if there was some terminal u also matching x with $t \prec u$. But if this were the case, t could not have been matched in the first place.

On the other hand, when using disambiguation functions, this sort of problem can occur occasionally. Specifically, if a token (t,x) is matched in a state n, the scanner must be re-run from the same position in state n' if the following four conditions are met:

1. There is another terminal u such that $x \in L(regex_{PT}(u))$.
2. $u \in validLA_{PT}(n') \setminus validLA_{PT}(n)$.
3. There is a disambiguation function df_A with $t,u \in A$.
4. $df_A(x) \neq t$.

The first condition is impossible to test without running the scanner, but the second and third can be easily tested, and the fourth can be tested at least on disambiguation groups. Then the scanner can be set to re-run if all the tested conditions are met.

4.7.3 Detection of lexical ambiguities.

The traditional use of the strict total order on regular expressions provides an implicit guarantee that no lexical ambiguities will be present in the finished scanner, *i.e.*, the

scanner will always be able, when called, to return one token that is the intended match at that point. Permitting there to be no precedence relation between certain terminals introduces the possibility of such ambiguities.

We have, however, a method to verify and make an *explicit* guarantee that there are no such ambiguities. This is done by, for each parse state n, checking the scanner to see that when it is run on the valid lookahead set for n, no matter what the input, no ambiguities may occur. This is a different method for each of the two implementations of a context-aware scanner detailed in chapter 5 on the following page, but both are essentially the same process. For each state k of the scanner DFA, the set of terminals accepted in k will be intersected with the valid lookahead set from the parse state n. If each such intersection is of cardinality no greater than 1, there are no ambiguities. But if the size of some intersection is greater than 1, this means that the scanner returns an ambiguous match, and the verifier will alert the grammar writer so that the ambiguity can be resolved, either by modifying the lexical specification or by using a disambiguation function.

4.7.4 Reporting scanning errors.

When a syntax error occurs, the parser's user expects to see an error message of the form "Expected a token of type `X`; instead matched token of type `Y` with lexeme `z`." With a context-aware scanner, however, if `Y` is not in the valid lookahead set, the scanner will simply fail scanning `z`, and not only will there be no way of knowing that the current input matches `Y`, but there will be no way of knowing that `z` is the lexeme in that case.

The solution is, if a scan fails, to run the scanner over the same input on the union of all the parser's valid lookahead sets: $validLA = \bigcup_{n \in States_{PT}} getValidLA_{PT}(n)$, the $getValidLA_{PT}$ function being that from Figure 4.2 on page 97. Then the scanner can report a list of the terminals matched, vis-a-vis the valid lookahead set $getValidLA_{PT}(n)$.

Chapter 5

Context-aware scanner implementations.

In this chapter we discuss two approaches to implementing a context-aware scanner, both employed within Copper; we also discuss optimizations to these approaches. One approach (discussed in section 5.2) performs disambiguation by context at scanner runtime, building a single DFA for the entire scanner — the same DFA as is built by a disjoint scanner — and annotating it with extra information necessary to perform the disambiguation by context. Another approach (discussed in section 5.3) is to perform the disambiguation by context at scanner compile time by building multiple DFAs — a different DFA for each valid lookahead set — which are traditional DFAs that run according to the traditional algorithm, with no annotations or runtime disambiguation by context.

The multiple-DFA approach is slightly faster than the single-DFA approach, but the single-DFA approach is more compact. At present, the multiple DFAs take up enough space as to be impractical on very large grammars; a direction for future work in this field is to devise optimizations to make the space requirements of the multiple-DFA

approach comparable to that of the single-DFA approach. This is discussed further in sections 5.3.3 and 10.2.1.

5.1 Background and preliminary definitions.

We first make precise definitions of the scanner DFA(s) used by a context-aware scanner and the process of constructing them. The definition differs from the definition of a DFA from section 2.2.2, in that it fails to include a set of final states; this is explained further below.

Definition 5.1.1. *Scanner DFA.*

> A *scanner DFA* M defined over an alphabet Σ consists of a triple $\langle Q_M, s_M, \delta_M \rangle$, where Q_M is the set of the DFA's states, $s_M \in Q_M$ is the DFA's *start state*, and $\delta_M : Q_M \times \Sigma \to Q_M$ is the DFA's transition function.

This definition is different from the standard definitions of DFAs discussed in section 2.2.2, in that it fails also to include a set of *final states*, or accept states, indicating that if the DFA finishes scanning in that state, the input string matches the DFA's corresponding regular expression. This is because the scanner DFA requires more sophisticated annotations to determine what regular expression or terminal is being matched. These annotations (*viz.*, accept, possible, and reject sets) are discussed further below.

The construction of a scanner DFA is defined on a set of terminals T with a corresponding mapping *regex* to a set of regular expressions R. T could be the full set of terminals from a grammar and *regex* the regular expression mapping from that grammar, or they could be subsets of those respective grammar components. The construction process — which is mostly identical to the traditional approach, except that it does not build the usual set of final states — is as follows:

1. For each $t \in T$, with $regex(t) = r$, build a nondeterministic finite automaton $M_t^n = \langle Q_t^n, F_t^n, s_t^n, \delta_t^n \rangle$ from r in the traditional manner.

2. Combine these NFAs into a single NFA M_T^n, again in the traditional manner, by taking the union of all their states and transitions and adding a new start state s_Γ with ε-transitions to every sub-NFA's start state, *i.e.*,

$$M_T^n = \left\langle Q_T^n = \bigcup_{r \in R} Q_t^n \cup \{s_\Gamma\}, \emptyset, s_T, \delta_T^n \ : \ Q_T^n \times \Sigma \cup \{\varepsilon\} \to \mathscr{P}(Q_T^n) \right\rangle$$

   where

$$\delta_T^n(n, \sigma) = \begin{cases} \delta_t^n(n, \sigma) & \text{if } n \in Q_t^n \\ \bigcup_{r \in R} s_t^n & \text{if } (n, \sigma) = (s_\Gamma, \varepsilon) \\ \emptyset & \text{otherwise} \end{cases}$$

3. Convert the NFA into a scanner DFA $M_T = \langle Q_T, s_T, \delta_T \rangle$, again in the traditional manner.

   Usually this process would also include a set of final states. But it will instead retain a mapping of what states in the original NFA M_T^n are "constituents" of the DFA states.

For a scanner DFA M_T built from an NFA in this manner, the mapping $constit_T : Q_T \to \mathscr{P}(Q_T^n)$ contains, for every state of M_T, the set of states in M_T^n that were incorporated into that state during M_T's construction.

We now define the adaptation of the final state set for context-aware scanners: the *accept set* assigned to each DFA state. This is essentially the set of terminals that could be validly matched to any string that will take the DFA to that state.

Definition 5.1.2. *Accept set, acc.*

In the context of a grammar Γ and a set of terminals $T \subseteq T_\Gamma$, let $R = \bigcup_{t \in T} \{regex(t)\}$ and M_T be a scanner DFA constructed from T by the method described above. An *accept set* for a state $q \in Q_T$ consists of the terminals that are matched if the scanner finishes scanning in state q. Let the mapping $acc_T : Q_T \to \mathscr{P}(T)$ represent the accept sets for all the states of the DFA.

A terminal is in the accept set of a state iff that state includes among its constituents a final state in the original NFA of the terminal's regular expression: $acc_T(q) = \left\{ t : F^n_{regex(t)} \cap constit(q) \neq \emptyset \right\}$. Note that this is roughly identical to the method used by traditional scanners.

To demonstrate the concept of an accept set in a more intuitive manner, we will take an example. Suppose a scanner is built for terminals i, k_1, and k_2, i matching the regular expression `[a-z]+`, k_1 matching `for`, k_2 matching `form`. The DFA in Figure 5.1 on the following page is built from these regular expressions using one of our approaches, the single-DFA approach (discussed in the next section). The start state q_0 in that DFA, being reached only when the empty string ε has been scanned, is not in any sets of final states. Hence, $acc_T(q_0) = \emptyset$. On the other hand, state q_4 is reached after scanning the string `form`. This matches the regular expressions of k_2 and i, so $acc(q_4) = \{i, k_2\}$.

5.2 Single DFA approach.

The first implementation approach we discuss is to compile the regular expressions of all terminals into one large DFA and annotate it with some metadata — three sets of terminals for each state — to accommodate scanning with numerous valid lookahead sets. This can be implemented with bit-vectors, so if T is the set of terminals for the grammar, this approach takes up $3 \cdot |T| \cdot |Q_T|$ bits more than the traditional approach.

However, computations on the terminal sets increase the time complexity from $O(n)$ to $O(n \cdot \lg|T|)$.

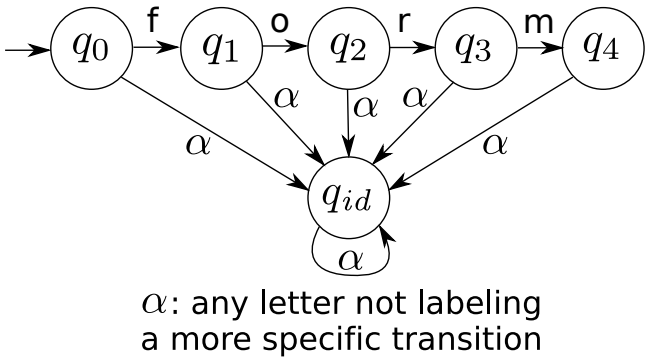

Figure 5.1: DFA for operation example of single-DFA approach.

See Figure 5.1 for the DFA that would be built from the lexical syntax enumerated above on the preceding page, using the single-DFA approach.

5.2.1 Possible sets.

Since a scanner DFA scans according to the principle of maximal munch, a scanner DFA must know when it has matched the longest possible match: know when it reaches a state from which no sequence of transitions leads to a state accepting any proper terminal, so it can stop scanning and return the longest match.

The traditional DFA, which always scans on the same set of terminals, has a fairly simple way of guaranteeing this, *viz.*, if it reaches a state with no transition marked for the next character in the input string, it is time to stop.

However, a DFA for a context-aware scanner may have to scan for a valid lookahead set that is a *subset* of the terminal set T for which the DFA was built. In order to ensure that a context-aware scanner stops scanning when it has matched the longest possible match in the given valid lookahead set, the DFA maintains a *possible set* enumerating

the terminals that might still be matched. The context-aware scanner will then intersect the possible set with the valid lookahead set to determine if there are any further matches possible.

Before defining possible sets, we must first define the transitive closure $\delta_T^\star$ of a DFA M_T's transition function δ_T. This is just, as the name suggests, the set of all states reachable by transition in M_T from the state that is the function's argument:

$$\delta_T^\star(q) = \left\{ q' \ : \ \begin{array}{r} \exists q_0, \ldots, q_n \in Q_T. \\ \exists \sigma_1, \ldots, \sigma_n \in \Sigma. \end{array} \ \left[\bigwedge_{i=1}^{n} (\delta_T(q_{i-1}, \sigma_i) = q_i) \wedge q = q_0 \wedge q' = q_n \right] \right\}$$

We now define possible sets, which are essentially the transitive closure of the accept sets; if there is a transition path (of zero or more transitions) from the current state q to a state matching terminal t, then t will be in the possible set of q.

Definition 5.2.1. *Possible set, poss.*

> In a scanner DFA M_T, the *possible set* for a state $q \in Q_T$ is the union of the accept sets of all states in the transitive closure of q. Let $poss_T : Q_T \to \mathscr{P}(T)$ represent the possible sets for all states of the DFA. Formally, $poss_T(q) = \bigcup_{q' \in \delta_T^\star(q)} acc(q')$.

The possible set of the start state will always be the full set of terminals T from which the scanner was built, as all states in the DFA are reachable by transition from the start state.

To return to the example on page 129 and in Figure 5.1, take state q_2. It is reached after scanning the string `fo`; hence, it accepts i but not k_1 or k_2, and $acc_T(q_2) = \{i\}$. The transitive closure of q_2, as can be seen, contains four states: q_2 itself, q_3, q_4, and q_{id}. All four of these states accept i. Additionally, q_3 (reached after scanning `for`) accepts k_1, and q_4 (reached after scanning `form`) accepts k_2. Hence, $poss_T(q_2) =$

$acc_T(q_2) \cup acc_T(q_3) \cup acc_T(q_4) \cup acc_T(q_{id}) = \{i\} \cup \{i, k_1\} \cup \{i, k_2\} \cup \{i\} = \{i, k_1, k_2\}$. By contrast, q_{id} has only one state in its transitive closure, *viz.*, q_{id} itself. Hence $poss_T(q_{id}) = acc_T(q_{id}) = \{i\}$.

5.2.2 Reject sets.

When using lexical precedence to disambiguate, if $s \prec t$ for two terminals s and t, s should not occur in the same accept set with t.

At first glance, it would seem that in the single-DFA approach, this only necessitates removing such terminals s from the accept sets containing both s and t, duplicating the traditional approach. However, this causes a problem when scanning on valid lookahead sets that contain s, but not t; the scanner will not recognize s where t matches, but if t is not in the valid lookahead set it will not recognize that *any* relevant terminal matches at that location, and incorrectly retain any *shorter* matches.

For example, suppose a keyword terminal k with $regex(k) =$`keyword` has precedence over an identifier terminal i with $regex(i) =$`[A-Za-z]+`. Then suppose the string "`keyword `" is being scanned with a valid lookahead set $\{i\}$. In this case it is proper for the scanner to terminate with an error, as according to the precedence relation `keyword` should not be matched to i under any circumstances. During scanning, the scanner will recognize as matching i, in order, `k`, `ke`, `key`, `keyw`, `keywo`, and `keywor`. It will then reach the state where k is in the accept set and i is not, and seeing no terminals in the accept set of that state that is in the valid lookahead set, it will not record a match. At the next transition it will stop scanning and, incorrectly, return the token $(i,$`keywor`$)$, this being the longest match recorded.

Hence, following the computation of the possible sets, the accept sets of the scanner DFA will be partitioned, with the s terminals going into a *reject set* that will tell the scanning algorithm to throw away any shorter matches. In the above example, this

would mean moving i to the reject set, causing the token $(i, \texttt{keywor})$ to be disregarded rather than returned.

Definition 5.2.2. *Reject set, rej_T.*

> In a scanner DFA M_T, the *reject set* for a state $q \in Q_T$ consists of those terminals in the accept set for which there are other terminals of higher precedence also in the accept set. Let $rej_T : Q_T \to \mathscr{P}(T)$ represent the reject sets for all states of the DFA.
>
> Formally, $rej_T(q) = \{t \in acc_T(q) : \exists u \in acc_T(q).\, [t \prec u]\}$.

We also define acc'_T to represent the "modified" accept set after the removal of the terminals in the reject set: $acc'_T(q) = acc_T(q) \setminus rej_T(q)$ for each state q, the maxima by precedence in the accept set.

In the example on page 129 and in Figure 5.1, q_0 has an empty accept set, while q_1, q_2, and q_{id} accept only one terminal (*id*); the accept sets of these states will not be split as their reject sets, as defined, are all empty.

However, in the case of q_3 and q_4, $acc_T(q_3) = \{i, k_1\}$ and $acc_T(q_4) = \{i, k_2\}$. Since $i \prec k_1$ and $i \prec k_2$, this means that i will be placed in the reject set of both these states, so:

- $rej_T(q_3) = rej_T(q_4) = \{i\}$;
- $acc'_T(q_3) = \{k_1\}$;
- $acc'_T(q_4) = \{k_2\}$.

This will enable the scanner to reject i in states q_3 and q_4 where terminals of higher precedence are valid matches.

function $singlescan(validLA, w)$: $\mathscr{P}(T) \times \Sigma^\star \to T \times \Sigma^\star$

1. $q = s_T$
2. $lexeme = \varepsilon$
3. $pos = \varepsilon$
4. $present = \emptyset$
5. $validLA' = validLA$
6. do
   (a) $possibles = poss_T(q) \cap validLA'$
   (b) if $pos = \varepsilon$ and $\$ \in validLA'$ then $possibles = possibles \cup \{\$\}$
   (c) if $possibles = \emptyset$ then break
   (d) $validLA' = validLA' \cap possibles$
   (e) $accepts = acc'_T(q) \cap validLA'$
   (f) $rejects = rej_T(q) \cap validLA'$
   (g) if $accepts \neq \emptyset$ then
      i. $present = accepts$
      ii. $lexeme = pos$
   (h) elseif $rejects \neq \emptyset$ then
      i. $present = \emptyset$
      ii. $lexeme = \varepsilon$
   (i) if $w = pos$ then break
   (j) let $c \in \Sigma, w = pos \cdot c \cdot x$ in
      i. $q = \delta_T(q, c)$
      ii. $pos = pos \cdot c$

   indefinitely
7. if $w = \varepsilon \wedge \$ \in validLA'$ then return $(\$, \varepsilon)$
8. elseif $present = \emptyset$ then return $(\perp, \varepsilon)$
9. else let $\{t\} = present$ in return $(t, lexeme)$

Figure 5.2: Single DFA implementation of context-aware scanner.

5.2.3 Scanning algorithm for single DFA approach.

See Figure 5.2 on the previous page for the pseudocode of the context-aware scanning algorithm using the single DFA, which implements the function $scan_{ca}$ from Definition 3.1.3 on page 74. Lines 1–5 initialize: the state is set to the scanner's start state s_T, the scanner's position in the input (represented by a prefix of w, *pos*) to the beginning, the set of most recently matched terminals *present* to $\emptyset$, and the working valid lookahead set $validLA'$ to *validLA*. (When using this algorithm with an LR parser, this will be $validLA_{PT}(n)$ rather than $getValidLA_{PT}(n)$, because the single-DFA algorithm uses reject sets instead of the expanded valid lookahead set of Figure 4.2 on page 97 to sort out precedence relations.)

The loop in line 6 iterates through the input string one character at a time. Line 6a obtains the set *possibles* of terminals that, based on the part of the input that has been read in already, could possibly be matched to a longer string. Line 6b adds the end-of-input symbol \$ to the possible set if the scanner is at the beginning of the input and \$ is being scanned for. Line 6c checks if the possible set is empty (*i.e.*, there are no longer matches than those already obtained); if it is, the set *present*, representing the longest matches found up to this point in the scan, is then known to contain the correct match, and the scan will be stopped so it can be returned. Line 6d throws unmatchable terminals out of the working valid lookahead set, thus ensuring that $validLA'$ contains only matchable terminals at any given point. Lines 6e and 6f build sets *accepts* and *rejects* constituting, respectively, the valid terminals that match the current prefix *pos* and are to be accepted, and the valid terminals that match *pos* but are to be actively rejected.

Line 6g checks to see if *pos* matches a terminal that is to be accepted. If it does, this terminal is stored in the set *present*. Since *present* is the result of a chain of intersection operations from *validLA*, there is a loop invariant that $present \subseteq validLA$, which is

useful in proving the algorithm's correctness.

Note that the assignment to *present* may replace a shorter match that was already in that variable. This implements the maximal munch principle. Note also that *present* is expected to hold only one terminal. If the parser and scanner have passed the analysis outlined in section 4.7.3 for detecting lexical ambiguities, there is guaranteed to be only one terminal in *present* at any one time (although if a disambiguation function is being used, this one terminal will be a Schrödinger terminal, as defined in Definition 4.3.1 on page 99).

Line 6h checks to see if *pos* matches a terminal that is to be actively rejected. If so, it will, like in the above line, remove any shorter matches from *present*, but will not replace them.

Line 6i checks to see if the end of input has been reached, and stops the scan if it has. Line 6j transitions the DFA. Firstly, it takes the character c at the current input position. To illustrate by example, if the input string were `foobar`, and the scanner had progressed past `foo`, $pos = \texttt{foo}$, $c = \texttt{b}$, and $x = \texttt{ar}$. It then calls the DFA's transition function, assigns the new state, and appends c to the end of *pos* to advance the input position.

Lines 7, 8 and 9 return the result of the scan. Line 7 checks whether or not conditions are met for the end-of-input symbol \$ to be matched — *i.e.*, the scanner was started at the end of the input and \$ is a valid terminal to match — and returns a token for \$ if they are. If they are not, line 9 returns a token for any other terminal.

5.2.4 Example of operation.

For an example of the algorithm's operation, take the lexical syntax provided as an example on page 129 whose corresponding single DFA is shown in Figure 5.1 on page 130. Suppose that this scanner is run from a state with the valid lookahead set

$\{i, k_2\}$ and input `form*`.

1. $q = s_T$, *lexeme* $= pos = \varepsilon$, $validLA' = \{i, k_2\}$, $present = \emptyset$.

   - $possibles = \{i, k_1, k_2\} \cap \{i, k_2\} = \{i, k_2\}$; $validLA'$ remains the same.
   - No terminal is accepted or rejected in this state: $accepts = \emptyset \cap \{i, k_2\} = \emptyset$; $rejects = \emptyset \cap \{i, k_2\} = \emptyset$.
   - $pos = \emptyset$ and $w =$`form*`, so $w \neq pos$.
   - $c =$`f`; $pos =$`f` and $q = \delta(s_T, \texttt{f}) = q_1$.

2. $q = q_1$, *lexeme* $= \varepsilon$, $pos =$ `f`, $validLA' = \{i, k_2\}$, $present = \emptyset$.

   - $possibles = \{i, k_1, k_2\} \cap \{i, k_2\} = \{i, k_2\}$; $validLA'$ remains the same.
   - The accept set in this state is $\{i\}$ and the reject set is empty: $accepts = \{i\} \cap \{i, k_2\} = \{i\}$; $rejects = \emptyset \cap \{i, k_2\} = \emptyset$.
   - Accept the token $(i, \texttt{f})$: $present = \{i\}$, *lexeme* $=$ `f`.
   - $w \neq pos$; $c =$`o`; $pos =$`fo` and $q = \delta(q_1, \texttt{o}) = q_2$.

3. $q = q_2$, *lexeme* $=$ `f`, $pos =$ `fo`, $validLA' = \{i, k_2\}$, $present = \{i\}$.

   - $possibles = \{i, k_1, k_2\} \cap \{i, k_2\} = \{i, k_2\}$; $validLA'$ remains the same.
   - The accept set in this state is $\{i\}$ and the reject set is empty: $accepts = \{i\} \cap \{i, k_2\} = \{i\}$; $rejects = \emptyset \cap \{i, k_2\} = \emptyset$.
   - Accept the token $(i, \texttt{fo})$: $present = \{i\}$, *lexeme* $=$ `fo`.
   - $w \neq pos$; $c =$`r`; $pos =$`for` and $q = \delta(q_2, \texttt{r}) = q_3$.

4. $q = q_3$, *lexeme* $=$ `fo`, $pos =$ `for`, $validLA' = \{i, k_2\}$, $present = \{i\}$.

- *possibles* $= \{i, k_1, k_2\} \cap \{i, k_2\} = \{i, k_2\}$; *validLA*$'$ remains the same.
- The accept set in this state is $\{k_1\}$ and the reject set $\{i\}$: *accepts* $= \{k_1\} \cap \{i, k_2\} = \emptyset$; *rejects* $= \{i\} \cap \{i, k_2\} = \{i\}$.
- Reject the token $(i, \texttt{for})$ by flushing $(i, \texttt{fo})$: *present* $= \emptyset$, *lexeme* $= \varepsilon$.
- $w \neq pos$; $c =$ `m`; *pos* $=$ `form` and $q = \delta(q_3, \texttt{m}) = q_4$.

5. $q = q_4$, *lexeme* $= \varepsilon$, *pos* $=$ `form`, *validLA*$' = \{i, k_2\}$, *present* $= \emptyset$.

   - *possibles* $= \{i, k_2\} \cap \{i, k_2\} = \{i, k_2\}$; *validLA*$'$ remains the same.
   - The accept set in this state is $\{k_2\}$ and the reject set $\{i\}$: *accepts* $= \{k_2\} \cap \{i, k_2\} = \{i\}$; *rejects* $= \{i\} \cap \{i, k_2\} = \{i\}$.
   - Accept the token $(k_2, \texttt{form})$: *present* $= \{k_2\}$, *lexeme* $=$ `form`.
   - $w \neq pos$; $c = *$; *pos* $=$ `form`$*$. Now there is no transition in this DFA for the character $*$, so a default, implicit "trap" state is used: $q = \delta(q_4, *) = trap$.

6. $q = trap$, *lexeme* $=$ `form`, *pos* $=$ `form`$*$, *validLA*$' = \{i, k_2\}$, *present* $= \{k_2\}$.

   - The possible, accept, and reject sets are all $\emptyset$ in this "trap" state. Consequently, *possibles* is set to $\emptyset$ and the loop breaks. The token $(k_2, \texttt{form})$ is returned.

5.2.5 Proof sketch of the algorithm's correctness.

Theorem 5.2.3. *The algorithm in Figure 5.2 on page 134 produces output in conformance with the requirements of a context-aware scanner as laid out in Definitions 3.1.3 and 4.2.2. Furthermore, with respect to incorporating lexical precedence in the LR framework, when called on a parse state n and validLA* $= validLA_{PT}(n)$*, it produces output in conformance with the additional requirements in Definition 4.2.3.*

To sketch a proof of this theorem, suppose that $(t,x) = singlescan(validLA, w)$. Then, according to Definitions 3.1.3, 4.2.2, and 4.2.3:

1. x must be a prefix of w and in the language of $regex_{PT}(t)$.
2. t must be in $validLA$.
3. There must be no longer prefix of w in the language of any regular expression in the given valid lookahead set, $\bigcup_{u \in validLA_{PT}(n)} L(regex_{PT}(u))$.
4. There must exist no terminal u matching x that takes precedence over t: $\neg\exists u \in T_{PT}.[t \prec u \wedge x \in L(regex_{PT}(u))]$.
5. If t does not have a valid parse action (*i.e.*, it is being scanned to satisfy a precedence relation), it must be the highest-precedence terminal in the grammar matching x: $t \notin validLA_{PT}(n) \Rightarrow \forall u \in T_{PT}.[x \in L(regex_{PT}(u)) \Rightarrow u \prec t]$.

Point 1 follows immediately from the DFA used in the scanner being correctly constructed.

There is a loop invariant in the scanner algorithm that *present* is always a subset of $validLA$. This is true because at any point in the loop, it will be equal either to the empty set, or to the contents of some set *accepts* created from intersection on $validLA' \subseteq validLA_{PT}(n) = validLA$. Point 2 follows immediately from this loop invariant.

For point 3, assume that there *is* such a longer prefix y of w matching a terminal $u \in validLA$. Then by construction, u must appear in the possible set of the state wherein x is matched to t. Thus the scanner will keep scanning on through the rest of y's characters, u being in the possible sets of these states, until the whole of y has been scanned and u is in the accept set, at which point u will be returned. But since this has not happened, and t has been matched, it follows that there is no such u.

Point 4 follows from the partitioning of accept sets; such a terminal t of non-maximal precedence as would be required to make the sentence untrue would never be matched in a state.

Point 5 also follows from *present* being a subset of $validLA'$, which contains only terminals in $validLA_{PT}(n)$.

5.2.6 Verifying determinism.

In section 4.7.3, we discussed the necessity for a new method to verify the lexical determinism of a context-aware scanner. In this section we discuss the specific method used in the single-DFA implementation.

Generally, a scanner is verified deterministic (*i.e.*, giving rise to no lexical ambiguities) by checking that no more than one terminal can be matched in any state of its DFA:

$$\forall q \in Q_T . \left(\left| acc'_T(q) \right| \leq 1 \right)$$

This is the method used in traditional scanner generation and also the method used in the multiple-DFA approach (see section 5.3), which uses an identical algorithm to that of the disjoint scanner. In the case of the disjoint scanner, this test examines all possible matches the scanner can make — *i.e.*, all possible scanner states — and verifies that they are all unambiguous.

But while a context-aware scanner based on the single DFA approach can in principle be verified to be deterministic in this manner, this would produce many "false positives" — specifically, on any grammar that needs a context-aware scanner to disambiguate properly. it is very possible that $|acc'_T(q)| > 1$ for some state q in the DFA, but that no valid lookahead set in the parser contains more than one of the terminals in

$acc'_T(q)$, so no lexical ambiguity would ever arise during scanning, despite the accept set containing more than one terminal.

The solution is an expanded test that checks the DFA against all the parser's valid lookahead sets to see if any ambiguities arise when scanning with that particular valid lookahead set. Formally, for each valid lookahead set *validLA*, the test must verify that $\forall q \in Q_T . (|validLA \cap acc'_T(q)| \leq 1)$.

In the abstract sense, this test does exactly the same thing as the simpler one above: it searches the space of all possible matches the scanner could make — all possible valid lookahead sets the scanner could be started on, and all possible states the scanner could reach during that scan — and verifies that they are all unambiguous. It is only that with a context-aware scanner, one has to take the parser state as well as the scanner state into account when enumerating possible "states."

This test does not produce any false positives, as if the test fails for some valid lookahead set and some $q \in Q_T$, then all that would have to be done to make an ambiguous scan is to give the parser as input, in the context where that valid lookahead set is used, the string w that transitions the scanner into state q.

In the LR framework, where the valid lookahead set is determined by the parse state, the test will check all parse states for an ambiguity, *i.e.*, check the truth of the statement

$$\forall n \in States_{PT} . \forall q \in Q_T . \left(\left| validLA_{PT}(n) \cap acc'_T(q) \right| \leq 1 \right)$$

5.2.7 Reporting scanning errors.

With the single-DFA approach, the method discussed in section 5.3.2 may be employed without any special steps; one can simply run the single DFA on the union of valid lookahead sets and report the result.

5.3 Multiple DFA approach.

Another approach is to compile a separate small DFA for each valid lookahead set and run the scanner on that DFA when scanning in a parser state with that valid lookahead set. This preserves the $O(n)$ runtime of the traditional approach at the cost of somewhat more space: in the worst case there will be a different valid lookahead for each parser state, so the multiple DFAs can take up an increased amount of space compared to the single DFA, potentially to the factor of the number of valid lookahead sets in the grammar — in the LR framework, $|States_{PT}|$ times more space.

5.3.1 Process.

The scanning algorithm used in this approach is exactly the same as the one used in a traditional disjoint scanner implementation; there are no reject or possible sets, and scanning is stopped when the scanner enters an implicit "trap" state from which no further states can be reached by transition, at which point the accept set in the state with the last match will be returned.

The differences from the traditional approach lie entirely in the manner in which the scanner functions are built. Specifically, for every valid lookahead set V, a different scanner DFA is built (in the traditional manner) from the set obtained from the $getValidLA_{PT}$ function in Figure 4.2 on page 97, which is then used to scan on any state that has V as its valid lookahead set.

Proving the correctness of the scanning apparatus is also very straightforward in this case, as the algorithm is identical to the traditional one; maximal munch is guaranteed, and the scanner is built from only the terminals in *validLA*, so no other terminals can match. In the LR framework, if the matched terminal is not in $validLA_{PT}(n)$, it is in $validLA \setminus validLA_{PT}(n) = \{t \in T_{PT} \setminus validLA_{PT}(n) \ : \ \exists u \in validLA_{PT}(n).\, [u \prec t]\}$,

matching the conditions of Definition 4.2.3.

5.3.2 Reporting scanning errors.

In the single-DFA approach, it is very straightforward just to run the scanner DFA on the union of all valid lookahead sets as discussed in section 5.2.7. However, in the multiple-DFA approach, another DFA must be constructed specifically for the union of all valid lookahead sets, to enable scanning on this set. We call this the *union DFA*.

5.3.3 Optimizations.

There is also a class of optimizations that can be performed on the multiple-DFA approach (in addition to any optimizations for traditional scanners, which can be applied equally well to these multiple-DFA scanners). These specific optimizations are heuristics aimed at reducing the number of separate DFAs in the final scanner far below the worst-case scenario of the number of valid lookahead sets, by allowing several valid lookahead sets to share the same DFA without having to introduce the annotations used in the single-DFA approach.

We have implemented and tested only one such heuristic, which is described below, but there is good reason to believe that there are many others from this same class that are useful, and hence in this section we also outline the general principles of the class. Possible directions for future work in this area are discussed further in section 10.2.1 on page 237.

Firstly, note that the worst case is a corner case. In practice, the number of valid lookahead sets is usually a small fraction of the number of parser states; many parser states will have identical valid lookahead sets and can use the same DFA. In several tests, we have found the ratio of valid lookahead sets to parse states to vary, although it

tends to decrease with larger grammars:

- A toy grammar used for testing (see appendix A.4) produces 6 valid lookahead sets covering 9 parse states (a ratio of two-thirds).
- A slightly more complex grammar, for four-function arithmetic, produces 5 valid lookahead sets covering 15 parse states (a ratio of one-third).
- ableC (see section 9.2 and appendix A.1) has 99 valid lookahead sets covering 476 parse states (a ratio of roughly 20%).
- ableJ (see section 9.4 and appendix A.3) has 133 valid lookahead sets covering 1107 parse states (a ratio of roughly 12%).

5.3.3.1 General principle of optimization.

The general principle to be used for these optimizations is the fact that a DFA built for a certain valid lookahead set V can be used to scan on any $S \subseteq V$, provided that no prefix of a lexeme matching a terminal in $V \setminus S$ matches a terminal in S. This is stated formally as follows.

Theorem 5.3.1. *Given a valid lookahead set V, $R = \{r : \exists t \in V.\, (r = regex(t))\}$, and the scanner built from the DFA $M_T = \langle Q_T, s_T, \delta_T \rangle$ that is built from the regular expressions of R, the scanner, called on another valid lookahead set $S \subseteq V$, will match the criteria in Definitions 3.1.3, 3.2.2 4.2.2, and 4.2.3 provided that*

$$\forall q \in Q_T.\, \left[acc_T(q) \cap S \neq \emptyset \Rightarrow \bigcup_{q' \in \delta_T^\star(q)} acc_T(q') \cap (V \setminus S) = \emptyset \right]$$

Proof. The DFA, being built from the union of the regular expressions of, among other terminals, all those in S, will match lexemes to such terminals correctly, unless either:

(1) there is a lexical ambiguity in some DFA state q between some $s \in S$ and some $t \in V \setminus S$, or (2) there is another state in the transitive closure of q accepting $t \in V \setminus S$, causing maximal munch to be applied inappropriately when this state is reached, matching t instead of failing and matching s, as it would using a DFA built from the terminals in S only.

The restrictions facially prevent the second occurrence. As to the first, since the DFA was built from a superset of the terminals of S, $q \in \delta_T^\star(q)$, the restrictions proscribe terminals in S and terminals not in S from occurring in the same accept set, preventing any lexical ambiguities or erroneous precedence-based disambiguations arising on that account. □

Although the theorem proves that such a DFA will work correctly when given syntactically *correct* input, it introduces the possibility that given syntactically *invalid* input, the DFA will return to the parser a token matching a terminal not in the valid lookahead set, which is not the intended behavior of a context-aware scanner. If this is a problem, a post-process check can be instituted to ensure that the token returned matches a terminal in the valid lookahead set (in the LR framework, this check would take the form of a modification to the method $runScan_{PT}$). If no terminal in the valid lookahead set is matched, the scanner may fail with a traditional "unexpected token" syntax error indicating that there is no parse action for the terminal matched by the scanner.

5.3.3.2 Union adequacy.

The heuristic we have implemented at present is fairly simple but effective, exploiting the presence of the union DFA for error-handling. It happens that the union DFA matches the constraints of Theorem 5.3.1 for a great many valid lookahead sets, which are then called *union adequate*. This approach has the advantage that the union DFA

has already been built and can be directly tested using the theorem without running the risk of building a DFA that will not be used.

The success of this heuristic varies from moderate to high; it cuts the number of DFA states needed in the *ableC* scanner by roughly half (which still leaves approximately 11,000 states), but also eliminates every DFA except the union DFA on simpler grammars where context-aware scanning is not strictly necessary.

Chapter 6

Modular determinism analysis.

Although context-aware scanning solves many of the problems attendant with extensible languages, and makes parsing of languages with heterogeneous lexical syntax much more practical, it does not solve an intrinsic flexibility limitation of the LALR(1) parser algorithm: namely, that the class of LALR(1) grammars is not closed under composition. This is particularly troublesome when seeking an automatic process for composing extensions that, individually, compose conflict-free with the host grammar.

The problem is to find an analysis that is performed at the time of those individual compositions, but provides a guarantee that the host grammar can be composed conflict-free with any combination of the tested extensions.

This chapter presents such an analysis, *Partitionable* (originally presented in our paper *Verifiable Composition of Deterministic Grammars* [SVW09], in which it was called *isComposable*). This analysis can recognize membership in a class of grammar extensions that can be deterministically composed with each other; it accomplishes this by determining that when the host and extension are composed, the part of the composed LR DFA that parses the host remains essentially the same as in the original host DFA, while all states in the LR DFA parsing extension constructs reside in their

own "partition."

6.1 Preliminary definitions.

In this section we set out some definitions of terms that will be used throughout the chapter.

Hitherto we have informally discussed *host* and *extension* grammars. We now make more formal definitions of these types of grammars — an extension grammar being a regular grammar that instead of a start symbol has a production tying it into the host grammar.

Definition 6.1.1. *Host and extension grammars, bridge production, marking terminal.*

A *host grammar* Γ^H is a context-free grammar that is to be extended by one or more extension grammars. An *extension grammar* Γ^E is defined with respect to a host grammar Γ^H. It is a 5-tuple $\langle T_E, NT_E, P_E, nt_H \rightarrow \mu_E s_E, regex_E \rangle$ — a context-free grammar with a *bridge production* replacing the start symbol. The grammar and bridge production must satisfy the following conditions:

1. The extension's terminal and nonterminal sets are disjoint from the host's: $T_H \cap T_E = NT_H \cap NT_E = \emptyset$.
2. For each $p_E \in P_E$, the left hand side nonterminal is in NT_E, while the symbols on the right hand side are in $T_E \cup NT_E \cup T_H \cup NT_H$.
3. The bridge production's left hand side must be a host nonterminal: $nt_H \in NT_H$.
4. μ_E is a *marking terminal*, unique to Γ^E, that is in neither T_H nor T_E. This is necessary so that the marking terminal may appear *only* in the

bridge production, which ensures the necessary degree of separation between the host and extension parts of the grammar, as discussed further below.

5. $dom(regex_E) = T_E \cup \{\mu_E\}$.

If the conditions in Definition 6.1.1 are matched, Γ^E is said to *extend* Γ^H.

Note that we have in practice relaxed the meaning of "extension" from this formal definition: under the relaxed definition, a language extension can have several bridge productions. In the formal sense, such an extension could be automatically rewritten to fit the stricter model, so we have chosen this formal definition to simplify the proofs of the modular determinism analysis.

We next define two operations by which grammars are composed: $\cup_G$ and $\cup_G^\star$. $\cup_G$ is a straightforward composition operation of one host grammar and one extension grammar, while $\cup_G^\star$ is the generalization of $\cup_G$ to one host grammar and several extension grammars.

Let CFG_L denote the set of context-free grammars, and CFG_E denote the set of extension grammars.

Definition 6.1.2. *Grammar composition ($\cup_G$).*

Let $\cup_G : CFG_L \times CFG_E \to CFG_L$ be a non-commutative, non-associative operation on one host grammar and one extension grammar that extends it, representing the composition of the two grammars. Specifically, if $\Gamma^C = \Gamma^H \cup_G \Gamma^E$, then

$$\Gamma^C = \begin{cases} \langle T_C, NT_C, P_C, s_H, regex_C \rangle & \text{if } \Gamma^E \text{ extends } \Gamma^H \\ \bot & \text{otherwise} \end{cases}$$

where $T_C = T_H \cup T_E \cup \{\mu_E\}$, μ_E being Γ^E's marking terminal; $NT_C = NT_H \cup NT_E$; $P_C = P_H \cup P_E \cup \{nt_H \rightarrow \mu_E s_E\}$ (Γ^E's bridge production); and

$$regex_C(t) = \begin{cases} regex_H(t) & \text{if } t \in T_H \\ regex_E(t) & \text{if } t \in T_E \text{ or } t = \mu_E \end{cases}$$

Definition 6.1.3. *Generalization of grammar composition ($\cup_G^\star$).*

Let $\cup_G^\star : CFG_L \times \mathscr{P}(CFG_E) \rightarrow CFG_L$ represent the generalization of $\cup_G$ for composing an unordered *set* of extensions with the host grammar they extend. Specifically:

$$\cup_G^\star\left(\Gamma^H, \left\{\Gamma_1^E, \ldots, \Gamma_n^E\right\}\right) = \begin{cases} \langle T_C, NT_C, P_C, s_H, regex_C\rangle & \text{if each } \Gamma_i^E \text{ extends } \Gamma^H \\ \bot & \text{otherwise} \end{cases}$$

where $T_C = T_H \cup \bigcup_{i=1}^n \left(T_i^E \cup \left\{\mu_i^E\right\}\right)$, $NT_C = NT_H \cup \bigcup_{i=1}^n NT_i^E$,
$P_C = P_H \cup \bigcup_{i=1}^n \left(P_i^E \cup \left\{nt_i^H \rightarrow \mu_i^E s_i^E\right\}\right)$, and

$$regex_C(t) = \begin{cases} regex_H(t) & \text{if } t \in T_H \\ regex_i^E(t) & \text{if } t \in T_i^E \text{ or } t = \mu_i^E \end{cases}$$

We next define sets *conflictFree* and *lexAmbigFree*. *conflictFree* contains those grammars that will not generate parse-table conflicts when compiled into an LALR(1) parse table PT_Γ, according to the definition of "conflict-free" on page 77. *conflictFree* can be decided by the LALR(1) compilation process.

lexAmbigFree contains those grammars Γ that give rise to no lexical ambiguities — *i.e.*, when Γ is compiled into a context-aware scanner, there is no string w that would cause it to encounter an unresolvable ambiguity. (Recall that according to the definition on page 76, regular expressions are included with the grammar, so lexical syntax may be considered with respect to a context-free grammar.) *lexAmbigFree* can be decided by whatever method is appropriate to the scanner implementation (*e.g.*, the method specified in section 5.2.6).

6.2 Formal requirements.

We now define a relation *isComposable* that encapsulates the minimum requirements for the modular determinism analysis: that it be defined on the host and a single extension, and that it guarantee conflict-free composability for any set of extensions that pass it.[1]

Later, we will show that the set of grammars passed by our modular determinism analysis is a subset of *isComposable* (*i.e.*, the analysis meets the requirements). We define the requirements separately because we recognize that our modular determinism analysis is not the only possible one (we have ourselves experimented with a tighter set of restrictions that provide the same guarantees [VWKSB07]), and we wish to draw clear lines in that regard.

Definition 6.2.1. *isComposable.*

Let Γ^H be a host grammar, and $\Gamma^E_1, \Gamma^E_2 \ldots, \Gamma^E_n$ be extension grammars extending Γ^H.

Then let $isComposable \subseteq CFG_L \times CFG_E$ refer to the set of host/extension pairs $\left(\Gamma^H, \Gamma^E\right)$ such that:

- Γ^H extends Γ^E.
- For each subset of *isComposable* sharing a common host grammar Γ^H (*i.e.*, each grammar set $G \subseteq isComposable$ such that $\left(\Gamma^H_1, \Gamma^E_1\right) \in G \wedge \left(\Gamma^H_2, \Gamma^E_2\right) \in G \Rightarrow \Gamma^H_1 = \Gamma^H_2$), all the extension grammars in G can be composed conflict-free with Γ^H; *i.e.*,

$$conflictFree\left(\Gamma^H \cup^{\star}_G \left\{\Gamma^E_i \ : \ \left(\Gamma^H, \Gamma^E_i\right) \in G\right\}\right)$$

[1] Note that in [SVW09], *isComposable* is used to refer to another set, *viz.*, grammars that pass the specific modular determinism analysis presented in this chapter. The *isComposable* of that paper is presented below, in Definition 6.3.10 on page 157, as *Partitionable*.

Corollary 6.2.2. *What isComposable guarantees.*

Given a host grammar Γ^H and a set of grammars $\left\{\Gamma^E_1, \ldots, \Gamma^E_n\right\}$ extending it,

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

6.3 Specification of the analysis.

Our modular determinism analysis, which recognizes membership in *isComposable* for a pair $\left(\Gamma^H, \Gamma^E_i\right)$, is, like *isComposable*, not defined on the extension grammar itself, but on two LR DFAs: that generated when compiling Γ^H, and that generated when compiling $\Gamma^H \cup_G \Gamma^E_i$. The idea behind the analysis is that if the DFA for $\Gamma^H \cup_G \Gamma^E_i$ can be divided into three state partitions — one that is identical to the DFA for Γ^H with a tightly circumscribed set of exceptions, and two containing states used to parse extension constructs — then when generating a DFA for $\Gamma^H \cup^\star_G \{\Gamma^E_1, \ldots, \Gamma^E_n\}$ (with $\forall i \in [1,n]\,.\mathit{isComposable}(\Gamma^H, \Gamma^E_i)$), each of the extension-construct partitions will remain separate and the host partition will remain unchanged from the Γ^H machine. This means that no new conflicts will be introduced to the combined DFA.

We first define some terms relating to the the LR DFAs generated from the various grammars involved with the analysis. Each such LR DFA will be denoted by the letter M with a superscript indicating the grammar from which it is generated. M^H will represent the LR DFA compiled from a host grammar Γ^H; M^{E_i}, the LR DFA compiled from the host composed with a single extension, $\Gamma^H \cup_G \Gamma^E_i$; M^C, the LR DFA compiled from the host composed with a set of extensions, $\Gamma^H \cup^\star_G \{\Gamma^E_1, \ldots, \Gamma^E_n\}$.

Figure 6.1 on the next page contains block diagrams illustrating the use of the analysis with regard to a host grammar Γ^H and two extensions Γ^E_i and Γ^E_j. Figure 6.1(a) diagrams what the modular determinism analysis checks for in the LR DFA of $\Gamma^H \cup_G \Gamma^E_i$;

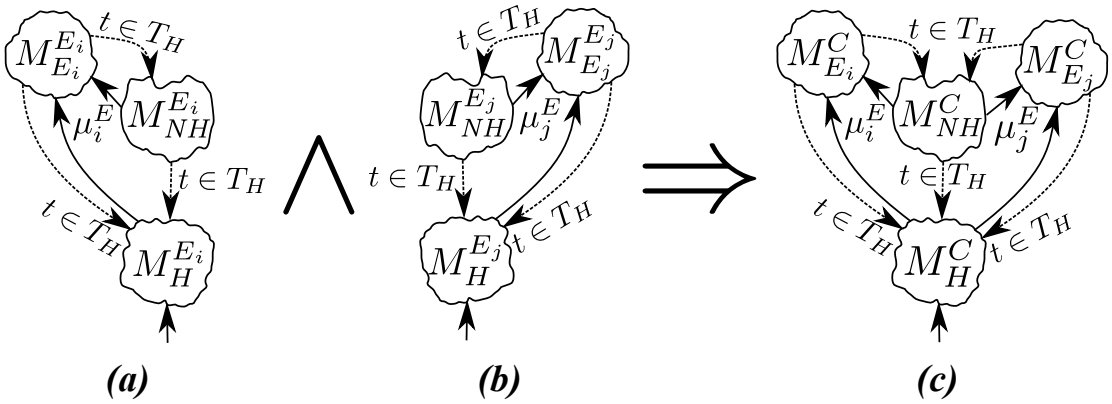

Figure 6.1: An illustration of how the modular determinism analysis works. From the left, block diagrams for the LR DFAs M^{E_i}, M^{E_j}, and M^C for, respectively, $\Gamma^H \cup_G \Gamma^E_i$, $\Gamma^H \cup_G \Gamma^E_j$, and $\Gamma^H \cup^\star_G \{\Gamma^E_i, \Gamma^E_j\}$.

Figure 6.1(b) diagrams the same thing for $\Gamma^H \cup_G \Gamma^E_j$; Figure 6.1(c) diagrams what the success of these checks guarantees of the LR DFA for $\Gamma^H \cup^\star_G \{\Gamma^E_i, \Gamma^E_j\}$. In Figures 6.1(a) and 6.1(b) are shown the partitioning of LR DFAs for passing extensions, with the three state partitions marked as blocks; these partitions are defined more precisely below. As these two sub-figures are roughly identical we will confine further discussion to Figure 6.1(a).

As can be seen, the LR DFA's start state is located in the partition $M^{E_i}_H$, representing the partition that is equivalent to the host language's LR DFA. Upon shifting the extension's marking terminal μ^E_i, as shown, the parser will enter the partition $M^{E_i}_{E_i}$, as it is now parsing an extension construct.

If the extension grammar contains back references to the host grammar (*i.e.*, a production with a host nonterminal on its right hand side) it may shift a host terminal and transition back to $M^{E_i}_H$, or into $M^{E_i}_{NH}$. $M^{E_i}_{NH}$ consists of states that parse only host-language constructs, but in new contexts. For example, in the parser of one of our language extensions, there is a state that parses Java array declarations in isolation from their usual context of expressions (see section 9.4.5 on page 224).

If the modular determinism analysis verifies two extensions Γ_i^E and Γ_j^E — *i.e.*, the LR DFAs for $\Gamma^H \cup_G \Gamma_i^E$ and $\Gamma^H \cup_G \Gamma_j^E$ are of the form shown in Figures 6.1(a) and 6.1(b) respectively — then the analysis guarantees that when both extensions are composed together with the host into $\Gamma^H \cup_G^\star \{\Gamma_i^E, \Gamma_j^E\}$, the resulting LR DFA may be partitioned as shown in Figure 6.1(c). Here, there is the same partition pattern: a partition M_H^C, still equivalent to the original host LR DFA; partitions $M_{E_i}^C$ and $M_{E_j}^C$, equivalent to $M_{E_i}^{E_i}$ and $M_{E_i}^{E_i}$ respectively; and M_{NH}^C, created through a merge of $M_{NH}^{E_i}$ and $M_{NH}^{E_j}$. Part of the analysis is to guarantee that this merging does not cause conflicts.

We now define several relations on LR DFA states concerning their item sets and lookahead sets; these are needed in the proofs of the modular determinism analysis as well as to specify the part of it relating to the merging of M_{NH}^C.

Definition 6.3.1. *I-subset,* $\subseteq_{\mathrm{I}}$.

> An LR DFA state s is an *I-subset* of another state t if s's item set is a subset of t's. Formally, $s \subseteq_{\mathrm{I}} t$ iff $\mathit{items}(s) \subseteq \mathit{items}(t)$.

Definition 6.3.2. *LR(0) equivalence,* $\equiv_0$.

> Two LR DFA states s and t are *LR(0)-equivalent* if they would be equal in an LR(0) DFA, *i.e.*, they have the same item sets. Denny and Malloy [DM08] term s and t *isocores*. Formally, $s \equiv_0 t$ iff $\mathit{items}(s) = \mathit{items}(t)$, or alternatively, $s \subseteq_{\mathrm{I}} t \wedge t \subseteq_{\mathrm{I}} s$.

Definition 6.3.3. *IL-subset,* $\subseteq_{\mathrm{IL}}$.

> An LR DFA state s is an *IL-subset* of another state t if $s \subseteq_{\mathrm{I}} t$ and, in addition, each lookahead set in s is a subset of the corresponding lookahead set in t. Formally, $s \subseteq_{\mathrm{IL}} t$ iff $s \subseteq_{\mathrm{I}} t \wedge \forall i \in \mathit{items}(s).\, [la_s(i) \subseteq la_t(i)]$.

Definition 6.3.4. *LR(1)-equivalence,* $\equiv_1$.

Two LR DFA states s and t are *LR(1)-equivalent* if they have the same item sets and each item has the same lookahead set. Formally, $s \equiv_1 t$ iff $items(s) = items(t) \wedge \forall i \in items(s).[la_s(i) = la_t(i)]$, or alternatively, $s \subseteq_{\text{IL}} t \wedge t \subseteq_{\text{IL}} s$.

When composing extensions with a host language, the junction points are the bridge productions and marking terminals. These, being unique to a single extension, will not be the cause of the sort of conflicts the analysis is aimed at preventing; hence, we provide the following two definitions to eliminate them from consideration. They are both defined on states n_H and n_C; the subscripts are in reference to the expectation that n_H will be a state of M^H and n_C a state of an LR DFA for some composed language, such as M^{E_i} or M^C.

Definition 6.3.5. *LR(0)-equivalence with exception for bridge items,* $\equiv_0^C$.

With respect to a composed grammar $\Gamma^H \cup_G^\star \{\Gamma_1^E, \ldots \Gamma_n^E\}$ and two LR DFA states n_H and n_C, $n_H \equiv_0^C n_C$ if $items(n_C) \setminus items(n_H) \subseteq \{nt_H \rightarrow \bullet \mu_E s_E\}$ (a set containing the bridge productions of all the composed extensions).

Definition 6.3.6. *LR(1)-equivalence with exceptions for marking terminal lookahead and bridge items,* $\equiv_1^C$.

With respect to a composed grammar $\Gamma^H \cup_G^\star \{\Gamma_1^E, \ldots \Gamma_n^E\}$ and two LR DFA states n_H and n_C, $n_H \equiv_1^C n_C$ if:

1. $n_H \subseteq_{\text{IL}} n_C$;
2. $items(n_C) \setminus items(n_H) \subseteq \{nt_H \rightarrow \bullet \mu_E s_E\}$ (a set containing the bridge productions of all the composed extensions).
3. $\forall i \in items(n_H).\left[la_{n_E}(i) \setminus la_{n_H}(i) \subseteq \{\mu_1^E, \ldots, \mu_n^E\}\right]$.

This is the same as LR(1)-equivalence, except that n_C is allowed to have extra items representing the bridge productions of the several extensions and any of the extensions' marking terminals added to any lookahead sets.

6.3.1 Formal specification.

In this section, we make a formal definition of the modular determinism analysis as a set $Partitionable \subseteq CFG_L \times CFG_E$. The analysis examines the DFAs M^H and M^{E_i} to ensure that each state in M^{E_i} fits into one of three state partitions; before we define *Partitionable* we define these three partitions.

Definition 6.3.7. *State partition* $M_H^{E_i}$.

This partition consists of states that "belong" to the host grammar, being LR(1)-equivalent to some state in M^H except for bridge items and marking terminal lookahead. Formally,
$M_H^{E_i} = \left\{ n_E \in States_{M^{E_i}} \ : \ \exists n_H \in States_{M^H} . \left[n_H \equiv_1^C n_E \right] \right\}$.

Definition 6.3.8. *State partition* $M_{E_i}^{E_i}$.

This partition consists of states that "belong" to the extension grammar, containing one or more items with an extension nonterminal on the left hand side. Formally,
$M_{E_i}^{E_i} = \{ n_E \in States_{M^{E_i}} \ : \ \exists (nt \rightarrow \alpha, k) \in items(n_E) . [nt \in NT_{E_i}] \}$.

Definition 6.3.9. *State partition* $M_{NH}^{E_i}$.

This partition consists of "new host" states — states that do not have any items with an extension nonterminal on the left hand side, and do not have an analogue in M^H. However, states in this partition are required to be IL-subsets of M^H states. Formally, $n_E \in M_{NH}^{E_i}$ iff:

1. $n_E \notin M_H^{E_i}$;
2. $\forall (nt \to \alpha) \in items(n_E).[nt \in NT_H]$;
3. $\exists n_H \in M_H^{E_i}.[n_E \subseteq_{\text{IL}} n_H]$; and
4. $\forall n_H \in M_H^{E_i}.[n_E \subseteq_{\text{I}} n_H \Rightarrow n_E \subseteq_{\text{IL}} n_H]$.

We are now ready to define the modular determinism analysis *Partitionable*.

Definition 6.3.10. *The analysis Partitionable.*

Let *Partitionable* $\subseteq (CFG_L, CFG_E)^2$ represent the modular determinism analysis. Specifically, $\left(\Gamma^H, \Gamma_i^E\right) \in$ *Partitionable* iff:

- $\forall nt \in NT_H.\left[follow_{E_i}(nt) \setminus follow_H(nt) \subseteq \left\{\mu_i^E\right\}\right]$: the follow set of any host language nonterminal must not vary between Γ^H and Γ_i^E, except for the possible addition of the marking terminal in the latter.
- $\forall n \in States_{M^{E_i}}.\left[n \in M_H^{E_i} \cup M_{E_i}^{E_i} \cup M_{NH}^{E_i}\right]$: each state in M_i^E must be able to be placed in one of the three partitions above.

6.4 Proof of correctness.

We now state the central theorem of this chapter.

Theorem 6.4.1. *Partitionable* $\subseteq$ *isComposable;* i.e.*, a set of host-extension pairs passing the analysis Partitionable meets the criteria laid out in Definition 6.2.1 on page 151, which ensure that any subset of Partitionable sharing a common host grammar can be composed without conflict, as illustrated by Corollary 6.2.2.*

[2] Note that this analysis *Partitionable* is called *isComposable* in [SVW09]. We have changed the name here because we feel it is more accurate if *isComposable* refers to *all* grammars that may be composed error-free rather than only those that pass this specific analysis.

Proof summary. If *Partitionable* $\subseteq$ *isComposable*, then in consideration of any set of extensions $\{\Gamma_1^E, \ldots, \Gamma_n^E\}$ with *Partitionable* (Γ^H, Γ_i^E) for each i, if they are all composed together with the host grammar into $\Gamma^H \cup_G^\star \{\Gamma_1^E, \ldots, \Gamma_n^E\}$, this grammar will compile conflict-free.

This can be verified by showing that, if each machine M^{E_i} built from $\Gamma^H \cup \Gamma^{E_i}$ can be partitioned as exemplified in Figure 6.1(a), then the states of M^C (the LR DFA built from $\Gamma^C = \Gamma^H \cup_G^\star \{\Gamma_1^E, \ldots, \Gamma_n^E\}$) can be separated as exemplified in Figure 6.1(c), into exactly $n+2$ partitions: M_H^C (corresponding to the partitions $M_H^{E_i}$, used in parsing host constructs), M_{NH}^C (corresponding to the partitions $M_{NH}^{E_i}$), and, for each i, $M_{E_i}^C$ (corresponding to the partitions $M_{E_i}^{E_i}$, used in parsing extension constructs), and all these partitions are conflict-free.

We first prove three lemmas used in the proof of the main theorem. Each lemma works toward the theorem's conclusion by starting with an arbitrary host language Γ^H and an arbitrary set of its extensions $\Gamma_1^E, \ldots, \Gamma_n^E$ that all pass the analysis (*i.e.*, *Partitionable* (Γ^H, Γ_i^E) for each Γ_i^E), and prove properties on their composition $\Gamma^H \cup_G^\star \{\Gamma_1^E, \ldots, \Gamma_n^E\}$, and the machines M^H, M^{E_i}, and M^C as defined above. Lemma 6.4.2 establishes that the partitions $M_{E_i}^C$ are distinct; Lemma 6.4.3, that *Partitionable*'s follow set restrictions carry over to Γ^C; Lemma 6.4.4, that bridge items and marking terminal lookahead, which *Partitionable* allows to be added in any state, do not give rise to conflicts in M^C.

Lemma 6.4.2. No items from two extensions in any state in M^C. If a state $n \in States_{M^C}$ has an item $(nt \to \alpha \bullet \beta) \in items(n)$ such that $nt \in NT_{E_i}$, it then follows that $\{nt : (nt \to \gamma \bullet \delta) \in items(n)\} \subseteq NT_H \cup NT_{E_i}$: *i.e.*, no state has items with left hand side nonterminals from different extensions.

Proof. To make this proof, we must show that if a state contains an item with a nonterminal from Γ_i^E on the left-hand side, it cannot contain an item with a nonterminal from

another extension Γ_j^E — and, by extension, that the partitions $M_{E_i}^C$ are disjoint.

By construction, all states n in M^C containing an item with a left hand side in NT_{E_i} must only be reachable by transition from the start state s_C via a state seeded solely from the item $h \to \mu_i^E \bullet s_i^E$: the state $n_{E_i}^S$ reached immediately after shifting the extension's marking terminal. Consequently, all such states n are in the transitive closure of $n_{E_i}^S$.

Also by construction, states in the transitive closure of $n_{E_i}^S$ will only contain items with left hand sides derivable from s_i^E, *i.e.*, host symbols and symbols from Γ_i^E. The only exception is if some derivable host nonterminal h_j is on the left hand side of a bridge production for another extension, $h_j \to \mu_j^E s_j^E$. But by the same construction, the start state $n_{E_j}^S$ for *this* extension precludes any symbols from any extension except Γ_j^E occurring in any states in *its* transitive closure. □

Lemma 6.4.3. Follow sets of Γ^C differ from follow sets of Γ^H only by addition of marking terminals. $\forall nt \in NT_H. \left[follow_C(nt) \setminus follow_H(nt) \subseteq \left\{\mu_1^E, \ldots, \mu_n^E\right\}\right]$.

Proof summary. *Partitionable* restricts follow sets in each grammar $\Gamma^H \cup_G \Gamma_i^E$ with respect to Γ^H, such that no new non-marking terminals can appear; these restrictions carry over to Γ^C as well.

This is not a trivial property to prove; the proof is by induction on a sequence of nonterminals derived by the rules used to create follow sets. These infer that any non-marking terminals introduced to Γ^C's follow sets would have to have been put there by a single extension, meaning that if any such terminals were in a follow set of Γ^C, one of the extensions comprising it would not have passed the *Partitionable* analysis.

Proof. Assume to the contrary that for some $nt_H \in NT_H$, $follow_C(nt_H) \setminus follow_H(nt_H)$ contains a terminal t that is not a marking terminal. Due to the constraints imposed by *Partitionable*, if this terminal t is not in $follow_H(nt_H)$, it is not in nt_H's follow set in any of the grammars Γ^{E_i} either.

Since follow sets are derived to a large degree from *first* sets, we must first consider how Γ^C's first sets are constructed. Recall that since extensions are only allowed one production with a host nonterminal on the left hand side (the bridge production) and since this will only introduce one new terminal to the host nonterminal's first set (the marking terminal), the only addition to the first set of any host nonterminal in $\Gamma^H \cup_G \Gamma_i^E$ as compared with Γ^H is the extension's marking terminal: $\forall nt_H \in NT_H. \left[first_{E_i}(nt_H) \setminus first_H(nt_H) \subseteq \left\{ \mu_i^E \right\} \right]$. Consequently, a host nonterminal will have only marking terminals added to its first set in Γ^C vis-a-vis its first set in Γ^H: $\forall nt_H \in NT_H. \left[first_C(nt_H) \setminus first_H(nt_H) \subseteq \left\{ \mu_1^E, \ldots, \mu_n^E \right\} \right]$. Furthermore, owing to the fact that bridge productions cannot make a nonterminal nullable, a nonterminal is nullable in Γ^C iff it is nullable in $\Gamma^H \cup_G \Gamma_i^E$ for some i, and (if a host nonterminal) in Γ^H.

Returning to the question of follow sets, the algorithm for follow set construction rules (see Figure 2.11 on page 39) provides that $t \in follow(nt_H)$ iff:

1. there is a production $X \rightarrow \alpha\ nt_H\ \beta\ \gamma$ such that α is a sequence of grammar symbols, β is a nullable sequence of grammar symbols and $t \in first(\gamma)$; or

2. there is a production $X \rightarrow \alpha\ nt_H\ \beta$ such that α is a sequence of grammar symbols, β is a nullable sequence of grammar terminals and $t \in follow(X)$.

Assume that statement (1) is true.

If X is a host nonterminal, then $X \rightarrow \alpha\ nt_H\ \beta\ \gamma$ is a host production (since the form precludes the possibility that it is a bridge production). Since t is not new to $first_C(\gamma)$ ($t \notin first_C(\gamma) \setminus first_H(\gamma)$), it would be in $follow_H(X)$, which is a contradiction.

If X is an extension nonterminal, $X \rightarrow \alpha\ nt_H\ \beta\ \gamma$ is still not a bridge production, and t is not new to $first_C(\gamma)$ (*i.e.*, $t \notin first_C(\gamma) \setminus first_{E_i}(\gamma)$), so the same reasoning applies.

Therefore, by elimination, statement (2) must be true: there is a production $X \rightarrow \alpha\ nt_H\ \beta$ with $t \in follow_C(X)$.

Now this is a recursive building block for follow sets — defining one follow set in terms of another. The essence of the rest of the proof is that if one in turn examines how t was placed in $t \in follow_C(X)$, and so on, one can trace t back to a first set for some symbol γ. Once it is established that t is in this first set, it can then be shown by induction that $t \in follow_H(nt_H)$ — a contradiction.

Formally unrolling the recursive definition, there exists a chain of productions of the following form for some integer k:

$X_0 \rightarrow \alpha_0 nt_H \beta_0$,

$X_{i+1} \rightarrow \alpha_{i+1} X_i \beta_{i+1}$ for $i < k-1$,

$X_k \rightarrow \alpha_k X_{k-1} \beta_k \gamma$

with $t \in first_C(\gamma)$, each β_i nullable, $t \in follow_C(X_k)$. Indeed, it follows immediately that $t \in follow_C(X_i)$ for each i. Also, since the first sets of γ did not change to admit t, then $t \in follow_{E_i}(X_k)$ (and, if $X_k \in NT_H$, in $follow_H(X_k)$ as well).

If for some pair (k_1, k_2) with $1 \leq k_1 < k_2 \leq k$ there exists some ℓ such that $X_i \in NT_H \cup NT_{E_\ell}$ for every i with $k_1 \leq i < k_2$, then the productions corresponding to the nonterminals $X_{k_1}, X_{k_1+1}, \ldots, X_{k_2}$ are in $P_H \cup P_{E_\ell}$ (all belonging only to the host or to the single extension). Therefore, if $t \in follow_{E_\ell}(X_{k_2})$, it is also in $follow_{E_\ell}(X_{k_2-1})$, and so on up to $follow_{E_\ell}(X_{k_1})$. Or, in summary, $t \in follow_{E_\ell}(X_{k_2}) \rightarrow t \in follow_{E_\ell}(X_{k_1})$. Due to the constraints imposed on the follow sets by *Partitionable*, if either X_{k_1} or X_{k_2} is a host nonterminal, H may be substituted for the respective occurrences of E_ℓ in the above.

If the entire *chain* of X nonterminals is in $NT_H \cup NT_{E_\ell}$ for some ℓ (the case in which $k_1 = 0$ and $k_2 = k$), it follows immediately that $t \in follow_{E_\ell}(X_n)$, and that $t \in follow_{E_i}(nt_H)$, therefore $t \in follow_H(nt_H)$. This is a contradiction, so there must be nonterminals from more than one extension in the X chain.

We now define two "filters" on the X chain. The first, E, will filter out any host nonterminals, leaving only extension nonterminals; the second, E', will further filter out subsequences of nonterminals from the same extensions, so the end product will be a chain of extension nonterminals in which each two adjacent nonterminals are from different extensions. For example, a chain $\left\langle nt_1^H, nt_1^{E_1}, nt_2^H, nt_1^{E_2}, nt_3^H, nt_2^{E_2}, nt_2^{E_1}, nt_3^H \right\rangle$ would be filtered first to $\left\langle nt_1^{E_1}, nt_1^{E_2}, nt_2^{E_2}, nt_2^{E_1} \right\rangle$, then to $\left\langle nt_1^{E_1}, nt_1^{E_2}, nt_2^{E_1} \right\rangle$.

Formally, E is defined as a sequence of integers such that X_{e_i} is the ith extension nonterminal in the X chain: $e_i = \arg\min_{j=e_{i-1}}^{k} X_j \notin NT_H$ (which is to say, the smallest j between e_{i-1} and k such that X_j is not a host nonterminal).

E' is then defined as a sequence of integers eliminating adjacent nonterminals in the same extension from the E sequence: $e'_i = \arg\min_{j=e'_{i-1}}^{k} \neg\exists m. \left[\left\{ X_{e_j}, X_{e'_{i-1}} \right\} \in NT_{E_m} \right]$. Note that no $X_{e'_i}$ and $X_{e'_{i+1}}$ are adjacent in the chain, since no production has a nonterminal from one extension on the left hand side and a nonterminal from another extension on the right hand side.

For every $X_{e'_i}$ and $X_{e'_{i+1}}$, if $X_{e'_i} \in NT_{E_i}$, then for every $e'_i \leq j < e'_{i+1}$, $X_j \in NT_H \cup NT_{E_i}$. Furthermore, at least one of these X_js must be in NT_H.

Now it can be shown by induction on E' that $t \in follow_H(nt_H)$, deriving a contradiction.

Induction statement. For $e'_{i-1} < j < e'_i$, it must be true that $t \in follow_H(X_j)$.

Base case. $e'_{|E'|}$. There are no nonterminals occurring after $X_{e'_{|E'|}}$ in the chain that are in a different extension. Therefore, the subsequence of nonterminals from $e'_{|E'|}$ to k are all in $NT_H \cup NT_{E_\ell}$ for some ℓ, and since $t \in follow_{E_\ell}(X_k)$, this means that $t \in follow_{E_\ell}(X_{e'_{|E'|}})$, and in $follow_H(X_{e'_{|E'|}})$ if $X_{e'_{|E'|}} \in NT_H$.

Step case. Follows the same pattern of reasoning as the base case with e'_i in the place of $e'_{|E'|}$ and e'_{i+1} in the place of k.

□

Lemma 6.4.4. The class of conflict-free states is closed under the introduction of bridge items and marking terminal lookahead. If a state n_H compiles to a conflict-free parse table row and $n_H \equiv_1^C n_C$, then n_C also compiles to a conflict-free parse table row.

Proof summary. Marking terminals only occur in one production, the bridge production $nt_H \rightarrow \mu_i^E s_i^E$. Therefore, any first set containing μ_i^E is also guaranteed to contain all the members of $\mathit{first}(nt_H)$ and any possible conflicts would also occur on all these terminals.

Proof. Adding a bridge item $nt_H \rightarrow \bullet\mu_i^E s_i^E$ to n_H to make n_C will cause an additional shift action to be placed in cell $\left(n, \mu_i^E\right)$ of the derivative parse table. Adding a marking terminal as lookahead to some item in $\mathit{items}_H(n)$ will possibly cause an additional reduce action to be placed in the parse table, also in cell $\left(n, \mu_i^E\right)$.

As can be seen, the only conflicts that arise from either of these two additions occur in the cell $\left(n, \mu_i^E\right)$.

Assume that $\left(n_C, \mu_i^E\right)$ contains a shift-reduce conflict. This means that there is some item $nt_H \rightarrow \bullet\mu_i^E s_i^E \in \mathit{items}_C(n_C)$ introducing a shift action into the cell and some item $it \in \mathit{items}_C(n_C)$ with $\mu_i^E \in la_{n_C}(it)$ introducing a reduce action.

However, $\mu_i^E \in \mathit{first}_C(nt_H)$, and its membership in any other nonterminal's first set is on account of its membership in $\mathit{first}_C(nt_H)$. Therefore, any lookahead set containing μ_i^E also contains the other members of $\mathit{first}_C(nt_H)$, *viz.*, the members of $\mathit{first}_H(nt_H)$.

Also, the presence of the item $nt_H \rightarrow \bullet\mu_i^E s_i^E$ implies (according to the LR DFA closure rule laid out in Figure 2.12 on page 40) that $\mathit{items}_C(n_C)$ also contains an item

$nt_H \to \bullet\alpha\beta$ for every $\alpha \in \mathit{first}_H(nt_H)$, where β is a sequence of zero or more grammar symbols. On a more abstract level, this is because $nt_H \to \bullet\mu_i^E s_i^E$ only appears in states where any construct deriving from nt_H may begin, which makes nt_H's entire first set valid lookahead set.

Therefore, if there is a shift-reduce conflict in cell $\left(n, \mu_i^E\right)$ there is also a shift-reduce conflict in every cell (n, α) with $\alpha \in \mathit{first}_H(nt_H)$, which is a contradiction as this would also cause n_H to compile with shift-reduce conflicts.

A reduce-reduce conflict compiled from n_C, similarly, would imply a reduce-reduce conflict in cells (n, α) with $\alpha \in \mathit{first}_H(nt_H)$. □

Proof of Theorem 6.4.1.

Proof. This proof works from the same starting point discussed on page 158: the arbitrary host language and set of extensions passing *Partitionable*, and the LR DFA M^C built from the composed grammar Γ^C.

We first give precise definitions of the $n+2$ partitions (M_H^C, $M_{E_i}^C$ for each extension i, and M_{NH}^C) mentioned above, then argue that the states of M^C can be separated into these partitions and that no state in M^C gives rise to a conflict.

The partition M_H^C. This partition contains any states n such that:

1. n is not in any of the partitions $M_{E_i}^C$, and
2. There exists a path from s_C (the start state of M^C) to n that does not pass through any states in any $M_{E_i}^C$ partition:

$$\exists \{n_0, (n_1, \sigma_1), \ldots, (n_k, \sigma_k)\}.$$
$$\left[n_0 = s_C \wedge n_k = n \wedge \delta_C(n_{i-1}, \sigma_i) = n_i \wedge \forall i. \forall j \in [1,k]\,.\left[n_j \notin M_{E_i}^C\right]\right]$$

Since each $\delta_C(n_{i-1}, \sigma_i)$ points to a state in M_H^C, each σ_i is a host symbol: $\sigma_i \in T_H \cup NT_H$ for each i.

Therefore, in each LR DFA M^{E_i}, the same path $\langle \sigma_1, \ldots, \sigma_n \rangle$ could be followed from its start state s_{E_i} and reach a state $n' \equiv_1^C n$.

This follows by construction, since host syntax is common to all such DFAs, and any states along the same path of host-only transitions must have identical sets of host-only items and host-only lookahead. Non-marking-terminal extension lookahead is also proscribed as contradicting the *Partitionable* analysis's constraints on host nonterminals' follow sets.

By Lemma 6.4.4, whereas the n' states are free of conflicts, so is n.

The partitions $M_{E_i}^C$. These partitions contain those states n such that $\exists (nt \to \alpha \bullet \beta) \in items_C(n).[nt \in NT_{E_i}]$: any states having an item with an extension nonterminal on the left hand side. By Lemma 6.4.2, the partitions $M_{E_i}^C$ are all disjoint.

The bridge production for Γ_i^E is $nt_H \to \mu_i^E s_i^E$. By construction, the only paths to any state $n \in M_{E_i}^C$ from M^C's start state s_C must run through the state containing the item $nt_H \to \mu_i^E \bullet s_i^E$, which was labeled $n_{E_i}^S$ above. Furthermore, there is at least one such path (since no DFA states are isolated) and at least one such path is acyclic (by definition).

Now if there is some state n_I on a transition path between $n_{E_i}^S$ and n such that n_I is *not* in the partition $M_{E_i}^C$, it follows that $n_{E_i}^S$ is on the path between n_I and n, making the path cyclic. Therefore, on the acyclic path, all states between $n_{E_i}^S$ and n are in $M_{E_i}^C$.

By construction, the transition symbols on this path are all in $T_H \cup NT_H \cup T_{E_i} \cup NT_{E_i}$: if one were not, this would mean that syntax from Γ_j^E was in the state preceding it, contradicting Lemma 6.4.2. Therefore, the only way into the block of states comprising $M_{E_i}^C$ is through the state $n_{E_i}^S$.

Consider the properties of $n^S_{E_i}$. Its item set consists of the closure of the item $nt_H \rightarrow \mu^E_i \bullet s^E_i$. There is also a state n_0 in $M^{E_i}_{E_i}$ seeded from the same state. In the case of both n_0 and $n^S_{E_i}$, there are transitions to it from every state, in M^{E_i} and M^C respectively, such that it contains an item $nt_H \rightarrow \bullet....$ It follows that the lookahead on the item $nt_H \rightarrow \mu^E_i \bullet s^E_i$ in both states is exactly the follow set of nt_H: $la_{n_0}(nt_H \rightarrow \mu^E_i \bullet s^E_i) = follow_{E_i}(nt_H)$ and $la_{n^S_{E_i}}(nt_H \rightarrow \mu^E_i \bullet s^E_i) = follow_C(nt_H)$. By Lemma 6.4.3, $follow_C(nt_H) \setminus follow_H(nt_H) \subseteq \{\mu^E_1, \ldots, \mu^E_n\}$; hence $n_0 \equiv^C_1 n^S_{E_i}$ and by Lemma 6.4.4, $n^S_{E_i}$ produces no conflicts. By straightforward structural induction on the non-marking-terminal transitions out from $n^S_{E_i}$, one can establish this for the entire partition as well.

The partition M^C_{NH}. This partition contains any states that are not in M^C_H or any $M^C_{E_i}$, which by elimination means states that have no item with an extension nonterminal on the left hand side, but that are not connected to M^C's start state except by paths through one of the extension start states $n^S_{E_i}$.

Let $Contrib : M^C_{NH} \rightarrow \mathscr{P}(\{\Gamma^E_1, \ldots, \Gamma^E_n\})$ represent, for each state n in $M^C_{E_i}$, the set of extension grammars Γ^E_i such that there exists a transition path from the extension's start state $n^S_{E_i}$ to n that does not pass through any extension's start state. Then:

- $|Contrib(n)| \geq 1$, since if the set were empty the LR DFA would be unconnected.
- There are no marking terminals on the path from $n^S_{E_i}$ to n (since the state pointed to by any marking terminal transition is always an extension start state).
- The path does not pass through any state $n_H \in M^C_H$, since all transitions on non-marking terminals out of n_H point to another state in M^C_H, which by induction would place n in M^C_H as well.

It is now established that, for each $\Gamma^E_i \in Contrib(n)$, there is a path from $n^S_{E_i}$ to n along transitions labeled only with terminals in $T_H \cup NT_H \cup T_{E_i} \cup NT_{E_i}$. By construction, this

means that there is an identical path in M^{E_i} from the extension start state to a state $n_i \equiv_0^C n$.

n_i is, by definition, not in $M_H^{E_i}$ or $M_{E_i}^{E_i}$, so it must be in $M_{NH}^{E_i}$. By construction, n, constructed by merging all such states n_i in the traditional LALR(1) manner, contains exactly all the lookahead from all the states n_i.

By the constraints of the *Partitionable* analysis, there is some state $n_i' \in M_H^{E_i}$ such that $n_i \subseteq_{\mathrm{I}} n_i'$, and furthermore that for all such states, $n_i \subseteq_{\mathrm{IL}} n_i'$.

Now if a bridge item is added to a state, it will also be added to all I-supersets of that state. Therefore, since bridge items are the only difference between the item sets of any of the states n_i, the space of I-supersets n_i' mentioned above will correspond exactly to the same set of states in M^H for each i (states that are identical to the n_i' states but shorn of bridge items and marking terminal lookahead).

Now the IL-subset relation is closed under union, *i.e.*, if a state consists of the union of the items and lookahead sets of several IL-subsets of the same state, it will also be an IL-subset of that state.

Consider a hypothetical state consisting of n shorn of all its bridge items and marking terminal lookahead. This state is an IL-subset of each member of the set of states in M^H mentioned above, establishing that it is conflict-free, which makes n conflict-free as well according to Lemma 6.4.4. □

We have now shown that if each of the machines M^{E_i}, compiled from extension grammars composed individually with the host, can be partitioned as shown in Figure 6.1(a), then M^C, compiled from all the extensions, can be partitioned as shown in Figure 6.1(c), meaning that the parse table generated from M^C will be free of parse table conflicts.

6.5 Grammar examples.

Several grammars that are practical applications of the modular determinism analysis are discussed at a high level in chapter 9 and listed in appendix A. In this section, we discuss, on the low level, grammars and grammar fragments illustrating each of the several ways an extension can fail the modular determinism analysis.

In this discussion we will quote rules and symbols from the grammars in appendix A. Here, as there, we use different fonts to denote host and extension symbols, terminals and nonterminals: ***host nonterminal***; **extension nonterminal**; *host terminal*; extension terminal.

In Copper's implementation of the analysis, failures are characterized in a manner intended to maximize an extension writer's intuitive understanding of what is wrong. To this end they are grouped into four categories: formal mismatches, lookahead spillage, follow spillage, and non-IL-subset conditions.

A formal mismatch is where an extension does not fit the form laid out in Definition 6.1.1, with a bridge production prefixing extension constructs with a unique marking terminal.

Lookahead spillage is where an extension introduces new non-marking-terminal lookahead into host DFA states, *i.e.*, there is a state n_E in M^{E_i} and a state n_H in M^H such that $n_E \equiv_0^C n_H$ but $n_E \not\equiv_1^C n_H$, so that n_E does not fit into the partition $M_H^{E_i}$; the lookahead spillage is then the new non-marking-terminal lookahead. To eliminate lookahead spillage, extension writers would look at the state with the spillage and trace it back to some reference their extension grammar made to a host nonterminal, which can then be altered as necessary.

Follow spillage is where an extension does not meet the analysis's restrictions on follow sets, where for some $nt \in NT_H$, $follow_{E_i}(nt) \setminus follow_H(nt) \not\subseteq \{\mu_i^E\}$; the follow

spillage is then those non-marking terminals added to *nt*'s follow set. Except for a few corner cases (discussed below), an extension with follow spillage usually has lookahead spillage as well.

A non-IL-subset condition is where there is a state n_{NH} in M^{E_i} that matches the first two conditions for admission into $M^{E_i}_{NH}$, but does not match one or both of the last two conditions. Like lookahed and follow spillage, these are resolved by examining the state in $M^{E_i}_{NH}$ that is not an IL-subset and tracing back into the extension.

6.5.1 Formal mismatch.

A simple example of a formal mismatch is an extension adding Java 1.5-style `foreach` loops to Java 1.4. Such an extension would extend Java by adding a rule so the Java nonterminal ***statement*** would derive *For Lparen* ***type*** *Id Colon* ***expression*** *Rparen* ***statement***. But the keyword *For*, beginning that construct, is not a unique marking terminal, so this is a formal mismatch.

If a new keyword is substituted for *For*, the formal mismatch is resolved. A version of the extension with such a new keyword is provided in appendix A.3.2.3.

6.5.2 Lookahead and follow spillage.

As lookahead spillage and follow spillage usually occur together, we will first consider them together, addressing them in isolation only as special cases.

The example we use is one in which lookahead and follow spillage do *not* occur: the boolean tables extension to ableJ, discussed in section 9.4.4 and appendix A.3.2.5. In that extension, there is only one back reference to the host grammar, when a row of the table is begun by a Java expression:

TableRow → ***expression*** *Colon* **TruthValueList**

This means that ***expression*** will have the terminal *Colon* added to its follow set, and to lookahead sets of items for its productions. Furthermore, the transitions on terminals in *first*(***expression***), such as *New*, will point back to states in $M_H^{E_i}$, adding *Colon* to lookahead sets there.

But *Colon* is already in the follow set of ***expression*** and the associated lookahead sets, as the conditional expression (`a ? b : c`) already provides for colons following expressions:

$$\textbf{\textit{ConditionalExpr}} \rightarrow \textbf{\textit{LogicalOrExpr}}\ \textit{Question}\ \textbf{\textit{Expr}}\ \textit{Colon}\ \textbf{\textit{ConditionalExpr}}$$

Hence, the use of *Colon* causes no lookahead or follow spillage in this extension.

For an example of when lookahead and follow spillage do occur, we will modify this extension grammar slightly, by replacing the colon with a new extension terminal t_E. With t_E in the place of the colon, t_E would be added to the follow set of ***expression*** and to the related lookahead sets, causing both follow spillage on ***expression*** and lookahead spillage in the states of $M_H^{E_i}$ used to parse Java expressions.

6.5.3 Lookahead spillage without follow spillage.

Lookahead spillage can occur without follow spillage when an extension introduces lookahead to host constructs in a new context.

For example, take the following grammar and extension.

Host productions:

$$\boldsymbol{S} \rightarrow \boldsymbol{A}\ a \mid b\ \boldsymbol{B}$$
$$\boldsymbol{A} \rightarrow c$$
$$\boldsymbol{B} \rightarrow \boldsymbol{A}\ c \mid c$$

Extension with bridge production $\boldsymbol{S} \rightarrow \mu\ \mathbf{E}$:

$\mathbf{E} \rightarrow \boldsymbol{A}\ c$

The host grammar makes use of the nonterminal $\boldsymbol{A}$ in two distinct contexts; firstly, at the start of a construct derived from $\boldsymbol{S}$ followed by the terminal a; secondly, at the end of a construct derived from $\boldsymbol{S}$, preceded by the terminal b and followed by the terminal c.

In the former context, shifting a terminal c will cause the parser to enter a state where the only valid action, on a, is to reduce on $\boldsymbol{A} \rightarrow c$. But in the latter, shifting c will cause the parser to enter a state where there are two valid actions: on end-of-input, \$, it will reduce on the production $\boldsymbol{B} \rightarrow c$, while on c it will reduce on the production $\boldsymbol{A} \rightarrow c$.

The extension adds a new reference to $\boldsymbol{A}$, which is followed by c. c is already in $\boldsymbol{A}$'s follow set, so there is no follow spillage caused by this back reference. But $\boldsymbol{A}$ is by itself in that context, so shifting c will cause the parser to move, not to the state where $\boldsymbol{B} \rightarrow c$ is also valid, but to the state where the only valid action is to reduce on $\boldsymbol{A} \rightarrow c$. Hence, c is added to the lookahead set of the item $\boldsymbol{A} \rightarrow c\bullet$ in that state, previously $\{a\}$; this constitutes lookahead spillage.

6.5.4 Follow spillage without lookahead spillage.

At first glance, it might appear that follow spillage never occurs without lookahead spillage also occurring. This would mean that the partitioning restrictions of the analysis *Partitionable* make its follow-set restrictions unnecessary; as follow sets can be constructed through union of lookahead sets, restrictions on lookahead sets, such as are a component of the paritioning restrictions, should guarantee that the follow sets do not change.

However, lookahead sets on host-language items within the partition $M_{E_i}^{E_i}$ are not

restricted by the analysis. Therefore, it is possible in some cases to have follow spillage without any lookahead spillage or non-IL-subset conditions; we present such a case in this section.

Host productions (start symbol is $\boldsymbol{A}$):

$$\boldsymbol{A} \rightarrow a \mid \boldsymbol{B}$$
$$\boldsymbol{B} \rightarrow b$$

Extension with bridge production $\boldsymbol{A} \rightarrow \mu\ \mathbf{E}$:

$$\mathbf{E} \rightarrow b\ \mathrm{c} \mid \boldsymbol{B}\ \mathrm{d}$$

Clearly, the production $\mathbf{E} \rightarrow \boldsymbol{B}$ d will result in follow spillage on $\boldsymbol{B}$. But there will be no lookahead spillage, as all the items with d as lookahead are in the partition $M_{E_i}^{E_i}$.

6.5.5 Non-IL-subset conditions.

As with the two sorts of spillage, it is rare to have a non-IL-subset condition that does not have the same cause as some lookahead spillage. The only practical example of this we have located to date is the ableJ extension for dimension types, discussed in section 9.4.5 and appendix A.3.2.6. In this section, we provide a simpler example.

Host productions:

$$\boldsymbol{S} \rightarrow a\,\boldsymbol{A} \mid b\,\boldsymbol{B}$$
$$\boldsymbol{A} \rightarrow c\ x$$
$$\boldsymbol{B} \rightarrow c\ y$$

Extension with bridge production $\boldsymbol{S} \rightarrow \mu\ \mathbf{E}$:

$$\mathbf{E} \rightarrow \boldsymbol{A} \mid \boldsymbol{B}$$

All items in the LR DFA for this grammar have the same lookahead set: the end-of-input $. Hence, the extension introduces no lookahead or follow spillage.

In the host grammar there is only one reference each to the nonterminals $\boldsymbol{A}$ and $\boldsymbol{B}$, the former being preceded by the terminal a, the second by the terminal b. This means that in the LR DFA for the host grammar, parsing of $\boldsymbol{A}$ and $\boldsymbol{B}$ takes place in strictly different parse states.

However, in the extension, **E** derives both $\boldsymbol{A}$ and $\boldsymbol{B}$. This means that in the start state for the extension there are two items, $\boldsymbol{A} \rightarrow \bullet c\ x$ and $\boldsymbol{B} \rightarrow \bullet c\ y$. Shifting a terminal c in this state will transition the parser into a state containing two items, $\boldsymbol{A} \rightarrow c \bullet x$ and $\boldsymbol{B} \rightarrow c \bullet y$. As this state contains only host-language items it cannot be part of $M_{E_i}^{E_i}$. But since it is parsing A and B in the same context, it is not an I- or IL-subset of any state in the LR DFA for the host grammar, where these constructs are parsed in separate parse states. This creates a non-IL-subset condition.

6.6 Lexical disambiguation.

In the previous sections, we have shown that the analysis *Partitionable* ensures a lack of parse table conflicts in the parser constructed from a composed language. We now consider the problem of *lexical* conflicts or ambiguities in that language. Context-aware scanning automatically resolves most of the scanner issues attendant with compiling a composed grammar, but other methods are needed for a few specific classes of conflict.

6.6.1 What context-aware scanning does and does not solve.

The analysis above focuses on the context-free part of the syntax. With a traditional scanner, most of it would be difficult to turn to an advantage, since terminals from the different extensions will all be valid for scanning in all contexts, and terminals from

separate extensions that do not cause lexical conflicts with the host terminals might still conflict with each other.

Context-aware scanning automatically resolves most of these lexical issues on account of Lemma 6.4.2. Since any state in the LR DFA M^C contains items from at most one extension, excepting bridge items and marking terminal lookahead, valid lookahead sets will not contain non-marking terminals from more than one extension, which means that any lexical conflicts occurring in M^C that do not involve marking terminals would also have been detected when building M^H or M^{E_i} and resolved by the extension writer.

The assumption that the scanners for M^H and M^{E_i} do not contain any lexical conflicts guarantees the absence of most sorts of lexical issues, including conflicts between an extension's marking terminal and non-marking terminals from the same extension. However, there are two sorts of lexical conflicts that may still appear.

Lexical conflicts between two marking terminals. These occur when two marking terminals μ_i^E and μ_j^E have overlapping regular expressions[3] and both marking terminals show up in the first set of the same nonterminal (*e.g.*, if the bridge productions for the two extensions share a left hand side).

Lexical conflicts between a marking terminal and a non-marking terminal from another extension. Since it is possible for the marking terminal of any extension to be valid lookahead in any state belonging to any other extension, this sort of conflict is also possible.

With regard to Figure 1.1 on page 9, it was noted that two extensions to Java, the SQL extension and the tables extension, both used a keyword `table`. Since the SQL

[3] If, as is usual, the marking terminals' regular expressions match only one string, this would mean they have the same regular expression.

extension's `table` never occurs in the same context as the beginning of a Java expression (the valid context for the tables extension's marking terminal) such a problem does not come up here, but if it were, there would be a conflict since both `table` keywords would be in the same valid lookahead set.

6.6.2 Marking terminal disambiguation by transparent prefix.

The two classes of lexical conflict enumerated above are impossible for any of the extension writers to resolve independently. Hence, these conflicts must be resolved by the non-expert programmer who composes the extensions, and there should be a simple and automatic way of resolving them.

One solution is to use transparent prefixes, in a fashion reminiscent of using a fully qualified name in Java to provide an unambiguous class name: one simply uses a transparent prefix to provide the "fully qualified name" of the marking terminal.

This strategy can be illustrated best by example. In our framework we intend for each extension grammar to have a name patterned after the Java package hierarchy (based on the unique domain name of the extension writer). For example, our SQL extension to Java has the name `edu:umn:cs:melt:ableJ:exts:sql`, while the tables extension is named `edu:umn:cs:melt:ableJ:exts:tables`. The grammar name of the extension, bookended with colons, would be given as the transparent prefix of the extension's marking terminal.[4]

Then, to continue the example, in the event that there was a lexical conflict on the keyword `table`, the programmer would instead type `:edu:umn:cs:melt:ableJ:exts:tables:table`. The scanner would read the transparent prefix, limit the valid lookahead set to the keyword `table` from the tables extension, and then scan the lexeme

[4] In practice, these extensions would likely come from different sources utilizing different Internet domains, meaning that there would be more significant differences to their grammar names and, hence, transparent prefixes.

`table` as that.

Default behavior (when a prefix is not provided). There is some leeway as to the default behavior — what should be done to resolve any lexical conflicts when a transparent prefix is not given by the programmer. This, however, can be resolved automatically by the extension writer or even the programmer, by giving each marking terminal one of the following three labels:

- *"Reserve against other terminals."* If a lexical conflict occurs between a marking terminal μ_i^E and another set of terminals X, immediately form a new precedence relation: $t \prec \mu_i^E$ for each $t \in X$. The use of this option should be avoided (see section 6.6.3).

- *"Prefer over other terminals."* If a lexical conflict occurs between μ_i^E and some set of terminals X in this case, a disambiguation function will be created for $X \cup \{\mu_i^E\}$ that returns μ_i^E. This still allows the marking terminal's lexeme to be matched by the member(s) of X in other contexts.

- *"Avoid in favor of host terminals."* If a lexical conflict occurs between μ_i^E and X in this case, a disambiguation function will be created that (if $X = \{t\}$) returns t, and (if $|X| \geq 2$) performs the same disambiguation action as was previously set to be performed on X. (N.B.: If this option is used, the only way to get the scanner to match μ_i^E is to provide the transparent prefix.)

6.6.3 Issues with lexical precedence.

If $t \prec_{PT} u$ for some pair of terminals t and u, then no lexeme $w \in L(regex_{PT}(u))$ can be matched to t in *any* context, even if u is not in the valid lookahead set. The most

common application of this is when t is an identifier and u is a keyword, and then the identifier terminal cannot match the keyword's lexeme and the keyword is reserved.

Now if $t \in T_H$ and $u \in T_{E_i}$ for some i, then composing Γ_i^E has effectively changed the language of Γ^H. If t is used in another extension Γ_j^E, no lexeme matching u will be able to be matched to t, even in the context of a state in $M_{E_j}^C$.

This is not a lexical conflict, but it does cause Γ_j^E to be parsed differently than if Γ_i^E had not been included in the composed grammar. It is therefore discouraged in our framework (although not expressly prohibited) for extension writers to define lexical precedence relations between host terminals and extension terminals.

6.7 Using operator precedence.

The use of operator precedence (resolving shift-reduce conflicts by the use of precedence rules) may cause trouble for the analysis in certain corner cases.

The lynchpin of the proof of Lemma 6.4.4 is that any parse table conflict that would occur in a parse table cell labeled by a marking terminal would also occur in the cells labeled by other members of $\mathit{first}_C(nt_H)$, where nt_H is the host nonterminal on the left hand side of the bridge production.

In most cases, this reasoning works; however, on the occasion that the cells of *all* other members of $\mathit{first}_C(nt_H)$ contained shift-reduce conflicts that were removed through the process of operator precedence, this reasoning fails and there could well be a conflict in the cell of a marking terminal.

The solution to this problem is to adopt a slightly more complex and stringent analysis. Let $\mathit{Interlopers}_j \subseteq NT_H$ refer to the set of host nonterminals contributing lookahead to any state in $M_{E_i}^{E_i}$ (*i.e.*, if a bridge production $nt_H \rightarrow \mu_j^E s_j^E$ is added, μ_j^E will show up as lookahead in a state in $M_{E_i}^{E_i}$ iff $nt_H \in \mathit{Interlopers}_j$). Then define a bijection supplying

a fresh new marking terminal to every member of $Interlopers_j$

$$mark \ : \ \left\{ \mu_1^\star, \ldots, \mu_{|Interlopers_j|}^\star \right\} \rightarrow Interlopers_j$$

and compile a new grammar consisting of $\Gamma^H \cup_G \Gamma_i^E$ composed with a set of productions $mark(\mu_i^\star) \rightarrow \mu_i^\star$.

This simulates every possible insertion of a marking terminal into the language by other extensions, with no interference from operator precedence declarations. If this can be done without raising a conflict, the validity of the proof is restored.

6.8 Time complexity analysis and performance testing.

Copper's implementation of the modular determinism analysis works by building two LR DFAs (M^H and M^{E_i}) from the analysis and comparing them. The complexity of comparing them (polynomial) is eclipsed by that of building them (potentially exponential). Therefore, the modular determinism analysis should take at most twice the time needed to build a parser for $\Gamma^H \cup_G \Gamma^{E_i}$.

To test this in practice, the modular determinism analysis simply needed to be run on extensions of varying sizes; we ran it on a number of Java extensions (both the ones that pass the analysis and the ones that fail it). See Figure 6.2 on the following page for the results, which showed that on a grammar Γ_i^E for which Copper took n seconds to compile $\Gamma^H \cup_G \Gamma_i^E$, the modular determinism analysis consistently ran between n and $2n$ seconds. There was one exception, in which the modular determinism analysis ran *faster* than the compilation.

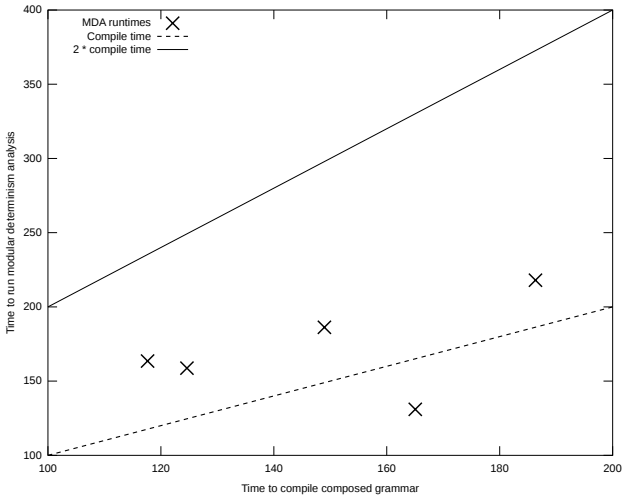

Figure 6.2: Runtimes of the modular determinism analysis.

6.9 Discussion.

In this chapter we have presented an analysis, *Partitionable*, embodying a set of restrictions verifying that any set of passing grammar extensions will compose without conflict.

At first glance, this may not seem very groundbreaking — imposing some restrictions on grammars to ensure they will compose conflict-free. Indeed, we ourselves experimented with tighter sets of restrictions that made the same guarantee — including a complete proscription on back references to the host grammar inside extensions. However, there are two factors that make this analysis a significant contribution.

The first factor is the wide range of extensions that pass the analysis, which includes many practical, pre-existing extensions that are discussed further in chapter 9. Each of our previous restrictions excluded a great many of these. The wide range of this analysis is partially due to the fact that if a host language is large and syntactically rich, although the probability that the language is conflict-free might be decreased, the probability that

extensions to it will pass the analysis is actually increased (larger follow and lookahead sets lead to fewer instances where an extension introduces new symbols to these sets).

The second factor is its broader implications for the field of extensible compilers, discussed further in section 10.1.2. The parsing stage has largely served as one of the last major roadblocks to the development of an extensible compiler where all language extensions are imported by the end user. However, with the aid of a context-aware scanner to handle the differences in lexical syntax between host and extensions in a systematic fashion, the modular determinism analysis is a step towards removing this roadblock.

Chapter 7

Parse table composition.

This chapter presents an extension or corollary of the modular determinism analysis presented in the previous chapter. The corollary (originallly presented in our paper *Verifiable Parse Table Composition for Deterministic Parsing* [SVW10]), instead of merely verifying that various extensions can be composed as grammars, enables the construction of separate parse table modules for each extension, which can then be composed quickly with the host parse table on-the-fly; an analogy is dynamic linking of libraries as opposed to static linking.

7.1 Statement of the problem.

One notable feature of the modular determinism analysis presented above is that it concerns only grammars, and not parsers: *i.e.*, once the analysis is completed and all extension grammars have been found to be inside the class *isComposable*, the grammar $\Gamma^H \cup^\star_G \{\Gamma^E_1, \ldots, \Gamma^E_n\}$ must be compiled into a parser from the ground up. There are three ways that this requirement could be a problem:

1. If the writer of the host grammar or an extension does not want the "source code"

of that grammar made available to all the end-user programmers. (Although concealing the *grammar* itself may not be a common practice, a parser will in practice contain blocks of code as semantic actions, which must also be revealed to the users for a parser to be built from the ground up.)

2. In applications where there are time constraints on building the composed parser (*e.g.*, if the parser is to be built at compiler runtime) and the potentially exponential process of building an LR DFA is insufficiently fast.

3. An LR DFA is a monolithic construct, and the analysis *Partitionable* must be more restrictive to ensure that a parser for $\Gamma^H \cup^{\star}_G \left\{\Gamma^E_1, \ldots, \Gamma^E_n\right\}$ can be built from the ground up, than it must be to ensure that, in general, an LALR(1) parser can be built for that grammar.

Fortunately, since the analysis *Partitionable* guarantees the partitioning of the composed LR DFA M^C into separate partitions for each extension, the derivative parse table is also ensured to be partitioned in the same manner; this offers a fairly straightforward basis for a procedure of parse table composition, in which parse tables pre-compiled by the individual extension writers — along with a small amount of additional metadata — are distributed to the end users and assembled rapidly into full parsers for the composed language, which in essence are just the individually compiled parse table pieces concatenated into a single table.

The outline of the rest of this chapter is as follows. Firstly, we define a relaxed version of the modular determinism analysis, $\mathit{Partitionable}_{PT}$, which guarantees that any set of passing extensions can have their parse tables composed correctly (though it is important to note that the original modular determinism analysis *Partitionable* also provides this guarantee). Secondly, we define the operation $\cup_T$ by which the parse table pieces are composed. Thirdly, we discuss the problem of building scanners for the

composed parse tables, and the ways of bundling the necessary scanners with the parse table pieces for each extension. Finally, we discuss two different ways to implement $\cup_T$ and analyze its time complexity.

7.2 The modified analysis *Partitionable*$_{PT}$.

The modular determinism analysis *Partitionable* is perfectly suitable for use in verifying parse table composition; *i.e.*, any set of extensions passing the analysis is guaranteed to compose without errors if the method described below is used. Those seeking to distribute extensions in both grammar and parse table form may therefore use *Partitionable* to guarantee composability in both cases.

However, the parse table composition operation couples its component parse table fragments more loosely than the grammar composition operation couples its component grammars; thus, some of the conditions needed to ensure grammar composability are not necessary to ensure parse table composability, and *Partitionable* may be relaxed to form a new analysis, *Partitionable*$_{PT} \supseteq$ *Partitionable*.

The exact nature of these relaxations is as follows. There are two ways to build an LALR(1) DFA corresponding to a given context-free grammar:

1. Build an LR(0) DFA for the grammar, then annotate it with lookahead as specified by the *Closure* and *Goto* rules (see Definitions 4.4.2 and 4.4.3 on page 110).

2. Build an LR(1) DFA for the grammar, then merge each set of LR(0)-equivalent states within it, the lookahead sets of the new states being the union of those in the states from which they were merged.

It is because of this potential for state merging that the restrictions on the state partitions $M_{NH}^{E_i}$ are needed. It is possible for some $n_i \in M_{NH}^{E_i}$ and $n_j \in M_{NH}^{E_j}$ to be LR(0)-equivalent

(*e.g.,* if the two extensions each derive the same host-language nonterminal in a different context from that in which it is derived in the host language). In this case, they will be merged when the composed grammar is compiled into the LR DFA M^C, meaning that the analysis *Partitionable* must ensure that such a merger will not cause a parse table conflict.

On the other hand, if these states are part of separate parse table pieces pre-built by their respective extension writers, they will not ever be merged and can be regarded, as the states in $M^{E_i}_{E_i}$ are, as unique to one extension. Therefore, when applied to extension grammars that are to be composed by merging parse tables, the analysis *Partitionable* can be modified to remove the constraints on the states in $M^{E_i}_{NH}$, leaving the constraints on $M^{E_i}_{H}$ and $M^{E_i}_{E_i}$ guaranteeing that the host and extension sections of the composed LR DFA will remain separate and hence a parse table composed in post-process will be functionally identical to one compiled from the grammar $\Gamma^H \cup^{\star}_G \{\Gamma^E_1, \ldots, \Gamma^E_n\}$.

Definition 7.2.1. $M^{E_i}_D$.

> Let $M^{E_i}_D$ represent a partition of M^{E_i} that has the same criteria as $M^{E_i}_{NH}$ (from Definition 6.3.9 on page 156) less the last two (the IL-subset conditions). Explicitly, $n_E \in M^{E_i}_D$ iff:
>
> 1. $n_E \notin M^{E_i}_H$;
> 2. $\forall (nt \rightarrow \alpha) \in items(n_E).[nt \in NT_H]$.

See Figure 7.1 on the next page for a block diagram along the lines of Figure 6.1(c), illustrating the difference between the sets *Partitionable* and $Partitionable_{PT}$. In Figure 6.1(c), transitions out of an extension partition ($M^C_{E_i}$ or $M^C_{E_j}$) can lead either to the partition M^C_H representing the original host machine, or to a common partition $M^{E_i}_{NH}$ representing any state parsing host constructs that does not fit in M^C_H. In Figure 7.1, by

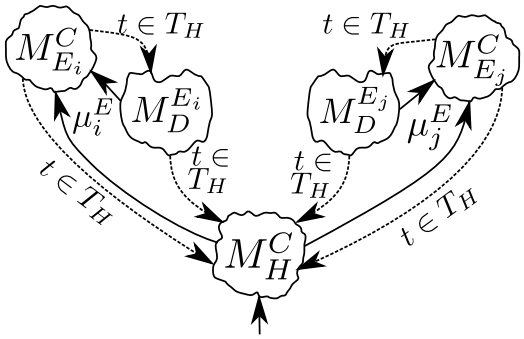

Figure 7.1: Block diagram of parse table for $\Gamma^H \cup^\star_T \{\Gamma^E_i, \Gamma^E_j\}$, by way of analogy with Figure 6.1(c).

contrast, instead of transitions leading into a common partition $M^{E_i}_{NH}$, there are transitions leading into separate partitions $M^{E_i}_D$ and $M^{E_j}_D$ — had the parser been recompiled from the grammar, the states of these partitions would have been merged to create $M^{E_i}_{NH}$.

Now that an explanation has been given of why the modular determinism analysis may be changed in this instance, we will define the relaxed analysis $\mathit{Partitionable}_{PT}$.

Definition 7.2.2. $\mathit{Partitionable}_{PT}$.

Let $\mathit{Partitionable}_{PT} \subseteq \left(\Gamma^H, \Gamma^E\right)$[1] represent the parse table version of the modular determinism analysis. It only differs from *Partitionable*, as laid out in Definition 6.3.10, in that each state must be able to be placed in one of $M^{E_i}_H$, $M^{E_i}_{E_i}$ or $M^{E_i}_D$, instead of $M^{E_i}_H$, $M^{E_i}_{E_i}$ and $M^{E_i}_{NH}$.

Explicitly, $\left(\Gamma^H, \Gamma^E_i\right) \in \mathit{Partitionable}_{PT}$ iff:

- $\forall nt \in NT_H . \left[\mathit{follow}_{E_i}(nt) \setminus \mathit{follow}_H(nt) \subseteq \left\{\mu^E_i\right\}\right]$;
- $\forall n \in \mathit{States}_{M^{E_i}} . \left[n \in M^{E_i}_H \cup M^{E_i}_{E_i} \cup M^{E_i}_D\right]$.

[1] Note that this analysis $\mathit{Partitionable}_{PT}$ is called $\mathit{isComposable}_{PT}$ in [SVW10]. We have changed the name here because we feel it is more accurate if $\mathit{isComposable}_{PT}$ refers to *all* grammars whose parse table fragments may be composed error-free rather than only those that pass this specific analysis.

There is also a modified definition of *isComposable*, $isComposable_{PT}$, which is the same as *isComposable* as laid out in Definition 6.2.1 on page 151 except that $\cup_G$ is replaced with $\cup_T$, the parse table compilation operation, which is defined in the next section.

7.3 The parse table composition operation $\cup_T$.

It was stated above that the parse table composition operation $\cup_T$ was a straightforward concatenation. In this section, the details of this operation are expounded.

We first define various fragments of the parse tables that will be composed into the parse table PT. $PT^H = \langle \Gamma^H, States_H, s_H, \pi_H \rangle$ is the complete parse table for the host grammar Γ^H, or the fragment corresponding to the partition M_H^C (the modular determinism analysis guarantees that these will be identical — all differences residing in the marking terminal columns, which are not part of PT^H). It is represented by the striped section in Figure 7.2 on the next page, which is discussed further below.

PT_i^E is the parse table fragment corresponding to an extension Γ_i^E. It consists of those rows in the parse table built from $\Gamma^H \cup_G \Gamma_i^E$ that correspond to the partitions $M_{E_i}^{E_i}$ and $M_D^{E_i}$ — the rows that "belong" to Γ_i^E. In Figure 7.2, there are two such fragments represented: Γ_1^E's by the wavy sections, Γ_2^E's by the checkered sections.

We can now make a formal definition of the composition operation.

Definition 7.3.1. *Parse table composition ($\cup_T$).*

> $\cup_T$ is the parse table analogue to the grammar composition operation $\cup_G$.
> If $PT^C = PT^H \cup_T PT_i^E$, then $PT^C = \langle \Gamma^H \cup_G \Gamma_i^E, States_C, s_H, \pi_C \rangle$, where:
>
> - $States_C = States_H \cup States_E$.

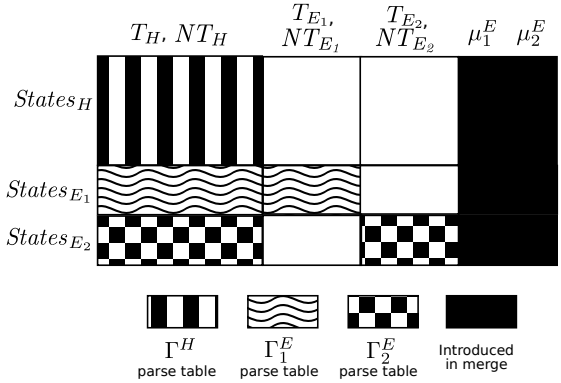

Figure 7.2: The sources of all actions in the composed parse table.

- $$\pi_C(n,\sigma) = \begin{cases} \pi_H(n,\sigma) & \text{if } n \in States_H \\ & \text{and } \sigma \in T_H \cup NT_H \\ \pi_E(n,\sigma) & \text{if } n \in States_E \\ & \text{and } \sigma \in T_H \cup T_E \cup NT_H \cup NT_E \\ \pi_\mu(n,\sigma) & \text{if } t = \mu_i^E \end{cases}$$

π_μ (see Definition 7.3.5 on page 190) represents the parse actions that need to be generated on the fly: shifts on bridge items and reductions on marking terminal lookahead.

Definition 7.3.2. *Generalization of parse table composition ($\cup_T^\star$).*

> The operation $\cup_T^\star$ is defined as the straightforward generalization of $\cup_T$ along the same lines as the operation $\cup_G^\star$ laid out in Definition 6.1.3 on page 150.

See Figure 7.2 for a pictorial outline of how $\cup_T^\star$, the generalization of $\cup_T$, operates. The parse table in the diagram is built from sections taken from completed parse tables for Γ^H, $\Gamma^H \cup_G \Gamma_1^E$, and $\Gamma^H \cup_G \Gamma_2^E$, as well as a section constructed at composition time. The striped section represents the host-language part of the table: the complete parse

table of a parser for Γ^H, consisting only of actions on host symbols ($T_H \cup NT_H$). The wavy sections represent the part of the table corresponding to $M_{E_1}^{E_1}$ and $M_D^{E_1}$, consisting only of actions on symbols in $T_H \cup NT_H$ and $T_{E_1} \cup NT_{E_1}$. The checkered sections, similarly, represent the part of the table corresponding to $M_{E_2}^{E_2}$ and $M_D^{E_2}$, consisting only of actions on terminals in $T_H \cup NT_H$ and $T_{E_2} \cup NT_{E_2}$. The solid white sections represent parts of the table where there are never any parse actions. The top two are on account of Lemma 6.4.3, which restricts follow sets on host nonterminals, thus ensuring that there will be no actions on extension terminals in the host partition; the bottom two are on account of Lemma 6.4.2, which guarantees that there will not be items from two extensions in any state, whence it follows that there will not be an action on a terminal $t_1 \in T_{E_1}$ in the same parse table row as an action on a terminal $t_2 \in T_{E_2}$.

The solid black section represents the block of columns pertaining to the several marking terminals, which must be generated on the fly. Building the marking terminal columns requires the maintenance of additional metadata to determine where a shift or reduce action should be placed in these columns (*i.e.*, when a bridge item $nt_H \rightarrow \mu_i^E s_i^E$ is in an item set or a marking terminal is in a lookahead set).

7.3.1 Needed additional metadata and definition of π_μ.

The additional metadata needed to build the marking terminal columns is presented here in the form of two maps, *initNTs* and *laSources*. It can also be placed in alternate forms in the event that a grammar writer wishes complete concealment of the grammar from the end users.

Recall from section 2.2.3.2 that an LALR(1) shift action is put into parse table cell (n,t) when, in the LR DFA from which the parse table was built, an item $(A \rightarrow \alpha \bullet t\beta)$ (α and β being sequences of zero or more grammar symbols) is a member of $items(n)$. If the item is of the form $A \rightarrow \bullet t\beta$ (*i.e.*, α is the empty sequence) then by the closure

rule (Definition 4.4.2 on page 110) there must also be an item $(B \rightarrow \gamma \bullet A\phi) \in items(n)$.

For example, in the start state of any LR DFA, there is an item of the form $S \rightarrow \bullet\alpha$, where S is the grammar's start nonterminal and α is a sequence of grammar symbols. In that state there is also an item $^\wedge \rightarrow \bullet S\$$, from which the item $S \rightarrow \bullet\alpha$ has been derived by closure. See Figure 2.14 on page 43 for a more specific example of this.

Any nonterminal A with such an item needs to be kept track of, as in the event that an extension introduces a bridge production $A \rightarrow \mu_i^E s_i^E$, all states containing such an item must be updated with a shift action on μ_i^E to the extension's start state.

Definition 7.3.3. *initNTs.*

> Let $initNTs : States \rightarrow \mathscr{P}(NT)$ represent these sets: $initNTS(n)$ consists of the set of nonterminals A such that there is an item in $items(n)$ with the bullet immediately preceding A.

initNTs enables on-the-fly addition of shift actions in the marking terminal columns; we also need metadata to handle reduce actions. Recall that a reduce action $reduce(A \rightarrow \alpha)$ is put into table cell (n,t) when there is an item $(A \rightarrow \alpha\bullet) \in items(n)$ such that t is in the lookahead set of $A \rightarrow \alpha\bullet$ in n. Recall also that this lookahead set is derived from the first sets of several nonterminals.

Specifically, if there is an item $A \rightarrow \alpha \bullet X\beta,\ z$ in a state's item set, the closure rule specifies that items for all productions with X on the left hand side are also in that item set. Then all symbols in $first(\beta)$ (and z, if β is nullable) are added to the lookahead sets of all these X items.

If the first non-nullable symbol in the sequence β is a nonterminal Y, it needs to be kept track of so that if Y is on the left hand side of Γ_i^E's bridge production, the extension's marking terminal μ_i^E can be added to the lookahead sets of the X items.

Definition 7.3.4. *laSources.*

Let $laSources : States \times NT \to \mathscr{P}(P)$ represent these sources: if the lookahead in the item $A \to \alpha\bullet$ is sourced from the nonterminal Y, as described above, then $(A \to \alpha) \in laSources(n, Y)$.

Definition 7.3.5. *The generated portion of the composed parse table,* π_μ.

π_μ must include:

- Shift actions: If the bridge production is $nt_H \to \mu_i^E s_i^E$, then

$$\forall n \in States_C. \left[nt_H \in initNTs(n) \to shift\left(n_{E_i}^S\right) \in \pi_\mu(n, \mu_i^E) \right]$$

- Reduce actions:

$$\forall n \in States_C. \forall p \in P \left[laSources(n, nt_H) \to reduce(p) \in \pi_\mu(n, \mu_i^E) \right]$$

7.4 Privacy of parse tables and metadata.

If no symbol or production of the grammar may be revealed to the end user, the parse table and the additional metadata in *initNTs* and *laSources* may be stored in the following manner:

- Parse tables may still be distributed in the traditional manner, using integers to represent terminals and nonterminals.
- Of productions in reduce actions, the parser only requires the symbol on the left hand side (to know what to look up in the goto table) and the number of symbols on the right hand side (to know how many stack elements to pop off beforehand, respectively).

- *initNTs* does not need to be stored at all, but can be inferred if necessary from the goto tables (*initNTs*(n) consists of exactly those nonterminals that have goto actions in n).

- Neither does *laSources* need to store all information about a production; only enough to make a reduce action based on it, as described above.

7.5 Composition of scanners.

Hitherto we have discussed only the problem of composing the parse table. However, if parsers are to be distributed in compiled form, some way must also be found for extension writers to compile and distribute scanners for their extensions, since simply generating a new scanner at composition time, as is done when compiling from grammars, will not work — the process is potentially exponential in time complexity.

Although the composition of parse tables is reasonably straightforward, scanner composition is not so much, as the lexical syntax of a composed language is far more monolithic and not so easily put apart into separate components. Furthermore, the scanner DFAs used to implement the lexical syntax are themselves very monolithic and do not admit being separated into "fragments" as the parse tables are. Hence, our solution to this problem involves extension writers compiling completely separate scanner DFAs for their extensions and distributing them along with the parse table.

There are two problems to be dealt with when building scanners for parse table composition:

1. **Scanner bundling problem.** A way must be found to bundle, along with the extension parse table sections, scanner DFAs that can scan on any state in such a section, while at the same time avoid, as much as possible, duplication of state

structure also found in the host language's scanner DFA, which cannot recognize extension terminals.

This poses a particular challenge because an extension can reference any host terminal or nonterminal; thus, a scanner for an extension potentially has to recognize all host terminals in addition to the extension's own.

2. **Marking terminal problem.** Precompiled scanner DFAs must be modified at composition time when a marking terminal is added to the valid lookahead in a particular parse state, as will frequently occur.

7.5.1 Scanner bundling problem.

Two solutions to this problem have been examined.

The first involves the use of the multiple-DFA context-aware scanner implementation described in section 5.3. *Prima facie,* this approach adapts itself well to the problem as there is one DFA built for each state in the parser, and one would simply concatenate scanner descriptions as with parse table rows. However, the heuristics needed to make this approach practical do not adapt themselves so well: if it is determined that one DFA can scan for two states, but the valid lookahead set in one subsequently changes, then the scanner DFA for that state would have to be rebuilt, negating the benefits of the approach.

The second, more practical, approach is to build separate DFAs for the host and extension sections of the parse table; then the DFA to scan on the states in $States_H$ can be distributed with the host parse table, and the DFA to scan on the states in $States_{E_i}$ can be distributed with the parse table section for the extension Γ_i^E.

Of course, there is some duplication of states necessary in this approach, but we have not found the number of duplicated states to be unreasonable in practice. For

ableJ extension	SQL	Tables	Foreach	Dimension analysis
$\lvert T_H \cup T_E \rvert$	194 (37 new)	161 (4 new)	158 (1 new)	181 (24 new)
$\lvert \{t : n \in States_E \wedge \pi(n,t) \neq \emptyset\} \rvert$ **(% of above)**	132 (68%)	69 (43%)	95 (60%)	105 (58%)
Total scanner states	930 (120 new)	852 (42 new)	813 (3 new)	856 (46 new)
Extension scanner states (% of above)	549 (59%)	253 (30%)	411 (51%)	345 (40%)

- Row 1: Number of terminals in composed grammar
- Row 2: Terminals containing actions in the extension part of the parse table
- Row 3: States in scanner for composed language's complete parse table $States_H \cup States_E$
- Row 4: States in scanner exclusively for extension partition $States_E$

Table 7.1: Statistics on scanners for extension states.

example, Table 7.1 shows test data on extensions to ableJ:

- The first row is the number of terminals in the Java host grammar and the extension combined ($T_H \cup T_E$). Alone, the Java host grammar has 157 terminals; the number of new terminals introduced range from the *foreach* extension, which introduces no new terminals except its marking terminal, to the SQL extension, which introduces many new terminals unique to SQL syntax.

- The second row gives the number of terminals in $T_H \cup T_E$ that are in the valid lookahead set of some state in an extension partition ($\bigcup_{n \in States_{E_i}} validLA_{PT}(n)$). These are made up mostly of host terminals, because in Java, many keywords can

begin an expression or statement, and expressions and statements are derived by many extension constructs.

- The third row gives the number of states in a scanner DFA that can scan on any state in the composed parse table (*i.e.*, the scanner built when $\Gamma^H \cup_G \Gamma_i^E$ was compiled from the grammar).

- The fourth row gives the number of states in a scanner DFA that can scan only on the states in some extension partition ($States_{E_i}$). These are the DFAs that will be bundled with the extension parse table pieces.

In the composed parser for all the extensions, the total number of scanner DFA states needed will be the summation of the fourth row, plus the number of states in the scanner DFA for the host grammar compiled alone. In the example case, the scanner for the Java host grammar contains 810 states; the total number of states needed in the composed parser is therefore $810 + 549 + 253 + 411 + 345 = 2368$. By comparison, if the composed parser is compiled from the grammar up, the resulting scanner will contain 1020 states, a 132% total increase in the scanner size.

This is a factor to be considered along with the benefits of rapid parse table composition, although in the present day with amounts of available memory increasing more quickly than the average size of grammars, it is not a very significant factor except for memory-critical applications, such as an embedded system.

7.5.2 Marking terminal problem.

It is very difficult to modify a scanner DFA, once built, to accommodate the scanning of new terminals. In the case where there is just one DFA for each partition of the parse table, the brute force approach of rebuilding the entire DFA to accommodate

marking terminals is potentially exponential in time complexity and would defeat the entire purpose of parse table composition to utilize.

A more efficient option exploits the facts that there are comparatively few marking terminals and that the languages of the regular expressions pertaining to marking terminals generally contain only one string (`table` for the tables extension to Java, *etc.*). Hence, the NFA constructed from any one of these regular expressions, or any union of them, is acyclic and requires only polynomial time to convert to a DFA. Therefore, a separate scanner DFA recognizing all the marking terminals in a composed grammar can be built in polynomial time.

To use this scanner DFA in parallel with the existing precompiled DFAs for the non-marking terminals, one uses a "dual-state" or "dual-DFA" arrangement in which the scanner engine will run the marking-terminal and non-marking-terminal DFAs in parallel wherever a marking terminal is valid lookahead. For the purposes of the scanning algorithm, the accept, reject and possible sets of the two scanner DFAs will each be merged as necessary at scanner runtime; if the marking terminal takes static precedence over another terminal in the accept set, that terminal will be moved to the reject set.

This dual-state operation incurs a negligible constant runtime penalty, but there is no increase in the asymptotic time complexity; see section 8.5 for more information.

7.6 Implementation.

There are several of ways to implement the composition and storage of the composed parse table laid out in Definitions 7.3.1 and 7.3.5.

It is a ubiquitous practice to store parse tables in two-dimensional arrays indexed

firstly by state number and secondly by a number representing a terminal or nonterminal. This is in general the only structure efficient enough to use in this application.

The most naïve process of implementation is to assemble one large array of states from the pieces in exactly the configuration shown in Figure 7.2. However, this requires that both the host and extension table pieces be copied piecemeal, at best row-by-row, into the composed table, which wastes time. To save time, the composed parse table would be stored in several pieces, which would then be used as one large "virtual" array made by mapping different indices of the "virtual" array to the various "real" arrays. This level of indirection incurs only a negligible increase in runtime (see section 8.5).

The extension parse table pieces would also be distributed in several arrays to enable fast copying: firstly, the goto table (nonterminal columns); secondly, the columns corresponding to host terminals T_H (leftmost in Figure 7.2); thirdly, the columns corresponding to extension terminals T_{E_i}. This allows the host section (rows from $States_H$ and columns from T_H, in the striped section in Figure 7.2) to be copied into the composed parser. This host section would be the first "real" array used by the composed "virtual" parse table. The second such array would be the marking terminal section (constructed rather than copied).

The remaining sections are then the several extension sections (represented by the wavy and checkered sections in Figure 7.2). These may be assembled in one of two ways:

1. Keep **a separate array** for each extension section, translating the exact array indices of states in $States_{E_i}$ at runtime. See Figure 7.3(a) on the following page for a diagram of this arrangement. This has the advantage of being able to be copied quickly, but the disadvantage of a more cluttered parse table.

2. Exploit the fact that there are no parse actions on terminals in T_{E_i} except in the

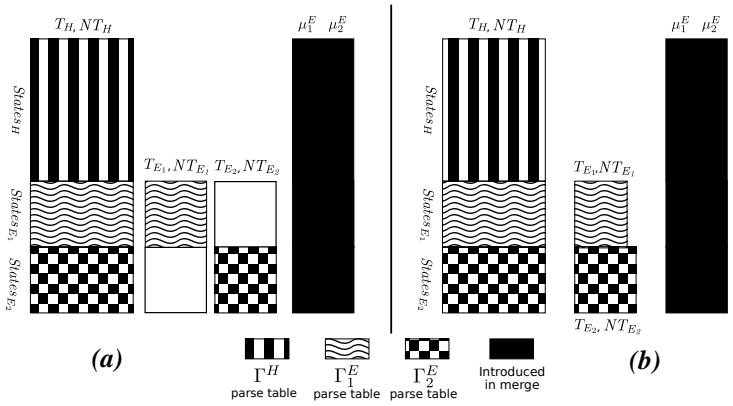

Figure 7.3: Assembling the composed parse table: (a) the multiple-array approach; (b) the three-array approach.

rows corresponding to states in $States_{E_i}$ (this is illustrated by the white squares in Figure 7.2) and merge the extension-symbol columns of all the extension pieces into a single array with $\max_{E_i} |T_{E_i}|$ columns, for a total of **three arrays**. Array indices are thus translated at compile time. See Figure 7.3(b) for a diagram of this arrangement.

This has the advantage of less clutter, but the disadvantage that more copying work must be done, since a byte-for-byte copy of extension parse tables cannot be made.

7.7 Time complexity analysis.

7.7.1 Composition time.

In composing parse tables, the majority of the computation consists of copying pieces of parse tables and calculating their destination. The multiple-array approach involves only copying $|States_H| \cdot |T_H \cup NT_H| + \sum_{E_i} |States_{E_i}| \cdot |T_{E_i} \cup NT_{E_i}|$ parse table cells. The

three-array approach involves copying $|States_H| \cdot |T_H \cup NT_H|$ parse table cells (the host language section, constituting the bulk of the table) and $\left(\left|States_H \cup \bigcup_{E_i} States_{E_i}\right| \cdot \left|T_H \cup NT_H \cup \bigcup_{E_i} (T_{E_i} \cup NT_{E_i})\right|\right)$ parse table cells in the merged extension arrays are filled. For each such cell, if the action within it has an extension state as a destination, it must be modified to point to its new location in the combined array — a constant-time operation. (The actions in the cells of the host language section will not have an extension state as a destination, as those are confined to the marking terminal columns.) The naïve single-array approach must instead fill *all* cells in this same way.

For all approaches, the parse table column for each extension's marking terminal ($\left|States_H \cup \bigcup_{E_i} States_{E_i}\right|$ states altogether) must be filled out. For each cell $\left(n, \mu_i^E\right)$:

- *initNTs* must be consulted one time to determine if a shift action should be inserted in the cell. If *initNTs* is implemented using a hash table, this takes constant time.

- *laSources* must be consulted one time to determine if a reduce action should be inserted in the cell. Again, if a hash table implementation is used, this takes constant time.

7.8 Conclusion.

In this chapter, we have presented a corollary of the modular determinism analysis that allows the automatic composition guaranteed by the analysis to be performed quickly, possibly even at parser runtime; it also permits extension writers to conceal their extension grammar specifications if they so desire.

Additionally, the looser coupling of composed parse tables allows for the use of a relaxed analysis $Partitionable_{PT}$ that can admit more extensions; see section 9.4.5 for an example of an extension that passes $Partitionable_{PT}$ but not the original *Partitionable*.

By shifting the time-consuming process of grammar compilation to be a task of the extension writer rather than the programmer who composes the extensions, parse table composition moves one step closer to a compiler in which language extensions can be imported in the same manner as libraries in Java. This is discussed further in chapter 10.

Chapter 8

Parse-time performance.

In this chapter, we present a time-complexity analysis of our scanning and parsing algorithms and runtime analyses of Copper, the scanner and parser generator based on them. These indicate that the runtime of our algorithms is comparable to a reasonable degree with that of the traditional LALR(1) algorithms. It is also established that fairly little new time-complexity analysis is needed on the algorithms.

8.1 Time complexity analysis.

Our algorithms, as used in Copper, do not differ greatly from those used in the traditional LALR(1) framework; hence, little new time complexity analysis is needed. Specifically, our parsing algorithm is only slightly modified from the algorithm used in traditional LALR(1) parsers, which is known to run in time linear with respect to the number of tokens in the input.

The traditional disjoint scanner algorithm also runs in linear time, though with respect to the number of characters in the input rather than the number of tokens. In chapter 5, we discussed two different implementations of the context-aware scanner:

the single-DFA implementation (discussed in section 5.2) that runs with one scanner DFA annotated with metadata allowing it to run on any of the valid lookahead sets it will be passed, and the multiple-DFA implementation (discussed in section 5.3) that relied on a series of DFAs, one built for each different valid lookahead set.

The multiple-DFA algorithm is identical to the traditional disjoint scanner algorithm, differing only in that it must be run in lock-step with the parser, since a different DFA will be used depending on which parser state the scanner is called from.

However, the single-DFA scanning algorithm differs in two respects from the traditional algorithm:

1. Scans for layout, transparent prefix and actual token are done separately, so at a position in which no layout separates two tokens and (although a prefix is valid) no prefix is used, three scans will be performed in the place of one — two returning empty strings, one returning the actual token.

   For example, in parsing the string `x+x` according to the grammar in appendix A.4, the multiple-DFA algorithm scans only three tokens: `x`, `+`, and `x`. On the other hand, the single-DFA algorithm scans for layout each time, returning the empty string, so the sequence is instead ε, `x`, ε, `+`, ε, `x`.

2. As each character is scanned, between three and five bit vector operations are performed (see lines 6a, 6b, 6d, 6e, and 6f of Figure 5.2 on page 134); each bit vector contains one bit for each terminal handled by the scanner.

The first difference, although it adds a constant factor to the time complexity, does not affect the asymptotic complexity. However, the second difference raises the time complexity of scanning from $O(n)$ to $O(|T| \cdot n)$, T being the set of terminals handled by the scanner.

8.2 Runtimes of the two scanner implementations.

The first question to be answered is: *How does the single-DFA scanner implementation having a time complexity of $t = O(|T| \cdot n)$ impact practical runtimes?*

Before addressing this question, we must take into consideration that Copper is the only implementation of context-aware scanners. Hence, to compare the runtimes of Copper to those of well-established tools like CUP/JFlex would not accurately reflect the runtime differential between the new and traditional *algorithms* as distinct from their implementations.

Fortunately, there is a benchmark that will eliminate most factors related to the implementation; Copper's multiple-DFA implementation does not differ in the algorithmic sense from the traditional scanner algorithm, and makes use of the same implementation of scanning and parsing engines as Copper's single-DFA implementation. Hence, the effects of the additional time complexity of the single-DFA algorithm can accurately be gauged by comparing the runtimes of those two implementations.

There is one other factor that must be addressed, however: the time it takes to load the parser into memory. Since the multiple-DFA scanners are presently much larger in size than the equivalent traditional scanners, we can expect the multiple-DFA scanners to load more slowly than their single-DFA counterparts. Hence, the parser's load time in each implementation — a constant figure — should be factored out of the calculations.

It follows that the method to test this question is to compile, for a certain grammar, parsers generated for CUP/JFlex, and for both Copper implementations, and then to run them on an assortment of files of varying sizes; then, having eliminated the factor of the loading time, to factor out all Copper-specific implementation issues by removing the difference between the runtime of the CUP/JFlex parser and the multiple-DFA Copper parser.

8.2.1 Tests on C source files.

588 C source files were selected at random from the source tree of the Linux kernel, version 2.6.23.1. Statistics on the sizes of these files are as follows:

- Mean size: 142,781 bytes.
- Median size: 184,607 bytes.
- Standard deviation: 84,138 bytes (normalized: 0.23577).
- Smallest file: 271 bytes.
- Largest file: 356,866 bytes.

Parsers were then built for the ableC grammar using the Java-based CUP parser generator and JFlex scanner generator in conjunction, as well as Copper (both the single- and multiple-DFA engines) and run on these source files. The system used for the testing was a 3.4 GHz Pentium-D with 4GB of RAM, running Ubuntu x86-64 release 6.06 (Dapper Drake).

The CUP/JFlex parser took 0.124 seconds to load into memory before any parsing began; the single-DFA Copper parser, 0.256 seconds; the multiple-DFA Copper parser, 0.376 seconds. These times have been factored out of our calculations.

Figure 8.1 shows the times taken to parse each C file, with the parser load time factored out. As expected, the data points for multiple-DFA Copper follow roughly the same trajectory as those for CUP, while the data points for single-DFA Copper follow a markedly steeper trajectory. Linear approximations of these three sets of data points using the sum of absolute differences formula, $\min_{k>0} \sum_n |t - k \cdot n|$, yielded the following slopes (t in seconds, n in bytes):

- CUP/JFlex: $t \approx 2.2175 \cdot 10^{-6} \cdot n$

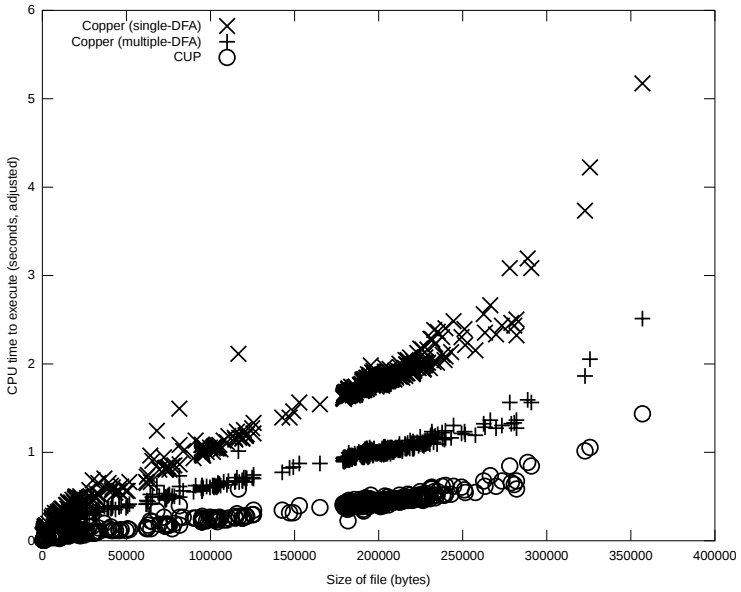

Figure 8.1: Time comparisons: CUP vs. single-DFA Copper vs. multiple-DFA Copper, runtimes on C files, with parser initialization time factored out.

- Multiple-DFA Copper: $t \approx 5.0625 \cdot 10^{-6} \cdot n$
- Single-DFA Copper: $t \approx 9.105 \cdot 10^{-5} \cdot n$.

The differing parser startup times being factored out, the difference between the CUP and multiple-DFA slopes is, as mentioned, attributable solely to implementation issues specific to Copper. It is the ratio of the single-DFA to the multiple-DFA slope that reflects the single-DFA algorithm's runtime differential; see Figure 8.2 on the following page for a graph of the ratios, which level out at approximately 1.8 as file sizes increase.

8.3 Effect of the bit vector operations.

Having examined the question of the practical consequences of the single-DFA implementation's $O(|T| \cdot n)$ runtime on a "real" language, C, we will now examine the significance of the effects of the extra factor $|T|$ in a starker context.

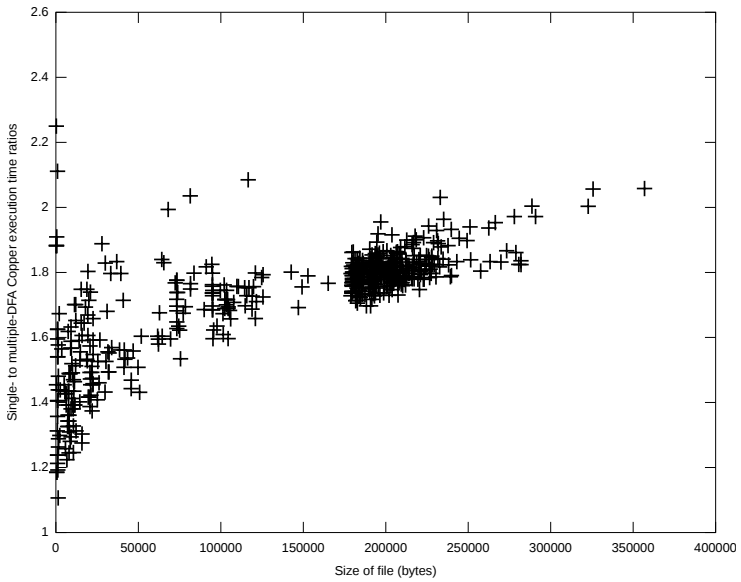

Figure 8.2: Ratios of the runtimes of multiple-DFA to single-DFA Copper.

This factor $|T|$ being solely the result of the three to five bit vector operations that are carried out during the scanning of each character in the input, we must eliminate any other factors except those bit vectors; and as it happens there is a simple way to do this. Copper permits the specification of "useless" terminals — terminals that are not used in the context-free syntax either as layout or as ordinary terminals. If specified, these will have no bearing on the construction of the parser or scanner, or on the valid lookahead sets; however, they will increase the size of the bit vectors used in the calculations.

We have implemented a "toy" grammar (see appendix A.4 on page 293) with 5 terminals, 3 nonterminals, and 5 productions. For this test, we formulated two different versions of this toy grammar: one the original and unmodified grammar, the other a version supplemented with 5,000 "useless" terminals having the same regular expression as the terminal `x`. Parsers based on the single- and multiple-DFA algorithms were built for each grammar.

A group of automatically generated test files, in sizes ranging from 1,000 to 499,000

bytes, were run on these four parsers; Figure 8.3 shows the runtimes. The addition of the useless terminals had a negligible effect on the runtime of the multiple-DFA algorithm (the parser of the grammar with the useless terminals taking on average 1.025 times longer to run), while the effect on the single-DFA algorithm was more pronounced (a ratio of 1.23).

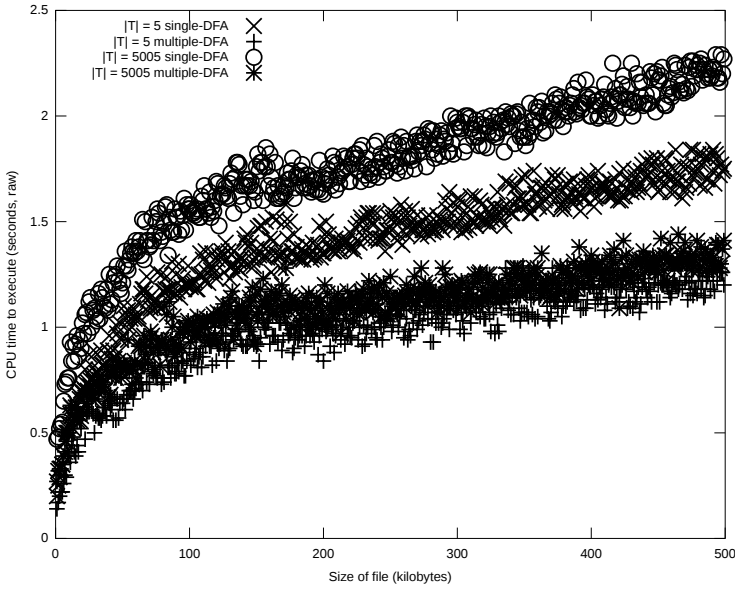

Figure 8.3: Runtimes of the single-DFA algorithm on test files for the toy grammar, with and without 5,000 useless terminals introduced to increase the size of the bit vectors.

8.4 Sizes of scanners for both implementations.

Having addressed the question of the runtime differential between scanners using the two different context-aware scanner implementations, we turn to the question of the additional space needed by a scanner for the multiple-DFA algorithm.

How many states are ultimately needed in the DFAs of such a scanner come down to two factors: firstly, the number of valid lookahead sets for which DFAs need to be

generated; secondly, the efficacy of the optimizations used to reduce the number of needed scanner DFAs by generating one DFA for several valid lookahead sets.

To test this, we compiled three parsers for each of a number of our test grammars. The first parser in each case used the single-DFA approach; the second, the multiple-DFA approach with no optimizations; the third, the multiple-DFA approach with the "union adequacy" optimization discussed in section 5.3.3.2.

	Toy grammar	**4-func. arith. grammar**	**ableC**	**ableC + Promela**	**Java**
# valid lookahead sets	4	5	99	168	194
# DFAs needed (post-opt.)	1	1	58	168	194
DFA states (single-DFA)	7	16	359	428	1020
DFA states (multiple-DFA unoptimized)	56	151	20621	22708	52155
DFA states (multiple-DFA optimized)	7	16	11329	22708	52155

Table 8.1: Size statistics for single- and multiple-DFA scanners on a number of test grammars.

See Table 8.1 for the test results. As may be seen, at present, the number of states needed in the multiple-DFA approach becomes unmanageably high with larger grammars — although the "union adequacy" optimization does cut down the number of states considerably, especially on syntactically poor grammars where there is no need for separate DFAs at all.

It is important to note, however, that these numbers represent only the implementation as it exists presently. It is very likely that further optimizations can be made that will reduce much further the number of states needed in the multiple-DFA approach,

as even the simple "union adequacy" optimization obtains a large reduction. It is also likely that there are optimizations that are specifically suited to syntactically rich languages where the "union adequacy" optimization fails.

8.5 Parse table composition and dual-DFA scanning.

In testing the effects of the parse table composition upon runtime performance, there were two separate questions to be considered: firstly, the effects of splitting the parse tables into several arrays instead of holding them in one, which creates "switching overhead" in terms of the time needed to map "virtual" parse table locations to "real" array locations; secondly, the effects of using the dual-DFA scanning arrangement discussed in section 7.5, in which there are separate scanner DFAs generated by the host writer and each extension writer, and another DFA for marking terminals only, compiled at composition time and run in parallel with the host or extension DFA when there is a marking terminal in the valid lookahead set.

Tests were run on a prototype implementation of the "virtual" parse table and the parallel scanner DFAs, using approximately 300 *ableJ* source files as a test bed. The Java host language was composed with the four extensions listed in Table 7.1 on page 193 and four parsers were thence compiled, covering every permutation of the two factors being tested:

1. A parser compiled from the grammar up, with a single parse table and a single scanner DFA (a standard Copper parser);

2. A parser with a split parse table but a single scanner DFA;

3. A parser with a single parse table but a separate scanner DFA for each extension;

4. A parser with both a split parse table and separate scanner DFAs (the intended output of the parse table composition).

This prototype implementation was only of the *runtime* parts of the new apparatus; the compile-time end is not yet implemented, and thus the compilation of the test parsers was done manually. The groups of terminals to turn into separate scanner DFAs — host, extension, and marking terminals — were manually separated, though the resulting scanners were identical to those that would have been compiled using the automated process. The marking terminal array was separated as shown, but the rest of the parse tables were separated arbitrarily by putting every third state in another array. Although this is not the arrangement that would be made by the composition process, it entails exactly the same amount of "switching overhead," as the parser must still determine which array to access to retrieve a given parse table cell, which is a separate question from what table is chosen by that process.

The tests showed that the table splitting produced only a negligible effect on the parser runtime, but that the introduction of the separate scanner DFAs and parallel scanning increased runtimes by an average of 10% over that of the established algorithms in Copper. See Figure 8.4 on the next page for a graph of the test data; this shows the runtimes of parsers #1 and #4, as the runtimes of parsers #2 and #3 respectively were so negligibly different as makes no distinction on the graph (less than 0.1% difference between the mean lines).

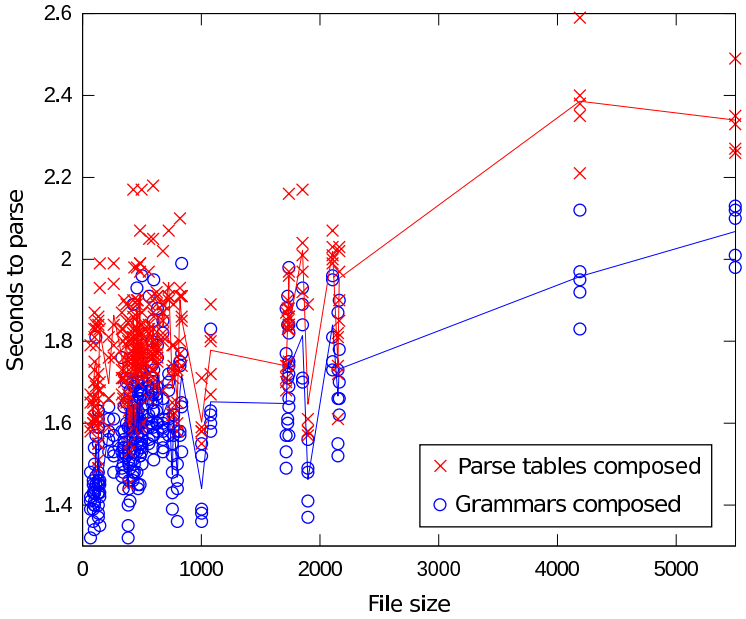

Figure 8.4: Performance statistics for parsers built by merging parse tables. Note that the bottom of this graph represents $y = 1.3$, rather than 0.

Chapter 9

Applications.

This chapter presents several existing applications of context-aware scanning and of the modular determinism analysis. Most are covered in summary fashion in our papers *Context-Aware Scanning for Parsing Extensible Languages* [VWS07] and *Verifiable Composition of Deterministic Grammars* [SVW09].

In presenting these examples — real established languages that have a history of being problematic to parse, and language extensions that were designed and built before the modular determinism analysis was conceived of — we aim to demonstrate that context-aware scanning and the modular determinism analysis can each independently be of immediate use to language developers. As mentioned in chapter 6, we have relaxed the meaning of "extension" to include grammar fragments with several bridge productions.

9.1 AspectJ.

The example of AspectJ [KHH+01], a significant extension to Java adding aspect constructs to the language, illustrates the balance of declarativeness, verifiability and expressivity struck by context-aware scanning. Although AspectJ does not pass the modular determinism analysis (it does not fit the extension form as many of its constructs begin with non-unique keywords), context-aware scanning allows it to be parsed deterministically as well as specified declaratively — previous solutions have only been able to offer one or the other.

AspectJ is a difficult language to parse, for several reasons, mostly relating to its lexical syntax. Firstly, it introduces a number of new reserved keywords into Java that must only be reserved in specific contexts — for example, the `after` keyword as shown in Figure 1.2 on page 12. Secondly, there are instances where the same portion of the input string might, in differing contexts, be validly broken into differing numbers of tokens; as in the example of the string `get*`, which in aspect constructs is matched as one token describing a pattern to match, but in Java constructs is matched as two tokens, an identifier with lexeme `get` followed by the multiplication operator `*`.

AspectJ does not have a "canonical" grammar like C's or Java's; the language's official maintainers, recognizing that AspectJ may present complications in compilation, are deliberately flexible in some of their definitions of the language [Asp03]. Hence, there are several different parsers existing for AspectJ, the actual language of each parser varying slightly.

However, despite this variety, our approach to parsing AspectJ is the only one thus far that is both declarative and deterministic. The parsers for the AspectJ compilers `ajc` and `abc` are deterministic but not declarative, and another solution employing SGLR is declarative but not deterministic.

AspectJ's reference implementation, `ajc` [Her02], takes an *ad hoc* component-based approach that uses a separate parser to parse AspectJ's *pointcut* constructs, which describe Java code patterns to match and identify where aspect constructs must be "woven" into the code. A pointcut construct is first matched as a "pseudotoken," or a single token encapsulating an entire pointcut; this is similar to the way that embedded regular expressions are often parsed (see page 107). The pseudotoken is then passed to the pointcut parser to be fully parsed [BETV06].

Another popular AspectJ implementation, the Aspect Bench Compiler (`abc`, for short) [HdMC04] takes a more monolithic approach, using a single LALR(1) grammar combined with a hand-coded scanner that separates the lexical syntax into four contexts or "modes." During scanning, the `abc` scanner maintains a stack on which nested modes are kept; the stack is pushed or popped when modes change. The modes are:

- **Java mode.** The initial and default mode, which is used for scanning Java constructs. Its lexical syntax is identical to Java's except for the addition of three new keywords: `aspect`, `privileged`, and `pointcut`.

- **Aspect mode.** Nested inside the Java mode, this is used for scanning aspect constructs. Like the Java mode, its lexical syntax is identical to Java's except for the addition of several new keywords.

- **Pointcut mode.** Nested inside the Aspect mode, this has a completely different lexical syntax. Perhaps the most significant departure is the introduction of a terminal for pattern matching on identifiers (*e.g.*, `get*` will match any identifier beginning with `get`).

  Hendren *et al.* [HdMC04] give, as an example of the difference, the scanning of the input `*1.Foo+.new(..)`. (Note that this string is syntactically invalid in the Java context, but can be validly scanned by the `abc` scanner in either context.)

In the Java context, this would be scanned as the following sequence of terminals:

Mul("*"), *Float*(1.0), *Identifier*("Foo"), *Add*("+"), *Dot*("."), *New*("new"), *Lparen*("("), *Dot*("."), *Dot*("."), *Rparen*(")")

In the pointcut context, with its own lexical syntax, it would instead be scanned as

idPatternTerm("*1"), *Dot*("."), *Identifier*("Foo")[1] *Add*("+"), *Dot*("."), *New*("new"), *Lparen*("("), *doubleDot*(".."), *Rparen*(")")

Note in particular that `*1.`, instead of being scanned as a multiplication operator followed by a floating-point number, is now scanned as a pattern terminal, `*1`, followed by a dot operator. Note also that `..` is scanned as one token, *doubleDot*, instead of an individual *Dot* token for each dot.

- **PointcutIfExpr mode.** Nested inside the Pointcut mode, this is for embedding conditional expressions inside pointcuts. It is identical to the Aspect mode except to handle the end, where it returns to the Pointcut mode rather than the Java mode.

The language of `abc` is different from that of `ajc` in several regards [HdMC], most notably in that with `ajc` the AspectJ keywords are not reserved, while with `abc` they are.

Bravenboer *et al.* [BETV06] have implemented an AspectJ parser in the SGLR framework, which attempts to maintain compatibility with both the `ajc` and `abc` variants of AspectJ by specifying several versions of the SGLR grammar specification. This solution is declarative (unlike `ajc` and `abc`); however, it exploits the nondeterminism of SGLR and thus is not deterministic like `ajc` and `abc`.

[1] In our solution, where the AspectJ and Java lexical contexts have their own identifier terminals, this would instead match the AspectJ identifier, *aspectJId*.

The SGLR-based declarative solution involves less explicit treatment of AspectJ's different lexical scopes; as mentioned in section 2.3.5, the framework's "scanning" is context-aware, so many of the scanning issues are resolved. However, they are forced to use a single identifier terminal for the entire language; since the SDF formalism used by these particular SGLR-based tools does not admit any hand-coding, and since the lexical syntax is compiled as a part of the parser, it is impossible to reserve all the new keywords only in their proper contexts.

Our solution to the AspectJ parsing problem [Sch09] exploits context-aware scanning and its practical modifications. It uses the single `abc` context-free grammar; the four "lexical modes" of the `abc` scanner are provided implicitly by the context-awareness, with only five disambiguation groups needed to resolve a few lexical conflicts (discussed further below).

This problem with the single identifier terminal is resolved by Copper's free-form lexical precedence relations. Since Copper does not use a total precedence ordering, precedences on each of the AspectJ keywords can be set without affecting the precedence ordering of any other terminals (not a possibility in a total ordering). Thus, our grammar for AspectJ uses two different identifier terminals: one for the Java context and one for the Aspect contexts. The Java identifier has none of the AspectJ keywords reserved against it, the AspectJ identifier has all of them reserved.

Going back to Figure 1.2, for example, the string `after` occurs in two contexts: Java and Aspect. In the Java context, `after` is not valid as a keyword and should be matched as an identifier. Since it is a Java context, this will be the Java identifier, against which `after` is not reserved. On the other hand, in the Aspect context, `after` is reserved as a keyword and should not be matched as an identifier. Hence, in places such as the declaration of `moves` in line 8, it is AspectJ identifiers that are valid and matched, and since the AspectJ keywords are reserved against the AspectJ identifier,

the AspectJ keywords cannot be matched as identifiers in this context.

Although this use of two identifier terminals automatically resolves nearly all conflicts between AspectJ keywords and Java identifiers, five of the 35 AspectJ keywords also occur in the same context as Java identifiers. This is because some of the aspect constructs back-reference Java host syntax, thus creating contexts where Java host constructs can begin at the same place as an AspectJ keyword can occur. For example, in aspect constructs, one can have constructs beginning with the keyword `after` as shown in Figure 1.2, as well as constructs beginning with any Java typename (in the Java grammar, typenames and identifiers are matched to the same terminal). These ambiguities can be resolved with the five disambiguation groups mentioned above. Each of these groups, defined on an AspectJ keyword and a Java identifier, disambiguate to the AspectJ keyword: for example, one of them is on the terminal set $\{After, Id\}$ and disambiguates to *After*.

However, this does not catch instances within aspect constructs in which the AspectJ keywords are not in the valid lookahead set and can be matched as identifiers, which is syntactically invalid. This must be watched for and caught in semantic analysis, as keywords and identifiers are distinguished in `ajc` [HdMC]. This problem is an area for future work; specifically an extension-specific form of lexical precedence, as discussed further in section 10.2.3.

We provide the grammar for AspectJ in appendix A.3.2.1 on page 271.

9.2 ableC.

ableC, an extensible adaptation of ANSI C (C89) with a syntactic extension to accommodate the use of `gcc` header files, is primarily an illustration of the flexibility and declarativeness gains of our framework, showing its greater capacity to “bend the

rules": accommodate grammars that *almost* fit within the confines of the strictly declarative formalism without introducing any more code than necessary. The ableC parser operates on source files after they have been run through the C preprocessor.

From the parsing standpoint, C's most interesting feature is the typename/identifier ambiguity. Although typenames and identifiers in C have the same regular expression (`[A-Za-z_\$][A-Za-z0-9_\$]*`), due to the structure of the C grammar, using the same terminal for typenames and identifiers (as is done in Java) makes the grammar non-LALR(1). Thus, typenames are generally distinguished from identifiers by maintaining a list of types defined by `typedef`. A `typedef` in C is represented in the C grammar by an ordinary variable declaration with the modifier `typedef`; at each one of these declarations, the scanner will add the declared typename to the list of types, only matching the regular expression `[A-Za-z_\$][A-Za-z0-9_\$]*` as a typename when the lexeme is on that list, and as an identifier when it is not.

```
1. typedef int length;
2. int main() {
3.     length x;
4.     int foo;
5.     foo = x;
6. }
```

Figure 9.1: C program illustrating typename/identifier ambiguity.

For example, in Figure 9.1, line 3 starts with `length`, which is a typename, having been `typedef`'d in line 1; line 5 starts with `foo`, which is an identifier. But since both typenames and identifiers are valid at the beginning of C statements, the scanner cannot distinguish between them.

Context-aware scanning cannot *per se* resolve this problem; however, it does greatly simplify the code needed to distinguish between typenames and identifiers, and places it

in a framework that permits enough use of code in the grammar specification to resolve the ambiguity without permitting enough to create an entire hand-coded scanner.

In a traditional disjoint scanner, the grammar writer is not allowed to assume that the scanner and parser will operate in lock-step: one must consider the possibility that the scanner will finish executing before the parser begins. Hence, the code needed to build and maintain the list of typenames must operate entirely in the scanner, with no context information of any kind from the parser.

On the other hand, a context-aware scanner operates in lock-step with the parser. This allows a specification of C to maintain the list of valid typenames as a parser attribute, able to be modified by semantic actions both on terminals and productions. This is simpler than having a field in the scanner containing the list, specifically because it is easier to tell when to add something to the list. See Figure 9.2 on the next page for an example of such an action. It is on a production with the left hand side **DirectDeclarator**, representing a variable declarator — the part of a `typedef` in which a new type is named. The action checks that two conditions are matched: firstly, that the name being declared is not already a typename;[2] secondly, that this declarator is within a `typedef` (this is gathered from the value of another parser attribute set in another parser action).

In the presence of the parser-attribute typename list, one then can simply use a disambiguation function to check the list and resolve the ambiguity. Pseudocode for this disambiguation function is shown in Figure 9.3 on the following page.

We provide the grammar for ableC in appendix A.1 on page 250. It is a direct adaptation of the ANSI C grammar in the second edition of Kernighan and Ritchie's

[2] Note that *Typename* is not in the valid lookahead set at this point, so a typename in the list *typenames* may be matched as an identifier. These instances must be checked for in semantic analysis, as is done in Java.

Parser action on production **DirectDeclarator** → *Identifier*, adding new typename to parser attribute *typenames*:
With *x* representing the lexeme of the right hand side terminal *Identifier*.

1. if <$x \notin$ *typenames*> and
   <*we are in a declaration that is a* `typedef` *but is not in a* `struct`> then
   (a) $typenames = typenames \cup \{x\}$

Figure 9.2: Parser action in the ableC parser that adds to the list *typenames*.

function $disambig_{TypenameID}(x) : \Sigma^\star \rightarrow \{Identifier, TypeName\}$
With parser attribute $typenames \subseteq \Sigma^\star$ representing all valid `typedef`'d typenames.

1. if $x \in typenames$ then return *Typename*
2. else return *Identifier*

Figure 9.3: Typename/identifier disambiguation function.

The C Programming Language [KR02] with some extensions added to parse GCC-specific extensions from header files.

9.3 Promela.

Promela [Hol03] (short for PROcess MEta-LAnguage) is the input language to the Spin model checker, which tests the consistency of models for, among other things, electronic communication and program control flow. It demonstrates context-aware scanning's utility in parsing embedded languages; specifically, a Promela source file may contain embedded C constructs.

Like AspectJ, it is an example of an "extended" language that was meant to be composed by the extension writer rather than the end user. It does not pass the modular determinism analysis on a minor technical matter: it defines a nonterminal, **Ansi_C_code**,

```
 1. int q;
 2. c_decl { int *p ;};
 3. init {
 4.     c_code{ *p = 0; *p++;};
 5.     c_code [p != 0]{ *p = &(now.q); };
 6.     c_code { printf("%d\n".Pinit->_pid); }
 7. }
 8. active proctype x1() {
 9.     printf("This is Spin version 4.0\n")
10. }
```

Figure 9.4: Example of Promela code.

which derives C constructs with no marking terminal to differentiate. However, all uses of this nonterminal are placed after proper marking terminals, and all C constructs are enclosed in brackets or curly braces, so a restructured grammar that eliminated this nonterminal would pass the modular determinism analysis.

The presence of the embedded C constructs makes Promela a parsing challenge in traditional frameworks, as the two languages have different lexical syntax, which would cause lexical conflicts if they were scanned with the same disjoint scanner. Furthermore, Promela was designed to resemble C closely in syntax; hence, although the two languages have different lexical syntax, a good deal of it is shared between them, and building separate parsers for the two languages is redundant (a large proportion of the lexical specifications and the grammar would have to be duplicated). This is illustrated by the example Promela program in Figure 9.4. Note particularly the similarities between Promela and C code (use of curly braces, *etc.*), and the presence of Promela-specific keywords (*e.g.*, `active`).

Before introducing context-aware scanning, we had developed a grammar for Promela that we compiled into a working traditional LALR(1) parser; however, due to this lexical issue and also our lack of a working C grammar we had had to exclude from it the few productions forming the bridge to the embedded C constructs. Later, after we implemented context-aware scanning in Copper and imported the ANSI C grammar as the nucleus of ableC, we decided to insert the bridge productions to compose the Promela and ANSI C grammars. The composed grammar compiled conflict-free in Copper without any tweaking or other modifications.

Context-aware scanning seamlessly solves the problem of parsing Promela because the languages can be joined by a simple, automatic composition with no conflicts, the context allowing the scanner to differentiate between C and Promela contexts. This is largely due to the C code being enclosed within brackets and curly braces, which provide the necessary degree of separation — the same general idea behind the modular determinism analysis.

We provide the grammar for Promela in appendix A.2 on page 255.

9.4 ableJ.

ableJ [VWKSB07] is an extensible version of Java 1.4. It also serves as our best example of a user-extensible language demonstrating the utility of the modular determinism analysis, as we have developed over 10 extensions to it, a few with no new syntax, many embedding entire large portions of pre-existing languages.

The most significant in parsing terms is AspectJ, which was given its own section (9.1, above). We also have extensions for autoboxing, native complex-number types, dimension type checking, `foreach` loop constructs, pattern matching (including the extensions of the *Pizza* language [OW97, OW98]), support for the randomized linear

perturbation used in computational geometry [VWJ04], SQL, and boolean tables. Below, we briefly discuss those extensions that are particularly interesting from the parsing perspective. We provide grammars for ableJ and its extensions in appendix A.3 on page 269.

9.4.1 Native complex-number types.

This extension adds native support for complex numbers to Java, overloading operators to allow arithmetic on complex numbers using primitive operations. In terms of concrete syntax, it introduces the following new constructs:

- A complex-number literal, in the form `complex`(ex_r, ex_i), where ex_r and ex_i are Java expressions representing the real and imaginary components of the complex number, respectively.
- Two complex number types, one primitive type `complex` and one reference type `Complex`.

As originally written, this extension, like Promela, fails the modular determinism analysis on a matter of form (the keyword `complex` being used in two different contexts) but would pass with slight modifications (if a different terminal could be used for one of these cases).

9.4.2 `foreach` loop constructs.

This extension adds a `foreach` loop construct to Java. In syntax it is identical to the `foreach` construct added to Java 1.5, with one exception: while a Java 1.5 `foreach` construct begins with the keyword `for`, a construct in our extension begins with a new keyword, `foreach`. This new keyword constitutes a marking terminal and the `foreach`

extension passes the modular determinism analysis. On the other hand, if `for` were to be used in that place, the extension would fail the modular determinism analysis as `for` is not a unique marking terminal.

9.4.3 SQL.

This extension allows the embedding of SQL expressions in Java, to provide compile time syntax and type checking on SQL expressions instead of passing them through string literals, as is done in the usual JDBC arrangement. With JDBC, the query string is not parsed or type-checked until *after* it is passed to the database server at runtime, which is when any errors in the expression are brought to light. By contrast, the use of this extension allows these errors to be caught at compile time. Being an embedded language with no back references to Java nonterminals, the SQL extension passes the modular determinism analysis.

Figure 1.1 on page 9 shows an example of the SQL extension's use.

9.4.4 Boolean tables.

```
table( a :  T * F *,
       b :  F T T *,
       c :  * F T T)
```

translates to

```
(a && !b) || (b && !c) || (!a && b && c) || c
```

Figure 9.5: A boolean table and its host translation.

This extension provides a "shorthand" for specifying Boolean expressions in disjunctive normal form, as tables in which the rows represent Boolean variables (*i.e.*,

Java expressions of type `boolean`) and the columns represent conjunctions of these variables.

See Figure 9.5 on the preceding page for an example of a table and its host language translation. `a`, `b`, and `c` are Java expressions of type `boolean`. The first column is `T-F-*`, meaning that for that conjunction, `a` is true, `b` is false, and the value of `c` does not matter. The other columns are likewise, and the entire table represents the disjunction of the columns.

From the parsing standpoint, the tables extension does not represent an embedded language quite so much as the SQL extension does. Of particular note is the derivation of a Java expression in each row, which is a new context for the expression, like for the ***primaryExpressionAndArrayCreation*** constructs in the dimension types extension (see the next section). However, in the tables extension, each expression is invariably followed by a colon. Expressions can also be followed by colons in the host contexts due to the conditional expression, `a ? b : c`. Hence, the tables extension adds nothing new to the follow set of a Java expression, nor to any lookahead sets of host states, and passes the modular determinism analysis.

9.4.5 Dimension type checking.

This extension to Java 1.5 adds concepts of units (*e.g.*, meters, kilograms) and dimensions (categories of units, *e.g.*, length, mass, time) as a compile-time supplement to scalar types to avoid confusion, such as the sort that caused the destruction of the Mars Climate Orbiter in 1998 when a file containing tables of impulse data in pound-seconds was read by a program that expected the data in newton-seconds, resulting in miscalculation of the spacecraft's intended trajectory [Boa99].

Dimensions are added to types in the same manner as type parameters; for example, to declare a variable of type `double` holding the gravitational constant $g \approx 9.81 \frac{m}{s^2}$, one

would enter:

- `Unit<double, length 1 m time -2 sec> g = 9.81;`

The dimension types extension is an example of an extension that fails the grammar-based modular determinism analysis *Partitionable*, but passes the relaxed analysis for parse table composition, $Partitionable_{PT}$.

Recall from Definition 6.3.9 on page 156 and section 7.2 that an extension failing *Partitionable* but passing $Partitionable_{PT}$ has a state in the partition $M_{NH}^{E_i}$ that is either (1) not an IL-subset of any state in $M_H^{E_i}$, or (2) an I-subset but not an IL-subset of such a state. This means that if two extensions that fail this part of the analysis derived the same host construct and consequently had LR(0)-equivalent states in their partitions $M_{NH}^{E_i}$, the analysis could not guarantee that, when those LR(0)-equivalent states were merged together in LALR(1) DFA construction, the merged state would not yield a conflict.

The dimension types extension has such a state in it. In our grammar for Java, there is a nonterminal ***primaryExpressionAndArrayCreation*** representing either primary expressions or array initializers (*e.g.*, `new int[100]`). Constructs derived from this nonterminal can occur in two contexts: as part of a constructor invocation, or as the beginning of a general expression.

The dimension types extension references this nonterminal in a new context, where items pertaining to general expressions and constructor invocations will not be present. This causes several $M_{NH}^{E_i}$ states to be created. Furthermore, the lookahead in this new context is different, causing the $M_{NH}^{E_i}$ states not to be IL-subsets of those that parse ***primaryExpressionAndArrayCreation*** constructs in the original contexts, which causes the modular determinism analysis to fail.

However, all other requirements of *Partitionable* are met; hence, the extension passes $Partitionable_{PT}$.

9.4.6 Discussion.

Although Java 1.4 itself is an LALR(1) language that can be parsed using the traditional framework, another approach is needed for parsing many of the extensions; the SQL extension, for example, with its different lexical syntax. Context-aware scanning and the modular determinism analysis help a good deal in this area, both for those extensions that are closer to embedded languages and those that heavily back-reference the host grammar; indeed, extensions to large and syntactically-rich languages are more likely to pass the modular determinism analysis.

Additionally, ableJ with its several extensions gave us a ready-made test bed for the modular determinism analysis of chapter 6; the practicality of the analysis was confirmed when several of our already-existing extensions passed it.

Chapter 10

Conclusion.

In this chapter we provide a further discussion of the three primary contributions of this thesis — context-aware scanning, the modular determinism analysis, and parse table composition — as well as listing some applications for context-aware scanning beyond that of extensible languages and speculating on the directions future work in this field could take.

10.1 Discussion of contributions.

In this section is provided an evaluation of the three primary contributions of this thesis and their relation to each other. The modular determinism analysis is considered to be the primary contribution, as it opens the door to the development of truly extensible compilers; parse table composition being a practical modification and context-aware scanning serving as an independently useful basis.

10.1.1 Context-aware scanning.

Although the modular determinism analysis has greater theoretical implications, we will discuss context-aware scanning first, as it was the basis making the analysis practical. We will first measure it according to the criteria listed in chapter 2, then briefly discuss some possible further applications for the framework and examine the question of the greater modularity of a disjoint scanner, arguing that in theory, little modularity is sacrificed when compared to approaches such as SGLR or PEGs.

10.1.1.1 Measuring the framework.

In chapter 2 we listed five criteria for evaluating parsers and parsing frameworks: verifiability, efficiency, expressivity, flexibility, and declarativeness, some of these being objective and easily quantified, others subjective. Here we argue that context-aware scanning's greatest strength is in its ability to provide guarantees of determinism, so it is best suited to applications where verifiability is critical.

Verifiability. We defined *verifiability* as being the framework's capacity to provide a guarantee that parsers built within it will function correctly on all inputs. Although this is broadly a subjective measurement, parts of it are objective, such as the question of whether there are ambiguities in a language's grammar.

Being LALR(1)-based, the framework retains the verifiability of the traditional LALR(1) framework, which is greater than that of a GLR-based or PEG framework. The GLR-based frameworks offer no determinism guarantee, while PEGs do not provide a determinism guarantee so much as an automatic compile-time *fix* for ambiguities: while the LALR(1) process can verify that there are no ambiguities in the grammar at all, the PEG process simply resolves the ambiguities, perhaps without the grammar writer knowing about them, by enforcing a precedence order on productions.

On the lexical side, since lexical precedence relations must be specified explicitly, the grammar writer is able to verify that the grammar contains no unintended lexical conflicts, rather than simply having such conflicts resolved automatically, as is done with a traditional scanner generator.

Efficiency. The multiple-DFA implementation of the context-aware scanner allows for a time complexity equivalent to that of the traditional framework, while the single-DFA implementation requires, for each character interpreted, three to five operations on bit vectors that are the length of the number of terminals. The runtime of the single-DFA implementation is still linear with respect to input size, so it is not nearly so time-complex as approaches such as GLR, but the constant multiplier is dependent on the size of the grammar. The runtimes of both implementations are reasonably comparable to parsers generated by CUP and JFlex.

Expressivity. We defined *expressivity* as the ability of a framework to build parsers for a broad range of grammars.

The framework has a level of expressivity lying between that of the traditional LALR(1) framework and that of a GLR-based framework. As the expressivity of the PEG framework is unclear, we make no comparison on this front.

The increase in expressivity the framework provides over the traditional LALR(1) framework is most visible in situations where disambiguation by syntactic context is needed: where if the scanner assumes all terminals in the grammar to be valid at any point, it will result in an incorrect parse.

AspectJ serves as an example of this, with context being needed to differentiate between the lexical modes used explicitly in the `abc` scanner, which can be handled automatically here instead of needing that explicit specification. A simpler example

may be found in the case of embedding regular expressions in programming languages, as discussed in section 4.4.3.

Flexibility. We defined *flexibility* as the subjective measure of the capacity of a framework to accommodate changes to parsers built within it; that a more flexible framework will be able to accommodate a broader range of changes to such a parser while keeping it within the framework.

As with expressivity, the flexibility of our framework lies between that of the traditional LALR(1) framework and the GLR-based frameworks. In chapter 2 we stated that the LALR(1) framework, while somewhat flexible on the token level, was less flexible on the character level due to the scanner-parser separation. By contrast, the context-aware scanner allows the token-level flexibility of the traditional LALR(1) parser to be extended to the character level as well; the strongest evidence of this is the heightened ability to add new extensions to languages without lexical conflicts.

Declarativeness. We defined *declarativeness* as the ability to specify a greater proportion of parsers within the framework without resorting to extraformal means, such as using custom blocks of code within the scanner specification.

In declarativeness, our framework yet again lies between the traditional LALR(1) framework and the GLR- and PEG-based frameworks. While the SGLR and PEG frameworks both do not allow for the use of custom code in directing the actions of the parser, our framework, like the traditional LALR(1) framework, permits some use of such code (via disambiguation functions, where arbitrary code can be used to resolve an ambiguity between several terminals; an example is the disambiguation function used to resolve the ambiguity between typenames and identifiers in C — see section 9.2).

However, the traditional LALR(1) framework is less declarative than ours, as a

greater number and proportion of languages can be specified declaratively in our framework, without hacks such as the *ad hoc* scanner "modes" used in, for example, the `abc` AspectJ parser.

10.1.1.2 Further applications.

Although this thesis is primarily concerned with context-aware scanning's application to extensible languages and its role as the basis for the modular determinism analysis, this is by no means its only application. It is useful for any one of the range of applications in which a grammar's lexical syntax is too complex to be scanned without lexical disambiguation by context. Here we discuss some examples: language embedding, comment parsing, and languages with unreserved keywords.

Language embedding. There are many instances in which code of two different languages is mixed together in one file. For example, the Web scripting languages JSP and ASP contain, respectively, Java and VBScript embedded in HTML, bookended by `<%` and `%>` tags; JavaScript may also be embedded in HTML, set off with `<script>` tags. In C++, one can specify a code block marked `extern "C"`, in which only C code is permitted. In C, there are `asm` blocks to embed assembly code within C source files.

Hitherto these grammars have mostly been parsed with *ad hoc* parsers built by hand. There have also been constructed new formalisms, such as island grammars [SCD03], to break up the file into "water" and "islands," the "islands" representing the code to be parsed by a particular parser.

One of the problems with using the traditional LALR(1) framework to parse embedded languages is that, while the grammars for both the embedding and embedded languages are often LALR(1) in their own right, they have completely different lexical syntax that makes it impossible to produce a unified scanner for a parser to parse their

composition.

Context-aware scanning solves this problem automatically and effortlessly. The bookending terminals such as `<%` and `<script>` are effectively marking terminals and serve the same purpose — providing a degree of separation between the parts of the parser that parse the embedding and embedded languages.

An example of the automatic way in which context-aware scanning solves this problem is Promela, the input language to the Spin model checker (see section 9.3), which embeds C constructs and hitherto has needed two parsers due to the fact that much of the lexical syntax of Promela and C overlaps.

Comment parsing. A similar application to language embedding is that of comment parsing. The popular API documentation generator for Java, *Javadoc*, its cross-language counterpart *Doxygen*, and other similar tools make use of specially formatted comments immediately preceding external declarations. The formatting of these comments may clash with the lexical syntax of the outside language, in which case they will cause lexical conflicts and must be parsed by a secondary parser. Context-aware scanning would allow the specification of the syntax of such specially formatted comments within the language's main grammar.

Languages with unreserved keywords. One of the problems inspiring the development of that non-deterministic scanning construct, the Schrödinger's Token (see [AH04] and section 2.3.6) is the problem of languages with unreserved keywords. Aycock and Horspool give the example of PL/I, in which keywords such as `IF` can also be used as identifiers, making it possible to write statements such as `IF IF = THEN THEN THEN = IF`.

This makes a scanner for the language very difficult to implement in a traditional LALR(1) framework, as one cannot use different regular expressions for keywords and

identifiers without reserving the keywords. Without using non-deterministic constructs such as the Schrödinger's Token, one must resort to a non-declarative scanner specification.

Our framework makes non-reserved keywords much easier to implement as separate terminals. If the keyword and identifier terminals are not valid in the same context, they will not conflict lexically. If they do, one can implement a disambiguation group for the keyword and identifier terminals that disambiguates to the keyword terminal.

10.1.1.3 The merits of the parser-scanner dividing line.

In section 1.3.1 we briefly discussed the question of keeping the scanner and parser disjoint, agreeing that this was a good principle of compiler design — evidenced by the comparatively few tools that do *not* maintain this separation — but arguing that it had to be bent for the sake of parsing many reasonable languages for which parsing context was needed by the scanner. Throughout the chapters following we provided several examples of such languages.

We also note that our framework does not constitute a *complete* departure from the principle of a disjoint scanner and parser. With a scannerless framework such as SGLR or PEGs, the scanning and parsing apparatus are *inextricably* mixed together. On the other hand, in our framework, despite Copper being an integrated scanner and parser generator, the context-aware scanner maintains at least a partially independent existence within a Copper parser.

This means that, in theory, given a grammar specification, one could build a stand-alone context-aware scanner, which would then be able to take in a valid lookahead set according to the form spelled out in Definition 3.2.2 and return the correct match.

10.1.2 Modular determinism analysis.

We argue that the modular determinism analysis is the most significant theoretical contribution presented in this thesis, because it removes one of the last major barriers to the development of an extensible compiler where the programmer decides which language extensions to import, following up the development of such modular tools for semantic analysis as attribute grammars.

In section 1.2, we described our vision for extensible compilers, and how parsing — especially with the traditional LALR(1) framework — forms an "Achilles' heel" preventing the full development of such a compiler.

This is not altogether a fault of the LALR(1) algorithm, however; despite the completely unstructured nature of the string of characters that forms the initial input to most compilers, as is shown in the proofs of the modular determinism analysis, an LALR(1) parser does allow for a reasonable amount of differentiation between host and extension constructs. However, the utility of this is limited so long as the scanner is obliged to operate with no input from the parser. It is this problem that context-aware scanning resolves, thus making this property of LALR(1) parsers exploitable. This introduces a degree of modularity to syntactic analysis, thus enabling the semantic analysis that follows to be used to its full potential.

Hence, the use of context-aware scanning to increase the modularity of the parsing apparatus and process, and the use of the modular determinism analysis to guarantee the lack of conflicts in the syntactic analysis phase, removes this "Achilles' heel."

10.1.2.1 Restrictions vs. guarantees.

The modular determinism analysis provides the guarantees necessary to remove parsing as an obstacle to implementation of the desired sort of extensible compiler, but it does so at the cost of imposing restrictions on the grammars of language extensions. On the

other hand, some other parsing frameworks, such as the PEG framework and SGLR, while lacking the guarantees, have been applied in the field of extensible languages and also the same time allow more expressivity than our modular determinism analysis; comparisons between these frameworks and ours were drawn in section 10.1.1.

It is quite valid to ask the question, "Are these restrictions too severe?" Our answer is, essentially, *no*: that the moderate loss of expressivity imposed by the restrictions is justified by the fact that such a static analysis can be performed by the extension writer.

In our considerations of restrictions, we have not considered only the set laid out in Definition 6.3.10 to define the set *Partitionable*. We previously [VWKSB07] experimented with a tighter set of restrictions that were simpler and easier to prove. Instead of simply requiring the use of a marking terminal on the bridge production, we also put restrictions on "back references" to the host grammar within the extension grammar; specifically, that if an extension nonterminal $nt_E \in NT_i^E$ derives a host nonterminal $nt_H \in NT_H$, then the production by which such a derivation is effected must take one of two forms:

- $nt_E \to \mu^L\ nt_H\ \mu^R$ — μ^L and μ^R are host-language terminals used to "bookend" host expressions to make it clear where they begin and end.
- $nt_E \to \mu^L\ nt_H$ — for use only at the end of an extension expression (*e.g.*, if nt_E is the extension's start nonterminal).

However, many of our previously implemented extensions did not fit these strictures.

We also considered a relaxed version of the restrictions, in which the analysis would take heed of the nonterminal on the left-hand side of the bridge production. Specifically, it would run the analysis with respect to a set A of host nonterminals; the analysis only guarantees the composability of the tested extension with extensions that have a nonterminal in the set A on the left-hand sides of their bridge productions. This

exploits the fact that most extensions derive from just one or two host nonterminals (*e.g.*, "expression" or "statement"); however, it increases the complexity of the analysis and it is unclear whether it has any practical benefits.

In summary, we have sought a middle ground in our choice of restrictions; seeking a set that would not restrict expressivity too greatly while not making an overly complicated test. Again, we believe that the restrictions are justified by the guarantees the modular determinism analysis provides.

10.1.3 Parse table composition.

Parse table composition is primarily a practical improvement on the modular determinism analysis rather than a theoretical contribution in its own right. Recall from chapters 6 and 7 that there are two versions of the modular determinism analysis, *Partitionable* and $Partitionable_{PT}$; that *Partitionable* guarantees that a set of extensions can be composed on the grammar level *or* the parse table level and will compile without error, while $Partitionable_{PT}$ only guarantees that on the parse table level; and that $Partitionable_{PT}$ is a superset of *Partitionable*, so any extension passing the original modular determinism analysis also passes the modified one.

However, most of the work necessary to prove the correctness of this modified analysis is laid out in the proofs for the original analysis, and for that reason we have described parse table composition as a *corollary* of the original determinism analysis work [SVW10].

That being said, parse table composition does offer more possibilities to the development of extensible compilers than the modular determinism analysis alone. Returning to the discussion of automatically generated compilers from the above section, and the analogy with importing libraries, while the modular determinism analysis can guarantee that a parser is able to be *compiled* automatically from several parts, such a parser

still has to be compiled *after* the parts are put together: analogous to the static linking of code libraries. On the other hand, parse table composition, with its pre-compiled parse tables, offers a first step toward a parser that is fully compiled prior to being put together — analogous to dynamic linking of libraries.

10.2 Future work.

There are several directions that future work on the ideas presented in this thesis could take.

10.2.1 Heuristics for scanner DFA sharing.

The multiple-DFA approach to context-aware scanning is currently usable to parse languages with smaller grammars, such as C, but is not practical for the larger languages, such as Java.

Already we have presented an heuristic for optimization as a start toward remedying this problem; future work in this area should focus on improvement of more heuristics. The final goal should be the ability to generate, automatically, a set of DFAs whose size is identical to the number of modes in the smallest hand-constructed "moded" scanner that could be made for the language. For example, the `abc` scanner for AspectJ (see section 9.1) makes use of only four different lexical contexts; the ideal heuristic would produce automatically for AspectJ a scanner with only four DFAs to it.

The reason why the multiple-DFA approach is impractical at present is that there is a great deal of unnecessary duplication of scanner DFA states; namely, that using only the "union adequacy" heuristic presented in section 5.3.3, a scanner DFA is built for a fair portion of the hundreds of valid lookahead sets in a fair-sized grammar, while just one DFA, or a handful of DFAs, may be sufficient to scan the language. This is

evidenced by the fact that many languages can be parsed perfectly well in the traditional LALR(1) framework with only one DFA, and that those needing further disambiguation by context can often be parsed with an *ad hoc* "moded" scanner making use of just a few "modes."

Theorem 5.3.1 gives a formal means by which one DFA may be used to scan on several valid lookahead sets. The problem then becomes one of optimization: given a set of valid lookahead sets, find the set of DFAs with the least number of states such that each valid lookahead set S can be mapped to a DFA that matches the constraints laid out in Theorem 5.3.1. Possible characteristics of the lexical syntax to exploit include:

- Many terminals are keywords, whose regular expressions have languages containing a single string.
- In an extensible language, the group of parse states belonging to a particular extension is more likely to be able to use a single scanner DFA.
- Examining the graph of the precedence relation may offer some insight into which groups of terminals are scanned for together.

10.2.2 Runtime parse table composition.

To date, we have only worked with composing extensions at compile time. Extensions that passed the modular determinism analysis were originally intended to be distributed as grammar fragments that were then composed (usually by concatenating grammar specifications) and compiled in the traditional way using a parser generator. Parse table composition added to that by allowing the grammar fragments to be pre-compiled.

Both of these approaches operate at *parser compile time* — when the parser is built rather than when it is run. However, there are also some applications for a process to compose extensions at the time when the parser is run.

This would be analogous to the practice in Java of using `import` statements to import library classes. If a parser for a programming language was built to recognize a similar import statement for *extensions*, then, it could compose the "imported" extension on the fly.

Although full recompilation of a grammar to fit the extension would take an impractical amount of time for this application, using pre-compiled extensions and parse table composition might make it practical, as one could simply assemble the host and extension parse tables, and generate a scanner DFA for the extension's marking terminal, on the fly as part of the semantic action on the production representing the import statement. One could also compose the extensions at compile time as usual, but "turn off" an extension's marking terminal — *i.e.*, prevent it from being matched in the scanner — unless the extension is imported. This would prevent any extension constructs at all from being parsed if the extension is not imported.

10.2.3 Extension-specific lexical precedence.

Lexical precedence relations as described in section 4.2 are convenient for resolving ambiguities in lexical syntax. However, as discussed in section 6.6.3, specifying precedence relations between extension and host terminals is discouraged, since these precedence relations have effect in *all* parse states and, if an extension terminal takes precedence over a host terminal, the set of strings matching the host terminal will be altered even in a host context.

This would not, however, be a problem if the effect of such precedence relations were limited to the extension-owned states. Then the extension terminal would take precedence while parsing extension constructs, and would not do so while parsing host constructs.

10.2.4 Informal guidelines for extension grammar design.

There exists a guide, based on practical experience, on how to "debug" grammars for the SGLR framework to remove any ambiguity [Vin07]. There is a similar body of empirical data used to guide writers of traditional LALR(1) grammars in resolving shift-reduce conflicts.

Unlike a grammar in the SGLR framework, an extension in our framework can be verified by the modular determinism analysis to compose deterministically with other extensions. However, like with conflicts in LALR(1) parse tables, it would be useful to have a practical guide on how to resolve any small issues that make an extension grammar fail the analysis.

References

[AH04] John Aycock and R. Nigel Horspool, *Schrödinger's token*, Software: Practice and Experience **31** (2004), no. 8, 803–814.

[Ana02] C. Scott Ananian, *Readme for CUP Java grammars*, On the World Wide Web at URL: `http://www.cs.princeton.edu/~appel/modern/java/CUP/javagrm-README.txt` and `http://www.cs.princeton.edu/~appel/modern/java/CUP/javagrm.tar.gz`, 2002.

[AP02] Andrew W. Appel and Jens Palsberg, *Modern compiler implementation in Java*, 2nd ed., Cambridge University Press, Cambridge, 2002.

[Asp03] AspectJ Team, *AspectJ programming guide, appendix C*, On the World Wide Web at URL: `http://www.eclipse.org/aspectj/doc/next/progguide/implementation.html`, 2003.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers – principles, techniques, and tools*, Addison-Wesley, Reading, MA, 1986.

[BETV06] Martin Bravenboer, Éric Tanter, and Eelco Visser, *Declarative, formal, and extensible syntax definition for AspectJ*, Proc. of Conf. on Object-oriented programming systems, languages, and applications (OOPSLA), ACM, 2006, pp. 209–228.

[Boa99] Mars Climate Orbiter Mishap Investigation Board, *Phase I Report*, On the Internet at URL: `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`, 1999.

[CFR06] Julien Cervelle, Rémi Forax, and Gilles Roussel, *Tatoo: an innovative parser generator*, Proc. Principles and practice of programming in Java (PPPJ), ACM, 2006, pp. 13–20.

[Den10] Joel Denny, *PSLR(1): Pseudo-scannerless minimal LR(1) for the deterministic parsing of composite languages*, Ph.D. thesis, CLemson University, 2010.

[DM07] Lars Dölle and Heike Manns, *The Styx handbook*, On the World Wide Web at URL: `http://speculate.de/styx/styx.html`, 2007.

[DM08] Joel E. Denny and Brian A. Malloy, *Ielr(1): practical lr(1) parser tables for non-lr(1) grammars with conflict resolution*, SAC '08: Proceedings of the 2008 ACM symposium on Applied computing (New York, NY, USA), ACM, 2008, pp. 240–245.

[EH07] Torbjörn Ekman and Görel Hedin, *The JastAdd extensible Java compiler*, Proc. Conf. on Object oriented programming systems and applications (OOPSLA), ACM, 2007, pp. 1–18.

[EKV09] Giorgios Economopoulos, Paul Klint, and Jurgen Vinju, *Faster scannerless glr parsing*, CC '09: Proceedings of the 18th International Conference on Compiler Construction (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 126–141.

[Far91] R.N. Farshi, *GLR parsing of e-Grammars*, Generalized LR Parsing (Masaru Tomita, ed.), Kluwer Academic Publishers, 1991, pp. 61–76.

[FEC04] Robert Filman, Tzilla Elrad, and Siobhán Clarke, *Aspect-oriented software development*, Addison-Wesley Professional, 2004.

[For04] Bryan Ford, *Parsing expression grammars: a recognition-based syntactic foundation*, Proc. of Symp. on Principles of Programming Languages (POPL), ACM, 2004, pp. 111–122.

[Gag98] Etienne Gagnon, *Sablecc, an object-oriented compiler framework*, In Proceedings of TOOLS, 1998, pp. 140–154.

[GBJL02] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen, *Modern compiler design*, John Wiley, 2002.

[GJS96] James Gosling, Bill Joy, and Guy Steele, *The java language specification*, 1 ed., Addison-Wesley, 1996.

[Gri06] Robert Grimm, *Better extensibility through modular syntax*, PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA), ACM Press, 2006, pp. 38–51.

[Gro89] Josef Grosch, *Generators for high-speed front-ends*, Compiler Compilers and High Speed Compilation **371** (1989), 81–92.

[HdMC] Laurie Hendren, Oege de Moor, and Aske Simon Christensen, *Differences with `ajc`, and limitations*, On the World Wide Web at URL: `http://abc.comlab.ox.ac.uk/differences`.

[HdMC04] ———, *The abc scanner and parser, including an LALR(1) grammar for AspectJ*, Available at `http://abc.comlab.ox.ac.uk/documents/scanparse.pdf`, September 2004.

[Her02] Jorrit N. Herder, *Aspect-oriented programming with AspectJ*, 2002.

[HFA+06] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, Andrew W. Appel, and Michael Petter, *Cup user's manual*, On the World Wide Web at URL: `http://www2.cs.tum.edu/projects/cup/manual.html`, 2006.

[Hol03] Gerard Holzmann, *Spin model checker, the: primer and reference manual*, Addison-Wesley Professional, 2003.

[JSE04] A. Johnstone, E. Scott, and G. Economopoulos, *Generalized parsing: Some costs*, Proc. International Conf. on Compiler Construction, Lecture Notes in Computer Science, vol. 2985, Springer-Verlag, 2004, pp. 89–103.

[Kan04] Isaiah Pinchas Kantorovitz, *Lexical analysis tool*, SIGPLAN Not. **39** (2004), no. 5, 66–74.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, *An overview of AspectJ*, ECOOP 2001 Object–Oriented Programming (J. L. Knudsen, ed.), LNCS, vol. 2072, 2001, pp. 327–353.

[Kle09] Gerwin Klein, *JFlex user's manual*, On the World Wide Web at URL: `http://jflex.de/manual.html`, 2009.

[Knu65] D. E. Knuth, *On the translation of languages from left to right*, Information and Control **8** (1965), no. 6, 607–639.

[KR02] Brian W. Kernighan and Dennis M. Ritchie, *The c programming language*, 2 ed., Prentice Hall, Englewood Cliffs, NJ, USA, 2002.

[Kra99] Douglas Kramer, *API documentation from source code comments: a case study of Javadoc*, SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation (New York, NY, USA), ACM, 1999, pp. 147–153.

[LMB92] John Levine, Tony Mason, and Doug Brown, *Lex & yacc*, 2nd ed., O'Reilly, 1992.

[MÖ6] Hanspeter Mössenböck, *The compiler generator Coco/R*, On the World Wide Web at URL: `http://www.ssw.uni-linz.ac.at/coco/Doc/UserManual.pdf`, 1990–2006.

[MN04] S. McPeak and G. C. Necula, *Elkhound: a fast, practical GLR parser generator*, Proc. International Conf. on Compiler Construction, Lecture Notes in Computer Science, vol. 2985, Springer-Verlag, 2004, pp. 73–88.

[Naw91] Jerzy R. Nawrocki, *Conflict detection and resolution in a lexical analyzer generator*, Inf. Process. Lett. **38** (1991), no. 6, 323–328.

[NS06] Mark-Jan Nederhof and Giorgio Satta, *Probabilistic parsing strategies*, J. ACM **53** (2006), no. 3, 406–436.

[OW97] Martin Odersky and Philip Wadler, *Pizza into Java: translating theory into practice*, Proc. of Symp. on Principles of Programming Languages (POPL), ACM Press, 1997, pp. 146–159.

[OW98] M. Odersky and P. Wadler, *Leftover Curry and reheated Pizza: How functional programming nourishes software reuse*, IEEE Fifth International Conference on Software Reuse (Vancouver, BC), 1998, Keynote address.

[PQ95] T. J. Parr and R. W. Quong, *ANTLR: a predicated-LL(k) parser generator*, Softw. Pract. Exper. **25** (1995), no. 7, 789–810.

[RDGN10] Lukas Renggli, StÃl'phane Ducasse, Tudor GÃőrba, and Oscar Nierstrasz, *Practical dynamic grammars for dynamic languages*, 4th Workshop on Dynamic Languages and Applications (New York, NY, USA), ACM, 2010.

[Rek92] J. Rekers, *Parser generation for interactive environments*, Ph.D. thesis, University of Amsterdam, 1992.

[RH98] T. Rus and T. Halverson, *A language independent scanner generator*, Paper available at `http://www.uiowa.cs.edu/~rus`, 1998.

[Sal90] Daniel Joseph Salomon, *Metalanguage enhancements and parser-generation techniques for scannerless parsing of programming languages*, Ph.D. thesis, University of Waterloo, 1990.

[SC89] D. J. Salomon and G. V. Cormack, *Scannerless NSLR(1) parsing of programming languages*, SIGPLAN Not. **24** (1989), no. 7, 170–178.

[SCD03] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean, *Robust multilingual parsing using island grammars*, CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, 2003, pp. 266–278.

[Sch09] August Schwerdfeger, *A declarative specification of a deterministic parser and scanner for AspectJ*, Tech. Report 09-007, University of Minnesota, 2009, Available at `http://www.cs.umn.edu`.

[Sip06] Michael Sipser, *Introduction to the theory of computation*, 2nd ed., Thomson Course Technology, Boston, 2006.

[SJE07] Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos, *BRNGLR: a cubic Tomita-style GLR parsing algorithm*, Acta Inf. **44** (2007), no. 6, 427–461.

[SVW09] August Schwerdfeger and Eric Van Wyk, *Verifiable composition of deterministic grammars*, Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, June 2009.

[SVW10] A. Schwerdfeger and E. Van Wyk, *Verifiable parse table composition for deterministic parsing*, Software Language Engineering, Lecture Notes in Computer Science, vol. 5969/2010, Springer-Verlag, 2010, pp. 184–203.

[Tom86] M. Tomita, *Efficient parsing for natural language.*, Int. Series in Engineering and Computer Science, Kluwer Academic Publishers, 1986.

[vdBSVV02] M. van den Brand, J. Scheerder, J. Vinju, and E. Visser, *Disambiguation filters for scannerless generalized lr parsers*, Proc. 11th International Conf. on Compiler Construction, LNCS, vol. 2304, 2002, pp. 143–158.

[Vin07] Jurgen Vinju, *SDF disambiguation medkit for programming languages*, On the World Wide Web at URL: `http://homepages.cwi.nl/~daybuild/daily-books/howto/sdf-disambiguation/sdf-disambiguation.html`, 2007.

[Vis97] Eelco Visser, *Scannerless generalized-LR parsing*, Tech. Report P9707, Programming Research Group, University of Amsterdam, August 1997.

[Vis01] ———, *Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5*, Rewriting Techniques and Applications (RTA'01) (A. Middeldorp, ed.), Lecture Notes in Computer Science, vol. 2051, Springer-Verlag, May 2001, pp. 357–361.

[VWJ04] E. Van Wyk and E. Johnson, *An extensible language framework for java*, Tech. report, University of Minnesota, Department of Computer Science and Engineering, 2004.

[VWKSB07] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin, *Attribute grammar-based language extensions for Java*, European Conf. on Object Oriented Programming (ECOOP), LNCS, vol. 4609, Springer-Verlag, July 2007, pp. 575–599.

[VWS07] Eric Van Wyk and August Schwerdfeger, *Context-aware scanning for parsing extensible languages*, Intl. Conf. on Generative Programming and Component Engineering, (GPCE), ACM Press, October 2007.

[WBGM10] Xiaoqing Wu, Barrett R. Bryant, Jeff Gray, and Marjan Mernik, *Component-based lr parsing*, Comput. Lang. Syst. Struct. **36** (2010), no. 1, 16–33.

[WDM08] Alessandro Warth, James R. Douglass, and Todd Millstein, *Packrat parsers can support left recursion*, PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (New York, NY, USA), ACM, 2008, pp. 103–110.

[WG97] Tim A. Wagner and Susan L. Graham, *Incremental analysis of real programming languages*, PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1997, pp. 31–43.

Appendix A

Grammars.

This appendix contains the full grammars for the applications of context-aware scanning discussed in chapter 9: ableC (section A.1, beginning on this page), Promela (section A.2, beginning on page 255), and ableJ with several of its extensions (section A.3, beginning on page 269). It also includes the toy grammar used in performance testing (section A.4, beginning on page 293).

In the grammar specifications, we use different fonts to denote host and extension symbols, terminals and nonterminals: ***host nonterminal***; **extension nonterminal**; *host terminal*; extension terminal.

A.1 ableC.

This grammar is a direct adaptation of the ANSI C grammar in the second edition of Kernighan and Ritchie's *The C Programming Language* [KR02] with some extensions added to parse GCC-specific extensions from header files. For brevity we do not enumerate the entire grammar, but list those of its terminals and nonterminals that are referenced in the Promela grammar or are pertinent to our system's innovations; it is

available for download at `http://melt.cs.umn.edu`.

Terminals:

- *Identifier* with regular expression `[A-Za-z_][A-Za-z_0-9]*`
- *LCurly* with regular expression `[\{]`
- *LParen* with regular expression `[\(]`
- *RCurly* with regular expression `[\}]`
- *RParen* with regular expression `[\)]`
- *STRUCT* with regular expression `struct`
- *TypeName* with regular expression `[A-Za-z_][A-Za-z_0-9]*`
- *UNION* with regular expression `union`

Nonterminals:

Ansi_C_DeclarationList, ***DeclarationSpecifiers***, ***Declaration***, ***Ansi_C_Expr***, ***Pointer***, ***Ansi_C_StmtList StorageClassSpecifier***, ***StructOrUnionSpecifier***, ***TypeSpecifier***,

Parser attributes:

- *typenames* : Σ^{***} (list of lists of strings) initialized to empty. Each list of strings represents a scope for variable declaration, while the list *typenames* represents a stack of these scopes.
- *sawTypedef* : $\mathbb{Z}$ initialized to 0
- *inTypedef* : $\mathbb{Z}$ initialized to 0
- *sawStruct* : $\mathbb{Z}$, measuring nesting within `struct` declarations, initialized to 0
- *sawPointer* : $\mathbb{Z}$ initialized to 0
- *passedPointer* : $\mathbb{Z}$ initialized to 0
- *sawParenthesis* : $\mathbb{Z}$, measuring parenthesis nesting, initialized to 0

Semantic actions:

- On terminal *LCurly*, push on a new variable scope inheriting the declarations of its parent (the top of the scope stack):

```
typenames = [listCopy(head(typenames))] +++ typenames;
```

- On terminal *RCurly*, pop off a variable scope:

```
typenames = tail(typenames);
```

- On terminal *LParen*, increment the parenthesis nesting counter:

```
sawParenthesis = sawParenthesis + 1;
```

- On terminal *RParen*, decrement the parenthesis nesting counter:

```
sawParenthesis = sawParenthesis - 1;
```

- On productions ***StructOrUnion*** $\rightarrow$ *STRUCT* and ***StructOrUnion*** $\rightarrow$ *UNION*, increment the struct nesting counter:

```
sawStruct = sawStruct + 1;
```

- On production ***StorageClassSpecifier*** → *Typedef*, mark that the keyword `typedef` has been seen:

```
sawTypedef = 1;
```

- On production ***DeclarationSpecifiers*** → ***DeclSpecList***, if the keyword `typedef` has been seen, mark the declarations derived from this production as typedefs:

```
inTypedef = sawTypedef;
```

- On production ***TypeSpecifier*** → ***StructOrUnionSpecifier***, which encloses an entire struct specification, decrement the struct nesting counter:

```
sawStruct = sawStruct - 1;
```

- On all productions with ***Pointer*** as a left hand side:

```
sawPointer = sawParenthesis;
```

- On productions ***Declaration*** → ***DeclarationSpecifiers InitDeclaratorList*** *Semi* and *hostntDeclaration* → ***DeclarationSpecifiers*** *Semi* representing an entire variable declaration, reset markers of typedefs and pointers:

```
inTypedef = 0;
sawTypedef = 0;
sawPointer = 0;
passedPointer = 0;
```

- On production ***DirectDeclarator*** → *Id* representing the variable name being declared in a declaration, check the parser attributes to see that (1) we are not in a `struct`; (2) we are in a `typedef`; (3) the name has not already been defined as a type. If all three conditions are met, add the name to the list of types.

```
local attribute pendingTypename :: String;
pendingTypename = id.lexeme;
local attribute addCondition :: Integer;
addCondition = if (sawStruct == 0 && inTypedef == 1) &&
                  passedPointer == 0 &&
                  !listContains(pendingTypename,
                                head(typenames))
               then 1
               else 0;
 typenames = if addCondition == 1
             then [head(typenames) ++ [pendingTypename]]
```

```
                    +++ tail(typenames)
                else typenames;
        passedPointer = if passedPointer == 0
                    then addCondition
                    else 1;
```

Disambiguation function: On terminal group {*Identifier*, *TypeName*}:

```
return if listContains(lexeme,head(typenames))
      then TypeName_t
      else Identifier_t;
```

A.2 Promela.

Start nonterminal is **Program**.

Marking terminals for C:

- Terminal C_CODE with regular expression `c_code`
- Terminal C_STATE with regular expression `c_state`
- Terminal C_EXPR with regular expression `c_expr`
- Terminal C_TRACK with regular expression `c_track`
- Terminal C_DECL with regular expression `c_decl`

Promela terminals:

- ACTIVE with regular expression `active`
- AND with regular expression `[&][&]`
- AND with regular expression `[&]`

- ASGN with regular expression `[=]`
- ASSERT with regular expression `assert`
- ATOMIC with regular expression `atomic`
- ATRATE with regular expression `[@]`
- BIT with regular expression `bit`
- BOOL with regular expression `bool`
- BREAK with regular expression `break`
- BYTE with regular expression `byte`
- $\text{BlockComment}_{\text{P}}$ with regular expression

  `[/][\*]([^\*]|[\r\n]|([\*]+([^\*/]|[\r\n])))*[\*]+[/]`
- CHAN with regular expression `chan`
- CHARLIT with regular expression `['](([^'])|[\\]['])*[']`
- CLAIM with regular expression `never`
- COMMA with regular expression `[,]`
- CONST with regular expression `(true)|(false)|(skip)|[0-9]*`
- DECR with regular expression `[\-][\-]`
- DIV with regular expression `[/]`
- DO with regular expression `do`
- D_PROCTYPE with regular expression `D[_]proctype`
- D_STEP with regular expression `d[_]step`
- EE with regular expression `[=][=]`
- ELSE with regular expression `else`
- EMPTY with regular expression `empty`
- ENABLED with regular expression `enabled`
- EVAL with regular expression `eval`
- FI with regular expression `fi`
- FULL with regular expression `full`
- GE with regular expression `[>][=]`
- GOTO with regular expression `goto`

- GT with regular expression `[>]`
- HIDDEN with regular expression `hidden`
- ID with regular expression `[a-zA-Z_][a-zA-Z_0-9]*`
- IF with regular expression `if`
- INAME with regular expression `[a-zA-Z_][a-zA-Z_0-9]*`
- INCR with regular expression `[\+][\+]`
- INIT with regular expression `init`
- INLINE with regular expression `inline`
- INT with regular expression `int`
- ISLOCAL with regular expression `local`
- LCURLY with regular expression `[\]`
- LE `with regular expression [<][=]`
- LEN `with regular expression len`
- LPAREN `with regular expression [\(]`
- LSHIFT `with regular expression [<][<]`
- LSQUARE `with regular expression [\[]`
- LT `with regular expression [<]`
- LineComment$_{\mathrm{P}}$ `with regular expression [/][/].*`
- MINUS `with regular expression [\-]`
- MOD `with regular expression [%]`
- MTYPE `with regular expression mtype`
- NE `with regular expression [!][=]`
- NEMPTY `with regular expression nempty`
- NFULL `with regular expression nfull`
- NONPROGRESS `with regular expression np[_]`
- OD `with regular expression od`
- OF `with regular expression of`
- OR `with regular expression [\|][\|]`
- OR `with regular expression [\|]`

- O_SND with regular expression `[!][!]`
- PC_VALUE with regular expression `pc[_]value`
- PID with regular expression `pid`
- PLUS with regular expression `[\+]`
- PNAME with regular expression `[a-zA-Z_][a-zA-Z_0-9]*`
- PRINTF with regular expression `printf`
- PRINTM with regular expression `printm`
- PRIORITY with regular expression `priority`
- PROCTYPE with regular expression `proctype`
- PROVIDED with regular expression `provided`
- RCURLY with regular expression `[\]`
- RCV with regular expression `[\?]`
- RPAREN with regular expression `[\)]`
- RSHIFT with regular expression `[>][>]`
- RSQUARE with regular expression `[\]]`
- RUN with regular expression `run`
- R_RCV with regular expression `[\?][\?]`
- SCOLON with regular expression `:`
- SEMI with regular expression `(\->)|(;)`
- SEP with regular expression `[\`
- SHORT with regular expression `short`
- SHOW with regular expression `show`
- SKIP with regular expression `skip`
- SND with regular expression `[!]`
- STAR with regular expression `[\*]`
- STRING with regular expression `["]([^"]|[\\]["])*["]`
- TILD with regular expression `[ ]`
- TIMEOUT with regular expression `timeout`
- TRACE with regular expression `(trace)|(notrace)`

- TYPEDEF with regular expression `typedef`
- UNAME with regular expression `[a-zA-Z_][a-zA-Z_0-9]*`
- UNLESS with regular expression `unless`
- UNSIGNED with regular expression `unsigned`
- WhiteSpace$_P$ with regular expression `[\t\n ]+`
- XOR with regular expression `[\^]`
- XU with regular expression `(xr)|(xs)`

Promela productions:

- **Aname** →

  ID | PNAME
- **Arg** →

  Expr COMMA **Arg** | **Expr**
- **Args** →

  Arg | ε
- **Asgn** →

  ASGN | ε
- **BaseType** →

  BYTE | BIT | SHORT | BOOL | INT | UNAME | CHAN | UNSIGNED | MTYPE | **Error** | PID
- **Body** →

  LCURLY **Sequence OS** RCURLY
- **C_Fcts** →

  Ccode | **Cstate**
- **Ccode** →

  C_CODE_nt | **C_DECL_nt**

- **Cexpr** →

  C_EXPR_nt

- **ChInit** →

  LSQUARE CONST RSQUARE OF LCURLY **TypList** RCURLY

- **Claim** →

  CLAIM **Body**

- **Cstate** →

  C$_S$TATE STRING STRING

  | C$_S$TATE STRING STRING STRING

  | C_TRACK STRING STRING STRING

  | C_TRACK STRING STRING

- **DeclList** →

  OneDecl | **OneDecl** SEMI **DeclList**

- **Decl** →

  DeclList | ε

- **Error** →

  Type SCOLON CONST

- **Events** →

  TRACE **Body**

- **Expr** →

  Expr AND **Expr**

  | **Expr** PLUS **Expr**

  | **Expr** XOR **Expr**

  | **Expr** STAR **Expr**

| MINUS **Expr**

| **Expr** LT **Expr**

| **Expr** EE **Expr**

| ENABLED LPAREN **Expr** RPAREN

| **Cexpr**

| **Expr** GT **Expr**

| **Varref** R_RCV LSQUARE **RArgs** RSQUARE

| **Varref** RCV LSQUARE **RArgs** RSQUARE

| LPAREN **Expr** SEMI **Expr** SCOLON **Expr** RPAREN

| **Expr** AND **Expr**

| PNAME SCOLON **Pfld**

| LEN LPAREN **Varref** RPAREN

| **Expr** GE **Expr**

| PNAME LSQUARE **Expr** RSQUARE ATRATE ID

| TILD **Expr**

| PNAME ATRATE ID

| **Expr** MOD **Expr**

| **Expr** RSHIFT **Expr**

| **Varref**

| **Expr** DIV **Expr**

| SND **Expr**

| LPAREN **Expr** RPAREN

| PNAME LSQUARE **Expr** RSQUARE SCOLON **Pfld**

| **Expr** OR **Expr**

| **Expr** LE **Expr**

| CONST

| **Expr** OR **Expr**

| TIMEOUT

| RUN **Aname** LPAREN **Args** RPAREN **OptPriority**

| CHARLIT

| PC_VALUE LPAREN **Expr** RPAREN

| NONPROGRESS

| **Expr** MINUS **Expr**

| **Expr** NE **Expr**

| **Expr** LSHIFT **Expr**

- **Expression** →

  Probe

  | LPAREN **Expression** RPAREN

  | **Expression** OR **Expression**

  | **Expr** AND **Expression**

  | **Expression** AND **Expression**

  | **Expr** OR **Expression**

  | **Expression** OR **Expr**

  | **Expression** AND **Expr**

- **FullExpr** →

  Expr | **Expression**
- **IDList** →

  IDList COMMA | **IDList** ID | ID
- **IVar** →

  VarDcl ASGN **ChInit** | **VarDcl** | **VarDcl** ASGN **Expr**
- **Init** →

  INIT **OptPriority Body**
- **Inline_Args** →

  ID COMMA **Inline_Args** | ID | ε
- **Inst** →

  ε

  | ACTIVE

  | ACTIVE LSQUARE CONST RSQUARE

  | ACTIVE LSQUARE ID RSQUARE
- **MArgs** →

  Expr LPAREN **Arg** RPAREN | **Arg**
- **MS** →

  SEMI | **MS** SEMI
- **NS** →

  INLINE ID LPAREN **Inline_Args** RPAREN **Statement**

  | INLINE INAME LPAREN **Inline_Args** RPAREN **Statement**
- **OS** →

  SEMI | ε

- **OneDecl** →

  BaseType VarList

  | **Vis BaseType Asgn** LCURLY **IDList** RCURLY

  | **BaseType Asgn** LCURLY **IDList** RCURLY

  | **Vis BaseType VarList**
- **OptEnabler** →

  ε

  | PROVIDED LPAREN **FullExpr** RPAREN

  | PROVIDED **Error**
- **OptPriority** →

  ε | PRIORITY CONST
- **Option** →

  SEP **Sequence OS**
- **Options** →

  Option | **Option Options**
- **Pfld** →

  ID | ID LSQUARE **Expr** RSQUARE
- **PrArgs** →

  ε | COMMA **Arg**
- **Probe** →

  NEMPTY LPAREN **Varref** RPAREN

  | EMPTY LPAREN **Varref** RPAREN

  | NFULL LPAREN **Varref** RPAREN

  | FULL LPAREN **Varref** RPAREN

- **ProcType** →

  PROCTYPE | D_PROCTYPE

- **Proc** →

  Inst ProcType ID LPAREN **Decl** RPAREN **OptPriority OptEnabler Body**

- **Program** →

  ε | **Units**

- **RArg** →

  EVAL LPAREN **Expr** RPAREN

  | MINUS CONST

  | **Varref**

  | CONST

- **RArgs** →

  RArg LPAREN **RArgs** RPAREN

  | **RArg**

  | LPAREN **RArgs** RPAREN

  | **RArg** COMMA **RArgs**

- **Sequence** →

  Sequence MS Step | **Step**

- **Special** →

  BREAK

  | **Varref** RCV **RArgs**

  | DO **Options** OD

  | ID SCOLON **Stmt**

| GOTO ID

| IF **Options** FI

| **Varref** SND **MArgs**

- **Statement** →

  ASSERT **FullExpr**

| **Varref** RCV LT **RArgs** GT

| ATOMIC LCURLY **Sequence OS** RCURLY

| ELSE

| **FullExpr**

| PRINTF LPAREN STRING **PrArgs** RPAREN

| PRINTM LPAREN CONST RPAREN

| **Varref** O_SND **MArgs**

| INAME LPAREN **Args** RPAREN

| PRINTM LPAREN **Varref** RPAREN

| **Varref** INCR

| **Varref** R_RCV LT **RArgs** GT

| **Ccode**

| **Varref** R_RCV **RArgs**

| **Varref** DECR

| **Varref** ASGN **Expr**

| D_STEP LCURLY **Sequence OS** RCURLY

| LCURLY **Sequence OS** RCURLY

- **Step** →

  Stmt

  | XU **VrefList**

  | ID SCOLON XU

  | **Stmt** UNLESS **Stmt**

  | ID SCOLON **OneDecl**

  | **OneDecl**
- **Stmt** →

  Special | **Statement**
- **TypList** →

  BaseType | **BaseType** COMMA **TypList**
- **Type** →

  BaseType
- **Unit** →

  Proc | **Error** | **Events** | **Init** | **OneDecl** | **C_Fcts** | **NS** | **Claim** | **Utype** | SEMI
- **Units** →

  Units Unit | **Unit**
- **Utype** →

  TYPEDEF ID LCURLY **DeclList** RCURLY
- **VarDcl** →

  ID SCOLON CONST

  | ID LSQUARE CONST RSQUARE

  | ID

- **VarList** →

  IVar | **IVar** COMMA **VarList**

- **Varref** →

  Varref LSQUARE **Expr** RSQUARE

  | ID

  | **Varref** STOP ID

- **Vis** →

  SHOW | HIDDEN | ISLOCAL

- **VrefList** →

  Varref | **Varref** COMMA **VrefList**

Bridge productions to C:

- **Ansi_C_code** →

  DeclarationList | ***StmtList*** | ***DeclarationList StmtList*** | ε

- **C_CODE_nt** →

  C_CODE LSQUARE ***Expr*** RSQUARE LCURLY **Ansi_C_code** RCURLY

  | C_CODE LCURLY **Ansi_C_code** RCURLY

- **C_DECL_nt** →

  C_DECL LCURLY ***DeclarationList*** RCURLY

- **C_EXPR_nt** →

  C_EXPR LSQUARE ***Expr*** RSQUARE LCURLY **Ansi_C_code** RCURLY

  | C_EXPR LCURLY **Ansi_C_code** RCURLY

  | C_EXPR LCURLY ***Expr*** RCURLY

A.3 ableJ.

A.3.1 Host.

As the ableJ host grammar, we have adapted an LALR(1) grammar for Java 1.4 written by C. Scott Ananian for the CUP tool [Ana02]. As with the ableC grammar, we only list those of its terminals and nonterminals as are referenced in extensions. It is available for download at `http://melt.cs.umn.edu`.

Terminals:

- *AndAnd* with regular expression `[&][&]`
- *Colon* with regular expression `[\:]`
- *Dot* with regular expression `[\.]`
- *Eq* with regular expression `[=]`
- *Extends* with regular expression `extends`
- *For* with regular expression `for`
- *Gt* with regular expression `[>]`
- *Id* with regular expression `[a-zA-Z_$][0-9a-zA-Z_$]*`
- *Implements* with regular expression `implements`
- *Intconst* with regular expression `([0-9]+|0[xX][0-9a-fA-F]+)[lL]?`
- *Lbrace* with regular expression `[\{]`
- *Lbrack* with regular expression `[\[]`
- *Lparen* with regular expression `[\(]`
- *Lt* with regular expression `[<]`
- *Minus* with regular expression `[\-]`
- *Mul* with regular expression `[\*]`
- *New* with regular expression `new`
- *Not* with regular expression `[!]`
- *OrOr* with regular expression `[\|][\|]`

- *Plus* with regular expression `[\+]`
- *Question* with regular expression `[\?]`
- *Rbrace* with regular expression `[\}]`
- *Rbrack* with regular expression `[\]]`
- *Rparen* with regular expression `[\)]`
- *Semicolon* with regular expression `[;]`
- *Stringconst* with regular expression

```
["]([^"\\]|
    [\\][btnfr"'\\]|
    [\\][0-7]|
    [\\][0-7][0-7]|
    [\\][0-3][0-7][0-7]|
    [\\][u]+[0-9a-fA-F][0-9a-fA-F]?[0-9a-fA-F]?[0-9a-fA-F]?
   )*["]
```

- *This* with regular expression `this`
- *Void* with regular expression `void`
- *Xor* with regular expression `[\^]`

Nonterminals: ***classMemberDefinition***, ***constructorBody***, ***expression***, ***expressionListOpt***, ***implementsClause***, ***initializer***, ***interfaceMemberDefinition***, ***methodBody***, ***modifier***, ***modifiersOpt***, ***nameConcrete***, ***names***, ***parameterDeclaration***, ***parameterDeclarationList***, ***primaryExpression***, ***primaryExpressionAndArrayCreation***, ***primitiveType***, ***qualifiedNameConcrete***, ***referenceType***, ***statement***, ***superClassClause***, ***throwsClause***, ***type***, ***typeDefinition***

A.3.2 Extensions.

A.3.2.1 AspectJ.

Terminals:

- Adviceexecution$_{kwd}$ with regular expression `adviceexecution`
- After$_{kwd}$ with regular expression `after`
- Args with regular expression `args`
- Around$_{kwd}$ with regular expression `around`
- Aspect$_{kwd}$ with regular expression `aspect`
- Before$_{kwd}$ with regular expression `before`
- Call$_{kwd}$ with regular expression `call`
- Cflow$_{kwd}$ with regular expression `cflow`
- Cflowbelow$_{kwd}$ with regular expression `cflowbelow`
- Declare$_{kwd}$ with regular expression `declare`
- Error$_{kwd}$ with regular expression `error`
- Execution$_{kwd}$ with regular expression `execution`
- Get$_{kwd}$ with regular expression `get`
- Handler$_{kwd}$ with regular expression `handler`
- Initialization$_{kwd}$ with regular expression `initialization`
- Issingleton$_{kwd}$ with regular expression `issingleton`
- Parents$_{kwd}$ with regular expression `parents`
- Percflow$_{kwd}$ with regular expression `percflow`
- Percflowbelow$_{kwd}$ with regular expression `percflowbelow`
- Pertarget$_{kwd}$ with regular expression `pertarget`
- Perthis$_{kwd}$ with regular expression `perthis`
- Pointcut with regular expression `pointcut`
- Precedence$_{kwd}$ with regular expression `precedence`
- Preinitialization$_{kwd}$ with regular expression `preinitialization`

- Privileged$_{kwd}$ with regular expression `privileged`
- Proceed$_{kwd}$ with regular expression `proceed`
- Returning$_{kwd}$ with regular expression `returning`
- Set$_{kwd}$ with regular expression `set`
- Soft$_{kwd}$ with regular expression `soft`
- Staticinitialization$_{kwd}$ with regular expression `staticinitialization`
- Target with regular expression `target`
- Throwing$_{kwd}$ with regular expression `throwing`
- Warning$_{kwd}$ with regular expression `warning`
- Within$_{kwd}$ with regular expression `within`
- Withincode$_{kwd}$ with regular expression `withincode`
- aspectjId with regular expression `[a-zA-Z_$][0-9a-zA-Z_$]*`
- cfQuote with regular expression `[\\]["]`
- doubleDot with regular expression `[\.][\.]`
- idPatternTerm with regular expression `[A-Za-z\*][A-Za-z0-9\*]*`
- quadDash with regular expression `[\-][\-][\-][\-]`

Bridge productions:

- ***classMemberDefinition*** →

  PointcutDecl

  | **AspectDecl**
- ***interfaceMemberDefinition*** →

  PointcutDecl

  | **AspectDecl**
- ***primaryExpression*** →

  Proceed$_{kwd}$ *Lparen* ***expressionListOpt*** *Rparen*

- ***typeDefinition*** →

  AspectDecl

Extension productions:

- **AdviceDecl** →

  modifiersOpt **AdviceSpec** ***throwsClause*** *Colon* **PointcutExpr** ***methodBody***
- **AdviceSpec** →

  After$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen*

  | After$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen* Returning$_{\text{kwd}}$ *Lparen* ***parameterDeclaration*** *Rparen*

  | After$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen* Returning$_{\text{kwd}}$

  | After$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen* Throwing$_{\text{kwd}}$ *Lparen* ***parameterDeclaration*** *Rparen*

  | ***type*** Around$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen*

  | After$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen* Throwing$_{\text{kwd}}$ *Lparen* *Rparen*

  | After$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen* Returning$_{\text{kwd}}$ *Lparen* *Rparen*

  | After$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen* Throwing$_{\text{kwd}}$

  | Before$_{\text{kwd}}$ *Lparen* ***parameterDeclarationList*** *Rparen*
- **AndClassNamePatternExpr** →

  UnaryClassNamePatternExpr

  | **AndClassNamePatternExpr** *AndAnd* **UnaryClassNamePatternExpr**

- **AndClassNamePatternExprNoBang** →

  BasicClassNamePattern

  | **AndClassNamePatternExprNoBang** *AndAnd* **UnaryClassNamePatternExpr**

- **AspectBody** →

  Lbrace **AspectBodyDecls** *Rbrace*

  | *Lbrace Rbrace*

- **AspectBodyDecl** →

  classMemberDefinition

  | **AdviceDecl**

  | **IntertypeDecl**

  | **DeclareDecl**

- **AspectBodyDecls** →

  AspectBodyDecls AspectBodyDecl

  | **AspectBodyDecl**

- **AspectDecl** →

  modifiersOpt $\text{Privileged}_{\text{kwd}}$ ***modifiersOpt*** $\text{Aspect}_{\text{kwd}}$ aspectjId ***superClassClause implementsClause*** **PerClauseOpt AspectBody**

  | ***modifiersOpt*** $\text{Aspect}_{\text{kwd}}$ aspectjId ***superClassClause implementsClause*** **PerClauseOpt AspectBody**

- **AspectJReservedIdentifier** →

  $\text{Returning}_{\text{kwd}}$

  | $\text{Preinitialization}_{\text{kwd}}$

| $\text{Privileged}_{\text{kwd}}$

| Args

| $\text{Precedence}_{\text{kwd}}$

| $\text{Throwing}_{\text{kwd}}$

| Call_{kwd}

| $\text{Staticinitialization}_{\text{kwd}}$

| $\text{Adviceexecution}_{\text{kwd}}$

| $\text{Execution}_{\text{kwd}}$

| $\text{Withincode}_{\text{kwd}}$

| $\text{Initialization}_{\text{kwd}}$

| $\text{Error}_{\text{kwd}}$

| Soft_{kwd}

| $\text{Cflow}_{\text{kwd}}$

| $\text{Handler}_{\text{kwd}}$

| $\text{Cflowbelow}_{\text{kwd}}$

| $\text{Warning}_{\text{kwd}}$

| Target

| $\text{Parents}_{\text{kwd}}$

| Set_{kwd}

| Get_{kwd}

| $\text{Aspect}_{\text{kwd}}$

- **BaseTypePattern** →

  NamePattern

  | ***primitiveType***

  | **NamePattern** *Plus*

- **BasicClassNamePattern** →

  NamePattern

  | **NamePattern** *Plus*

  | *Lparen* **ClassNamePatternExpr** *Rparen*

- **BasicPointcutExpr** →

  Get_{kwd} *Lparen* **FieldPattern** *Rparen*

  | $\text{Adviceexecution}_{\text{kwd}}$ *Lparen Rparen*

  | $\text{Cflowbelow}_{\text{kwd}}$ *Lparen* **PointcutExpr** *Rparen*

  | Set_{kwd} *Lparen* **FieldPattern** *Rparen*

  | Call_{kwd} *Lparen* **MethodConstructorPattern** *Rparen*

  | $\text{Within}_{\text{kwd}}$ *Lparen* **ClassNamePatternExpr** *Rparen*

  | $\text{Staticinitialization}_{\text{kwd}}$ *Lparen* **ClassNamePatternExpr** *Rparen*

  | $\text{Execution}_{\text{kwd}}$ *Lparen* **MethodConstructorPattern** *Rparen*

  | *This Lparen* **TypeIdStar** *Rparen*

  | $\text{Cflow}_{\text{kwd}}$ *Lparen* **PointcutExpr** *Rparen*

  | $\text{Preinitialization}_{\text{kwd}}$ *Lparen* **ConstructorPattern** *Rparen*

  | Args *Lparen* **TypeIdStarListOpt** *Rparen*

  | *If Lparen* ***expression*** *Rparen*

| **aspectj_name** *Lparen* **TypeIdStarListOpt** *Rparen*

| *Lparen* **PointcutExpr** *Rparen*

| Withincode$_{\text{kwd}}$ *Lparen* **MethodConstructorPattern** *Rparen*

| Handler$_{\text{kwd}}$ *Lparen* **ClassNamePatternExpr** *Rparen*

| Target *Lparen* **TypeIdStar** *Rparen*

| Initialization$_{\text{kwd}}$ *Lparen* **ConstructorPattern** *Rparen*

- **BasicTypePattern** →

  Lparen **TypePatternExpr** *Rparen*

  | **BaseTypePattern** ***declaratorBrackets***

  | *Void*

  | **BaseTypePattern**

- **ClassNamePatternExpr** →

  AndClassNamePatternExpr

  | **ClassNamePatternExpr** *OrOr* **AndClassNamePatternExpr**

- **ClassNamePatternExprList** →

  ClassNamePatternExprList *Comma* **ClassNamePatternExpr**

  | **ClassNamePatternExpr**

- **ClassNamePatternExprNoBang** →

  AndClassNamePatternExprNoBang

  | **ClassNamePatternExprNoBang** *OrOr* **AndClassNamePatternExpr**

- **ClassTypeDotId** →

  Lparen **TypePatternExpr** *Rparen Dot* **SimpleNamePattern**

  | **NamePattern** *Plus Dot* **SimpleNamePattern**

| **SimpleNamePattern**

| **NamePattern** *Dot* **SimpleNamePattern**

| **NamePattern** doubleDot **SimpleNamePattern**

- **ClassTypeDotNew** →

  NamePattern doubleDot *New*

  | *Lparen* **TypePatternExpr** *Rparen Dot New*

  | **NamePattern** *Plus Dot New*

  | **NamePattern** *Dot New*

  | *New*

- **ConstructorPattern** →

  ModifierPatternExpr ClassTypeDotNew *Lparen* **FormalPatternListOpt** *Rparen* **ThrowsPatternListOpt**

  | **ClassTypeDotNew** *Lparen* **FormalPatternListOpt** *Rparen* **ThrowsPatternListOpt**

- **DeclareDecl** →

  $\text{Declare}_{\text{kwd}}$ $\text{Parents}_{\text{kwd}}$ *Colon* **ClassNamePatternExpr** *Implements* ***names*** *Semicolon*

  | $\text{Declare}_{\text{kwd}}$ $\text{Warning}_{\text{kwd}}$ *Colon* **PointcutExpr** *Colon Stringconst Semicolon*

  | $\text{Declare}_{\text{kwd}}$ $\text{Parents}_{\text{kwd}}$ *Colon* **ClassNamePatternExpr** *Extends* ***names*** *Semicolon*

  | $\text{Declare}_{\text{kwd}}$ $\text{Precedence}_{\text{kwd}}$ *Colon* **ClassNamePatternExprList** *Semicolon*

  | $\text{Declare}_{\text{kwd}}$ $\text{Error}_{\text{kwd}}$ *Colon* **PointcutExpr** *Colon Stringconst Semicolon*

  | $\text{Declare}_{\text{kwd}}$ Soft_{kwd} *Colon* ***type*** *Colon* **PointcutExpr** *Semicolon*

- **FieldPattern** →

  TypePatternExpr ClassTypeDotId

  | **ModifierPatternExpr TypePatternExpr ClassTypeDotId**
- **FormalPattern** →

  doubleDot

  | **TypePatternExpr**

  | *Dot Dot*
- **FormalPatternList** →

  FormalPatternList *Comma* **FormalPattern**

  | **FormalPattern**
- **FormalPatternListOpt** →

  ε

  | **FormalPatternList**
- **IntertypeDecl** →

  modifiersOpt nameConcrete *Dot New Lparen*
  parameterDeclarationList *Rparen* ***throwsClause constructorBody***

  | ***modifiersOpt type qualifiedNameConcrete*** *Lparen*
  parameterDeclarationList *Rparen* ***throwsClause methodBody***

  | ***modifiersOpt type qualifiedNameConcrete*** *Semicolon*

  | ***modifiersOpt type qualifiedNameConcrete*** *Eq* ***initializer*** *Semicolon*
- **MethodConstructorPattern** →

  ConstructorPattern

  | **MethodPattern**

- **MethodPattern** →

  ModifierPatternExpr TypePatternExpr ClassTypeDotId *Lparen* **FormalPatternListOpt** *Rparen* **ThrowsPatternListOpt**

  | **TypePatternExpr ClassTypeDotId** *Lparen* **FormalPatternListOpt** *Rparen* **ThrowsPatternListOpt**

- **ModifierPatternExpr** →

  Not ***modifier***

  | **ModifierPatternExpr** *Not* ***modifier***

  | **ModifierPatternExpr** ***modifier***

  | ***modifier***

- **NamePattern** →

  NamePattern doubleDot **SimpleNamePattern**

  | **NamePattern** *Dot* **SimpleNamePattern**

  | **SimpleNamePattern**

- **OrPointcutExpr** →

  OrPointcutExpr *OrOr* **UnaryPointcutExpr**

  | **UnaryPointcutExpr**

- **OrTypePatternExpr** →

  OrTypePatternExpr *OrOr* **UnaryTypePatternExpr**

  | **UnaryTypePatternExpr**

- **PerClause** →

  $\text{Issingleton}_{\text{kwd}}$

  | $\text{Pertarget}_{\text{kwd}}$ *Lparen* **PointcutExpr** *Rparen*

| Percflowbelow$_{\text{kwd}}$ *Lparen* **PointcutExpr** *Rparen*

| Perthis$_{\text{kwd}}$ *Lparen* **PointcutExpr** *Rparen*

| Issingleton$_{\text{kwd}}$ *Lparen Rparen*

| Percflow$_{\text{kwd}}$ *Lparen* **PointcutExpr** *Rparen*

- **PerClauseOpt** →

  ε

  | **PerClause**

- **PointcutDecl** →

  modifiersOpt Pointcut aspectjId *Lparen* ***parameterDeclarationList*** *Rparen* *Colon* **PointcutExpr** *Semicolon*

  | ***modifiersOpt*** Pointcut aspectjId *Lparen* ***parameterDeclarationList*** *Rparen* *Semicolon*

- **PointcutExpr** →

  OrPointcutExpr

  | **PointcutExpr** *AndAnd* **OrPointcutExpr**

- **SimpleNamePattern** →

  AspectJReservedIdentifier

  | aspectjId

  | idPatternTerm

  | *Mul*

- **ThrowsPattern** →

  Not **ClassNamePatternExpr**

  | **ClassNamePatternExprNoBang**

- **ThrowsPatternList** →

  ThrowsPattern

  | **ThrowsPatternList** *Comma* **ThrowsPattern**
- **ThrowsPatternListOpt** →

  Throws **ThrowsPatternList**

  | ε
- **TypeIdStar** →

  type

  | doubleDot

  | *Mul*

  | ***type*** *Plus*
- **TypeIdStarList** →

  TypeIdStar

  | **TypeIdStarList** *Comma* **TypeIdStar**
- **TypeIdStarListOpt** →

  ε

  | **TypeIdStarList**
- **TypePatternExpr** →

  TypePatternExpr *AndAnd* **OrTypePatternExpr**

  | **OrTypePatternExpr**
- **UnaryClassNamePatternExpr** →

  BasicClassNamePattern

  | *Not* **UnaryClassNamePatternExpr**

- **UnaryPointcutExpr** →

  BasicPointcutExpr

  | *Not* **UnaryPointcutExpr**
- **UnaryTypePatternExpr** →

  Not **UnaryTypePatternExpr**

  | **BasicTypePattern**
- **aspectj_name** →

  aspectj_simple_name

  | **aspectj_qualified_name**
- **aspectj_qualified_name** →

  aspectj_name *Dot* aspectjId
- **aspectj_simple_name** →

  aspectjId

Disambiguation groups:

- Disambiguate $\{\text{Proceed}_{\text{kwd}}, Id\} \rightarrow \text{Proceed}_{\text{kwd}}$
- Disambiguate $\{\text{Declare}_{\text{kwd}}, Id\} \rightarrow \text{Declare}_{\text{kwd}}$
- Disambiguate $\{\text{Around}_{\text{kwd}}, Id\} \rightarrow \text{Around}_{\text{kwd}}$
- Disambiguate $\{\text{After}_{\text{kwd}}, Id\} \rightarrow \text{After}_{\text{kwd}}$
- Disambiguate $\{\text{Before}_{\text{kwd}}, Id\} \rightarrow \text{Before}_{\text{kwd}}$

A.3.2.2 Native complex numbers.

Marking terminals:

- ComplexRef with regular expression `Complex`
- Complex with regular expression `complex`

Productions (both bridge productions):

- ***primaryExpression*** →

  Complex *Lparen* ***expression*** *Comma* ***expression*** *Rparen*

  | ComplexRef *Lparen* ***expression*** *Comma* ***expression*** *Rparen*
- ***referenceType*** →

  ComplexRef

  | Complex

A.3.2.3 `foreach` loops.

Marking terminal:

- NewFor with regular expression `foreach`

Bridge production:

- ***statement*** →

  Enhanced_For_Stmt

Extension production:

- **Enhanced_For_Stmt** →

  NewFor Lparen ***type*** *Id Colon* ***expression*** *Rparen* ***statement***

A.3.2.4 SQL.

Marking terminals:

- Conn with regular expression `connection`
- Establish with regular expression `establish`
- Register with regular expression `register`
- Using with regular expression `using`

Extension terminals:

- ALL with regular expression `(ALL)|(all)`
- BOOLEAN with regular expression `BOOLEAN`
- DISTINCT with regular expression `(DISTINCT)|(distinct)`
- Driver with regular expression `driver`
- FLOAT with regular expression `FLOAT`
- FROM with regular expression `(FROM)|(from)`
- Float_Const with regular expression

```
(((([0-9]+[\.][0-9]*|[\.][0-9]+)([eE][\-\+]?[0-9]+)?|
   [0-9]+[eE][\-\+]?[0-9]+
  )[fFdD]?
 )|([0-9]+([eE][\-\+]?[0-9]+)?[fFdD])
```

- INTEGER with regular expression `INTEGER`
- Int_Const with regular expression `([0-9]+|0[xX][0-9a-fA-F]+)[lL]?`
- LP with regular expression `[\(]`
- Query with regular expression `query`
- RP with regular expression `[\)]`
- SELECT with regular expression `(SELECT)|(select)`
- And with regular expression `[aA][nN][dD]`
- DOT with regular expression `[\.]`

- Dash with regular expression `[\-]`
- Divide with regular expression `[/]`
- EQ with regular expression `[=]`
- GTEQ with regular expression `[>][=]`
- GT with regular expression `[>]`
- Id with regular expression `[a-zA-Z_$][0-9a-zA-Z_$]*`
- LTEQ with regular expression `[<][=]`
- LT with regular expression `[<]`
- Mul with regular expression `[\*]`
- NEQ with regular expression `[<][>]`
- Not with regular expression `[nN][oO][tT]`
- Or with regular expression `[oO][rR]`
- Plus with regular expression `[\+]`
- String_Const with regular expression

```
["]([^"\\]|
    [\\][btnfr"'\\]|
    [\\][0-7]|
    [\\][0-7][0-7]|
    [\\][0-3][0-7][0-7]|
    [\\][u]+[0-9a-fA-F][0-9a-fA-F]?[0-9a-fA-F]?[0-9a-fA-F]?
   )*["]
```

- Table with regular expression `table`
- VARCHAR with regular expression `VARCHAR`
- WHERE with regular expression `(WHERE)|(where)`
- With with regular expression `with`

Bridge productions:

- ***classMemberDefinition*** →

  Conn *Id Stringconst* With **TableDcls** *Semicolon*
- ***primaryExpression*** →

  Using *Id* Query *Lbrace* **Stmt** *Rbrace*
- ***statement*** →

  Establish *Id Semicolon*

  | Conn *Id Stringconst* With **TableDcls** *Semicolon*

  | Register Driver *Stringconst Semicolon*
- ***typeDefinition*** →

  Conn *Id Stringconst* With **TableDcls** *Semicolon*

Extension productions:

- **FieldDcl** →

  Type Id
- **FieldDcls** →

  FieldDcl *Comma* **FieldDcls** | **FieldDcl**
- **Arith_Expr** →

  Arith_Expr Plus **Arith_Term**

  | **Arith_Expr** Dash **Arith_Term**

  | **Arith_Term**
- **Arith_Factor** →

  Primary | Plus **Primary** | Dash **Primary**

- **Arith_Term** →

  Arith_Term Divide **Arith_Factor** | **Arith_Term** Mul **Arith_Factor** | **Arith_Factor**
- **Boole_Expr** →

  Boole_Term | **Boole_Expr** Or **Boole_Term**
- **Boole_Factor** →

  Rel_Expr | Not **Rel_Expr**
- **Boole_Term** →

  Boole_Term And **Boole_Factor** | **Boole_Factor**
- **Expr** →

  Boole_Expr
- **Primary** →

  Int_Const | Id | String_Const | LP **Expr** RP | Id DOT Id | Float_Const
- **Rel_Expr** →

  Rel_Expr GT **Arith_Expr**

  | **Rel_Expr** EQ **Arith_Expr**

  | **Rel_Expr** GTEQ **Arith_Expr**

  | **Rel_Expr** LT **Arith_Expr**

  | **Rel_Expr** NEQ **Arith_Expr**

  | **Rel_Expr** LTEQ **Arith_Expr**

  | **Arith_Expr**
- **Stmt** →

  SELECT **Select_Qualifier Select_List** FROM **Table_List** **Where_Clause**

- **Type** →

  FLOAT | INTEGER | BOOLEAN | VARCHAR
- **Select_List** →

  Mul | **Select_SubList**
- **Select_Qualifier** →

  ε | ALL | DISTINCT
- **Select_SubList** →

  Expr | **Select_SubList** *Comma* **Expr**
- **TableDcl** →

  Table Id *Lbrack* **FieldDcls** *Rbrack*
- **TableDcls** →

  TableDcl *Comma* **TableDcls** | **TableDcl**
- **Table_List** →

  Id *Comma* **Table_List** | Id
- **Where_Clause** →

  ε | WHERE **Expr**

A.3.2.5 Boolean tables.

Terminals:

- FalseTV with regular expression `F`
- Table with regular expression `table`
- TrueTV with regular expression `T`

Productions:

- ***primaryExpression*** →

  Table *Lparen* **TableRows** *Rparen*
- **TableRow** →

  expression *Colon* **TruthValueList**
- **TableRows** →

  TableRow | **TableRows** *Comma* **TableRow**
- **TruthValueList** →

  TruthValue | **TruthValue TruthValueList**
- **TruthValue** →

  TrueTV | *Mul* | FalseTV

A.3.2.6 Dimension types.

Marking terminals:

- Call with regular expression `call`
- Units with regular expression `Unit`

Extension terminals:

- Celsius with regular expression `C`
- Centimeters with regular expression `cm`
- Days with regular expression `days`
- Fahrenheit with regular expression `F`
- Feet with regular expression `ft`
- Grams with regular expression `g`
- Hours with regular expression `hr`

- Inches with regular expression `in`
- Kilograms with regular expression `kg`
- Kilometers with regular expression `km`
- LengthDim with regular expression `L`
- Length with regular expression `length`
- MassDim with regular expression `M`
- Mass with regular expression `mass`
- Meters with regular expression `m`
- Miles with regular expression `mi`
- Minutes with regular expression `min`
- Seconds with regular expression `sec`
- TempDim with regular expression `Te`
- Temp with regular expression `temperature`
- TimeDim with regular expression `T`
- Time with regular expression `time`

Bridge productions:

- ***expression*** →

  Lparen Units *Lt* ***type*** *Comma* **DimensionExpr** *Gt Rparen* ***expression***
- ***primaryExpression*** →

  Call *Lparen* ***primaryExpressionAndArrayCreation*** *Dot Id Lparen* ***expressionListOpt*** *Rparen Rparen*
- ***type*** →

  Units *Lt* ***type*** *Comma* **DimensionExpr** *Gt*

Extension productions:

- **DimExpr** →

  DimTerm
- **DimFactor** →

  MassDim | TempDim | TimeDim | *Lparen* **DimExpr** *Rparen* | *Id* | LengthDim
- **DimPower** →

  DimFactor *Xor* **PosNegInt** | **DimFactor**
- **DimTerm** →

  DimPower | **DimTerm DimFactor**
- **DimensionExpr** →

  ExplicitDimExprList | **DimExpr**
- **ExplicitDimExpr** →

  Mass **Exponent_Literal MassUnit** | Length **Exponent_Literal LengthUnit** | Temp **Exponent_Literal TempUnit** | Time **Exponent_Literal TimeUnit**
- **ExplicitDimExprList** →

  ExplicitDimExpr | **ExplicitDimExpr ExplicitDimExprList**
- **Exponent_Literal** →

  Intconst | *Minus Intconst*
- **LengthUnit** →

  Miles | Centimeters | Feet | Inches | Kilometers | Meters
- **MassUnit** →

  Grams | Kilograms
- **PosNegInt** →

  Minus Intconst | *Intconst*

- **TempUnit** →

  Celsius | Fahrenheit

- **TimeUnit** →

  Seconds | Days | Hours | Minutes

A.4 Toy grammar for tests.

Terminals: Start nonterminal is ***E***.

- *y* with regular expression `y`
- *x* with regular expression `x+`
- + with regular expression `\+`
- *ws* with regular expression `[ ]*`

Productions:

- ***E*** → ***T***

  | ***T*** *y*

  | ***T*** + ***E***

- ***T*** → *x*

Index