5-2010

PSLR(1): Pseudo-Scannerless Minimal LR(1) for the Deterministic Parsing of Composite Languages

Joel Denny
Clemson University, joeldenny@joeldenny.org

PSLR(1): Pseudo-Scannerless Minimal LR(1) for the Deterministic Parsing of Composite Languages

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Joel E. Denny
May 2010

Accepted by:
Dr. Brian A. Malloy, Committee Chair
Dr. Harold C. Grossman
Dr. Jason Hallstrom
Dr. Stephen T. Hedetniemi

Abstract

Composite languages are composed of multiple sub-languages. Examples include the parser specification languages read by parser generators like Yacc, modern extensible languages with complex layers of domain-specific sub-languages, and even traditional programming languages like C and C++. In this dissertation, we describe PSLR(1), a new scanner-based LR(1) parser generation system that automatically eliminates scanner conflicts typically caused by language composition. The fundamental premise of PSLR(1) is the pseudo-scanner, a scanner that only recognizes tokens accepted by the current parser state. However, use of the pseudo-scanner raises several unique challenges, for which we describe a novel set of solutions. One major challenge is that practical LR(1) parser table generation algorithms merge parser states, sometimes inducing incorrect pseudo-scanner behavior including new conflicts. Our solution is a new extension of IELR(1), an algorithm we have previously described for generating minimal LR(1) parser tables. Other contributions of our work include a robust system for handling the remaining scanner conflicts, a correction for syntax error handling mechanisms that are also corrupted by parser state merging, and a mechanism to enable scoping of syntactic declarations in order to further improve the modularity of sub-language specifications. While the premise of the pseudo-scanner has been described by other researchers independently, we expect our improvements to distinguish PSLR(1) as a significantly more robust scanner-based parser generation system for traditional and modern composite languages.

Table of Contents

Title Page . . . . . **i**

Abstract . . . . . **ii**

List of Tables . . . . . **v**

List of Figures . . . . . **vi**

1 Introduction **1**
- 1.1 Problem Statement 2
- 1.2 Contributions of the Work 2
- 1.3 Thesis Statement and Evaluation 4
- 1.4 Dissertation Organization 4

2 Background **5**
- 2.1 Notation 5
- 2.2 Scanner-based LR(1) Parser Generation 6
  - 2.2.1 Scanners 6
  - 2.2.2 LR(1) Parsers 9
- 2.3 Composite Languages 15
  - 2.3.1 Parser Specifications 15
  - 2.3.2 Regular Sub-languages 19
  - 2.3.3 Subtle Sub-languages 20
  - 2.3.4 Extensible Languages 22
- 2.4 Summary 22

3 Methodology **24**
- 3.1 Pseudo-scanner 24
- 3.2 Lexical Precedence 26
  - 3.2.1 Guiding Principles 27
  - 3.2.2 Traditional Rules 29
  - 3.2.3 Non-traditional Rules 31
  - 3.2.4 Ambiguities 33
  - 3.2.5 Self-consistency 38
  - 3.2.6 Pseudo-scanner Tables 42
  - 3.2.7 Resolver Algorithm 45
  - 3.2.8 Summary 51
- 3.3 Lexical Ties 51
- 3.4 Minimal LR(1) 57
  - 3.4.1 LALR(1) Versus Canonical LR(1) 58
  - 3.4.2 IELR(1) 59

3.4.3 IELR(1) Extension for PSLR(1) . . . 63
3.4.4 Summary . . . 68
3.5 Syntax Error Handling . . . 68
3.5.1 Pseudo-scanner . . . 68
3.5.2 Parser . . . 70
3.5.3 Summary . . . 73
3.6 Whitespace and Comments . . . 73
3.7 Scoped Declarations . . . 77
3.8 Summary . . . 79

4 Studies and Evaluation . . . 81
4.1 PSLR(1) Bison . . . 81
4.2 Levine SQL . . . 82
4.3 ISO C99 . . . 84
4.4 Template Argument Lists . . . 86
4.5 Results . . . 87
4.5.1 Grammars and Parser Tables . . . 87
4.5.2 Readability and Maintainability . . . 91

5 Related Work . . . 97
5.1 Nawrocki . . . 97
5.2 Keynes . . . 99
5.3 Van Wyk and Schwerdfeger . . . 103
5.4 Scannerless GLR . . . 107

6 Merits of the Work . . . 110

Bibliography . . . 114

List of Tables

2.1 Parser Tables for Template Argument Lists 12
2.2 A Template Argument List Parse 14

3.1 Unambiguous Sequential Lexical Precedence Function 35
3.2 Opposing Lexical Precedence 41
3.3 IELR(1) Case Studies 60
3.4 IELR(1) Parser Tables 61
3.5 IELR(1) Action Corrections 62

4.1 Grammar Sizes 88
4.2 Parser Tables 88
4.3 Lexical Reveals 88
4.4 Specification Sizes 92
4.5 Complexity Counts 95
4.6 Complexity Frequencies. 95

List of Figures

2.1 Scanner-based Parsing 6
2.2 Scanner Conflict Resolution 6
2.3 Template Argument Lists 10
2.4 Yacc Parser Specification Language 16
2.5 Lex Start Conditions 18
2.6 Scoped Declarations 21

3.1 Traditional Lexical Precedence Rules 29
3.2 Non-traditional Lexical Precedence Rules 31
3.3 Intransitive Lexical Precedence 36
3.4 Token Precedence Mixed with Lexeme Precedence 36
3.5 Shortest Match Mixed with Longest Match 36
3.6 Lexical Ties 52

Chapter 1

Introduction

Grammar-dependent software is omnipresent in software development [21]. For example, compilers, document processors, browsers, import/export tools, and generative programming tools are used in software development in all phases. These phases include comprehension, analysis, maintenance, reverse-engineering, code manipulation, and visualization of the application program under study. However, construction of these tools relies on the correct recognition of the language constructs specified by the grammar.

Some aspects of grammar engineering are reasonably well understood. For example, the study of grammars as definitions of formal languages, including the study of LL, LR, LALR, and SLR algorithms and the Chomsky hierarchy, form an essential part of most computer science curricula. Nevertheless, parsing as a disciplined study must be reconsidered from an engineering point of view [21, 22]. Many parser developers eschew the use of parser generators because it is too difficult to customize the generated parser or because the generated parser requires considerable modification to incorporate sufficient power to handle complex grammars such as the C++ and C# grammar. Thus, industrial strength parser development requires considerable effort, and many approaches to parser generation are ad hoc [35, 36].

In this dissertation, we address difficulties in the generation of scanner-based LR(1) parsers for composite languages. Composite languages are composed of multiple sub-languages, each of which may have a wholly different syntax. For example, the language of the parser specification file read by the parser generator Yacc is composed of grammar productions and declarations but also passages of the programming language C [9, 19]. Most traditional programming languages like C or

C++ also contain minor sub-languages in the form of comments and literal strings [8, 10]. The need for general techniques for parsing composite languages is growing with the popularity of modern extensible languages, which may contain complex layers of sub-languages composed arbitrarily for the requirements of specific domains [12, 13, 18, 40].

1.1 Problem Statement

Traditional scanner-based LR(1) parser generation encounters a serious difficulty for composite languages. Specifically, there are often points in a parse at which the input character sequence matches token definitions from multiple sub-languages. However, a traditional scanner has no automatic mechanism for determining which sub-languages are currently valid and thus which tokens to recognize. The result is one or more scanner conflicts that must be manually resolved or eliminated. Unfortunately, traditional scanner-generator tools like Lex [9, 23] provide only weak or cumbersome formal mechanisms for handling scanner conflicts, sometimes leading tool users to develop complex ad-hoc solutions.

1.2 Contributions of the Work

In this dissertation, we describe PSLR(1) (Pseudo-scannerless Minimal LR(1)), a new scanner-based LR(1) parser generation system that automatically eliminates many of the scanner conflicts typically caused by language composition. Our PSLR(1) generator reads a unified scanner and parser specification and then generates a deterministic minimal LR(1) parser with a tightly coupled scanner that we call a pseudo-scanner. Unlike a traditional scanner, whose token recognition power is limited to an FSA (finite state automaton), a pseudo-scanner's power is augmented by the parser's stack. Specifically, the pseudo-scanner examines the current parser state, which indicates the syntactic left context of the current point in the parse and thus indicates which sub-languages are currently valid. By only recognizing tokens accepted by the current parser state, the pseudo-scanner automatically eliminates conflicts with tokens from other sub-languages.

The premise of the pseudo-scanner has been described independently by several other researchers [20, 27, 40]. As some of them note, use of the pseudo-scanner raises several new challenges. For example, for many programming languages like C, keywords are reserved words. That is, a scan-

ner is expected not to mistake a keyword for an identifier even if the keyword is erroneous in the current syntactic left context. Unlike a traditional scanner, a pseudo-scanner makes this mistake by default. For some of these challenges, we observe that existing solutions are similar to the solutions we have devised for PSLR(1). In other cases, we feel that existing solutions are unintuitive for tool users or have undesirable side-effects, so in this dissertation we describe alternative solutions.

No existing technique that we have reviewed handles the most difficult challenge of the pseudo-scanner. All existing parser generation systems based on the premise of the pseudo-scanner that we have found employ LALR(1), a popular LR(1) parser table generation technique that merges parser states for efficiency. It is well know that merging parser states can render parser tables less powerful than canonical LR(1) and thus less intuitive to the parser developer [15, 16, 17, 32, 33]. We observe that merging parser states also merges those states' sets of acceptable tokens and thus can induce incorrect pseudo-scanner behavior including new conflicts, undermining the pseudo-scanner's advantage over traditional scanners. As part of our preliminary work, we described and implemented IELR(1), a minimal LR(1) parser table generation algorithm that eliminates parser table inadequacies induced by LR(1) state merging but which does not consider the effect of LR(1) state merging on a pseudo-scanner [15, 16]. In this dissertation, we describe an IELR(1) extension to eliminate incorrect behavior induced in the pseudo-scanner as well.

Our work includes many other contributions to the field of parser generation. For example, there can exist scanner conflicts that cannot be eliminated by the basic behavior of the pseudo-scanner or by our IELR(1) extension. We describe a robust system for detecting such conflicts, reporting them, and resolving them according to user declarations. We also describe syntax error handling mechanisms for PSLR(1). One such mechanism that is relevant to both PSLR(1) and traditional scanner-based LR(1) is an LR(1) parsing algorithm extension that we call LAC (lookahead correction), which overcomes syntax error handling problems caused by parser state merging. We also describe a mechanism that enables scoped declarations, declarations like precedence and associativity that are rendered effective only for particular portions of a grammar in order to improve the modularity of sub-language specifications.

1.3 Thesis Statement and Evaluation

We have implemented our PSLR(1) generator as an extension of Bison [2], the GNU implementation of Yacc. In this dissertation, we refer to our extended Bison as PSLR(1) Bison. Internally, Bison employs a traditional scanner-based LR(1) parser for analyzing its input parser specifications. Thus, we initially implemented PSLR(1) Bison to employ traditional scanner-based LR(1) internally as well. The language of these parser specifications is a challenging composition of multiple sub-languages especially in the case of PSLR(1) Bison because it unifies scanner and parser specifications. Thus, PSLR(1) Bison's internal parser is itself a candidate for PSLR(1).

As part of the evaluation of our work, we have implemented PSLR(1) Bison's internal parser a second time using PSLR(1) instead of traditional scanner-based LR(1). We have also implemented a PSLR(1) version of a traditional scanner-based LR(1) parser for SQL written by John R. Levine for his text book, *flex & bison* [24]. For each of these parsers, we collect and compare readability and maintainability statistics for the PSLR(1) specification versus the traditional scanner-based LR(1) specification. In addition, we introduce a metric that we call lexical reveals, which measures the corrections made by PSLR(1)'s IELR(1) extension, and we use this metric to examine four case studies. The results from all of these studies support our thesis that PSLR(1) is a significantly more robust parser generation system for composite languages than traditional scanner-based LR(1) parser generation or the approach of coupling LALR(1) with a pseudo-scanner.

1.4 Dissertation Organization

We organize the remainder of this dissertation as follows. In chapter 2, we provide a more detailed background on scanner-based LR(1) parser generation and composite languages. In chapters 3 and 4, we describe our PSLR(1) system and our evaluation of its success. In chapter 5, we review the literature and compare PSLR(1) to several other modern systems for generating parsers for composite languages. Finally, in chapter 6, we select the most well known and robust of these systems, scannerless GLR, and analyze its advantages and disadvantages in order to explain how PSLR(1) is more robust.

Chapter 2

Background

In this chapter, we explain fundamental concepts on which PSLR(1) is based. In section 2.1, we clarify a few aspects of the notation that we employ. We discuss traditional scanner-based LR(1) parser generation in section 2.2. In section 2.3, we discuss composite languages in terms of traditional scanner-based LR(1) parser generation. We summarize in section 2.4.

2.1 Notation

In this paper, we italicize the first occurrence of a formal term that has already been established by existing literature. We use bold italic to indicate original terminology that this paper introduces. This distinction should help the reader determine when we are referencing known concepts and when we are introducing new concepts.

We employ a mathematical notation that we feel communicates our formal models precisely and succinctly. Most of our notation is standard and should be familiar to the reader. However, we now disambiguate a few symbols that are not always used consistently in existing literature:

1. The symbol ":" is consistently read "such that."
2. "$\{i : c\}$" is read "the set of all i such that c is true."
3. "$\exists i : c$" is read "there exists an i such that c is true."
4. "$\forall i : c, s$" is read "for every i such that c is true, s is true."

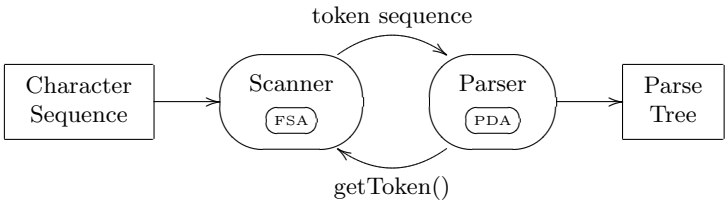


Figure 2.1: Scanner-based Parsing. In scanner-based parsing, a scanner employs a finite state automaton to convert a character sequence into a token sequence, and a parser employs a pushdown automaton to convert that token sequence into a parse tree.

(a)
```
int          return INT;
[a-zA-Z]+    return ID;
```
(b) `integer` (c) `int`

Figure 2.2: Scanner Conflict Resolution. (a) In this Lex specification, the regular expressions for the two tokens overlap. (b) First, the scanner selects the longest matching tokens, leaving only `ID` in this case. (c) Second, the scanner selects the first token declared, `INT` in this case.

---

5. "$\forall i \in I, s$" is read "for every i in I, s is true."

6. "σ" indicates the same sequence as "$\sigma[1..|\sigma|]$", but the latter notation explicitly indexes the range of all elements.

2.2 Scanner-based LR(1) Parser Generation

As illustrated in Figure 2.1, scanner-based parsing splits lexical analysis and syntactic analysis of an input character sequence into two loosely coupled phases. The lexical analysis is performed by a *scanner*, which we discuss in section 2.2.1. The syntactic analysis is performed by a *parser*, which we discuss in section 2.2.2. In some contexts, the term *parser* is also used to refer to the combination of the scanner and parser, and lexical analysis is considered part of syntactic analysis. In this paper, we assume deterministic scanners and parsers, so the output either corresponds to a single parse tree or is a report of a syntax error. We also assume that the parsing technique is from the LR(1) family as in the case of traditional parser generation tools like Yacc.

2.2.1 Scanners

A scanner generator tool, such as Lex, can be employed to generate a scanner from a formal *scanner specification*. This scanner specification consists primarily of a lexical syntax specification, which defines *tokens* as regular expressions. The scanner specification also usually contains literal

code in a general-purpose programming language like C, some of which is an integral part of the lexical syntax specification. For example, the scanner specification for Lex in Figure 2.2a defines two tokens, the keyword `int` and the identifier. The return statements specify formal C names for the tokens, `INT` and `ID`.

As illustrated in Figure 2.1, the generated scanner employs an FSA (finite state automaton) to read the input character sequence, match contiguous subsequences against the specified regular expressions, and produce a token sequence. We introduce the following definitions to facilitate our discussion of this behavior.

Definition 2.2.1 (Input Character Set)
Throughout this paper, we refer to the input character set as Ξ. That is, the character sequence read by the scanner is from the set Ξ^*. $\square$

Definition 2.2.2 (Lexeme)
Given a character sequence $\lambda \in \Xi^+$ and a token t, then the relation $t \cong \lambda$ holds iff λ in its entirety matches the regular expression for t. λ is then called a *lexeme* for t. $\square$

Definition 2.2.3 (Prefix)
Given two character sequences $\xi \in \Xi^*$ and $\xi' \in \Xi^*$, then the relation $\xi' \preceq \xi$ holds iff $\exists \varrho \in \Xi^* : \xi = \xi'\varrho$. In this case, $\xi - \xi' = \varrho$. Also, the relation $\xi' \prec \xi$ holds iff $\xi' \preceq \xi$, $\xi' \neq \xi$, and thus $|\xi - \xi'| > 0$. $\square$

Definition 2.2.4 (Character Sequence Matches)
Given a character sequence $\xi \in \Xi^*$ and a set of tokens T, then the set of *matches* for ξ over T is the set $\mathscr{M}(\xi, T) = \{(\lambda, t) \in (\Xi^+, T) : \lambda \preceq \xi \wedge t \cong \lambda\}$. Notice that, because a lexeme cannot be the empty string, $\mathscr{M}(\xi, T) = \emptyset$ if $|\xi| = 0$. $\square$

In order to progress through an input character sequence ξ, a scanner must select a single match (λ, t) from the set $\mathscr{M}(\xi, T)$, and the next match then comes from the set $\mathscr{M}(\xi - \lambda, T)$. How to select a single match is the main focus of our work.

Definition 2.2.5 (Scanner Conflict)

Given a character sequence $\xi \in \Xi^*$ and a set of tokens T, then a ***scanner conflict*** for ξ over T is any set $C \subseteq \mathscr{M}(\xi, T)$ such that $|C| > 1$. We now identify several types of scanner conflicts:

1. Iff $C = \mathscr{M}(\xi, T)$, then C is the ***complete scanner conflict*** for ξ over T.
2. Iff $|C| = 2$, then C is a ***pairwise scanner conflict*** for ξ over T. Let $C = \{(\lambda, t), (\lambda', t')\}$. There are two types of pairwise scanner conflicts:
   (a) An *identity conflict* occurs when $\lambda = \lambda'$ and thus $t \neq t'$.
   (b) We introduce the term ***length conflict*** for the case when $\lambda \neq \lambda'$ regardless of whether $t = t'$. We have chosen this term because, by Definition 2.2.4, $(\lambda \preceq \xi) \wedge (\lambda' \preceq \xi)$, and thus $(\lambda \prec \lambda') \vee (\lambda' \prec \lambda)$, so $|\lambda| \neq |\lambda'|$. For the subtype of length conflict where $t = t'$, we introduce the term ***autolength conflict***.

□

The term identity conflict comes from Nawrocki [27]. Nawrocki also defines a subset of length conflicts that he calls *LM-conflicts* (longest match conflicts), which we explain in section 5.1. We dislike Nawrocki's term in this case as it implies the traditional view that the longest lexeme is always preferable.

For example, both the keyword's regular expression and the identifier's regular expression in Figure 2.2a can match the first three characters of the character sequence of Figure 2.2b. Thus, the keyword token has an identity conflict with the identifier token for this character sequence. However, the identifier's regular expression can also match any prefix of the character sequence. Thus, the identifier has length conflicts with the keyword and with itself. The identifier's length conflicts with itself are autolength conflicts.

Definition 2.2.6 (Traditional Lexical Precedence)

Given a character sequence $\xi \in \Xi^*$ and a set of tokens T, then a scanner generated by a traditional scanner generator like Lex defines a *highest precedence match* from $\mathscr{M}(\xi, T)$ and thus *resolves* the complete scanner conflict for ξ over T using two *lexical precedence rules* in the following order:

1. The scanner resolves all length conflicts by selecting the lexeme $\lambda : \exists t : (\lambda, t) \in \mathscr{M}(\xi, T) \wedge \forall (\lambda', t') \in \mathscr{M}(\xi, T), |\lambda| \geq |\lambda'|$. λ is said to have precedence over other lexemes that prefix ξ.

This rule is usually called *longest match* or *maximal munch*.

2. Given the selected λ, if $|\{t : (\lambda, t) \in \mathscr{M}(\xi, T)\}| > 1$, then the scanner resolves the resulting identity conflicts by selecting the token t whose matching regular expression is declared earliest in the scanner specification. t is said to have precedence over other tokens for which λ is a lexeme.

□

For example, the keyword from Figure 2.2a matches only the first three characters of the character sequence in Figure 2.2b, but the identifier matches the entire character sequence. Thus, the traditional scanner selects the entire character sequence as the lexeme and selects the identifier as the token. Both the keyword and the identifier match the entire character sequence of Figure 2.2c. The traditional scanner selects the keyword in this case because its regular expression is declared first in the scanner specification.

2.2.2 LR(1) Parsers

A parser generator tool, such as Yacc, can be employed to generate a parser from a formal *parser specification*. In this paper, we assume the parsing technique is from the LR(1) family, including LALR(1), canonical LR(1), or some form of minimal LR(1). The parser specification consists primarily of a syntax specification, which contains a context-free grammar and related declarations like precedence and associativity. The parser specification may also contain semantic declarations and literal code in a general-purpose programming language like C. We discuss the various components of a Yacc parser specification in more detail in section 2.3.1. In this section, we discuss how the parser interacts with the scanner, and we formally model the LR(1) parser tables generated from the context-free grammar.

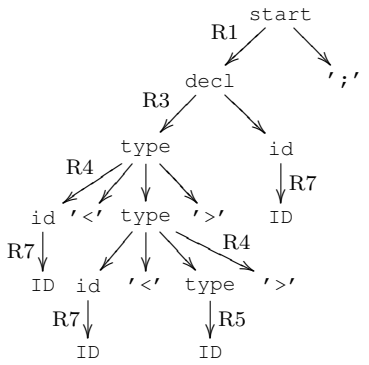
Definition 2.2.7 (Context-free Grammar)
A *grammar* is a tuple $G = (V, T, P, S)$, such that V is a set of *nonterminals*, T is a set of *terminals*, P is a set of *productions*, and S is the *start symbol*. The set $\{V \cup T\}$ is the set of all the grammar's *symbols*, and $S \in V$. In this paper, we are concerned only with *context-free grammars*, so $\forall p \in P, \exists \ell \in V : \exists \varrho \in \{V \cup T\}^* : p = (\ell \rightarrow \varrho)$. □

(a)
```
vector<list<string>> v;
```

(b)
```
a >> b;
```

(c)
```
%token-re ID ([a-zA-Z_][a-zA-Z_0-9]*)
%token-re YYLAYOUT ([\t\n\r ])
%lex-no-tie '>' '>>'
%%
start
  : decl ';'          /*1*/
  | expr ';'          /*2*/
  ;
decl
  : type id           /*3*/
  ;
type
  : id '<' type '>'   /*4*/
  | ID                /*5*/
  ;
expr
  : id '>>' id        /*6*/
  ;
id:
  ID                  /*7*/
  ;
```

(d)



(e)

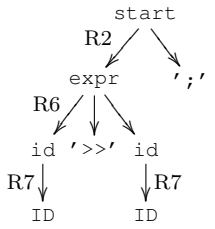


Figure 2.3: Template Argument Lists. (a) In C++0x, the character sequence ">>" can close two template argument lists such that one list is nested within the other. (b) It can also be the bitwise right shift operator. (c) The PSLR(1) parser for this unified scanner and parser specification accepts (d) the first example sentence with this parse tree and (e) the second example sentence with this parse tree.

For example, consider the specification in Figure 2.3c. For now, ignore the declarations preceding the "`%%`" as they would not be permitted by a traditional LR(1) parser generator, which expects the scanner to be generated separately. Instead, consider only the grammar appearing after the "`%%`". All grammar symbols appearing immediately before a "`:`", such as `start` or `decl`, are the grammar's nonterminals. All other grammar symbols, such as `ID` or `'>>'`, are the grammar's terminals. Productions are grouped by their LHS's, and the RHS's for each LHS are separated with a "`|`". Thus, the LHS of productions 1 and 2 is "`start`", the RHS of production 1 is "`decl ';'`", and the RHS of production 2 is "`expr ';'`". By default, the start symbol is the first LHS, which is `start` in our example.

As illustrated in Figure 2.1, the generated parser employs a PDA (pushdown automaton) to read the token sequence produced by the scanner and, using the tokens as terminals in the specified grammar, construct a *parse tree* in a bottom-up fashion. The parse tree derives the token sequence from the grammar's start symbol via the grammar productions. For example, the parser for the grammar in Figure 2.3c constructs the parse tree in Figure 2.3d for the character sequence in Figure

2.3a as long as the scanner transforms that character sequence into the token sequence formed by the leaves of the tree. When performed as an action in a bottom-up parser, a production is referred to as a *reduction*, abbreviated with an "R" in the figure.

Definition 2.2.8 (Augmented Context-free Grammar)
Given a context-free grammar $G = (V, T, P, S)$, then its *augmented grammar* is $\mathscr{G}(G) = (V', T', P', S')$ such that $V' = V \cup \{S'\}$, $T' = T \cup \{\#\}$, $P' = P \cup \{(S' \rightarrow S\#)\}$, $S' \notin \{V \cup T\}$, $\# \notin \{V \cup T\}$, and $\#$ is the *end token*, a special token marking the end of any token sequence. □

Given the grammar G from the parser specification, a parser generator like Yacc often employs $\mathscr{G}(G)$ instead of G to make detecting the end of the token sequence easier. Moreover, scanners generated by Lex return the # token upon reaching the end of the input character sequence. Thus, the generated parser has successfully parsed and accepted the token sequence as soon as it performs the reduction $(S' \rightarrow S\#)$.

The parser's PDA is encoded in a set of *parser tables*. For example, the first column of Table 2.1 shows the canonical LR(1) parser tables for the grammar of Figure 2.3c. Each numbered cell describes an LR(1) parser *state*. Each row within a state represents an LR(1) *item*, which consists of a dotted production, a *lookahead set*, and an LR(1) *action*. We now present a formal model for LR(1) parser tables.

Definition 2.2.9 (LR(1) Item)
Given a context-free grammar $G : \mathscr{G}(G) = (V', T', P', S')$, then an *LR(1) item* for G is a tuple $m = (p, d, K)$ such that:

1. $\exists \ell : \exists \varrho : p = (\ell \rightarrow \varrho) \in P'$.
2. d is an integer such that $1 \leq d \leq |\varrho| + 1$ to specify the index before which to insert a dot in the sequence ϱ. The dot indicates a position in a parse.
3. The tuple (p, d) is the *core* of m.
4. K is a set of terminals called the *lookahead set*.

□

Canonical LR(1)				LALR(1)			
0.	start′ →·start #	{}	G2	0.	start′ →·start #	{}	G2
	start →·decl ';'	{#}	G3		start →·decl ';'	{#}	G3
	→·expr ';'	{#}	G5		→·expr ';'	{#}	G5
	decl →·type id	{';'}	G4		decl →·type id	{';'}	G4
	expr →·id '>>' id	{';'}	G6		expr →·id '>>' id	{';'}	G6
	type →·id '<' type '>'	{ID}	G6		type →·id '<' type '>'	{ID}	G6
	→·ID	{ID}	S1		→·ID	{ID}	S1
	id →·ID	{'>>','<'}	S1		id →·ID	{'>>','<'}	S1
1.	type → ID·	**{ID}**	R5	1.	type → ID·	**{ID,'>'}**	R5
	id → ID·	{'>>','<'}	R7		id → ID·	{'>>','<'}	R7
2.	start′ → start·#	{}	S7	2.	start′ → start·#	{}	S7
3.	start → decl·';'	{#}	S8	3.	start → decl·';'	{#}	S8
4.	decl → type·id	{';'}	G10	4.	decl → type·id	{';'}	G10
	id →·ID	{';'}	S9		id →·ID	{';'}	S9
5.	start → expr·';'	{#}	S11	5.	start → expr·';'	{#}	S11
6.	expr → id·'>>' id	{';'}	S13	6.	expr → id·'>>' id	{';'}	S13
	type → id·'<' type '>'	{ID}	S12		type → id·'<' type '>'	{ID}	S12
7.	start′ → start #·	{}	Acc	7.	start′ → start #·	{}	Acc
8.	start → decl ';'·	{#}	R1	8.	start → decl ';'·	{#}	R1
9.	id → ID·	{';'}	R7	9.	id → ID·	{';'}	R7
10.	decl → type id·	{';'}	R3	10.	decl → type id·	{';'}	R3
11.	start → expr ';'·	{#}	R2	11.	start → expr ';'·	{#}	R2
12.	type → id '<'·type '>'	**{ID}**	G14	12.	type → id '<'·type '>'	**{ID,'>'}**	G14
	→·id '<' type '>'	{'>'}	G15		→·id '<' type '>'	{'>'}	G15
	→·ID	{'>'}	**S18**		→·ID	{'>'}	**S1**
	id →·ID	{'<'}	**S18**		id →·ID	{'<'}	**S1**
13.	expr → id '>>'·id	{';'}	G16	13.	expr → id '>>'·id	{';'}	G16
	id →·ID	{';'}	S9		id →·ID	{';'}	S9
14.	type → id '<' type·'>'	**{ID}**	S17	14.	type → id '<' type·'>'	**{ID,'>'}**	S17
15.	type → id·'<' type '>'	{'>'}	**S19**	15.	type → id·'<' type '>'	{'>'}	**S12**
16.	expr → id '>>' id·	{';'}	R6	16.	expr → id '>>' id·	{';'}	R6
17.	type → id '<' type '>'·	**{ID}**	R4	17.	type → id '<' type '>'·	**{ID,'>'}**	R4
18.	**type → ID·**	**{'>'}**	**R5**				
	id → ID·	**{'<'}**	**R7**				
19.	**type → id '<'·type '>'**	**{'>'}**	**G20**				
	→·id '<' type '>'	**{'>'}**	**G15**				
	→·ID	**{'>'}**	**S18**				
	id →·ID	**{'<'}**	**S18**				
20.	**type → id '<' type·'>'**	**{'>'}**	**S21**				
21.	**type → id '<' type '>'·**	**{'>'}**	**R4**				

Table 2.1: Parser Tables for Template Argument Lists. These are the canonical LR(1) and LALR(1) parser tables for the grammar of Figure 2.3c. Differences are shown in bold.

Definition 2.2.10 (LR(1) Action)

Given a context-free grammar $G : \mathscr{G}(G) = (V', T', P', S')$, then an *LR(1) action* for G is a tuple (a_t, a_p, a_s) such that any one of the following is true:

1. a_t = "S" to indicate a *shift* action, which removes a token from the input and pushes the LR(1) state a_s onto the parser stack. In this case, a_p is left undefined. We model LR(1) states below in Definition 2.2.11 to contain LR(1) actions.

2. a_t = "G" to indicate a *goto* action, which removes a nonterminal from the input and pushes

the LR(1) state a_s onto the parser stack. Again, a_p is left undefined.

3. a_t = "R" to indicate a *reduce* action such that $a_p \in P'$ is the production by which to reduce the parser stack. That is, given that $a_p = (\ell \rightarrow \varrho)$, then $|\varrho|$ states are popped from the stack, and ℓ is inserted at the front of the input. In this case, a_s is left undefined.

4. During parser conflict resolution, a may be given other values, but that topic is beyond the scope of this paper.

□

Definition 2.2.11 (LR(1) State)

Given a context-free grammar $G : \mathscr{G}(G) = (V', T', P', S')$, then we model an *LR(1) state* s_p for G as a set of LR(1) items augmented with associated LR(1) actions. Thus, $\forall m = (p, d, K, a) \in s_p$, given that $p = (\ell \rightarrow \varrho) \wedge a = (a_t, a_p, a_s)$, then:

1. (p, d, K) is an LR(1) item for G.

2. $\forall m' = (p', d', K', a') \in s_p : m \neq m', (p, d) \neq (p', d')$.

3. a is the LR(1) action for G that is associated with (p, d, K) in that, before conflict resolution is performed on s_p, a is determined by m's core as follows:

   (a) Iff $d < |\varrho| + 1$, then:

      i. $\varrho[d] \in T' \Leftrightarrow a_t =$ "S".

      ii. $\varrho[d] \in V' \Leftrightarrow a_t =$ "G".

      iii. $\varrho[d]$ is the input symbol upon which the action should be performed, so the chosen parser table generation algorithm computes a_s based on $\varrho[d]$.

   (b) Iff $d = |\varrho| + 1$, then $a_t =$ "R" $\wedge\ a_p = p$. K contains the input tokens upon which this action should be performed.

Given an LR(1) state s_p, then the set $\{(p, d) : \exists (p, d, K, a) \in s_p\}$ is the *core* of s_p. □

Table 2.2 illustrates how the canonical LR(1) parser tables of Table 2.1 parse the input token sequence formed by the leaves of the parse tree in Figure 2.3d. The parser stack is initialized by pushing the start state, state 0, as shown in the first row of the parse. The parser's next action

Stack	Input	Action
0	`ID '<' ID '<' ID '>' '>' ID ';' #`	S1
0,1	`'<' ID '<' ID '>' '>' ID ';' #`	R7
0	`id '<' ID '<' ID '>' '>' ID ';' #`	G6
0,6	`'<' ID '<' ID '>' '>' ID ';' #`	S12
0,6,12	`ID '<' ID '>' '>' ID ';' #`	S18
0,6,12,18	`'<' ID '>' '>' ID ';' #`	R7
0,6,12	`id '<' ID '>' '>' ID ';' #`	G15
0,6,12,15	`'<' ID '>' '>' ID ';' #`	S19
0,6,12,15,19	`ID '>' '>' ID ';' #`	S18
0,6,12,15,19,18	`'>' '>' ID ';' #`	R5
0,6,12,15,19	`type '>' '>' ID ';' #`	G20
0,6,12,15,19,20	`'>' '>' ID ';' #`	S21
0,6,12,15,19,20,21	`'>' ID ';' #`	R4
0,6,12	`type '>' ID ';' #`	G14
0,6,12,14	`'>' ID ';' #`	S17
0,6,12,14,17	`ID ';' #`	R4
0	`type ID ';' #`	G4
0,4	`ID ';' #`	S9
0,4,9	`';' #`	R7
0,4	`id ';' #`	G10
0,4,10	`';' #`	R3
0	`decl ';' #`	G3
0,3	`';' #`	S8
0,3,8	`#`	R1
0	`start #`	G2
0,2	`#`	S7
0,2,7		R0

Table 2.2: A Template Argument List Parse. This table shows how the canonical LR(1) parser tables of Table 2.1 parse the input token sequence show in the initial configuration above. A scanner might return this token sequence for the character sequence in Figure 2.3a, for example.

---

is always dictated by the state at the top of the stack and the next symbol in the input, `ID` in this case. State 0 has an item with its dot before `ID`, and the item's action is S1, shift to state 1. Thus, the parser removes `ID` from the input and pushes state 1 onto the stack. The dot in the second item of state 1 is at the end of the RHS of production 7, and the item's action is thus R7, reduce by production 7. The item's lookahead set contains the token `'<'`, which is the next symbol in the input. Thus, the parser pops one state off the parser stack for each RHS symbol of production 7 and inserts the LHS symbol at the front of the input. Such reduce actions construct the edges of the parse tree. Now state 0 is at the top of the stack again and the next symbol is `id`. State 0 has an item with its dot before `id`, and the item's action is G6, goto state 6. Thus, the parser removes `id` from the input and pushes state 6 onto the stack. In this paper, we use the term goto such that a goto always follows a reduce and thus is always performed upon seeing a nonterminal in the input. The only exception is that, upon reaching the action R0, the parser terminates with a success status.

Definition 2.2.12 (Accepts)

Given the LR(1) parser state s_p, then $acc(s_p) = \{t : \exists((\ell \rightarrow \varrho), d, K, (a_t, a_p, a_s)) \in s_p : (a_t = \text{“S”} \wedge \varrho[d] = t) \vee (a_t = \text{“R”} \wedge t \in K)\}$. $\square$

When the parser is in state s_p, if the scanner returns a token that is not in the set $acc(s_p)$, the parser reports a syntax error because there is no appropriate parser action on that token. The scanner and parser always stay in sync, so it would seem the scanner could examine the current $acc(s_p)$ before selecting the next token in order to avoid inducing a syntax error. Unfortunately, when using traditional scanner-based LR(1) parser generation tools like Lex and Yacc, the only tool-supported communication between the scanner and parser is the token sequence produced by the scanner and the parser's requests for new tokens. Any other desired communication must be manually coded by the user of the tools. Thus, unless the user manually extends the scanner's capabilities, the scanner's recognition power is limited to regular languages. In the next section, we further examine this shortcoming of traditional scanner-based LR(1) parsing in the context of composite languages.

2.3 Composite Languages

Most practical languages are the product of some degree of language composition. In section 2.3.1, we explore one of the most obvious examples, the parser specification languages accepted by parser generators like Yacc. In section 2.3.2, we present several examples of regular sub-languages from traditional programming languages like C and C++. In section 2.3.3, we present some less obvious examples of sub-languages. In section 2.3.4, we discuss the complex form of language composition employed by modern extensible languages.

2.3.1 Parser Specifications

As depicted in Figure 2.4a, Yacc's internal parser parses a parser specification, p_{yacc}, written in the language L_{yacc} in order to generate the source code, p_{code}, that is compiled into the specified parser, p_{gen}. Figure 2.4b presents an example p_{yacc} containing an expression grammar. As this example demonstrates, L_{yacc} is a composite language with several easily distinguishable sub-languages, passages of which are boxed and labeled in the figure. The primary sub-language is L_{decl}, which

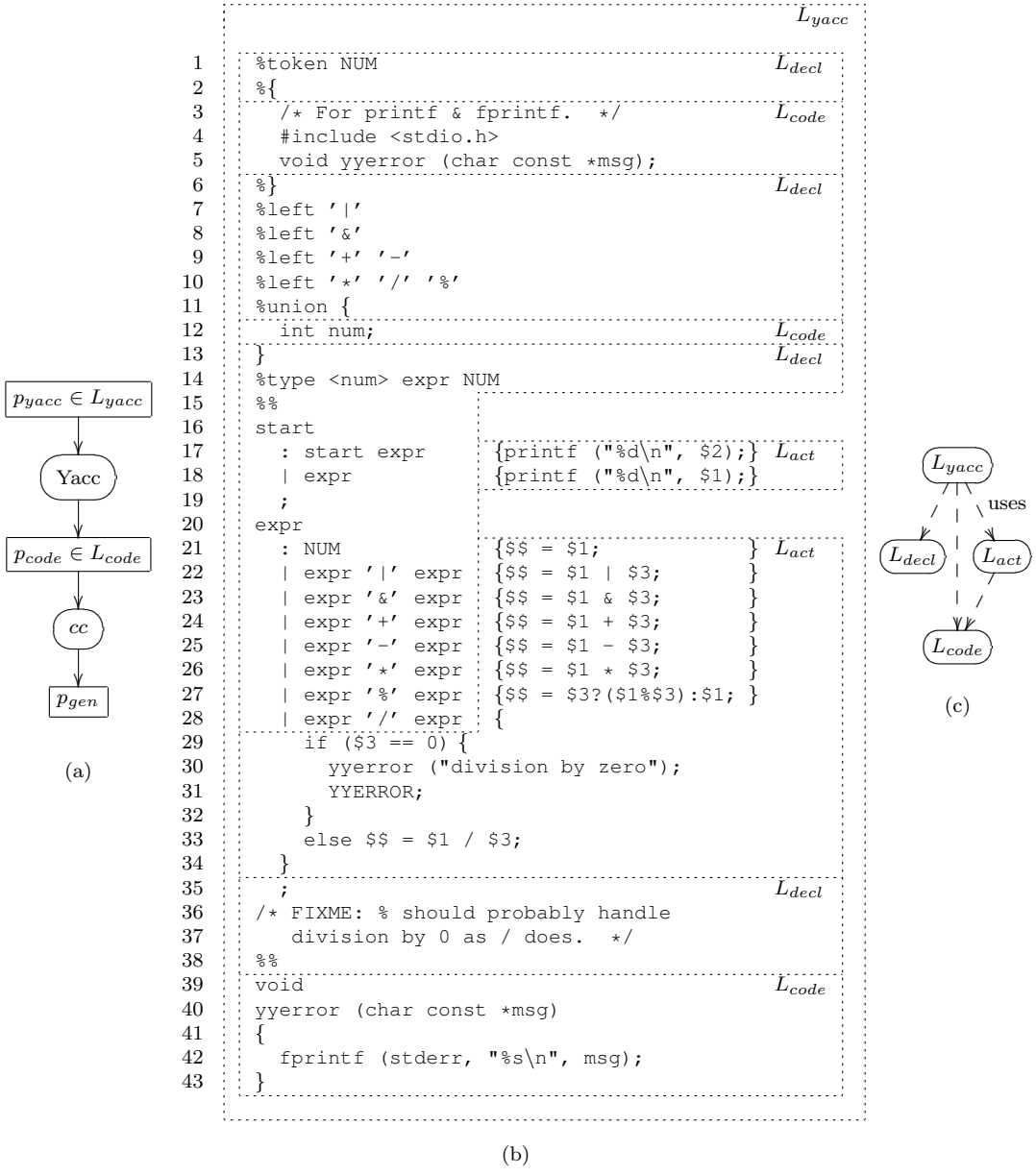


Figure 2.4: Yacc Parser Specification Language. (a) L_{yacc} is the language of a parser specification, p_{yacc}, read by Yacc's internal parser in order to generate the code, p_{code}, for the specified parser, p_{gen}. (b) Each passage within this example p_{yacc} is written in the sub-language L_{decl}, L_{code}, or L_{act}. (c) Thus, L_{yacc} is a composite language.

consists of grammar productions and other declarations. The other two main sub-languages are L_{code} and L_{act}. L_{code} consists of literal code in the programming language C to be printed verbatim to p_{code}. L_{act} is used for semantic actions and is an extension of L_{code} with "`$`" tokens to refer to semantic values. These composition relationships are depicted in Figure 2.4c.

The syntax of L_{decl} is quite different than the syntax of L_{code} and L_{act}, and Yacc must parse each sub-language correctly. For example, in L_{decl}, the character "`:`" is a token that follows the LHS of a grammar production as on lines 17 and 21 of Figure 2.4b. In L_{code} or L_{act}, "`:`" may appear in a number of places, such as in the C ternary operator on line 27, but there is no grammar production syntax in these sub-languages. There are also many notable differences at the lexical level. For example, in L_{decl}, the character sequence "`%left`" is a single token used to declare left associativity as on lines 7-10. In L_{code} or L_{act}, "`%`" is the modulo operator token as on line 27, and so "`%left`" would be the modulo operator token followed by an identifier token, "`left`". In L_{decl}, the character "`|`" is a token separating RHS's of productions that are paired with the same LHS as on lines 18-28, and so two adjacent occurrences of "`|`" would be two tokens with an empty RHS in between. In L_{code} or L_{act}, the character "`|`" is the bitwise OR operator token as on line 22, and two adjacent occurrences of "`|`" would be the single logical OR operator token.

In this section, we assume a Yacc implementation whose internal parser is a traditional scanner-based LR(1) parser. Thus, it is the scanner's job to recognize character sequences as tokens using an FSA without the help of the parser. Because the lexical rules differ in L_{decl}, L_{code}, and L_{act}, the scanner can encounter conflicts when trying to determine which tokens from which of these sub-languages should be matched for character sequences like "`%left`". While the traditional lexical precedence rules from Definition 2.2.6 might resolve a conflict appropriately for one sub-language, they cannot be appropriate for all sub-languages because different sub-languages require different tokens to be selected.

The scanner can completely avoid conflicts among tokens from different sub-languages by recognizing the boundaries between passages of the sub-languages. To illustrate, consider again the example p_{yacc} in Figure 2.4b. At the beginning of p_{yacc} on line 1, Yacc's internal scanner must start in a state where it recognizes only tokens from L_{decl}. Upon recognizing a "`%{`" token or upon recognizing a "`%union`" token followed by a "{" token in L_{decl} as on lines 2 and 11, the scanner must enter a special state where it recognizes only tokens from L_{code}. Upon recognizing a "{" token after a grammar production's RHS in L_{decl} as on lines 17-28, the scanner must enter a special state

```
 1 <L_DECL>{
 2   "%token"                    return PERCENT_TOKEN;
 3   [a-zA-Z._][a-zA-Z._0-9]*    yylval = strdup (yytext); return NAME;
 4   "%{"                        BEGIN L_CODE; return CODE_START;
 5   [ \t\r\n]                   /*Discard whitespace.*/
 6 }
 7 <L_CODE>{
 8   "%}"                        BEGIN L_DECL; return CODE_END;
 9   .|\n                        yylval = strdup (yytext); return CODE;
10 }
```

Figure 2.5: Lex Start Conditions. This Lex specification handles some of the transitions between the L_{decl} and L_{code} sub-languages of L_{yacc} as well as a few tokens.

where it recognizes only tokens from L_{act}. When the scanner later recognizes a matching "`%}`" or "}" token, it must reenter the initial L_{decl} state. Also, upon recognizing the second occurrence of the "`%%`" token in the L_{decl} state as on line 38, the scanner must leave the L_{decl} state and enter the L_{code} state for the remainder of p_{yacc}.

These L_{decl}, L_{code}, and L_{act} states can be implemented as *start conditions* when using Lex. A start condition is simply an FSA state to which the scanner transitions by default at the end of each token to prepare for the beginning of the next token. Thus, the use of start conditions does not inherently extend the language recognition power of the scanner beyond a pure FSA. For example, Figure 2.5 shows a Lex specification that handles some of the transitions between L_{decl} and L_{code} as well as a few tokens. Consider how the generated scanner would behave for lines 1-2 in Figure 2.4b. The scanner starts in the L_{decl} start condition and transitions back to it at the end of the "`%token`" and at the end of the "`NUM`" because the rules for these tokens on lines 2 and 3 in Figure 2.5 do not specify a transition to a different start condition. However, at the end of the "`%{`" token, the scanner transitions to the L_{code} start condition. As shown on line 4 in Figure 2.5, the Lex user must explicitly specify such a transition to a different start condition by using the `BEGIN` macro in C.

Look at the semantic action for the "`/`" operator on lines 28-34 in Figure 2.4b. Within a passage of L_{act} between a pair of "{" and "}", Yacc's internal scanner must balance nested occurrences of "{" and "}" so that it can determine when to transition back to the L_{decl} start condition. However, it is well known that a language containing nested constructs is not a regular language, and thus an FSA is not powerful enough to recognize it [11]. In cases like this, the Lex user often resorts to ad-hoc code that uses a variable to maintain the count of unclosed braces. The code increments the count for each "{" and decrements the count for each "}". This code is

equivalent in power to a stack that pushes for each "{" and pops for each "}". Use of a stack is so often appealing in a scanner that Flex [3], the BSD implementation of Lex, provides a start condition stack, which can also be used for recognizing nested structures. However, like any transition among start conditions, manipulation of this stack must be explicitly coded in C by the Flex user.

An FSA combined with a stack is a PDA. Thus, when the Lex or Flex user adds code to emulate, implement, or manipulate a stack, he is really just reimplementing machinery the parser already possesses. Moreover, regardless of whether the scanner requires a stack because of nested structures, the scanner often returns tokens like "{", "}", "%{", and "%}" to the parser so that the parser can track the transitions between sub-languages for the purposes of syntactic analysis as specified by the grammar. In this case, the contextual information recorded by the scanner's current start condition or on the scanner's stack is duplicated on the parser stack. Unfortunately, when using traditional scanner and parser generator tools like Lex and Yacc, the scanner has no way to access the parser's stack, and so the user must code and maintain the same contextual logic in both the scanner specification and the parser specification.

2.3.2 Regular Sub-languages

In Figure 2.4b, consider the C-style comments on line 3 and lines 36-37, the single-quoted character literals on lines 22-28, and the double-quoted string literals on lines 17-18. L_{yacc} permits such comments and literals to appear in each of L_{decl}, L_{code}, and L_{act}.[1] Within a comment or literal, characters like "%" do not represent the same tokens they do outside the comment or literal. For example, any occurrence of "%}" or "}" embedded in a comment or literal does not indicate a transition from the L_{code} or L_{act} start condition back to the L_{decl} start condition. Thus, even though Yacc can treat most characters in L_{code} and L_{act} as inert text to be printed verbatim to p_{code}, it must fully parse comments and literals in L_{code} and L_{act} in order not to misinterpret the characters contained within.

Because the syntax of the comment, character literal, and string literal each is regular, each can be specified as a single token with a single regular expression within each sub-language of L_{yacc}. When using a scanner-generator like Lex, this approach requires that the scanner recognize each comment and literal as a whole without the possibility of separate actions for individual parts of the comment or literal. For example, the syntax of the string literal includes a set of escape sequences,

[1] Yacc does not actually permit string literals to appear within L_{decl}, but Bison does.

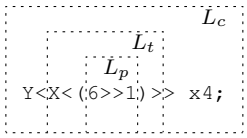
such as the "\n" in the "%d\n" shown on line 17 in Figure 2.4b. If the string literal is specified as a single token with a single regular expression, then "%d\n" must be fully recognized and later rescanned in order to convert the contained "\n" into a newline. Also, if the closing quotes were missing on line 17, the scanner would reject the entire string literal with a generic lexical error message with no acknowledgement that the text is the beginning of a string literal. However, a user-friendly scanner should instead report different error messages for the missing quotes and for each invalid escape sequence or other invalid character within the string literal.

The developers of Bison's internal scanner specification apparently realized that the structure of the comment, character literal, and string literal each is complex enough to be handled as a separate sub-language in the scanner. Thus, Bison's internal scanner specification assigns a unique start condition to each and breaks up each regular expression into multiple parts with separate actions. From the viewpoint of a robust scanner then, most traditional programming languages, such as C or C++, exhibit at least a simple composition of languages because they contain regular sub-languages like comments and literals. Lex's start conditions are sufficient to handle these regular sub-languages without a stack, but adding the associated start conditions further complicates the scanner specification and worsens maintainability.

2.3.3 Subtle Sub-languages

L_{decl}, L_{code}, and L_{act} are clear examples of distinct sub-languages within L_{yacc}. Because comments, character literals, and string literals can be expressed with regular expressions, the motivation to classify them as sub-languages was less obvious until we considered them from the viewpoint of a robust scanner. In this section, we explore examples of sub-languages that are perhaps even less obvious than these.

Consider extending the Lex specification in Figure 2.5 to handle all transitions between L_{decl} and L_{code}. If we ignore the issue of comments and literals, these lexical rules already handle the code between lines 2 and 6 in Figure 2.4b, where a "%}" cannot appear because it indicates a transition from L_{code} back to L_{decl}. However, these lexical rules do not handle the code between lines 11 and 13. In this case, "}" instead indicates the transition back to L_{decl}. In general, "{" and "}" must be balanced here as in L_{act}. Moreover, after line 38, the scanner should not treat any of these character sequences specially because a transition back to L_{decl} is not possible. In other words, from the viewpoint of the scanner, L_{code} really consists of three sub-languages whose



(a)

(b) `%lex-prec '>' -< '>>'` for L_c and L_p

(c) `%lex-prec '>>' -< '>'` for L_t

(d) `template_id: template_name '<' template_argument_list_opt '>' ;`

(e) `primary_expression: '(' expression ')' ;`

Figure 2.6: Scoped Declarations. Let L_c be the main C++0x language, let L_t be the template argument list sub-language, and let L_p be the parenthesized expression sub-sub-language. (a) We have copied this example sentence from the C++0x proposal for handling right angle brackets [37]. The interpretation of the character sequence ">>" is different in (b) L_c and L_p than in (c) L_t. (d) In this C++ grammar production, `template_argument_list_opt` is a start symbol for the grammar of L_t. (e) In this production, `expression` is a start symbol for the grammar of L_p.

---

only distinction is how to handle the tokens that delimit passages of the sub-language. Each such sub-language requires its own start condition, which further complicates the scanner specification for L_{yacc} and worsens maintainability.

Another subtle example of a sub-language appears in the evolving C++0x specification [37]. Figure 2.3a shows an example C++ statement involving templates. It declares the variable `v` to be a `vector` of `list`'s of `string`'s. That is, `list<string>` is the argument of the template `vector`, and `string` is the argument of the nested template `list`. Intuitively then, the character sequence ">>" contains two distinct right angle bracket tokens such that the first closes the template argument list of `list` and the second closes the template argument list of `vector`. However, Figure 2.3b shows another example C++ statement where ">>" is a single token, the bitwise right shift operator. Previous versions of C++ required two consecutive right angle brackets to be separated by whitespace in order not to be interpreted as the single bitwise right shift operator token, so Figure 2.3a contains a syntax error in that case. However, C++0x proposes that, within template argument lists but not elsewhere, ">>" should be interpreted as two tokens regardless of intervening whitespace.

In this way, the C++0x template argument list requires different lexical rules than other parts of a C++0x program. From the viewpoint of the scanner then, the C++0x template argument list is a distinct sub-language. A "<" indicates a transition into this sub-language. Because of nesting, a scanner must employ a stack in order to recognize which ">" indicates a transition back out.

To complicate matters further, C++0x proposes that nested parentheses and square brackets appearing in the template argument list sub-language delimit a sub-sub-language where ">>" is once

again interpreted as a single bitwise right shift operator token. Of course, to recognize the transition out of this sub-sub-language, the scanner must employ a stack to balance parentheses and square brackets. Figure 2.6a shows an example sentence copied from the C++0x proposal for handling right angle brackets [37]. This sentence employs all three layers of sub-languages, passages of which we have boxed and labeled in the figure. We discuss Figures 2.6b–2.6e in section 3.7.

2.3.4 Extensible Languages

A composite language like those we have discussed so far is often specified along with all of its sub-languages in a single scanner specification and a single parser specification. However, there is a modern movement to specify sub-languages in separate modules so that both language experts and non-experts can more easily compose selected sub-languages into arbitrarily complex layers in order to generate custom parsers for specific domains. Such languages are often called *extensible languages* [12, 13, 18, 40]. For example, Van Wyk and Schwerdfeger embed SQL schemas, SQL queries, and condition tables into an extensible version of Java [40]. Grim extends C with an AOP (aspect-oriented programming) notation and embeds C within Java [18]. Bravenboer et al. develop an extensible specification of AspectJ, which is in turn an AOP extension of Java [12]. Their goal is to facilitate the specification of new Java language extensions as researchers experiment with evolving AOP concepts and syntaxes.

Because extensible languages permit a complex and arbitrary style of language composition, they exacerbate the difficulties that traditional scanner-based LR(1) parser generators already pose for simpler composite languages. Specifically, the user's task of manually replicating each newly developed composite grammar's sub-language transition logic into a separate scanner specification erodes the modularity of sub-language specifications and thus impedes the development and maintenance of new composite language specifications. The difficulty of this task may even be preventative if the user is not a language expert.

2.4 Summary

A composite language is composed of multiple sub-languages. Language composition is obvious when some sub-languages are already specified independently. For example, the parser specification language read by a parser generator like Yacc contains grammar productions and dec-

larations as well as passages of the programming language C. When languages are composed in this way, sub-language transitions during parsing usually entail corresponding transitions in the lexical rules. Thus, the scanner must recognize sub-language transitions in order to avoid scanner conflicts among the various sub-languages. The scanner for a traditional programming language like C or C++ must sometimes recognize similar transitions in its lexical rules. From the viewpoint of the scanner then, such languages are also composite languages.

Scanner-generator tools like Lex provide start conditions for implementing sub-language transitions in the scanner, but the user must specify these transitions explicitly in the scanner specification. In a parser specification, the user often must specify the same sub-language transitions implicitly in the grammar. The contextual information recorded by the scanner's current start condition at run time is then duplicated on the parser's stack. Nevertheless, traditional scanner and parser generators attempt to generate loosely coupled scanners and parsers, so the user must maintain these tightly coupled scanner and parser specifications separately but consistently. Scanner and parser specifications would be significantly more maintainable if all sub-language transitions were instead computed from a grammar by a parser generator and recognized automatically by the scanner using the parser's stack. The need to automate sub-language transitions in the scanner in this way is growing with the popularity of modern extensible languages, which may contain complex layers of sub-languages composed arbitrarily for the requirements of specific domains.

Chapter 3

Methodology

In this chapter, we detail the functionality of our PSLR(1) generator and the tightly coupled minimal LR(1) parsers and pseudo-scanners that it constructs. We also describe the lexical formalisms that we add to the language of Bison parser specifications to form PSLR(1)'s unified specification language. In section 3.1, we explain how a pseudo-scanner generated from such a PSLR(1) specification automatically resolves scanner conflicts among tokens from multiple sub-languages. In section 3.2, we describe a lexical precedence system for resolving remaining scanner conflicts. In sections 3.3–3.6, we describe mechanisms to address a number of new challenges that are not exhibited by traditional scanners. In section 3.7, we describe a mechanism to permit scoped syntactic declarations. We summarize in section 3.8.

3.1 Pseudo-scanner

Consider Figure 2.3c on page 10, which presents a PSLR(1) specification for a scanner and parser that accept the sentences in Figures 2.3a and 2.3b with the parse trees in Figures 2.3d and 2.3e, respectively. The tokens within it are named `';'`, `'<'`, `'>'`, `'>>'`, and `ID`. The tokens whose names are quoted are literal character sequences, and so their regular expressions are implicit. For example, the token `';'` matches only the semicolon. The ***token regular expression directive***, denoted `%token-re`, specifies regular expressions for other tokens. For example, the character sequences in Figures 2.3a and 2.3b that match the regular expression specified for the token `ID` are "`vector`", "`list`", "`string`", "`v`", "`a`", and "`b`". The token `YYLAYOUT` is special in that it

specifies characters that the pseudo-scanner should discard as whitespace. We ignore `YYLAYOUT` for the remainder of this section and discuss it in detail in section 3.6. We discuss the `%lex-no-tie` directive in section 3.3.

In section 2.2.2, we assumed that a scanner would return the proper token sequence for the character sequence in Figure 2.3a, and we discussed how an LR(1) parser for the grammar of Figure 2.3c would then parse that token sequence. However, a traditional scanner actually encounters a length conflict upon reaching ">>" in Figure 2.3a as well as in Figure 2.3b. That is, the scanner could recognize the first ">" as a `'>'` token, or it could recognize the entire ">>" as a `'>>'` token. For both sentences, the traditional scanner resolves this conflict using longest match by default and thus selects the `'>>'` token. Consider again the first column of Table 2.1, which shows the canonical LR(1) parser tables for the grammar of Figure 2.3c. When the parser reaches the ">>" in Figure 2.3b, the parser's current state, s_p, is state 1, and so `'>>'` is in $acc(s_p)$ but `'>'` is not. Because the traditional scanner returns `'>>'`, the parser proceeds without error. However, as illustrated in Table 2.2, when the parser reaches the ">>" in Figure 2.3a, s_p is state 18, and so `'>'` is in $acc(s_p)$ but `'>>'` is not. Because the traditional scanner again returns `'>>'`, the parser reports a syntax error even though Figure 2.3c specifies that this sentence is acceptable.

As we explained for C++0x in section 2.3.3, we say that the ">>" appears in a different sub-language in Figure 2.3a than in Figure 2.3b. With a traditional scanner generator tool like Lex, the user must manually specify transitions between the sub-languages in order to eliminate the conflict and select the correct token at the ">>" in each sentence. However, as we just demonstrated for both sentences, $acc(s_p)$ already contains the correct token for the current sub-language at this point without the conflicting token from the other sub-language. Thus, if the scanner were to recognize and return only tokens that are in $acc(s_p)$, it would detect sub-language transitions automatically based on syntactic left context as indicated by s_p. This exploitation of the parser's stack is the simple premise of the pseudo-scanner, which we now formalize.

Definition 3.1.1 (Pseudo-scanner)

Given an LR(1) parser whose state set is Σ_p, then a scanner behaves as a ***pseudo-scanner*** for that parser iff, $\forall \xi \in \Xi^*, \forall s_p \in \Sigma_p : \mathscr{M}(\xi, acc(s_p)) \neq \emptyset$, when the input character sequence is ξ and the parser is in state s_p, the scanner selects its match from $\mathscr{M}(\xi, acc(s_p))$. □

According to Definition 3.1.1, a pseudo-scanner for the canonical LR(1) parser in Table 2.1

returns a `'>'` token at the ">>" in Figure 2.3a, and it returns a `'>>'` token at the ">>" in Figure 2.3b. In other words, the pseudo-scanner eliminates the traditional scanner's conflict automatically, it returns the correct token for both sentences, and the parser proceeds without error.

3.2 Lexical Precedence

In Definition 3.1.1, it is possible that $|\mathscr{M}(\xi, acc(s_p))| > 1$. That is, while a pseudo-scanner helps to eliminate scanner conflicts between tokens from different sub-languages, some scanner conflicts may remain. We refer to these remaining scanner conflicts as ***pseudo-scanner conflicts***.

Definition 3.2.1 (Pseudo-scanner Conflict)
Given a character sequence $\xi \in \Xi^*$ and an LR(1) parser state s_p, then a ***pseudo-scanner conflict*** for ξ in s_p is a scanner conflict for ξ over $acc(s_p)$. □

The cause of a pseudo-scanner conflict is that syntactic left context as tracked on the parser's stack is not powerful enough to invalidate some of the sub-languages that have conflicting tokens. Moreover, the conflict may be induced by an ambiguity within the PSLR(1) specification. In any case, the user can try to rewrite his PSLR(1) specification to eliminate the conflict, or he can depend on lexical precedence rules to define a highest precedence match from $\mathscr{M}(\xi, acc(s_p))$ and thus resolve the conflict.

PSLR(1) specifications can employ the ***lexical precedence directive***, denoted `%lex-prec`, for explicitly declaring several kinds of lexical precedence rules to resolve pseudo-scanner conflicts without the need for traditional start conditions or other ad-hoc code. In section 3.2.1, we explain the guiding principles we followed while developing the `%lex-prec` directive. In section 3.2.2, we explain how the `%lex-prec` directive can be used to declare traditional lexical precedence rules in accordance with these guiding principles. In section 3.2.3, we describe support for several non-traditional rules. In sections 3.2.4 and 3.2.5, we discuss unambiguous and ambiguous uses of the various lexical precedence rules. In section 3.2.6, we describe the pseudo-scanner tables that the PSLR(1) generator constructs to encode the lexical precedence rules selected by the user. In section 3.2.7, we describe the generator's algorithm for discovering, resolving, and reporting pseudo-scanner conflicts in those tables. We summarize in section 3.2.8.

3.2.1 Guiding Principles

A traditional scanner generator like Lex resolves all scanner conflicts using the lexical precedence rules given in Definition 2.2.6 without warning the user. This approach is dangerous especially in the face of software evolution. As our example in section 3.1 demonstrates, when a conflict arises for an unambiguous scanner and parser specification because the scanning and parsing method is not quite powerful enough to implement that specification, any resolution of that conflict can eliminate one or more sentences from the language. If, instead, the specification is ambiguous, any resolution of the conflict can at least eliminate one possible parse tree. Thus, if the user makes a change to an existing specification and unexpectedly induces a new conflict that the generator resolves quietly, there may be severe and undesirable changes in behavior without warning, and exhaustive testing is vital to reveal these changes. This problem is even more dangerous in the case of modern extensible languages for which sub-languages may be composed and reorganized arbitrarily. Unlike traditional scanner generators and scanner conflicts, traditional LR(1) parser generators do report parser conflicts to the user. Our PSLR(1) generator extends the latter practice for pseudo-scanner conflicts.

Recall the distinction between a complete scanner conflict and a pairwise scanner conflict from Definition 2.2.5, and recall that a pseudo-scanner conflict is a kind of scanner conflict as described in Definition 3.2.1. A pseudo-scanner's behavior is not fully defined unless all complete pseudo-scanner conflicts are resolved. That is, for every character sequence $\xi \in \Xi^*$ and for every LR(1) parser state s_p such that $|\mathscr{M}(\xi, acc(s_p))| > 1$, the complete pseudo-scanner conflict for ξ in s_p is the set of matches $\mathscr{M}(\xi, acc(s_p))$, which must be resolved by defining a highest precedence match from that set. Lexical precedence rules declared with `%lex-prec` in a PSLR(1) specification can define this highest precedence match by resolving pairwise pseudo-scanner conflicts contained within that complete pseudo-scanner conflict. However, it is not always necessary to resolve all such pairwise pseudo-scanner conflicts. For example, given some lexeme $\lambda \preceq \xi$ and given a set of three unique tokens $\{t, t', t''\} \subseteq acc(s_p)$, assume that $\mathscr{M}(\xi, acc(s_p)) = \{(\lambda, t), (\lambda, t'), (\lambda, t'')\}$. To use `%lex-prec` to specify that (λ, t'') is the highest precedence match, it is sufficient to specify rules for identity conflicts such that $t < t''$ and $t' < t''$ without specifying any relationship between t and t'. Thus, the complete pseudo-scanner conflict $\{(\lambda, t), (\lambda, t'), (\lambda, t'')\}$ is resolved without resolving the pairwise pseudo-scanner conflict $\{(\lambda, t), (\lambda, t')\}$.

Our PSLR(1) generator reports every complete pseudo-scanner conflict that the user has not explicitly resolved with `%lex-prec`. Moreover, once all complete pseudo-scanner conflicts are resolved, our generator reports any *useless* lexical precedence rule the user has declared using `%lex-prec`. A useless lexical precedence rule is a lexical precedence rule that does not specify any pairwise pseudo-scanner conflict resolution that is employed in at least one complete pseudo-scanner conflict resolution. In this way, when a new complete pseudo-scanner conflict arises, rather than quietly resolving it and producing potentially undesirable behavior, our PSLR(1) generator alerts the user immediately.

For the same reasons, lexical precedence relations are not implicitly transitive. For example, if for identity conflicts the user declares that token t has lower precedence than token t' and declares that token t' has lower precedence than token t'', then the only identity conflicts these rules resolve are identity conflicts between t and t' and identity conflicts between t' and t''. These rules do not imply any relationship between t and t'', for which there might not yet be any identity conflict that needs to be resolved in order to resolve complete pseudo-scanner conflicts. If such an identity conflict between t and t'' does arise and there is no separate rule that resolves it, our generator reports to the user the complete pseudo-scanner conflicts in which the identity conflict appears.

We make one exception to the above principles. When a token has an unresolved pairwise pseudo-scanner conflict with itself, our generator resolves the conflict using longest match without warning the user. For the case of identity conflicts, the justification is simple. By Definition 2.2.5, a token cannot have an identity conflict with itself. For the case of autolength conflicts, the justification is threefold. First, by far the most common rule we have found useful in this case is the traditional rule, longest match. Second, in our experience, autolength conflicts are usually no surprise to the user. Autolength conflicts are often obvious from a repetition operator appearing in the token's regular expression. For example, in Figure 2.2a, the "+" in the identifier's regular expression is the source of the identifier's autolength conflicts. Third, there are often many tokens in a scanner specification that have autolength conflicts. Declaring lexical precedence rules for all of these tokens would be uselessly tedious. However, we have found a few cases where a rule of shortest match is useful for autolength conflicts, so longest match is simply the default rule.

```
%token-re ID ([a-zA-Z]+)
%lex-prec ID <∼ 'int'
```
(a)

```
%token-re OCTAL (0[0-7]+)
%lex-prec OCTAL -∼ '0'
```
(b)

```
%token-re NUM ([0-9])
%lex-prec NUM <- '0'
```
(c)

Figure 3.1: Traditional Lexical Precedence Rules. (a) Traditional rules involving token precedence and longest match can be specified together in a PSLR(1) specification. However, when only (b) length conflicts or (c) identity conflicts need to be resolved, then only the corresponding component of the `%lex-prec` operator should be specified.

Definition 3.2.2 (Default Lexical Autolength Precedence)

Given any two matches (λ, t) and (λ', t) for the same character sequence such that $|\lambda| < |\lambda'|$ and such that no lexical autolength precedence rule is declared for t, the relative precedence of these matches is $(\lambda, t) < (\lambda', t)$. $\square$

3.2.2 Traditional Rules

Consider again the Lex specification in Figure 2.2a. As we explained in section 2.2.1, the identifier has both an identity conflict and length conflicts with the keyword, and the identifier has autolength conflicts. We can rewrite the specification from Figure 2.2a as a PSLR(1) specification while specifying a traditional scanner's resolution of these conflicts. The new specification appears in Figure 3.1a. The "<∼" is an example of a `%lex-prec` operator. The first character in a `%lex-prec` operator always specifies how identity conflicts between the operands should be resolved, and the second character always specifies how length conflicts between the operands should be resolved. In this case, the "<" specifies that identity conflicts between the identifier and keyword should be resolved by selecting the keyword. The "∼" specifies that length conflicts between them should be resolved using longest match. However, if there exists no parser state that accepts both the identifier and the keyword, then these scanner conflicts are not pseudo-scanner conflicts according to Definition 3.2.1, and so there exists no complete pseudo-scanner conflict that these rules help to resolve. In that case, our PSLR(1) generator reports that both rules specified by this `%lex-prec` declaration are useless. Our generator quietly resolves the identifier's autolength conflicts using longest match.

The specification in Figure 3.1b contains two tokens, `'0'` and `OCTAL`. It assumes that no identity conflict between those tokens needs to be resolved, and it specifies that length conflicts between them should be resolved using longest match. The order of operands for "−∼" is irrelevant.

Assume there exists a parser state, s_p, that accepts both tokens and no other tokens. Thus, there exists a complete pseudo-scanner conflict that this rule helps to resolve so that the rule is useful. As this specification evolves, a user might naively change the "+" in the `OCTAL` token's regular expression to a "$\star$", creating an identity conflict with the `'0'` token. A traditional scanner generator would quietly resolve the new conflict based on the order in which the tokens happen to be declared, and so the generated scanner would then match the `OCTAL` token instead of the `'0'` token for the single character "`0`". Our PSLR(1) generator instead reports that there is now a complete pseudo-scanner conflict for "`0`" in s_p that is unresolved.

The specification in Figure 3.1c contains two tokens, `'0'` and `NUM`. It specifies that an identity conflict between those tokens should be resolved by selecting the `'0'` token, and it assumes that no length conflict between them needs to be resolved. Assume there exists a parser state, s_p, that accepts both tokens and no other tokens. Thus, there exists a complete pseudo-scanner conflict that this rule helps to resolve so that the rule is useful. As this specification evolves, a user might naively append a "+" to the `NUM` token's regular expression, creating length conflicts. A traditional scanner generator would quietly resolve all the new conflicts using longest match. Our PSLR(1) generator does the same for the `NUM` token's autolength conflicts. However, it reports that there are now unresolved complete pseudo-scanner conflicts involving the `NUM` and `'0'` tokens in s_p.

We now define the `%lex-prec` operators from this section formally.

Definition 3.2.3 (Traditional Lexical Precedence Operators)
Given any two matches (λ, t) and (λ', t') for the same character sequence, the lexical precedence operators "$<\sim$", "$<-$", and "$-\sim$" can specify the relative precedence of those matches as follows:

1. $(t <\sim t') \Leftrightarrow (t <- t') \wedge (t -\sim t')$.
2. $(t <- t') \Rightarrow (t \neq t')$.
3. $(t <- t') \wedge (\lambda = \lambda') \Rightarrow (\lambda, t) < (\lambda', t')$.
4. $(t -\sim t') \wedge (|\lambda| < |\lambda'|) \Rightarrow (\lambda, t) < (\lambda', t')$.
5. $(t -\sim t') \wedge (|\lambda'| < |\lambda|) \Rightarrow (\lambda', t') < (\lambda, t)$.

Notice that $(t -\sim t') \Leftrightarrow (t' -\sim t)$. $\square$

```
%token-re WORD ([a-zA-Z](-?[a-zA-Z])*)
%token-re NON  (non-)
%lex-prec WORD -< NON
%%
word
  :     WORD {$$ = new_word ($1);         }
  | NON WORD {$$ = new_negated_word ($2);}
  ;
```

(a)

```
%token-re WORD ([a-zA-Z](-?[a-zA-Z])*)
%token-re NON  (non-?)
%lex-prec WORD << NON
%%
word
  :     WORD {$$ = new_word ($1);         }
  | NON WORD {$$ = new_negated_word ($2);}
  ;
```

(b)

```
%token-re COM_START ("/*"([^*]|\*+[^*/])*\**)
%token-re COM       ({COM_START}"*/")
%lex-prec COM_START -< COM // or -~
```

(c)

```
%token-re COM_START ("/*"(.|\n)*)
%token-re COM       ({COM_START}"*/")
%lex-prec COM -s COM
%lex-prec COM_START << COM
```

(d)

Figure 3.2: Non-traditional Lexical Precedence Rules. (a) PSLR(1) supports declarations for resolving length conflicts by assigning precedence to tokens. (b) When there are identity conflicts as well, they must be resolved using the same precedence relationship. (c) When autolength conflicts are resolved using the rule of longest match, the syntax of C's multiline comment is difficult to express as a regular expression, but (d) the rule of shortest match makes it easier.

3.2.3 Non-traditional Rules

So far in this section, we have resolved all length conflicts using longest match. In other words, we have always assigned precedence to the longest lexeme. In contrast, we have always resolved identity conflicts by assigning precedence to tokens instead, and there are cases where it is useful to do the same for length conflicts. For example, consider the PSLR(1) specification in Figure 3.2a. According to this specification, the `WORD` token matches any series of letters such that any consecutive pair of letters may be separated by a single hyphen. The `NON` token matches only the character sequence "`non-`". There is no identity conflict between the `WORD` token and the `NON` token, but there are length conflicts between them. If these length conflicts were resolved using longest match, the first grammar production would match a character sequence like "`non-euclidean`" in its entirety. However, this specification says that these length conflicts should instead be resolved by selecting the `NON` token. Thus, the pseudo-scanner splits a character sequence like "`non-euclidean`" into two tokens, which are matched by the second grammar production. Because the grammar does not accept the `NON` token alone, it is a syntax error if the character sequence "`non-`" appears alone.

The specification in Figure 3.2b revises the specification in Figure 3.2a so that the pseudo-

scanner also extracts the prefix "non" and thus splits a character sequence like "nonacid" into two tokens, which are matched by the second grammar production. Notice that this change creates an identity conflict between the WORD and NON tokens, and we resolve it by selecting the NON token just as we do for length conflicts. By requiring the user to add an explicit lexical precedence rule for the new identity conflict rather than resolving it implicitly, the PSLR(1) generator motivates the user to consider the consequences of the new conflict and its resolution. Specifically, WORD and thus the first grammar production no longer match the character sequence "non", so it is now also a syntax error if the character sequence "non" appears alone.

The specification in Figure 3.2c defines the COM token for C's multiline comment, and it defines the COM_START token for such a comment that is not properly closed. Because the regular expression of COM starts with exactly the regular expression of COM_START, the regular expression of COM_START is referenced as "{COM_START}" to avoid repetition. Of course, there are length conflicts between these two tokens. Resolving them with longest match is equivalent to giving COM higher precedence because, for any given input character sequence, any match for COM is always longer than any match for COM_START.

Figure 3.2c's regular expressions are surprisingly complex given the simplicity of C's multiline comment syntax, and those regular expressions become even worse for any similar syntactic structure whose closing character sequence is longer than "*/". In contrast, the regular expressions appearing in Figure 3.2d are simple, and the closing character sequence can easily be extended. However, in the latter case, COM has autolength conflicts. If those autolength conflicts were resolved using the default rule of longest match, then COM would incorrectly match the following character sequence up to the last "*/":

```
/*com*/ str = "*/";
```

Instead, this specification explicitly resolves autolength conflicts for COM using the %lex-prec operator "-s", which specifies the rule of *shortest match*, also called *minimal munch*. Thus, COM matches only up until the first "*/", as desired. For the autolength conflicts of COM_START, the specification accepts the default rule of longest match because shortest match would cause it to always match exactly "/*" and nothing else. All conflicts between COM and COM_START are resolved by giving COM higher precedence regardless of which token has the shortest or longest match. For this reason, COM_START is guaranteed not to match beyond the first "*/" either.

We now formally define the `%lex-prec` operators from this section as well as the operator "`<s`".

Definition 3.2.4 (Non-traditional Lexical Precedence Operators)
Given any two matches (λ, t) and (λ', t') for the same character sequence, the lexical precedence operators "`<<`", "`-<`", "`<s`", and "`-s`" can specify the relative precedence of those matches as follows:

1. $(t{<<}t') \Leftrightarrow (t{<-}t') \wedge (t{-<}t')$.
2. $(t{-<}t') \Rightarrow (t \neq t')$.
3. $(t{-<}t') \wedge (\lambda \neq \lambda') \Rightarrow (\lambda, t) < (\lambda', t')$.
4. $(t{<}\mathtt{s}t') \Leftrightarrow (t{<-}t') \wedge (t{-}\mathtt{s}t')$.
5. $(t{-}\mathtt{s}t') \wedge (|\lambda| < |\lambda'|) \Rightarrow (\lambda', t') < (\lambda, t)$.
6. $(t{-}\mathtt{s}t') \wedge (|\lambda'| < |\lambda|) \Rightarrow (\lambda, t) < (\lambda', t')$.

Notice that $(t{-}\mathtt{s}t') \Leftrightarrow (t'{-}\mathtt{s}t)$. □

We have introduced our non-traditional lexical precedence operators in order to further minimize the ad-hoc coding required in scanner specifications. The full power of these operators becomes more apparent in section 3.6, where we introduce a mechanism to handle whitespace and comments, and in section 3.7, where we introduce scoped declarations.

3.2.4 Ambiguities

As we demonstrated in the previous sections, the lexical precedence rules for a given PSLR(1) specification consist of (1) all rules the user specifies in `%lex-prec` declarations and (2) the rule of longest match for autolength conflicts that are not resolved by the user's `%lex-prec` declarations. In this section, we introduce a formal model to reveal ambiguity in the way the user can interpret these rules.

Definition 3.2.5 (Lexical Precedence Function)

Any set of lexical precedence rules R defines a ***lexical precedence function***, Δ, such that, given any set of matches M for some character sequence, then:

1. Iff there exists some $m \in M$ such that $\forall m' \in M : m \neq m', m' < m$ according to R, then $\Delta(M) = m$.
2. Iff there exists no such m, then $\Delta(M) =$ undefined.

□

Definition 3.2.6 (Set Ordering)

Given a sequence σ and a set S, then σ is an *ordering* of S iff $|\sigma| = |S|$ and $\forall s \in S, s \in \sigma$. □

Definition 3.2.7 (Sequential Lexical Precedence Function)

Any lexical precedence function Δ defines a ***sequential lexical precedence function***, $\mathscr{F}$, such that, given any sequence of matches μ for some character sequence, there exists a sequence $\rho : |\rho| = |\mu| \wedge \rho[1] = \mu[1] \wedge \rho[|\rho|] = \mathscr{F}(\mu) \wedge \forall i : 1 < i \leq |\rho|, \rho[i] = \Delta(\{\rho[i-1], \mu[i]\})$. □

We have introduced the concept of a sequential lexical precedence function to model how we assume that users will intuitively interpret lexical precedence rules. For example, let's say the user has written the specification in Figure 3.1a and wishes to determine what match the pseudo-scanner should select for the character sequence "`integer`" when the parser is in a state that accepts exactly the tokens `ID` and `'int'`. To do so, the user must compare all matches from $\mathscr{M}$("`integer`", {ID,`'int'`}) using the lexical precedence rules from his specification in order to determine the highest precedence match. To perform all these comparisons, we assume that the user will arbitrarily choose a particular ordering of the matches, μ, and then simply compare the matches sequentially so that he can finish in linear time. The result of such a sequential comparison is $\mathscr{F}(\mu)$ such that $\mathscr{F}$ is the sequential lexical precedence function defined by the specification's lexical precedence rules. However, there are many possible orderings of the matches to choose from. The user might, for example, choose the ordering μ shown in Table 3.1a. According to Definition 3.2.7, $\mathscr{F}$ computes each element in ρ in order by applying the specification's lexical precedence function, Δ, to the previous element in ρ and the current element in μ. In other words, the current element in

	μ	rule	ρ
1	(“i”, ID)		(“i”, ID)
2	(“in”, ID)	ID −∼ ID	(“in”, ID)
3	(“int”, ID)	ID −∼ ID	(“int”, ID)
4	(“int”, ’int’)	ID <- ’int’	(“int”, ’int’)
5	(“inte”, ID)	ID −∼ ’int’	(“inte”, ID)
6	(“integ”, ID)	ID −∼ ID	(“integ”, ID)
7	(“intege”, ID)	ID −∼ ID	(“intege”, ID)
8	(“integer”, ID)	ID −∼ ID	(“integer”, ID)

(a)

	μ'	rule	ρ
1	(“i”, ID)		(“i”, ID)
2	(“in”, ID)	ID −∼ ID	(“in”, ID)
3	(“int”, ID)	ID −∼ ID	(“int”, ID)
4	(“inte”, ID)	ID −∼ ID	(“inte”, ID)
5	(“integ”, ID)	ID −∼ ID	(“integ”, ID)
6	(“intege”, ID)	ID −∼ ID	(“intege”, ID)
7	(“integer”, ID)	ID −∼ ID	(“integer”, ID)
8	(“int”, ’int’)	ID −∼ ’int’	(“integer”, ID)

(b)

Table 3.1: Unambiguous Sequential Lexical Precedence Function. For the specification in Figure 3.1a and the character sequence “integer”, the order in which matches are compared does not affect the highest precedence match.

ρ is the highest precedence match found so far. The highest precedence match found after iterating the entire sequence is then the last element of ρ, which is $\mathscr{F}(\mu) = (\text{“integer”}, \texttt{ID})$.

Because $\mathscr{F}$ can be applied to any ordering of $\mathscr{M}(\text{“integer”}, \{ID, \texttt{'int'}\})$, it is worthwhile to explore how different orderings affect the highest precedence match. For example, we adjust μ slightly to produce μ' shown in Table 3.1b. Notice that $\mathscr{F}(\mu) = \mathscr{F}(\mu')$. Moreover, for any other sequence μ'' that is an ordering of $\mathscr{M}(\text{“integer”}, \{ID, \texttt{'int'}\})$, $\mathscr{F}(\mu) = \mathscr{F}(\mu'')$ also. In this case, we say that $\mathscr{F}$ is ***unambiguous*** for $\mathscr{M}(\text{“integer”}, \{ID, \texttt{'int'}\})$.

Definition 3.2.8 (Ambiguous vs Unambiguous)

Given a set of matches M for some character sequence, then a sequential lexical precedence function $\mathscr{F}$ is ***ambiguous*** for M iff there exists a pair of sequences μ and μ' such that each of μ and μ' is an ordering of M, $\mathscr{F}(\mu)$ and $\mathscr{F}(\mu')$ are defined, and $\mathscr{F}(\mu) \neq \mathscr{F}(\mu')$. Otherwise, $\mathscr{F}$ is ***unambiguous*** for M. □

If we could guarantee in general that the sequential lexical precedence functions defined by PSLR(1) specifications are always unambiguous for all complete pseudo-scanner conflicts, then the user could compare matches in linear time in any order he wished as he considered the effect of his lexical precedence declarations. Because the user wouldn't have to consider alternate orderings, the task of writing, modifying, and understanding PSLR(1) specifications would be simplified.

The traditional lexical precedence rules from Definition 2.2.6 always define unambiguous sequential lexical precedence functions. However, the precedence rules specified using `%lex-prec` operators sometimes do not. We have identified three properties of the `%lex-prec` operators that cause this difference:

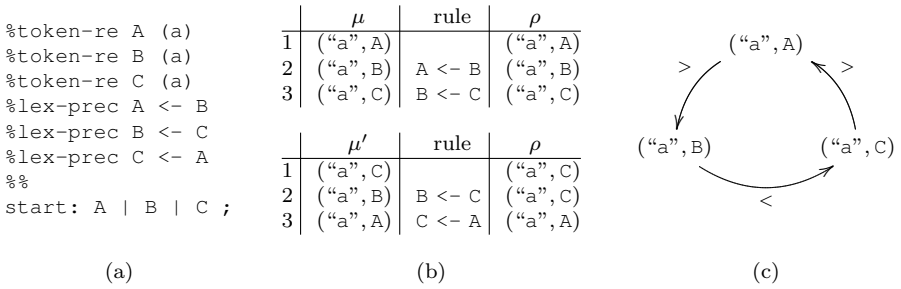


Figure 3.3: Intransitive Lexical Precedence. (a) Specifying an intransitive lexical precedence relation can define ambiguous sequential lexical precedence functions. (b) For the start state of the parser and the character sequence “a”, the order in which matches are compared affects the highest precedence match. (c) The trouble is that a cycle exists in the precedence relation over the matches.

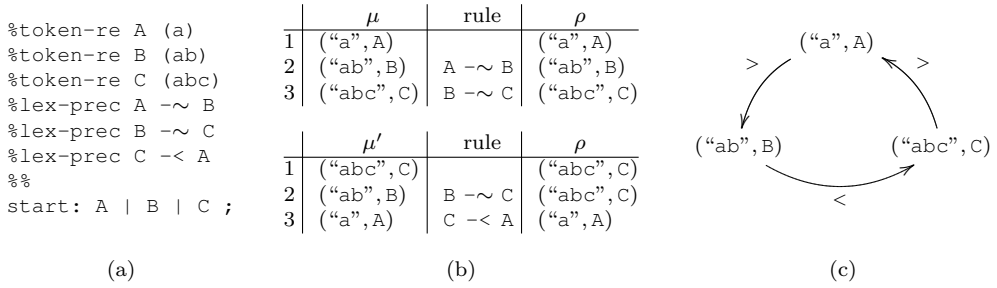


Figure 3.4: Token Precedence Mixed with Lexeme Precedence. (a) Mixing precedence of tokens with precedence of lexemes for resolving length conflicts can define ambiguous sequential lexical precedence functions. (b) For the start state of the parser and the character sequence “abc”, the order in which matches are compared affects the highest precedence match. (c) The trouble is that a cycle exists in the precedence relation over the matches.

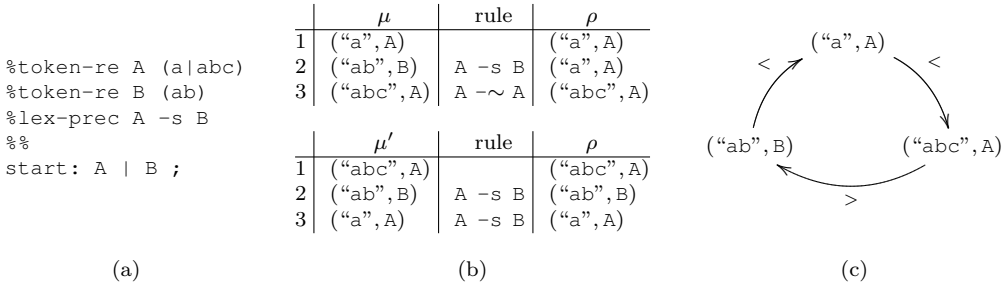


Figure 3.5: Shortest Match Mixed with Longest Match. (a) Mixing the rule of shortest match with the rule of longest match (implicit for autolength conflicts in this case) can define ambiguous sequential lexical precedence functions. (b) For the start state of the parser and the character sequence “abc”, the order in which matches are compared affects the highest precedence match. (c) The trouble is that a cycle exists in the precedence relation over the matches.

---

1. The `%lex-prec` operators do not require lexical precedence relations to be transitive.

2. The `%lex-prec` operators permit precedence of tokens to be mixed with precedence of lexemes for resolving length conflicts.

3. The `%lex-prec` operators permit the rule of shortest match to be mixed with the rule of longest match for resolving length conflicts.

For example, Figures 3.3a, 3.4a, and 3.5a present PSLR(1) specifications revealing the first, second, and third property, respectively. In each example, the start state of the parser accepts all tokens from the specification. Each of Figures 3.3b, 3.4b, and 3.5b shows two orderings of the matches for an example character sequence while the parser is in the start state such that the highest precedence match differs for the two orderings. Each of Figures 3.3c, 3.4c, and 3.5c reveals a cycle that exists in the precedence relation over those matches. Because of the third property of the `%lex-prec` operators, we can also create such a cycle by changing the "`<<`" operator to "`<s`" or "`<~`" in Figure 3.2d. In general, such a cycle is the cause of an ambiguous sequential lexical precedence function.

As we explained in section 3.2.1, to fully define the pseudo-scanner's behavior, it is not always necessary that a PSLR(1) specification's lexical precedence rules resolve all pairwise pseudo-scanner conflicts. In that case, for some match orderings, it is possible for Δ and thus $\mathscr{F}$ to return undefined. To determine a highest precedence match in spite of this result, we assume that the user will simply alter the match ordering so that $\mathscr{F}$ does not return undefined. For example, let the conflict C be the set of matches $\{m, m', m'', m'''\}$ and let the only precedence relationships defined for those matches be $m < m''$, $m'' < m'$, $m' < m'''$, and $m''' < m$. If the user first tries the match ordering $\mu = (m, m', m'', m''')$, then he immediately encounters an undefined result for $\Delta(\{m, m'\})$. We assume that the user will simply look for the first match with which m does have a relationship, he will find m'', and then he will continue his comparisons from there. Thus, he converts the ordering μ to the ordering $\mu' = (m, m'', m', m''')$, for which no undefined result is encountered, and $\mathscr{F}(\mu') = m'''$. However, given the ordering $\mu'' = (m''', m, m'', m')$, then $\mathscr{F}(\mu'') = m'$, so $\mathscr{F}$ is actually ambiguous for C, and C is unresolved. Thus, even when there are unresolved pairwise pseudo-scanner conflicts, it is still sometimes possible to compare matches sequentially, and ambiguity can still result.

In some cases, $\mathscr{F}$ returns undefined for every ordering of a set of matches. For example, let the conflict C be the set of matches $\{m, m', m''\}$ and let the only precedence relationships defined for those matches be $m < m'$ and $m < m''$. For every possible ordering μ of C, $\mathscr{F}(\mu) =$ undefined.

In this case, $\mathscr{F}$ is not ambiguous for C according to Definition 3.2.8 even though C is unresolved. The reason we don't include this case in our definition of an ambiguous sequential lexical precedence function is that, when the user compares matches sequentially in this case, he can never derive a highest precedence match, and so he will never be led to the erroneous conclusion that C is resolved.

The ambiguities permitted by the `%lex-prec` operators must be addressed in order for the operators' meaning to be defined in all cases. One solution would be to design an algorithm that selects a single ordering of any set of matches and thus quietly resolves all such ambiguities in order to compute the highest precedence match in linear time. However, this solution would burden the user with discovering these ambiguities and understanding their implicit resolution. Instead, we have designed an algorithm, which we describe in section 3.2.7, that can detect and report to the user whether a PSLR(1) specification defines an ambiguous sequential lexical precedence function for any complete pseudo-scanner conflict. However, the algorithm does not distinguish between ambiguous sequential lexical precedence functions and other unresolved conflicts. Instead, it actually reports any complete pseudo-scanner conflict for which the lexical precedence rules do not define a highest precedence match uniquely. In other words, our algorithm employs the lexical precedence function, Δ, from Definition 3.2.5 rather than sequential lexical precedence function, $\mathscr{F}$, from Definition 3.2.7 for resolving complete pseudo-scanner conflicts.

Definition 3.2.9 (Resolved Scanner Conflict)
Given the lexical precedence function Δ defined by a set of lexical precedence rules R and given a scanner conflict C, then C is resolved by R iff $\Delta(C) \neq$ undefined. $\square$

As long as our generator reports that all complete pseudo-scanner conflicts are resolved and thus guarantees that the sequential lexical precedence function defined by the PSLR(1) specification is unambiguous for all such conflicts, then the user can select any ordering of the matches in such a conflict and compare them sequentially to correctly determine the highest precedence match in linear time.

3.2.5 Self-consistency

As we demonstrated in the previous section, `%lex-prec` operators can be combined in a PSLR(1) specification in such a way that ambiguous sequential lexical precedence functions are defined. However, for most of our `%lex-prec` operators, a single use of one of the operators alone

can never do so. We say that each such operator is ***always self-consistent***.

Definition 3.2.10 (Self-consistent vs Self-contradictory)

Given:

1. A lexical precedence operator $\odot$.
2. A character sequence $\xi \in \Xi^*$.
3. A pair of tokens t and t'.

Let R be the following set of lexical precedence rules:

1. $t \odot t'$.
2. The default rule of longest match for autolength conflicts.

The operator $\odot$ is ***self-consistent*** for ξ over $\{t, t'\}$ iff the sequential lexical precedence function that is defined by R is unambiguous for $\mathscr{M}(\xi, \{t, t'\})$. Otherwise, $\odot$ is ***self-contradictory*** for ξ over $\{t, t'\}$. Iff $\odot$ is self-consistent for every possible ξ and $\{t, t'\}$ described in the above given, then we say $\odot$ is ***always self-consistent***. □

Theorem 3.2.11 (Self-consistent Lexical Precedence Operators)

The following lexical precedence operators are always self-consistent:

1. "<∼"
2. "<–"
3. "–∼"
4. "<<"
5. "–<"
6. "–s" when the operands are the same token.

□

Theorem 3.2.11 is straightforward to prove informally. We refer to the sequential lexical precedence function defined by the set of lexical precedence rules R as $\mathscr{F}$. First, consider the case where $\odot$ is "<∼". We can be sure that the lexeme for the highest precedence match computed by $\mathscr{F}$ is the longest lexeme regardless of the ordering of matches because the rule of longest match resolves all length conflicts. If both t and t' match the longest lexeme, then the token for the highest precedence match computed by $\mathscr{F}$ is t' regardless of the ordering of matches because t' is chosen for all identity conflicts. If only one of the tokens matches the longest lexeme, then that token is the token for the highest precedence match computed by $\mathscr{F}$ regardless of the ordering of matches.

Second, consider the case where $\odot$ is "<<". If there is any match for t', then the longest match for t' is the highest precedence match computed by $\mathscr{F}$ regardless of the ordering of matches because t' is chosen in any pairwise conflict between t and t'. If there is no match for t', then the longest match for t is the highest precedence match computed by $\mathscr{F}$ regardless of the ordering of matches. Third, consider the case where $\odot$ is "<–", "–∼", or "–<". Each of these operators is exactly the same as either "<∼" or "<<" except that either identity or length conflicts are left unresolved. In general, the only effect that resolving less pairwise conflicts can have on $\mathscr{F}$ for some ordering of matches is to cause $\mathscr{F}$ to return an undefined result, and an undefined result does not create an ambiguity according to Definition 3.2.8. Finally, consider the case where $\odot$ is "–s" and $t = t'$. For the highest precedence match computed by $\mathscr{F}$, there is only one possible token, and the lexeme is the shortest lexeme regardless of the ordering of matches. Thus, in all cases for all of these operators, the ordering of matches does not affect the highest precedence match computed by $\mathscr{F}$, and so the operators are always self-consistent.

Consider again the example in Figure 3.5 in terms of Definition 3.2.10. That is, $\odot$ is "–s", ξ is "abc", t is A, t' is B, and all pairwise scanner conflicts for "abc" over {A,B} are resolved by A –s B plus the default rule of longest match for autolength conflicts. As demonstrated in Figure 3.5b, the sequential lexical precedence function that is defined by A –s B plus the default rule of longest match for autolength conflicts is ambiguous for $\mathscr{M}$("abc", {A,B}), so the operator "–s" is not always self-consistent. If we replace "–s" with "<s", we have an example demonstrating that the operator "<s" is not always self-consistent either. To make the latter example more realistic, we can create identity conflicts by changing the regular expression for B to "a|ab". In general, the trouble is that a shortest match rule for length conflicts between different tokens sometimes contradicts the default longest match rule for either token's autolength conflicts.

Consider again the specification in Figure 3.2b. Recall that, when the input character sequence consists only of "non", the pseudo-scanner returns the token NON alone, which the grammar does not accept. It may be tempting to reverse the precedence for just the identity conflict between the tokens WORD and NON so that the pseudo-scanner instead returns the token WORD, which is accepted by the first grammar production. Thus, the %lex-prec operator "<<" would be replaced by ">" in the specification. However, consider what match the sequential lexical precedence function, $\mathscr{F}$, then computes for the character sequence "nonacid". Using μ from Table 3.2a as the ordering of matches, the highest precedence match computed by $\mathscr{F}$ is ("non", NON). Thus, the prefix "non"

(a)

	μ	rule	ρ
1	(“n”, WORD)		(“n”, WORD)
2	(“no”, WORD)	WORD -∼ WORD	(“no”, WORD)
3	(“non”, WORD)	WORD -∼ WORD	(“non”, WORD)
4	(“nona”, WORD)	WORD -∼ WORD	(“nona”, WORD)
5	(“nonac”, WORD)	WORD -∼ WORD	(“nonac”, WORD)
6	(“nonaci”, WORD)	WORD -∼ WORD	(“nonaci”, WORD)
7	(“nonacid”, WORD)	WORD -∼ WORD	(“nonacid”, WORD)
8	(“non”, NON)	WORD -< NON	(“non”, NON)

(b)

	μ'	rule	ρ
1	(“n”, WORD)		(“n”, WORD)
2	(“no”, WORD)	WORD -∼ WORD	(“no”, WORD)
3	(“non”, WORD)	WORD -∼ WORD	(“non”, WORD)
4	(“non”, NON)	WORD >- NON	(“non”, WORD)
5	(“nona”, WORD)	WORD -∼ WORD	(“nona”, WORD)
6	(“nonac”, WORD)	WORD -∼ WORD	(“nonac”, WORD)
7	(“nonaci”, WORD)	WORD -∼ WORD	(“nonaci”, WORD)
8	(“nonacid”, WORD)	WORD -∼ WORD	(“nonacid”, WORD)

Table 3.2: Opposing Lexical Precedence. If we replace the “<<” operator with “><” in the specification in Figure 3.2b, then, for the character sequence “nonacid”, the ordering of matches affects the highest precedence match computed by the sequential lexical precedence function.

---

is extracted as we expected in section 3.2.3. However, using μ' from Table 3.2b as the ordering of matches, the highest precedence match computed by $\mathscr{F}$ is (“nonacid”, WORD). By Definition 3.2.10 then, the “><” operator is self-contradictory for “nonacid” over {WORD, NON}. In general when “><” is self-contradictory, the highest precedence match computed by $\mathscr{F}$ depends on whether the last pair of matches compared between the two tokens is a length conflict, as in Table 3.2a, or an identity conflict, as in Table 3.2b. In this way, the opposing precedence within the operator is the cause of its self-contradictory nature.

Even though “><” may seem like an obvious extension of the other %lex-prec operators we have implemented, we have concluded that, because of its self-contradictory nature and because we have found no practical use for it, it is not worth the confusion it would likely cause the user. For the same reasons, we had originally planned in general not to implement any lexical precedence operator that is not always self-consistent. However, after realizing that, as demonstrated by Figures 3.2c and 3.2d, the operator “-s” can be quite useful for autolength conflicts, for which it is always self-consistent according to Theorem 3.2.11, we decided to permit the operators “-s” and “<s” for length conflicts between different tokens, for which they are not always self-consistent. It will be interesting to see whether the always self-consistent property proves to be the litmus test for the usefulness of a lexical precedence operator.

3.2.6 Pseudo-scanner Tables

In this section, we describe the pseudo-scanner tables that our PSLR(1) generator constructs to encode the lexical behavior defined by the user's PSLR(1) specification. Let G be the context-free grammar from the specification such that $\mathscr{G}(G) = (V', T', P', S')$, and let R be the set of lexical precedence rules from the specification. Let Σ_p be the set of LR(1) parser states for G, and let s_{p0} be the index of the start state within Σ_p. If not all complete pseudo-scanner conflicts are resolved by R, then our generator reports an error and does not bother to construct the pseudo-scanner tables, so this section's definitions assume that all complete pseudo-scanner conflicts are resolved.

The first set of tables our PSLR(1) generator constructs for the pseudo-scanner encodes a single deterministic FSA that matches all tokens in T' against the regular expressions defined by the user's PSLR(1) specification. Let Σ_s be that FSA's set of states, and let s_{s0} be the index of the start state within Σ_s. To facilitate our explanations, we employ δ as a function that can examine the transitions recorded in any FSA state. That is, $\forall s_s : 1 \leq s_s \leq |\Sigma_s|, \forall s'_s : 1 \leq s'_s \leq |\Sigma_s|, \forall c \in \Xi$, the statement $\delta(\Sigma_s[s_s], c) = \Sigma_s[s'_s]$ holds iff there is a transition from state $\Sigma_s[s_s]$ on c to state $\Sigma_s[s'_s]$. Iff either (1) there is no transition from $\Sigma_s[s_s]$ on c or (2) the destination of that transition is the error state, then the statement $\nexists\delta(\Sigma_s[s_s], c)$ holds.

Σ_s is nearly appropriate as the FSA for a traditional scanner with only one start condition that recognizes all tokens in T'. Algorithms to construct such an FSA are well known and involve converting the regular expressions for all tokens in T' to a single non-deterministic FSA and then converting the non-deterministic FSA to a deterministic FSA [11]. The only difference is that our generator does not encode the resolution of identity conflicts in Σ_s. That is, our generator maintains a set of accepted tokens per accepting state rather than always reducing the set to the token declared earliest in the scanner specification. For example, assume the user's PSLR(1) specification is the specification from Figure 3.1a. If $\delta^*(\Sigma_s[s_{s0}],\text{“}\texttt{int}\text{”}) = \Sigma_s[s_s]$, then $\Sigma_s[s_s]$ is an accepting state for both the `ID` token and the `'int'` token. Thus, the identity conflict between `ID` and `'int'` remains. Our generator also does not encode the resolution of length conflicts in Σ_s, but neither does a traditional scanner generator. Continuing our example, if $\delta(\Sigma_s[s_s],\text{“}\texttt{s}\text{”}) = \Sigma_s[s'_s]$, then $\Sigma_s[s'_s]$ accepts the `ID` token. When the traditional scanner reaches $\Sigma_s[s'_s]$, it forgets that $\Sigma_s[s_s]$ is an accepting state because it always resolves length conflicts using longest match, but the pseudo-scanner might decide that a match represented by $\Sigma_s[s_s]$ has higher precedence than the match

represented by $\Sigma_s[s'_s]$. Also as in a traditional scanner, Σ_s does not encode the basic pseudo-scanner behavior defined in Definition 3.1.1. That is, regardless of which tokens are accepted by the current parser state, $\Sigma_s[s_s]$ accepts `ID` and `'int'`, and $\Sigma_s[s'_s]$ accepts `ID`.

Definition 3.2.12 (Accepts for Scanner)
Given the LR(1) parser state s_p and the scanner FSA state s_s, we extend Definition 2.2.12 to overload *acc* as follows:

1. $acc(s_s)$ is the set of tokens accepted by s_s.
2. $acc(s_p, s_s) = acc(s_p) \cap acc(s_s)$.

□

Rather than encoding conflict resolution and the basic pseudo-scanner behavior directly in Σ_s, our generator encodes them in separate tables, which we now define. Because these tables encode which tokens should be accepted by which states in Σ_s, our generator discards the sets of accepted tokens in Σ_s after constructing these tables.

Definition 3.2.13 (*state_to_accepting_state*)
$\forall s_s : 1 \leq s_s \leq |\Sigma_s|, state\_to\_accepting\_state[s_s]$ is either:

1. Undefined iff $acc(\Sigma_s[s_s]) = \emptyset$.
2. The index that $\Sigma_s[s_s]$ would have in a Σ'_s that is formed by copying Σ_s and then, $\forall s'_s : 1 \leq s'_s \leq |\Sigma_s| \wedge acc(\Sigma_s[s'_s]) = \emptyset$, removing $\Sigma_s[s'_s]$.

□

The purpose of *state_to_accepting_state* is to reduce the size of tables like *scanner_accepts*, which we now define.

Definition 3.2.14 (*scanner_accepts*)
$\forall s_p : 1 \leq s_p \leq |\Sigma_p|, \forall s_s : 1 \leq s_s \leq |\Sigma_s|$, when the parser is in state $\Sigma_p[s_p]$ and the pseudo-scanner is in state $\Sigma_s[s_s]$:

1. If the pseudo-scanner should not recognize a match for any token, then either:

(a) $state\_to\_accepting\_state[s_s]$ = undefined.

(b) $scanner\_accepts[s_p][state\_to\_accepting\_state[s_s]]$ = undefined.

2. Otherwise, $scanner\_accepts[s_p][state\_to\_accepting\_state[s_s]]$ is the token for which the pseudo-scanner should recognize a match.

□

scanner_accepts resolves all pseudo-scanner identity conflicts by selecting a single token per Σ_s accepting state, which corresponds to a column in *scanner_accepts*. However, because of the basic pseudo-scanner behavior, it does so per parser state, which corresponds to a row in *scanner_accepts*. Sometimes an accepting state in Σ_s does not need an accepted token to be specified for a particular parser state either because no token is accepted by both it and the parser state or because there is always a shorter match that has higher precedence. In these cases, *scanner_accepts* leaves the accepted token undefined.

Definition 3.2.15 (*length_precedences*)
$\forall t \in T', \forall t' \in T', length\_precedences[t][t'] =$ true iff $(t\text{-}<t') \in R \vee (t\text{-}\sim t') \in R$. □

While scanning the input, the pseudo-scanner uses *length_precedences* to determine how the user has specified that length conflicts be resolved. In other words, $length\_precedences[t][t'] =$ true iff matches for t have lower precedence than longer matches for t' according to R. If R has no rule for lexical length precedence between t and t', then either there are no complete pseudo-scanner conflicts whose resolution requires the resolution of those length conflicts, or our generator reports an error. In either case, the value of $length\_precedences[t][t']$ is irrelevant.

Definition 3.2.15 is the first place we have formally modeled R as a set, so we now clarify a couple of points about the contents of R. By Definition 3.2.3, $(t\text{-}\sim t') \Leftrightarrow (t'\text{-}\sim t)$. Thus, we assume that $(t\text{-}\sim t') \in R \Leftrightarrow (t'\text{-}\sim t) \in R$. Likewise, by Definition 3.2.4, $(t\text{-}\mathtt{s}t') \Leftrightarrow (t'\text{-}\mathtt{s}t)$, so we also assume that $(t\text{-}\mathtt{s}t') \in R \Leftrightarrow (t'\text{-}\mathtt{s}t) \in R$. The first point affects Definition 3.2.15, and both points affect definitions in the next section.

Given the above tables, the pseudo-scanner algorithm is straight-forward. Let ξ be the portion of the input character sequence that has not yet been tokenized. We call the pseudo-scanner function *pseudo_scan*. The parser passes the index of the current parser state as an argument to

pseudo_scan, and *pseudo_scan* either returns the token it selects from all the tokens that match ξ, returns undefined if no token matches ξ, or returns # if ξ is empty.

Definition 3.2.16 (*pseudo_scan*(s_p))

```
1  if (|ξ| = 0) do:
2  ┗ return #.
3  let i_best = 1.
4  let t_best = undefined.
5  let s_s = s_s0.
6  for (let i = 1; i ≤ |ξ| ∧ ∃δ(Σ_s[s_s], ξ[i]); i = i + 1) do:
7  ┣ let s'_s : Σ_s[s'_s] = δ(Σ_s[s_s], ξ[i]).
8  ┣ set s_s = s'_s.
9  ┣ let s_a = state_to_accepting_state[s_s].
10 ┗ if (s_a ≠ undefined) do:
11    ┣ let t = scanner_accepts[s_p][s_a].
12    ┗ if (t ≠ undefined ∧ (t_best = undefined ∨ length_precedences[t_best][t])) do:
13       ┣ set i_best = i + 1.
14       ┗ set t_best = t.
15 set ξ = ξ[i_best..|ξ|].
16 return t_best. □
```

As we discussed earlier, algorithms to construct Σ_s are well know. Converting our definitions of the *state_to_accepting_states* and *length_precedences* tables to algorithms that construct those tables is straight-forward. Thus, we do not describe such algorithms in this paper. However, our algorithm to construct *scanner_accepts* is more complex, and we describe it in the next section. Later in this chapter, we revise our definitions of *scanner_accepts*, *length_precedences*, and *pseudo_scan* to implement features such as error handling.

3.2.7 Resolver Algorithm

Discovering, resolving, and reporting parser conflicts in LR(1) parser tables is relatively straight-forward because there is a finite number of states, a finite number of actions that can conflict per state, and thus a finite number of conflicts. For a traditional scanner, the number of states is finite also, but scanner length conflicts span multiple states across state transitions, and transition loops can exist. Thus, the number of complete scanner conflicts and the number of matches per complete scanner conflict can be infinite. Nevertheless, the traditional lexical precedence rules are straight-forward to implement because all complete scanner conflicts are resolved by resolving every pairwise scanner conflict in isolation as described in the previous section. Moreover, because

all scanner conflicts are resolved implicitly, the scanner generator never reports any of them.

For PSLR(1), the number of complete pseudo-scanner conflicts and the number of matches per complete pseudo-scanner conflict can be infinite for the same reasons as for complete scanner conflicts for a traditional scanner, but it is not sufficient to resolve pairwise pseudo-scanner conflicts in isolation. Instead, because of the complexity of PSLR(1)'s lexical precedence rules, the PSLR(1) generator must examine a complete pseudo-scanner conflict as a whole to determine if and how it is resolved, and the generator must report it to the user if it is unresolved. The only way to devise an algorithm that terminates but still manages to handle every one of the potentially infinite number of complete pseudo-scanner conflicts is to divide those conflicts into a finite number of categories such that all conflicts in a category can be discovered, resolved, and reported in the same manner. Our primary mechanism for categorizing complete pseudo-scanner conflicts is the ***scanner conflict profile***.

Definition 3.2.17 (Scanner Conflict Profile)

Given:

1. A set of lexical precedence rules R.
2. A set of tokens T.
3. A character sequence $\xi \in \Xi^*$.

let $\xi_s = \xi[1..|\xi| - 1]$, which is the empty string if $|\xi| = 1$. The ***scanner conflict profile*** for ξ over T in the context of R is then the tuple (T_s, t_s, T_ℓ) such that:

1. $T_s = \{t : \exists\lambda : (\lambda, t) \in \mathscr{M}(\xi_s, T)\}$, which might be $\emptyset$. Thus, we say that T_s is the set of all tokens matching the *shorter* lexemes.
2. Either:
   (a) $\exists m_s \in \mathscr{M}(\xi_s, T) : \forall m \in \mathscr{M}(\xi_s, T) : m \neq m_s, m < m_s$ according to R. In this case, t_s is such that $\exists\lambda_s : m_s = (\lambda_s, t_s)$. Thus, we say that t_s is the token with the highest precedence match for the shorter lexemes.
   (b) $t_s =$ undefined iff no such m_s exists.
3. $T_\ell = \{t : (\xi, t) \in \mathscr{M}(\xi, T)\}$, which might be $\emptyset$. Thus, we say that T_ℓ is the set of all tokens matching the *longest* lexeme.

□

Definition 3.2.17 only makes sense for a complete scanner conflict because it considers all matches for a character sequence, ξ, over a set of tokens, T. However, there is no restriction on T, so a scanner conflict profile can be computed for any complete scanner conflict including a complete pseudo-scanner conflict. Our PSLR(1) generator is only concerned with complete pseudo-scanner conflicts, and it examines each parser state in isolation, so T is always the set of tokens accepted by a particular parser state. As a result, $|T|$ is finite, the number of possible sets $T_s \subseteq T$ is finite, the number of possible tokens $t_s \in T_s$ is finite, and the number of possible sets $T_\ell \subseteq T$ is finite. Thus, the number of possible combinations for T_s, t_s, and T_ℓ to form a scanner conflict profile (T_s, t_s, T_ℓ) must be finite. That is, for any given PSLR(1) specification, scanner conflict profiles divide complete pseudo-scanner conflicts into a finite number of categories as desired.

We now explore whether every complete pseudo-scanner conflict with the same profile can be discovered, resolved, and reported in the same manner. The resolution of conflicts is the key to discovering and reporting them, so we consider resolution first.

Observation 3.2.18 (Resolution for Scanner Conflict Profile)
Given a scanner conflict profile (T_s, t_s, T_ℓ) in the context of the set of lexical precedence rules R, then:

1. If $t_s \neq$ undefined and if, $\forall t \in T_\ell, (t \mathrel{-\!\!\prec} t_s) \in R \vee (t \mathrel{-\mathtt{s}} t_s) \in R$, then the highest precedence match according to R is the same as the highest precedence match when the last character is removed from the input character sequence. In Definition 3.2.17, this match is called m_s. Its token is t_s.

2. Otherwise, if $\exists t_\ell \in T_\ell : (\forall t \in T_\ell : t \neq t_\ell, (t \mathrel{\prec\!\!-} t_\ell) \in R) \wedge (\forall t \in T_s, (t \mathrel{-\!\!\prec} t_\ell) \in R \vee (t \mathrel{-\!\!\sim} t_\ell) \in R)$, then $\exists \lambda_\ell : (\lambda_\ell, t_\ell)$ is the highest precedence match. λ_ℓ is always the entire input character sequence.

3. Otherwise, the conflict is unresolved.

□

The justification for Observation 3.2.18's algorithm is straight-forward. As in Definition 3.2.17, let ξ be the input character sequence, and let T be the set of tokens such that the complete

scanner conflict is $\mathscr{M}(\xi, T)$. In both cases where the algorithm claims the conflict is resolved, the algorithm defines the highest precedence match as the match $m \in \mathscr{M}(\xi, T) : \forall m' \in \mathscr{M}(\xi, T) : m \neq m', m' < m$ according to R. The algorithm claims the conflict is unresolved iff there is no such m. Thus, by Definitions 3.2.9 and 3.2.5, the algorithm correctly resolves complete scanner conflicts.

By resolving conflicts based on their profiles, Observation 3.2.18's algorithm immediately yields a strategy for reporting unresolved conflicts. That is, the algorithm reveals that the profile for a conflict contains enough information to determine whether the PSLR(1) specification's lexical precedence rules, R, resolve the conflict. Thus, to describe all deficiencies in R to the user so that he can then adjust R to resolve any remaining unresolved conflicts, we assert that it is sufficient for the PSLR(1) generator to report one unresolved conflict per profile. With this strategy, the generator's conflict report for a PSLR(1) specification with an infinite number of conflicts can remain comprehensive and yet finite.

Observation 3.2.18 is only a vague outline of how to resolve conflicts. There are several points we must address in order to develop it into a concrete algorithm. First, Observation 3.2.18's algorithm does not provide an obvious way to actually discover conflicts so that the generator can even consider resolving and reporting them. Second, even though a conflict's profile is enough to determine whether the conflict is resolved, which token belongs to the highest precedence match, and how to select the lexeme for the highest precedence match, the profile alone is not enough information to compute the actual lexeme for the highest precedence match. We must also know the input character sequence, ξ. Third, the algorithm is recursive. That is, in order to select the highest precedence match for ξ, we need to know the highest precedence match for the character sequence $\xi_s = \xi[1..|\xi| - 1]$. The derivation of a conflict's profile from the conflict is recursive in the same manner because T_s and t_s from Definition 3.2.17 are computed from ξ_s. To address all of these points, it seems that we need to examine every possible ξ for which there are matches while making sure to examine every prefix of a ξ before examining ξ. However, there are an infinite number of possible values for ξ, so we must divide them into a finite number of categories.

Every ξ for which there are matches corresponds to some state transition path from $\Sigma_s[s_{s0}]$ through Σ_s, but the same is not true for every ξ for which there are no matches. Thus, examining all such paths rather than all possible character sequences would avoid some character sequences for which there are no matches and thus no conflicts. A depth-first iteration of the states would examine all such paths while visiting prefixes of any ξ before ξ so that the algorithm can exploit the

recursive nature of Observation 3.2.18 and Definition 3.2.17. Moreover, this iteration would visit every accepting state in Σ_s so that, as we discussed in the previous section, the algorithm could construct *scanner_accepts*. However, the number of paths and the lengths of paths through Σ_s can still be infinite because of transition loops. We must divide this infinite number of paths into a finite number of categories, and we must choose a finite path for each category.

Our PSLR(1) generator's resolution algorithm for complete pseudo-scanner conflicts achieves the above goals by maintaining a scanner conflict profile map, $profile\_map$, during its depth-first iteration through the states of Σ_s. Each key in $profile\_map$ is a profile, and each value is the set of indices for the Σ_s states at which the corresponding profile has already been encountered. We model $profile\_map$ as an associative array. When it is cleared, the value for each possible key becomes $\emptyset$. We also assume the existence of a $report\_conflict$ function for reporting one complete pseudo-scanner conflict to the user.

Definition 3.2.19 (*compute_scanner_accepts*)

```
1  for (let s_p = 1; s_p ≤ |Σ_p|; s_p = s_p + 1) do:
2      for (let s_s = 1; s_s ≤ |Σ_s|; s_s = s_s + 1) do:
3          let s_a = state_to_accepting_state[s_s].
4          if (s_a ≠ undefined) do:
5              set scanner_accepts[sp][sa] = undefined.
6  for (let s_p = 1; s_p ≤ |Σ_p|; s_p = s_p + 1) do:
7      clear profile_map.
8      resolve(s_p, s_s0, ∅, undefined). □
```

Definition 3.2.20 ($resolve(s_p, s_s, T_s, t_s)$)

```
1  for (let i = 1; i ≤ |Ξ|; i = i + 1) do:
2      if (∃δ(Σ_s[s_s], Ξ[i])) do:
3          let s'_s : Σ_s[s'_s] = δ(Σ_s[s_s], Ξ[i]).
4          let T_ℓ = acc(Σ_p[s_p], Σ_s[s'_s]).
5          let entry = profile_map[(T_s, t_s, T_ℓ)].
6          if (s'_s ∉ entry) do:
7              let t_ℓ = undefined.
8              if (T_ℓ = ∅ ∨ (t_s ≠ undefined ∧ ∀t ∈ T_ℓ, (t-<t_s) ∈ R ∨ (t-st_s) ∈ R)) do:
9                  set t_ℓ = t_s.
10             else do:
11                 set t_ℓ : (t_ℓ ∈ T_ℓ) ∧ (∀t ∈ T_ℓ : t ≠ t_ℓ, (t<-t_ℓ) ∈ R)
12                                  ∧(∀t ∈ T_s, (t-<t_ℓ) ∈ R ∨ (t-∼ t_ℓ) ∈ R).
13                 if (t_ℓ ≠ undefined) do:
14                     set scanner_accepts[s_p][state_to_accepting_state[s'_s]] = t_ℓ.
15             else if (entry = ∅) do:
```

16 $\quad\quad\quad \llcorner\ \mathit{report\_conflict}(s_p, s'_s, T_s, t_s, T_\ell)$.

17 $\quad\quad$ set $\mathit{profile\_map}[(T_s, t_s, T_\ell)] = \mathit{entry} \cup \{s'_s\}$.

18 $\quad\quad \llcorner\ \mathit{resolve}(s_p, s'_s, \{T_s \cup T_\ell\}, t_\ell)$. □

The *compute_scanner_accepts* function in Definition 3.2.19 initializes *scanner_accepts* and, for each parser state, invokes the recursive *resolve* function to initiate the depth-first iteration starting at $\Sigma_s[s_{s0}]$ through Σ_s. The *resolve* function in Definition 3.2.20 implements that depth-first recursion, applies Observation 3.2.18 in order to resolve complete pseudo-scanner conflicts, computes *scanner_accepts* based on that resolution, and reports conflicts that are not resolved. On line 14 in Definition 3.2.20, notice that, when a new token is stored in *scanner_accepts*, the cell in which to store it is selected uniquely for each combination of s_p and s'_s. T_ℓ is also computed entirely from s_p and s'_s, and the new token to be stored is always the token with the highest lexical identity precedence in T_ℓ. In other words, there is one possible token for each cell in *scanner_accepts*, and once that token is stored, it is never changed. The main importance of this observation is that we do not have to worry that recording the resolution of one conflict can alter the recorded resolution of another conflict.

Throughout the depth-first iteration, *profile_map* has two purposes. The first purpose is achieved on line 6 in Definition 3.2.20, where *profile_map* is used to determine if, for the current parser state, the current pseudo-scanner state in the depth-first iteration has been visited previously with the same conflict profile. If so, then the iteration does not recurse any deeper. The justification is that, because the parser state, pseudo-scanner state, and profile are the same, the conflict resolution result would be the same, the arguments to the nested *resolve* invocation would be the same, and so all actions the algorithm would perform by continuing into the recursion have already been initiated. Moreover, the actions that were initiated last time the algorithm visited this parser state, pseudo-scanner state, and profile might not yet have completed when the current *resolve* invocation was encountered. That is, the last *resolve* invocation for this parser state, pseudo-scanner state, and profile might still be on the call stack. In that case, ending the recursion here helps to avoid an infinite loop. In general, we can be sure our algorithm always terminates because there are only a finite number of combinations of profiles and states. In this way, we achieve our goal of dividing a potentially infinite number of complete pseudo-scanner conflicts into a finite number of categories. That is, for each parser state, conflicts are categorized by profile plus the last pseudo-scanner state visited when the conflict is discovered.

The second purpose of $profile\_map$ is to enable us to report one unresolved conflict per profile. The condition on line 15 achieves this purpose. However, the algorithm actually reduces the conflict report further. The $T_\ell = \emptyset$ condition on line 8 guarantees no conflict is reported when there are no matches for the full character sequence being examined. The justification is simply that the complete conflict has no more matches than the complete conflict for the character sequence with one less character, which the algorithm just examined in the predecessor pseudo-scanner state. Thus, there is no new conflict to report. Thus, the only reason to bother storing a profile in $profile\_map$ when $T_\ell = \emptyset$ is to avoid repeating the recursion the next time this parser state, pseudo-scanner state, and profile are visited.

3.2.8 Summary

The basic behavior of a pseudo-scanner automatically eliminates many of the scanner conflicts among tokens from different sub-languages. Our PSLR(1) generator supports a lexical precedence directive to resolve remaining scanner conflicts in a careful, explicit, and declarative manner in order to further avoid the start conditions, other ad-hoc coding, and implicit conflict resolution of a traditional scanner generator like Lex. These features make our lexical precedence directive an important part of our unified scanner and parser specification language. As such, the directive will facilitate the modularization and arbitrary composition of sub-languages for the sake of modern extensible languages. However, as part of this effort, we have permitted lexical precedence rules that are more flexible but more complex than traditional lexical precedence rules. Determining which rules are useful enough in practice to be worth the complexity of the user's task of comprehending them is part of our ongoing exploration of practical applications of PSLR(1). Nevertheless, for any combination of these rules that the user specifies, we have devised an algorithm with which our PSLR(1) generator discovers the potentially infinite number of remaining scanner conflicts, resolves as many as possible according to the user's specified rules, and produces a comprehensive but finite report of unresolved conflicts in order to aid the user in further developing his PSLR(1) specification.

3.3 Lexical Ties

A pseudo-scanner, as specified in Definition 3.1.1, always selects a syntactically acceptable match based on left context. However, for some languages, the correct match is not always a

(a)
```
struct int;
```

(b)
```
do {} whiles;
```

(c)
```
void *i = &&j;
```

(d)
```
%token-re ID ([a-zA-Z_][a-zA-Z_0-9]*)
%symbol-set keywords
  'struct' 'int' 'do'
  'while' 'void'
%lex-tie ID keywords
%lex-prec ID <~ keywords
%lex-tie '&' '&&'
%lex-prec '&' -~ '&&'
```

Figure 3.6: Lexical Ties. By default, a pseudo-scanner always returns a syntactically acceptable token even though it may not be the correct token. In C, for example, (a) a keyword like `int` may be mistaken for an identifier, (b) a keyword like `while` may be broken off of an identifier like `whiles` or off of a similar keyword, and (c) an operator like `&` may be broken off of another operator like `&&`. (d) To avoid this problem, tokens that should not be confused but that match similar lexemes should be declared as lexically tied.

---

syntactically acceptable match. For example, in a language like C, all keywords are *reserved words*. Thus, even though keywords and identifiers match similar lexemes, a scanner for C should never mistake a keyword for an identifier, and it should always select the longest matching keyword or identifier even if the selected match is a syntax error in the current context. Figure 3.6 shows a few cases where a pseudo-scanner for C would thus make the wrong choice. For the character sequence "`int`" in Figure 3.6a, the match ("`int`",`ID`) is syntactically acceptable based on left context, but the correct match, ("`int`",`'int'`), is not. For the character sequence "`whiles`" in Figure 3.6b, the match ("`while`",`'while'`) is syntactically acceptable based on left context, leaving the trailing "`s`" for a subsequent scanner invocation, but the correct match, ("`whiles`",`ID`), is not. Operators with similar lexemes can cause the same trouble as identifiers and keywords. For example, for the character sequence "`&&`" in Figure 3.6c, the match ("`&`",`'&'`) is syntactically acceptable based on left context, leaving the trailing "`&`" for a subsequent scanner invocation, but the correct match, ("`&&`",`'&&'`), is not.[1]

In contrast, a scanner generated by a traditional scanner-generator tool like Lex usually handles the above examples correctly. When the user does not employ start conditions, the scanner always recognizes all tokens regardless of the current parser state. Thus, the scanner never ignores the correct keyword, identifier, or operator match simply because it is not syntactically acceptable. When the user does employ start conditions, he must ensure that every start condition that recognizes any keyword or identifier also recognizes every other keyword and identifier regardless of where they are syntactically acceptable. When the scanner is in such a start condition, it then never ignores

[1] We assume ISO C99 without the label address operator extension supported by compilers like GCC.

the correct keyword or identifier match. When the scanner is in a start condition that accepts no keyword or identifier but a keyword or identifier appears in the input, a syntax error is guaranteed to be detected as expected. Likewise, the user must ensure that similar operators are always recognized by the same start conditions.

For a pseudo-scanner then, groups of tokens like keywords and identifiers or similar operators must somehow be tied together so that the scanner always recognizes all tokens from such a group where it recognizes any of them. However, requiring users to study sub-language transitions or, worse, parser tables in order to ensure that tokens are properly tied together worsens the complexity of developing and maintaining a PSLR(1) specification. Instead, a PSLR(1) specification can employ the ***lexical tie directive***, denoted `%lex-tie`, to specify groups of tokens that the generator should automatically tie together in the pseudo-scanner. To make the declarations a little more succinct and maintainable, a PSLR(1) specification can also employ a ***symbol set directive***, denoted `%symbol-set`.

For example, Figure 3.6d declares all of the keywords from the examples in Figures 3.6a, 3.6b, and 3.6c to be in a symbol set bearing the user-supplied name `keywords`. It then declares all keywords and the identifier to be lexically tied. Next, it declares traditional lexical precedence rules for conflicts between the identifier and any keyword, giving higher precedence to the keyword in the case of identity conflicts. Finally, in a separate group, it declares two similar operators, `'&'` and `'&&'`, to be lexically tied, and it declares the traditional longest match rule for the length conflict between them. Thus, because a match for the identifier token is syntactically acceptable at the "`int`" in Figure 3.6a, the pseudo-scanner behaves as if a match for the `'int'` token is as well. The highest precedence match is then ("`int`",`'int'`). Similarly, because a match for the `'while'` token is syntactically acceptable at the "`whiles`" in Figure 3.6b, the pseudo-scanner behaves as if a match for the identifier token is as well. The highest precedence match is then ("`whiles`",`ID`). Finally, because a match for the `'&'` token is syntactically acceptable at the "`&&`" in Figure 3.6c, the pseudo-scanner behaves as if a match for the `'&&'` token is as well. The highest precedence match is then ("`&&`",`'&&'`). In every case, the pseudo-scanner returns the highest precedence match to the parser, which reports the expected syntax error.

Formally, lexical tie declarations modify the definition of *acc* from Definition 2.2.12 as follows.

Definition 3.3.1 (Lexical Ties)

Given the token t and the set T of all tokens to which t is lexically tied, then $ties(t) = \{t\} \cup T$. $\square$

Definition 3.3.2 (Accepts with Lexical Ties)

Given the LR(1) parser state s_p, then $acc(s_p) = \{t : \exists((\ell \rightarrow \varrho), d, K, (a_t, a_p, a_s)) \in s_p : (a_t =$ "S" $\wedge\ \varrho[d] \in ties(t)) \vee (a_t =$ "R" $\wedge\ K \cap ties(t) \neq \emptyset)\}$. $\square$

This change to *acc* affects the basic definition of pseudo-scanner behavior from Definition 3.1.1, the definition of pseudo-scanner conflicts from Definition 3.2.1, and conflict resolution as performed by the *resolve* function from Definition 3.2.20 to construct the *scanner_accepts* table. That is, the purpose of declaring lexical ties is to expand sets of matches, possibly creating new pseudo-scanner conflicts. Thus, `%lex-prec` declarations like those appearing in Figure 3.6d might not be necessary until the associated tokens are declared to be lexically tied.

There are a few important subtleties for lexical ties that need to be clarified. First, the lexical tie relation for any PSLR(1) specification is implicitly reflexive. For example, given a token `T`, the declaration "`%lex-tie T T`" is redundant because, in Definition 3.3.1, $ties(t)$ always contains t. Second, the lexical tie relation for any PSLR(1) specification is implicitly symmetric. That is, the order of operands for `%lex-tie` is irrelevant. For example, given the tokens `A` and `B`, our PSLR(1) generator interprets the declaration "`%lex-tie A B`" to mean that the pseudo-scanner should recognize `A` in all syntactic contexts in which `B` is recognized and that the pseudo-scanner should recognize `B` in all syntactic contexts in which `A` is recognized. Reversing this statement to reflect the opposite order of operands in the declaration "`%lex-tie B A`" does not change its meaning. Third, the lexical tie relation for any PSLR(1) specification is implicitly transitive. Thus, in Definition 3.3.1, $ties(t)$ includes all tokens to which t is explicitly declared to be lexically tied, but $ties(t)$ also includes all tokens to which those tokens are lexically tied, and so on. While transitivity for the lexical tie relation might seem inconsistent with our decision that the lexical precedence relation should not be implicitly transitive, transitivity is actually an unavoidable consequence of the intuitive meaning of lexical ties. Continuing our example for the `A` and `B` tokens and given the token `C`, our generator interprets the declaration "`%lex-tie B C`" to mean that the pseudo-scanner should recognize `B` in all syntactic contexts in which `C` is recognized and that the pseudo-scanner should recognize `C` in all syntactic contexts in which `B` is recognized. However, given both declarations, the only way to recognize `C` in all contexts in which `B` is recognized is to recognize `C` in all contexts in which `A` is

recognized, and the only way to recognize `A` in all contexts in which `B` is recognized is to recognize `A` in all contexts in which `C` is recognized. Thus, `A` and `C` are lexically tied implicitly.

There are also a few subtleties when lexical ties and symbol sets are combined. Given the symbol set `set` and the token `T`, then "`%lex-tie set T`" or the equivalent "`%lex-tie T set`" lexically ties `T` to every token t in `set` iff there is pairwise scanner conflict between `T` and t. Given the symbol sets `set-a` and `set-b`, then "`%lex-tie set-a set-b`" or the equivalent "`%lex-tie set-b set-a`" lexically ties every token t in `set-a` to every token t' in `set-b` iff there is pairwise scanner conflict between t and t'. For example, if `punctuators` is a symbol set containing all punctuators from a language, then "`%lex-tie punctuators punctuators`" ensures that punctuators like `'+'` and `'++'` are lexically tied while it has no effect on punctuators like `'.'` and `'/'`. Our justification for requiring conflicts is that, given the transitivity of lexical ties, avoiding unnecessary lexical ties can potentially avoid the creation of a large set of unnecessary pseudo-scanner conflicts. However, that requirement is relaxed when both operands are tokens. For example, given the tokens `A` and `B`, "`%lex-tie A B`" lexically ties `A` and `B` regardless of whether `A` and `B` have any pairwise scanner conflict. We are not yet sure that there is ever a practical reason to lexically tie tokens that do not conflict, but we permit this possibility in case it proves useful as we continue to explore practical applications of PSLR(1).

Any pair of tokens that have a scanner conflict is a candidate for lexical tying if one of those tokens appears without the other in any parser state's set of acceptable tokens.

Definition 3.3.3 (Lexical Tie Candidates)
Given any two different tokens t and t' and an LR(1) parser whose state set is Σ_p, then (t, t') is a ***lexical tie candidate*** for that parser iff $\exists \xi \in \Xi^* : \exists \lambda : \exists \lambda' : \{(\lambda, t), (\lambda', t')\} \subseteq \mathscr{M}(\xi, \{t, t'\})$ and $\exists s_p \in \Sigma_p : (t \in acc(s_p)) \neq (t' \in acc(s_p))$. □

Our PSLR(1) generator reports all lexical tie candidates to aid the user in developing a correct PSLR(1) specification. The user can disable this report for a pair of tokens that should not be lexically tied by using the `%lex-no-tie` directive. For example, the PSLR(1) specification in Figure 2.3c declares "`%lex-no-tie '>' '>>'`" to specify that, when the character sequence ">>" is encountered in a syntactic context where only `'>'` is acceptable, it is indeed the intention of the specification author that the pseudo-scanner should select the match (">",`'>'`) and ignore the possibility of (">>",`'>>'`). It is not possible to override the transitivity of lexical ties using

`%lex-no-tie`. That is, if the `A` and `B` tokens are lexically tied and the `B` and `C` tokens are lexically tied, it is nonsensical to declare that `A` and `C` should not be lexically tied. As for `%lex-tie`, the order of operands of `%lex-no-tie` is irrelevant. That is, like the lexical tie relation, the lexical no-tie relation is implicitly symmetric. However, unlike the lexical tie relation, the lexical no-tie relation is not implicitly reflexive or transitive.

For some PSLR(1) specifications, the lexical tie candidate report might prove to require more work from the user than it is worth. For example, if a language contains no reserved keywords, constantly reassuring the PSLR(1) generator that basic pseudo-scanner behavior is indeed preferred might become tedious for the user. However, our PSLR(1) generator supports `yyall` as a built-in symbol set that contains all symbols. Thus, the user can suppress the lexical tie candidate report entirely by declaring "`%lex-no-tie yyall yyall`". More specific declarations always override more generic declarations. Thus, if the user decides that keywords should be reserved but doesn't want a report of all other lexical tie candidates, he can override "`%lex-no-tie yyall yyall`" for the `ID` token and the `keywords` symbol set by also declaring "`%lex-tie ID keywords`".

As future work, we are considering extending our PSLR(1) generator with ***asymmetric lexical ties*** to improve support for languages, such as PL/I, that have non-reserved keywords, and for languages, such as SQL, that have both non-reserved and reserved keywords. For example, assume the user declares the rule of longest match for all length conflicts, and assume he declares that all keywords have higher lexical identity precedence than `ID`. Assume the user has declared the symbol set `non-reserved` containing all non-reserved keywords including `'for'`. The user could then declare "`%lex-tie ID -> non-reserved`" as an asymmetric lexical tie indicating that the pseudo-scanner should recognize the `ID` token in all syntactic contexts in which non-reserved keywords are recognized. This would ensure that the pseudo-scanner selects the match ("`foreach`",`ID`) instead of ("`for`",`'for'`) for a variable name like "`foreach`" when `'for'` is acceptable in the current syntactic context but `ID` is not. However, the reverse asymmetric lexical tie is not declared. Thus, in a syntactic context where `ID` is acceptable but `'for'` is not, the pseudo-scanner would select the match ("`for`",`ID`) instead of ("`for`",`'for'`) for the character sequence "`for`" as desired. Asymmetric lexical ties would be combined with symmetric lexical ties to form a reflexive and transitive relation. Thus, if `'foreach'` is actually a reserved keyword and there is also a symmetric lexical tie between `ID` and a symbol set containing reserved keywords, then the pseudo-scanner would select the match ("`foreach`",`'foreach'`) for the character sequence

"`foreach`" in any syntactic context where `ID`, `'for'`, or `'foreach'` is acceptable.

There is one final caveat about lexical ties, whether symmetric or asymmetric. Before lexical ties expand $acc(s_p)$, there might exist some token $t \in acc(s_p)$ such that every match for t has lower lexical precedence than some match for some other token in $acc(s_p)$ either before or after lexical ties expand $acc(s_p)$. Thus, due to lexical precedence and possibly lexical ties, t might not actually be acceptable in the syntactic contexts represented by s_p even though the grammar implies it should be. So far, there is nothing problematic about this scenario as it reflects the user's PSLR(1) specification. However, our PSLR(1) generator evaluates lexical ties to adjust $acc(s_p)$ before discovering, resolving, and reporting complete pseudo-scanner conflicts. Thus, the generated pseudo-scanner recognizes every token $t' \in ties(t)$ in every syntactic context represented by s_p even though t itself is not recognized in any of those contexts. This result seems contrary to the concept of a lexical tie. That is, if t is not recognized in a syntactic context, then why is t' added to that context? Solving this problem appears to be a chicken-and-egg problem: lexical ties are an input to conflict resolution, so there seems to be no obvious way to allow conflict resolution to affect the computation of lexical ties. Instead of trying to change this behavior, we take the approach of documenting it. If the user does not want this behavior, he must rewrite his grammar not to permit t to appear in syntactic contexts where his lexical precedence rules and lexical ties make t unacceptable.

3.4 Minimal LR(1)

Practical LR(1) parser table generation algorithms merge canonical LR(1) parser states. The most popular such algorithm is LALR(1). In section 3.4.1, we discuss the reasons why state merging is often desired, and we discuss the loss of language recognition power that it can cause relative to canonical LR(1). In section 3.4.2, we summarize our IELR(1) algorithm, which is a minimal LR(1) algorithm in that it achieves the best of both canonical LR(1) and LALR(1) for traditional parsers. However, IELR(1) does not address problems that state merging causes for the behavior of a pseudo-scanner. Thus, in section 3.4.3, we present an IELR(1) extension that we have implemented as part of our PSLR(1) generator to handle those problems as well. We summarize in section 3.4.4.

3.4.1 LALR(1) Versus Canonical LR(1)

Consider again the PSLR(1) specification in Figure 2.3c. Recall that the PSLR(1) parser and pseudo-scanner it specifies accept the sentences in Figures 2.3a and 2.3b with the parse trees in Figures 2.3d and 2.3e, respectively. In section 3.1, we showed that the scanner conflict between the `'>'` and `'>>'` tokens at the character sequence ">>" in both sentences is not a pseudo-scanner conflict when the parser employs canonical LR(1) parser tables. Examining the canonical LR(1) parser tables, shown in the first column of Table 2.1, it is easy to see why. When the parser reaches the ">>" in Figure 2.3b, the parser's current state, s_p, is state 1, and so `'>>'` is in $acc(s_p)$ but `'>'` is not. When the parser reaches the ">>" in Figure 2.3a, s_p is state 18, and so `'>'` is in $acc(s_p)$ but `'>>'` is not. The pseudo-scanner always selects the token from $acc(s_p)$.

Now consider what happens when the parser employs LALR(1) parser tables instead. The LALR(1) algorithm merges canonical LR(1) states 1 and 18 to form LALR(1) state 1, shown in the second column of Table 2.1. As a result, at the ">>" in both of our example sentences, s_p is LALR(1) state 1, $acc(s_p)$ contains both the `'>'` and `'>>'` tokens, and so there is now a pseudo-scanner conflict. In general, by merging canonical LR(1) parser states, the LALR(1) algorithm sometimes merges the left contexts that distinguish between sub-languages. When this happens, the parser can lose the power to determine which token from which sub-language it should accept. The result is sometimes a new pseudo-scanner conflict as in our example. It is also possible that existing pseudo-scanner conflicts from different left contexts may merge in such a way that, for some left contexts, the lexical precedence rules select a different match than they did before the merge.

The effect that parser state merging has on a pseudo-scanner is similar to a well-known effect it has on the parser itself. That is, because of the merging of left contexts, the parser sometimes loses the power to determine the correct parser action on a given lookahead token [15, 16, 17, 28, 29, 30, 32, 33]. The result may be the creation of a new parser conflict or the merging of existing parser conflicts such that precedence rules now select a different action for some left contexts than they did before the merge. We have previously introduced the term *LR(1)-relative inadequacies* to refer to all such changes in the parser's behavior [15, 16]. We now introduce the term ***PSLR(1)-relative inadequacies*** to refer collectively to such changes in the parser's behavior and the pseudo-scanner's behavior. That is, given a set of parser tables, if the pseudo-scanner and parser do not accept exactly the same set of sentences each with the same parse tree as they would with canonical LR(1) parser

tables, then the given set of parser tables is inadequate for PSLR(1).

Our choice of canonical LR(1) as the standard against which to compare other parser tables is not arbitrary. LR parsing is "the most general [deterministic][2] shift-reduce parsing method known", and canonical LR(1) is the most general technique for generating LR(1) parser tables [11]. As a result, when the user of our PSLR(1) generator considers the general behavior of deterministic shift-reduce parsing with one token of lookahead but without the complexity of any further parsing restriction, the behavior he should expect from the generated parser and pseudo-scanner is the behavior produced by canonical LR(1) parser tables. In contrast, in order to understand the behavior produced by parser tables with PSLR(1)-relative inadequacies, the user must consider not only the behavior of deterministic shift-reduce parsing but also the complex details of parser state construction and merging. In this way, eliminating all PSLR(1)-relative inadequacies provides the user with a simpler model of how the parser and pseudo-scanner should behave and thus facilitates the development and maintenance of a correct PSLR(1) specification.

Unfortunately, canonical LR(1) parser tables tend to be an order of magnitude larger than LALR(1) parser tables for practical languages [11]. There are at least two effects. First, canonical LR(1) parser tables at one time were considered to require "too much space and time to be useful in practice" [11]. However, the validity of this statement is fading with the increasing memory capacity and processing power of modern computers. Second, the extra states often contain unnecessary duplicates of conflicts, and so the difficulty of debugging conflicts can increase an order of magnitude as well [16].

3.4.2 IELR(1)

As part of our preliminary work, we described and implemented IELR(1) [15, 16]. IELR(1) is a *minimal LR(1)* parser table generation algorithm because it generates parser tables that are nearly the size of LALR(1) parser tables but with the full language recognition power of canonical LR(1) when the parser is not coupled with a pseudo-scanner. It does so by computing the source of each LR(1)-relative inadequacy in the LALR(1) parser tables and then splitting states to eliminate it. Thus, at any point in the parse of a syntactically acceptable sentence, an IELR(1) parser performs the same parser action as a canonical LR(1) parser does and thus constructs the same parse tree. Other

[2]GLR (Generalized LR) does not employ backtracking but is more general than LR [34]. Thus, we replace the word "nonbacktracking" in this quote with the word "deterministic", which excludes both backtracking and GLR.

Grammar	Version	$\|T\|$	$\|V\|$	$\|T \cup V\|$	$\|P\|$
Gawk	Gawk 3.1.0	61	45	106	163
Gpic	Groff 1.18.1	138	45	183	247
C	GCC 4.0.4	92	208	300	573
Java	GCC 4.2.1	109	164	273	516
C++	ISO 2003	117	184	301	481

Table 3.3: IELR(1) Case Studies. These counts measure the size of each case study's grammar $G = (V, T, P, S)$, such that V is the set of nonterminals, T is the set of terminals or tokens, P is the set of productions, and S is the start symbol. These counts include the productions and nonterminals that Bison generates implicitly for mid-rule actions.

---

minimal LR(1) algorithms we have found handle at most LR(1) grammars [28, 29, 30]. However, IELR(1) also correctly handles non-LR(1) grammars coupled with a specification for resolving parser conflicts.

We implemented IELR(1) as an extension of Bison, and we also parameterized IELR(1) to generate full canonical LR(1) parser tables. IELR(1) and canonical LR(1) are scheduled to appear in Bison 2.5. Bison has always implemented LALR(1). Table 3.3 characterizes grammars from a series of case studies that we have previously presented in order to compare LALR(1), IELR(1), and canonical LR(1) using Bison [15, 16]. We now summarize these case studies and their results to demonstrate the success of IELR(1).

The first four case studies are mature parser specifications from widely used software applications that employ LALR(1) parser generators. Gawk (GNU AWK), a text-based data processing language, was first written in 1986 but is based on the original AWK, which was written in 1977 and is standardized in SUSv3 (the Single UNIX Specification, Version 3) [9, 4]. Groff (GNU Troff) is a document formatting system for UNIX that includes Gpic (GNU Pic), a Groff preprocessor for specifying diagrams. Groff was first released in 1990 and is based on Troff which has existed since the early 1970's [14, 6]. We copied our C and Java parser specifications from GCC (the GNU Compiler Collection), which is a widely used collection of compilers developed by the GNU Project [5].

The latest version of the C++ programming language is C++ 2003. Annex A of the C++ 2003 specification presents a formal C++ grammar [8]. As our final case study, we formatted this grammar as a Bison parser specification file except that, for section A.2, Lexical conventions, we (1) replaced the *integer_literal*, *character_literal*, *floating_literal*, and *string_literal* nonterminals

Grammar	States			S/R			R/R		
	LA	IE	Canon	LA	IE	Canon	LA	IE	Canon
Gawk	320	329	2359	65	65-0	265-200	0	0-0	0-0
no prec/assoc	320	320	2467	410	410-0	3209-2799	0	0-0	0-0
Gpic	423	428	4834	0	0-0	0-0	0	0-0	0-0
no prec/assoc	426	426	4871	803	803-0	7576-6773	8	8-0	24-16
C	933	933	4108	13	13-0	29-16	0	0-0	0-0
no prec/assoc	933	933	4108	329	329-0	3731-3402	0	0-0	0-0
Java	792	792	6161	0	0-0	0-0	62	62-0	660-598
C++	822	836	9849	407	410-3	2871-2464	135	169-34	3130-2995

Table 3.4: IELR(1) Parser Tables. In this table, we describe the parser tables that the Bison LALR(1) implementation, our IELR(1) implementation, and our canonical LR(1) implementation generate for our case studies. We report the number of states and the number of parser conflicts left unresolved by the user. For canonical LR(1) and IELR(1), we show adjustments to account for such unresolved parser conflicts that are perfectly duplicated among states with identical cores.

with tokens, and (2) removed all productions that only those nonterminals depend upon.

Table 3.4 describes the parser tables that the Bison LALR(1) implementation, our IELR(1) implementation, and our canonical LR(1) implementation generate for each of our case studies. Because some of our case studies include precedence and associativity declarations that resolve most of their parser conflicts, we also describe their parser tables when generated without these declarations in order to better demonstrate the complexity of the parser specification analysis. For example, the "no prec/assoc" row beneath the "Gpic" row reveals that the LALR(1) and IELR(1) algorithms must actually examine 803 S/R conflicts even though all parser conflicts are ultimately resolved by user declarations.

When IELR(1) or canonical LR(1) splits an LALR(1) state, conflicts in the LALR(1) state might be perfectly duplicated among some of the new states. Bison counts the conflict separately for each such duplicate, but the multiple count is a misleading representation of complexity because the same precedence and associativity declarations would resolve all duplicates. Therefore, from each IELR(1) or canonical LR(1) unresolved conflict count in Table 3.4, we subtract all but one copy of each unresolved conflict that is perfectly duplicated among states with identical cores.

Table 3.5 describes the parser actions that are corrected in the parser tables by switching from LALR(1) to IELR(1) or to canonical LR(1). We were surprised to discover that mature parser specifications from widely used software products employing LALR(1) parser generators should suffer from any incorrect parser actions that result from the misuse of the LALR(1) algorithm. The Gawk

Grammar	Actions		States		Tokens	
	IE	Canon	IE	Canon	IE	Canon
Gawk	9-0	90-81	3-0	30-27	3	3
Gpic	2-0	16-14	1-0	8-7	2	2
C	0-0	0-0	0-0	0-0	0	0
Java	0-0	0-0	0-0	0-0	0	0
C++	4-0	37-33	4-0	37-33	2	2

Table 3.5: IELR(1) Action Corrections. For each of our case studies, this table reports the number of parser actions that are corrected by switching from LALR(1) to IELR(1) or to canonical LR(1), the number of parser states containing corrected parser actions, and the number of unique tokens in the grammar on which there are corrected parser actions. We also show adjustments to account for action corrections that are perfectly duplicated among states with identical cores.

---

and Gpic case studies provide strong evidence that such incorrect actions do occur in real-world parsers. Such actions are unintuitive and thus may impede the development of a correct parser.

Our canonical LR(1) experiments led to an interesting insight. As had been observed previously in the literature, canonical LR(1) parser tables tend to be an order of magnitude larger than LALR(1) parser tables for practical LR(1) parser specifications [11]. The state counts in Table 3.4 are consistent with this observation. However, because of the increased splitting of states in canonical LR(1) relative to IELR(1), the number of conflicts that are perfectly duplicated and the number of action corrections usually increases an order of magnitude. Thus, the difficulty of investigating conflicts while developing a parser specification increases an order of magnitude as well. Interestingly, for all of our case studies, every new conflict is a perfect duplicate, so the adjusted conflict counts are the same as for IELR(1).

The IELR(1) algorithm consists of 6 phases, which we outline here. We have previously described the algorithm in full detail [16].

- Phase 0: LALR(1). This phase computes LALR(1) parser tables, which fully merge all LR(1) parser states with identical cores and thus contain some form of every possible LR(1)-relative inadequacy that can exist after any possible combination of such merges.

- Phase 1: Compute Auxiliary Tables. From the LALR(1) parser tables, this phase computes a number of additional tables employed by later phases.

- Phase 2: Compute Annotations. This phase identifies each parser conflict in the LALR(1) parser tables, traces each conflict back through all predecessor states that contribute to the

conflict, and adds annotations to the visited states to record the nature of the states' contributions.

- Phase 3: Split States. This phase effectively splits the LALR(1) states to eliminate all LR(1)-relative inadequacies. However, the algorithm actually recomputes all parser states in a manner similar to phase 0. The main difference is that, when considering whether to merge parser states, it employs a stricter state compatibility test based on the LALR(1) states' annotations from phase 2.

- Phase 4: Compute Reduction Lookaheads. This phase recomputes the reduction lookahead sets throughout the recomputed parser states.

- Phase 5: Resolve Remaining Conflicts. This phase resolves all remaining parser conflicts by eliminating parser actions with the lowest precedence.

3.4.3 IELR(1) Extension for PSLR(1)

As shown in Table 2.1, the LALR(1) parser tables for the PSLR(1) specification in Figure 2.3c have no parser conflicts and thus no LR(1)-relative inadequacies. For this reason, the LALR(1) parser tables are also IELR(1) parser tables. However, as we explained in section 3.4.1, these tables do contain other PSLR(1)-relative inadequacies. Thus, our IELR(1) algorithm in its original form is not always adequate for PSLR(1). In this section, we explain how we extend IELR(1) to eliminate all PSLR(1)-relative inadequacies.

Because state merging is the cause of all PSLR(1)-relative inadequacies including LR(1)-relative inadequacies, much of our original IELR(1) algorithm can be reused for PSLR(1). We extend IELR(1) for PSLR(1) in two steps. First, we extend IELR(1) phase 2 to annotate parser states based on their contributions to pseudo-scanner conflicts. Second, we extend IELR(1) phase 3 by adjusting its state compatibility test to consider these extended annotations. This state compatibility test extension dictates the form of the annotations for phase 2, so we focus our discussion on phase 3.

Our extension to the state compatibility test of IELR(1) phase 3 must determine whether two states s_p and s'_p can be merged without introducing a PSLR(1)-relative inadequacy. The extension does not need to consider PSLR(1)-relative inadequacies that are LR(1)-relative inadequacies because the latter are already considered by IELR(1)'s existing state compatibility test.

Definition 3.4.1 (State Compatibility Test)

Given a set of lexical precedence rules R, which defines a lexical precedence function Δ, and given two LR(1) parser states s_p and s'_p, then PSLR(1)'s extension to the IELR(1) state compatibility test considers s_p and s'_p to be compatible in the context of R iff, $\forall \xi \in \Xi^*$, either:

1. $\mathscr{M}(\xi, acc(s_p)) = \emptyset \vee \mathscr{M}(\xi, acc(s'_p)) = \emptyset$.

2. $\Delta(\mathscr{M}(\xi, acc(s_p))) = \Delta(\mathscr{M}(\xi, acc(s'_p))) = \Delta(\mathscr{M}(\xi, acc(s_p) \cup acc(s'_p)))$.

□

Point 1 in Definition 3.4.1 checks for the case when pseudo-scanner conflicts for ξ are *irrelevant* in the syntactic contexts represented by either s_p or s'_p. The concept of irrelevant pseudo-scanner conflicts resembles the concept of irrelevant parser conflicts in the original IELR(1) state compatibility test [16]. For example, if $\mathscr{M}(\xi, acc(s_p)) = \emptyset$, then we say that pseudo-scanner conflicts for ξ are irrelevant in the syntactic contexts represented by s_p. In other words, in the syntactic contexts represented by s_p, there are no acceptable tokens that match ξ, and so any match selected for ξ would correctly be detected as a syntax error by the parser. Thus, even though merging s_p and s'_p causes the selected match to become $\Delta(\mathscr{M}(\xi, acc(s'_p)))$, the pseudo-scanner's behavior is still correct.

When pseudo-scanner conflicts for ξ are relevant in the syntactic contexts represented by s_p and in the syntactic contexts represented by s'_p, point 2 in Definition 3.4.1 checks that merging s_p and s'_p does not change the highest precedence match for ξ in any of those syntactic contexts. To determine what the highest precedence match becomes, the third segment of the equality in point 2 evaluates Δ for the merged state. However, we prove that it is actually redundant to evaluate Δ for the merged state. In the terminology of our original IELR(1) algorithm, this means that a lexical precedence function for a PSLR(1) specification is always *merge-stable* [16].

Theorem 3.4.2 (Merge-stable Lexical Precedence Function)

Given a lexical precedence function Δ, given two LR(1) parser states s_p and s'_p, and given a character sequence $\xi \in \Xi^*$, then the following two conditions are equivalent:

1. $\Delta(\mathscr{M}(\xi, acc(s_p))) = \Delta(\mathscr{M}(\xi, acc(s'_p)))$.

2. $\Delta(\mathscr{M}(\xi, acc(s_p))) = \Delta(\mathscr{M}(\xi, acc(s'_p))) = \Delta(\mathscr{M}(\xi, acc(s_p) \cup acc(s'_p)))$.

□

Proof (Theorem 3.4.2)

If point 1 in Theorem 3.4.2 is false, then point 2 in Theorem 3.4.2 is trivially false because it includes point 1. For the rest of this proof then, we only need to prove that, when point 1 is true, point 2 is also true. For brevity, let $M = \mathscr{M}(\xi, acc(s_p))$, and let $M' = \mathscr{M}(\xi, acc(s'_p))$. Thus, by Definition 2.2.4, $\mathscr{M}(\xi, acc(s_p) \cup acc(s'_p)) = M \cup M'$. Let $m = \Delta(M)$, and thus $m = \Delta(M')$ because we assume point 1 is true. Our goal then is to prove that $m = \Delta(M \cup M')$ so that point 2 is true:

1. Consider the case where $m =$ undefined. That is, by Definition 3.2.5, there exists no match in M that has higher precedence than all other matches in M, and the same is true for M'. Because neither M nor M' has a highest precedence match, then neither does $M \cup M'$, so $\Delta(M \cup M') =$ undefined $= m$.

2. Consider the case where $m \neq$ undefined. That is, by Definition 3.2.5, m has higher precedence than all other matches in M and all other matches in M', so m has higher precedence than all other matches in $M \cup M'$. Moreover, by Definition 3.2.5, $m \in M$ and $m \in M'$, so $m \in M \cup M'$. Thus, $m = \Delta(M \cup M')$.

□

In section 3.2.4, we considered whether it would be reasonable to use the sequential lexical precedence function, $\mathscr{F}$, defined by Δ in place of Δ so that the highest precedence match could be computed in linear time. However, Theorem 3.4.2 would not necessarily be valid if we did so. The trouble would be that, if $\mathscr{F}$ were ambiguous, the highest precedence match for s_p, for s'_p, and for the merged state would depend on how the match ordering would be chosen in each case. Fortunately, Δ returns undefined when $\mathscr{F}$ is ambiguous, and our generator reports an unresolved conflict, so we can use Theorem 3.4.2 to simplify our state compatibility test.

Definition 3.4.3 (State Compatibility Test, *simplified*)

Given a set of lexical precedence rules R, which defines a lexical precedence function Δ, and given two LR(1) parser states s_p and s'_p, then PSLR(1)'s extension to the IELR(1) state compatibility test considers s_p and s'_p to be compatible in the context of R iff, $\forall \xi \in \Xi^*$, either:

1. $\mathscr{M}(\xi, acc(s_p)) = \emptyset \vee \mathscr{M}(\xi, acc(s'_p)) = \emptyset$.

2. $\Delta(\mathscr{M}(\xi, acc(s_p))) = \Delta(\mathscr{M}(\xi, acc(s'_p)))$.

□

Regardless of whether we use Definition 3.4.1 or 3.4.3, we have not yet considered whether the state compatibility test actually makes sense when the complete pseudo-scanner conflict for ξ is unresolved in either s_p or s'_p. Without loss of generality, we only examine the case where it is unresolved in s_p. That is, $\Delta(\mathscr{M}(\xi, acc(s_p)))$ = undefined. There are two subcases to consider. First, if $\Delta(\mathscr{M}(\xi, acc(s'_p))) \neq$ undefined, then $\Delta(\mathscr{M}(\xi, acc(s_p))) \neq \Delta(\mathscr{M}(\xi, acc(s'_p)))$, so s_p and s'_p are not merged. This makes sense because merging s_p with s'_p would have one of two effects: (1) a match from the syntactic contexts represented by s'_p would become the highest precedence match in the syntactic contexts represented by s_p and thus suppress the conflict report for the latter syntactic contexts, or (2) the complete pseudo-scanner conflict in the syntactic contexts represented by s'_p would become unresolved even though the user's lexical precedence rules were sufficient to resolve it. Either effect is undesirable, so refusing to merge the states is the correct decision.

The second subcase is when $\Delta(\mathscr{M}(\xi, acc(s'_p)))$ = undefined. Thus, $\Delta(\mathscr{M}(\xi, acc(s_p))) = \Delta(\mathscr{M}(\xi, acc(s'_p)))$, so s_p and s'_p are merged. Unfortunately, this means the generator is forced to report a merged version of the unresolved complete pseudo-scanner conflict instead of reporting only the matches that are actually present in each syntactic context. To avoid this problem, we could choose instead to never merge s_p and s'_p in this subcase as long as $s_p \neq s'_p$. However, if the user has just written the first draft of his PSLR(1) specification and has not yet resolved any conflicts because he wants to read the generator's conflict report first, then states with pseudo-scanner conflicts would only be merged with identical states. Thus, the generated parser tables could be as large as canonical LR(1) parser tables. As we explained earlier in this paper, canonical LR(1) parser tables can severely complicate the process of debugging conflicts because the number of parser states can increase by an order of magnitude. Thus, our approach is simply to document that unresolved complete pseudo-scanner conflicts from multiple syntactic contexts are sometimes merged in the conflict report. A similar problem can occur for parser conflicts that are left unresolved by the user but happen to have the same default resolution, so this is not a new phenomenon. If the user believes that canonical LR(1) parser tables can aid in the debugging of parser and pseudo-scanner conflicts in these cases, our PSLR(1) generator accepts an option to switch the parser table generation algorithm from IELR(1) to canonical LR(1).

The conflict resolution algorithm presented in section 3.2.7 might be too time-consuming to perform before every merge that must be considered during IELR(1) phase 3. Thus, our PSLR(1) generator does not implement the state compatibility test exactly as written in Definition 3.4.1 or Definition 3.4.3. We observe that, if R selects the same highest precedence match for every pairwise pseudo-scanner conflict in s_p as in s'_p, then it is guaranteed to select the same highest precedence match for every complete pseudo-scanner conflict in s_p as in s'_p because any complete scanner conflict is a collection of pairwise scanner conflicts according to Definition 2.2.5. Thus, phase 3 applies our state compatibility test from Definition 3.4.3 to pairwise pseudo-scanner conflicts instead of complete pseudo-scanner conflicts. This shift to the pairwise level improves the performance of phase 3 because, instead of using our algorithm from 3.2.7 to analyze the exact combination of tokens present in each parser state encountered during phase 3, our generator can iterate the pseudo-scanner's FSA once before IELR(1) begins and summarize how all possible pairwise scanner conflicts for each pair of tokens are resolved. It is important to understand that we shift to the pairwise level in this manner only for the sake of parser table construction. Afterwards, complete pseudo-scanner conflicts are fully discovered, resolved, and reported for the resulting parser tables using our algorithm from section 3.2.7.

Because not all pairwise pseudo-scanner conflicts need to be resolved in order to resolve all complete pseudo-scanner conflicts, our state compatibility test is more strict than necessary, and unnecessary state splitting can occur as a result. As we discuss in section 4, this effect does not prove to be problematic for our case studies. Part of the reason is our decision, discussed earlier in this section, that unresolved pseudo-scanner conflicts should not prevent state merging. That is, when all complete pseudo-scanner conflicts are resolved, then pairwise pseudo-scanner conflicts that remain unresolved do not cause unnecessary state splitting. Of course, when we discussed this decision earlier, we discussed it at the level of complete pseudo-scanner conflicts rather than at the level of pairwise pseudo-scanner conflicts, and we said that complete pseudo-scanner conflicts might be merged in the conflict report as a result. This conflict report problem still exists after our shift to the pairwise level because, when all pairwise pseudo-scanner conflicts remain unresolved, then complete pseudo-scanner conflicts remain unresolved. Fortunately, after our shift to the pairwise level, that problem is lessened when just a few of the associated pairwise pseudo-scanner conflicts are resolved differently in s_p than in s'_p.

3.4.4 Summary

In this section, we have discussed the trade-offs between the LALR(1) and canonical LR(1) parser table generation algorithms. We have also explained how our IELR(1) algorithm is a minimal LR(1) algorithm in that it achieves the best of both canonical LR(1) and LALR(1) for traditional parsers. Finally, we presented an IELR(1) extension required by PSLR(1) parsers and their pseudo-scanners.

3.5 Syntax Error Handling

There are three tasks for handling syntax errors, and all are ultimately performed by the PSLR(1) parser: detecting, reporting, and recovering. However, before the parser can perform these three tasks, the pseudo-scanner must first select some match to return to the parser upon encountering the syntax error. This match has an important influence on the way in which the parser performs its three tasks, so the pseudo-scanner needs to make a reasonable selection. First, the match should be syntactically unacceptable so that the parser can detect the syntax error. Second, the match should be a reasonable guess at interpreting the erroneous input character sequence so that the parser can construct a reasonable syntax error message to report to the user of the parser. Third, if the parser implements a syntax error recovery mechanism, such a reasonable guess also improves the parser's chances of recovering from the syntax error and correctly parsing the remaining input. Finally, in cases where there is no obvious way to make a reasonable guess, it is still helpful to document some deterministic algorithm to select a match so that the author of a PSLR(1) specification can more easily develop formal test suites for his generated PSLR(1) parsers and pseudo-scanners. In section 3.5.1, we discuss the pseudo-scanner's mechanisms for selecting a match upon encountering a syntax error. In section 3.5.2, we discuss the parser's mechanisms for detecting and reporting a syntax error. The choice of a syntax error recovery mechanism for the parser is orthogonal to the choice between PSLR(1) and traditional scanner-based LR(1), so it is beyond the scope of this paper. We summarize in section 3.5.3.

3.5.1 Pseudo-scanner

The basic pseudo-scanner behavior defined in Definition 3.1.1 dictates that, for the character sequence ξ and the parser state s_p, the pseudo-scanner always selects a match from $\mathscr{M}(\xi, acc(s_p))$.

This definition seems to imply that the pseudo-scanner only selects matches that are syntactically acceptable. In that case, it would be impossible for there to be a syntax error unless $\mathscr{M}(\xi, acc(s_p)) = \emptyset$. However, as we explained in section 3.4, LR(1) state merging sometimes adds to $acc(s_p)$ tokens that are not always syntactically acceptable when the parser is in state s_p. Also, as we explained in section 3.3, the user can declare lexical ties, which can add other syntactically unacceptable tokens to $acc(s_p)$. Thus, the PSLR(1) parser and pseudo-scanner must have syntax error handling mechanisms for the case when $\mathscr{M}(\xi, acc(s_p)) = \emptyset$ and for the case when $\mathscr{M}(\xi, acc(s_p)) \neq \emptyset$.

If $\mathscr{M}(\xi, acc(s_p)) \neq \emptyset$ when a syntax error is encountered, then the pseudo-scanner is not aware of the syntax error. In this case, the pseudo-scanner always selects a match $(\lambda, t) \in \mathscr{M}(\xi, acc(s_p))$ just as it would if there were no syntax error. If t is lexically tied to some token that is syntactically acceptable, then the pseudo-scanner's selection of the match (λ, t) is not merely a reasonable guess. It is an exact selection according to the PSLR(1) specification's lexical ties and lexical precedence rules. If, instead, t is not lexically tied to some token that is syntactically acceptable, then the reason for t's appearance in $acc(s_p)$ must be LR(1) state merging. In this case, the pseudo-scanner's selection of the match (λ, t) is a reasonable guess because state merging means that there exists some similar syntactic context in which (λ, t) is acceptable.

If $\mathscr{M}(\xi, acc(s_p)) = \emptyset$, then the pseudo-scanner is aware of the syntax error and must expand its search to tokens outside of $acc(s_p)$ in order to find a match. In this case, given the PSLR(1) specification's grammar $G : \mathscr{G}(G) = (V', T', P', S')$, then the pseudo-scanner tries to select a match from $\mathscr{M}(\xi, T')$. To support this mechanism, our PSLR(1) generator extends our *scanner_accepts* table from Definition 3.2.14 with a ***fallback*** row. Our generator computes the fallback row in nearly the same way it computes a row for an actual parser state. That is, it uses the functions *compute_scanner_accepts* and *resolve* from Definitions 3.2.19 and 3.2.20 with two small modifications. First, in place of $acc(s_p)$, it uses T'. Second, even though it attempts to resolve all complete scanner conflicts using the lexical precedence rules specified in the PSLR(1) specification, there may exist complete scanner conflicts that are not complete pseudo-scanner conflicts and thus are not resolved by these lexical precedence rules. In this case, the generator falls back on traditional lexical precedence rules. Of course, the pseudo-scanner's behavior must reflect this choice while scanning the input, so the generator also modifies the *length_precedences* table from Definition 3.2.15 to specify the rule of longest match for all length conflicts that are left unresolved by the PSLR(1) specification. Finally, the *pseudo_scan* function from Definition 3.2.16 is extended to check for

matches in the fallback row of *scanner_accepts* until it finds a match in the current parser state's row. If it never finds a match in the current parser state's row, it returns the best match from the fallback row.

It is also possible that both $\mathscr{M}(\xi, acc(s_p)) = \emptyset$ and $\mathscr{M}(\xi, T') = \emptyset$. That is, there might exist no match in the current parser state's row or the fallback row in *scanner_accepts*. In this case, the pseudo-scanner returns a special token that matches only the lexeme $\xi[1]$. Such a token is often called a *character token*. The next time the parser requests a token, the pseudo-scanner looks for a match for $\xi - \xi[1]$. In other words, the pseudo-scanner returns one character at a time until it can match a token in $acc(s_p)$ or T'.

3.5.2 Parser

In every case in which a syntax error is encountered, the pseudo-scanner mechanisms we discussed in the previous section select a match for the input character sequence. The selected match cannot be syntactically acceptable because, in the case of a syntax error, there exists no syntactically acceptable match for the input character sequence. Because the match is not syntactically acceptable, the parser is guaranteed to detect the syntax error after it receives the match from the pseudo-scanner. The parser then reports the syntax error and lists the tokens for which there are acceptable matches in the current syntactic context.

Unfortunately, the parser can sometimes experience delayed syntax error detection and can report an incorrect list of accepted tokens. There are three causes: LR(1) state merging, default reductions, and explicit error actions in the parser. In this section, we explain these problems in detail, and we explain how we fix them. The problems and our fix are relevant for both PSLR(1) parsers and traditional scanner-based LR(1) parsers, and so they are actually orthogonal to our PSLR(1) work. Nevertheless, we have chosen to fix the problems as part of our PSLR(1) work because we feel that they are too severe to be ignored in any type of parser generation system.

LR(1) state merging, default reductions, and explicit parser error actions all have a common effect that leads to delayed syntax error detection and incorrect accepted token lists: they cause parser states to accept tokens that are not actually syntactically acceptable. As we discussed in section 3.4, LR(1) state merging does so by merging lookahead sets from different syntactic contexts. Default reductions are a parser table optimization that removes the largest lookahead set from each parser state. The reduction with which a removed lookahead set was associated becomes the default

reduction in that parser state. That is, when the parser is in such a state and encounters a token that has no parser action in that state, the parser must perform the default reduction in case that token might have been a member of the removed lookahead set. Thus, all tokens become acceptable according to that parser state.

Explicit parser error actions are the way in which S/R parser conflicts are resolved if the conflicted token is declared to be non-associative. In Yacc and Bison, non-associative tokens are declared with the directive `%nonassoc`. The trouble is that parser tables are constructed before conflict resolution, and so there may still exist reduction lookahead sets that contain the non-associative token in parser states other than the conflicted parser state. Removing the non-associative token from those lookahead sets might not be possible without splitting the containing states because those states might have some syntactic contexts in which the S/R conflict for the non-associative token would never be encountered.

Regardless of the reason why a syntactically unacceptable token is acceptable according to the current parser state, when the parser encounters such a token, the parser performs reductions until it finally reaches a state that either has an explicit error action for the token or has no explicit action for the token and no default reduction. In other words, the parser's detection of the syntax error is delayed as erroneous reductions and associated semantic actions are performed. Moreover, the state the parser has reached when it finally detects the syntax error is not the original parser state in which the token was encountered, so it might not accept the same tokens as the original parser state. Of course, due to the three causes we have been discussing, the original parser state might not accept exactly the tokens that are actually syntactically acceptable anyway. Thus, whether delayed detection happens or not, the list of tokens the parser reports based on the current state might be incorrect. It might contain syntactically unacceptable tokens, and it might omit syntactically acceptable tokens.

It might seem that lexical ties can cause the parser state to accept syntactically unacceptable tokens because lexical ties add tokens to $acc(s_p)$. However, our PSLR(1) generator does not compute parser actions for lexically tied tokens. Lexical ties only affect (1) the *scanner_accepts* table constructed for the pseudo-scanner and (2) the state compatibility test of PSLR(1)'s IELR(1) extension because this extension must consider pseudo-scanner conflict resolution.

As we explained in section 3.4, canonical LR(1) never merges states with different lookahead sets. Though the default reduction optimization can be applied to any set of LR(1) parser tables,

it is not considered part of canonical LR(1). However, explicit parser error actions like those set by `%nonassoc` do cause parser states to accept syntactically unacceptable tokens in the case of canonical LR(1). Thus, for non-LR(1) grammars with S/R conflicts resolved by `%nonassoc`, even canonical LR(1) is not immune to the problems of delayed syntax error detection and incorrect accepted token lists in syntax error reports.

We call our solution to these problems ***LAC*** (lookahead correction). LAC is a straightforward extension to the LR(1) parsing algorithm as follows. Whenever the parser requires a new token from the scanner so that it can determine what parser action to perform, we say that token is the *lookahead*, and we say the current parser stack represents the *initial context* of that lookahead. Upon receiving this lookahead from the scanner, the parser immediately performs an exploratory parse to see if the returned token is syntactically acceptable. During this exploratory parse, any reductions of the parser stack are performed on a temporary copy of the parser stack so that the initial context of the lookahead is not lost. Moreover, this exploratory parse is entirely syntactic in that the user's semantic actions are not performed because the effect of those actions during an erroneous parse might be undesirable. If the exploratory parse reaches a shift action, then the lookahead is syntactically acceptable, so the parser discards the temporary stack and resumes a full parse on the permanent stack. If the exploratory parse reaches a syntax error instead, then the parser discards the temporary stack and reports the syntax error. To build the list of syntactically accepted tokens for the syntax error, the parser performs exploratory parses for every token in the grammar. For each of these exploratory parses, the permanent stack, which still represents the initial context of the token, is copied to a temporary stack.

Because LAC requires many parse actions to be performed twice, it can have a performance penalty. However, not all parse actions must be performed twice. If a state has only one action and that action is a default reduction, then the parser does not need a lookahead. Thus, during a contiguous series of shift actions and such reduce actions, the parser never has to initiate an exploratory parse. Moreover, the most time-consuming tasks in a parse are often the file I/O, the lexical analysis performed by the scanner, and the user's semantic actions, but none of these are performed during the exploratory parse. In our experience, the performance penalty of LAC has proven insignificant for practical grammars.

As we explained in section 3.4, for traditional LR(1) parsing, IELR(1) is guaranteed to perform exactly the same parse actions as canonical LR(1) for any syntactically acceptable input.

For PSLR(1) parsing, we need to add PSLR(1)'s IELR(1) extension for this guarantee to hold. For traditional LR(1) parsing or PSLR(1) parsing, when we also add LAC, IELR(1) is guaranteed to perform exactly the same parse actions as canonical LR(1) for *any* input. If there are explicit error actions due to `%nonassoc`, then this guarantee only holds if LAC is added to canonical LR(1) as well.

3.5.3 Summary

There are three tasks for handling syntax errors, and all are ultimately performed by the PSLR(1) parser: detecting, reporting, and recovering. The pseudo-scanner's selection of a match to return to the parser upon encountering a syntax error has an important influence on the way in which the parser performs its three tasks. In this section, we have discussed the pseudo-scanner's mechanisms for selecting this match, and we have discussed the parser's mechanisms for detecting and reporting a syntax error. We also explained how LR(1) state merging, default reductions, and explicit parser error actions cause delayed syntax error detection and incorrect accepted token lists in syntax error messages. We then described LAC, an LR(1) parsing algorithm extension that we devised to fix these problems.

3.6 Whitespace and Comments

Grammars for programming languages like C and C++ usually do not specify the acceptable locations for whitespace and comments. The trouble is that the grammar symbols for whitespace and comments would have to be inserted between nearly every pair of tokens in the grammar, severely impairing the grammar's readability and maintainability. Thus, for traditional scanner-based parsing, whitespace and comments are usually processed entirely in the scanner, and the scanner does not return tokens for them to the parser.

In Figure 2.3c, we introduced the `YYLAYOUT` token as our solution for whitespace. In general, our PSLR(1) generator identifies any token whose name starts with or is equal to "`YYLAYOUT`" as a ***layout token***. The generated pseudo-scanner handles layout tokens in the same way a traditional scanner handles whitespace and comments. That is, the pseudo-scanner recognizes layout tokens in all parser states but discards them rather than returning them to the parser. Thus, layout tokens require that the definition of *acc* from Definition 3.3.2 be modified to add all layout tokens to

$acc(s_p)$ for every parser state s_p. Moreover, layout tokens require that the *pseudo_scan* function from Definition 3.2.16 be modified so that, after recognizing a layout token, *pseudo_scan* immediately restarts itself in order scan for a subsequent token to return to the parser.

As for lexical ties, the modifications to *acc* required by layout tokens can produce new pseudo-scanner conflicts that must be resolved with `%lex-prec`. For example, within C's literal string, layout tokens should not be recognized. Thus, if a user chooses to specify the syntax of C's literal string using a grammar rather than a single regular expression, he must use the lexical precedence operators "<<", "-<", and "<-" to declare lower precedence for layout tokens than for the conflicting tokens within that grammar. If the user needs to specify different lexical precedence rules for whitespace and for each of the various kinds of comments, he can take advantage of the ability to specify multiple layout tokens. For example, as we discussed in section 3.2.3, the regular expression for C's multiline comment benefits from the rule of shortest match, but the rule of longest match might be desirable for other layout tokens, such as the single-line comment.

PSLR(1)'s IELR(1) extension can be optimized for layout tokens by identifying any match for a layout token as an *always contribution* in a pairwise pseudo-scanner conflict. That is, because every layout token appears in $acc(s_p)$ for every parser state s_p, no amount of state splitting can remove a layout token from a syntactic context. Thus, it is *useless* for IELR(1) phase 2 to create annotations for pairwise pseudo-scanner conflicts in which a layout token's match has higher lexical precedence. In other words, the highest precedence match in this case is considered *split-stable*. We introduced the concepts of always contributions, useless annotations, and split-stability as part of our original IELR(1) algorithm [16].

For a non-layout token, PSLR(1) specifications can employ the ***token action directive***, denoted `%token-action`, to declare a semantic action that the pseudo-scanner should perform upon matching the non-layout token. That token action can construct the semantic value to be returned along with the non-layout token to the parser. For example, in a code transformer, the semantic value might simply contain the lexeme matched for the non-layout token. Token actions are useless for constructing semantic values for layout tokens because layout tokens are never returned to the parser because they are not syntactically acceptable according to the grammar. However, some applications, such as code transformers, need to preserve the text of whitespace and comments. Thus, we further extend *pseudo_scan* so that it accumulates lexemes from all contiguous layout tokens and then makes the accumulated layout text accessible in the token action for the following

non-layout token. The accumulated lexemes for the layout tokens appearing after the final token from the input are made accessible in the token action for the end token, #. Thus, for any non-layout token t, the user can write a token action that attaches the text for the preceding whitespace and comments as a property on the semantic value of t, which is returned to the parser. To avoid writing individual token actions for every non-layout token in the grammar, a PSLR(1) specification can employ a form of `%token-action` that specifies a single token action for all tokens declared to have semantic values of a common type. As future work, we are also considering providing a mechanism so that the accumulated value for layout tokens is not limited to lexemes. The user would be able to specify a token action to construct a semantic value of any type for a layout token, that value would then be made accessible in the token action for the next layout token, and so on until the token action for a non-layout token would be reached.

We foresee one significant limitation of layout tokens, and we outline a solution that we plan to implement as part of our future work. The problem is that the syntax of whitespace and comments may not always be regular, especially if comments are required to follow a strict documentation standard. Sometimes, even if the syntax is regular, it might simply be more convenient to specify the syntax with a context-free grammar. One possible solution is to allow the user to declare layout nonterminals that our PSLR(1) generator would insert implicitly between every pair of tokens throughout the grammar. However, because parser tables would be generated from this cluttered grammar, this approach would make parser conflicts difficult to debug.

Instead, we are planning to add a ***lexical directive***, denoted `%lex`, to declare a nonterminal to be a ***lexical nonterminal***. For each lexical nonterminal, our generator would construct a ***lexical parser*** that sits between the main parser and the pseudo-scanner. The lexical parser's job would be to parse the grammar for the lexical nonterminal and to return the lexical nonterminal as a token to the main parser. For any production for which the lexical nonterminal is the LHS, the RHS would be required to start and end with a token. These tokens would become the lexical nonterminal's ***start tokens*** and ***end tokens***. So that start tokens would always indicate the start of lexical nonterminals, no start token would be allowed to appear anywhere in the main grammar. So that end tokens would always indicate the end of lexical nonterminals, a lexical nonterminal's end tokens would not be allowed to appear anywhere else within that lexical nonterminal's grammar. For any main parser state s_p that accepts a lexical nonterminal, $acc(s_p)$ would be extended with that lexical nonterminal's start tokens in the eyes of the pseudo-scanner. Whenever the pseudo-scanner would

recognize a start token of a lexical nonterminal, the main parser would become inactive and the corresponding lexical parser would become active. When the pseudo-scanner would recognize an end token of the lexical nonterminal, the lexical parser would treat it as # and would return the lexical nonterminal as a token to the main parser, which would then become active. Any lexical nonterminal named like a layout token would be treated similar to the way layout tokens are currently treated. That is, the pseudo-scanner would recognize the lexical nonterminal's start tokens in any main parser state, and the lexical parser would discard the lexical nonterminal without returning it to the main parser. Thus, whitespace and comments would no longer be constrained to regular syntax.

Lexical nonterminals and lexical parsers would be useful for more than just whitespace and comments. For example, as discussed in section 2.3.1, the syntax for a block of C code embedded in a Yacc parser specification is not regular because of nested braces. Thus, the C block syntax needs to be specified with a grammar. However, when the parser encounters a C block's opening brace in a context where a C block is not syntactically acceptable, the parser then rejects the C block's opening brace, reports a syntax error for just the opening brace, and initiates syntax error recovery starting inside the C block. Specifying the C block syntax with a lexical nonterminal would allow an open brace to be treated as a start token. That start token could then become a fallback token so that the C block's lexical parser would have a chance to parse the C block in full before returning it to the main parser, which would then report it as a syntax error and initiate syntax error recovery after the C block.

We also plan to permit a hierarchy of lexical nonterminals and lexical parsers in case a token in a lexical nonterminal's grammar needs to be expressed in context-free form as well. During parsing, a stack would have to be maintained to record the current nesting of lexical parsers. The main parser would always be at the bottom of the stack. One interesting area of research would be to figure out how to extend modern syntax error recovery mechanisms to handle a stack of lexical parsers. It would likely involve popping the stack of lexical parsers whenever the syntax error recovery mechanism pops the start token for the corresponding lexical nonterminal.

3.7 Scoped Declarations

In our discussion of extensible languages in section 2.3.4, we mentioned the modern movement to specify sub-languages in separate modules in order to facilitate their composition into custom languages for specific domains. However, the lexical declarations we have discussed so far and the declarations supported by traditional parser generators like Yacc erode modularity because the effects of these declarations are global in scope. For example, if one sub-language's PSLR(1) specification declares traditional lexical precedence rules for conflicts between a particular pair of tokens, then the generator uses traditional lexical precedence rules to resolve conflicts between that pair of tokens in every sub-language. Any declaration of non-traditional lexical precedence rules for conflicts between those tokens appearing in another sub-language's PSLR(1) specification would then create a contradiction, and our PSLR(1) generator would report an error. In this section, we discuss two solutions to this problem that we plan to implement as part of our future work.

One solution to the scoping problem is to ensure that symbols from different sub-language specifications always have different names even if those symbols are otherwise identical. This solution could be automated if the user were able to declare different namespace names for different sub-languages. For example, assume two different sub-language specifications use the same token `'&'`, but the namespace for one sub-language is `foo`, and the namespace for the other is `bar`. While composing the sub-languages' specifications into the composite language specification, our PSLR(1) generator could automatically rename `'&'` to `foo.'&'` throughout the first sub-language's specification and to `bar.'&'` throughout the second's. Thus, every declaration in the computed composite language specification would be effectively scoped to the sub-language to whose namespace the declaration's operands belonged.

While the namespace solution might be reasonable in some cases, there are a couple of potential problems. First, it would duplicate sub-language grammar symbols, even those for which there are no contradictions among the sub-languages. This duplication would unnecessarily expand the size of composite grammars and thus the parser tables, impairing efficiency and complicating the task of debugging. Second, in the case of subtle sub-languages, scoped declarations might be needed in places where the proper partitioning of the grammar symbols into namespaces is not obvious.

The template argument list example from C++0x, which we described in section 2.3.3, is an example of a subtle sub-language where a namespace partitioning is not obvious. Consider

again the example C++0x sentence in Figure 2.6a. Recall that the sub-languages L_c and L_p require that the `'>>'` token have lexical precedence over the `'>'` token as expected according to the traditional rule of longest match. However, the sub-language L_t requires the reverse precedence so that nested template argument lists can be terminated without inserting unexpected whitespace into ">>". Figures 2.6b and 2.6c show the contradictory `%lex-prec` declarations that are thus required for these sub-languages. To avoid the contradiction, each declaration must be scoped to its own sub-language. However, the grammars of L_t and L_p reuse major portions of the grammar of L_c in a mutually recursive manner, and the tokens `'>'` and `'>>'` appear throughout all of the sub-languages' grammars. Thus, it is not obvious how the C++ grammar's symbols should be partitioned into namespaces.

Another possible solution for the scoping problem requires the user to identify the start symbols for the sub-languages' grammars. These start symbols are specific occurrences of nonterminals in the RHS's of grammar productions, and each sub-language may have more than one start symbol. In the case of C++, identifying the start symbols is more straightforward than partitioning the grammar symbols into namespaces. For example, in the production shown in Figure 2.6d, the symbol `template_argument_list_opt` is a start symbol for L_t. In the production shown in Figure 2.6e, `expression` is a start symbol for L_p.

To implement scoped declarations based on sub-language grammar start symbols, our PSLR(1) generator would recursively record the scope of each item in each parser state based on the nonterminal from which the item was derived. For some items in some parser states, multiple scopes would be recorded because left context is not always powerful enough to determine a single derivation in bottom-up parsers. A parser state would have a ***scope conflict*** if contradictory declarations applied because of the presence of multiple scopes. As for pseudo-scanner conflicts and parser conflicts, some scope conflicts could result from the merging of parser states. To eliminate such scope conflicts, we could extend the LALR(1) parser table construction in IELR(1) phase 0 so that it would not merge states whose items are from different scopes. However, merging states whose items are from different scopes would not always produce a scope conflict because there might not happen to be any contradictory declarations that apply, so this approach might result in parser tables that are larger than necessary. If that proves to be problematic in practice, we could instead extend IELR(1) to detect scope conflicts that result from the merging of parser states and then to eliminate these conflicts by splitting states. Because the canonical LR(1) algorithm does not

track scopes and merges states that are identical in all other ways, either the LALR(1) extension or IELR(1) extension could split even canonical LR(1) states, potentially producing more powerful tables than canonical LR(1).

Notice how we combined the "−<" lexical precedence operator with scoping in our C++ example. While the declaration in Figure 2.6b is equivalent to the traditional rule of longest match, the declaration in Figure 2.6c has no traditional equivalent. Its purpose in our example is to prevent the `'>>'` token from being recognized within L_t except within L_p. In non-composite languages, it would be schizophrenic to include a token in a grammar and then add a declaration that prevents its recognition entirely. However, in this case, the grammar of L_t, where `'>>'` must not be recognized, reuses portions of the grammar of L_c, where `'>>'` must be recognized. We predict that one of the greatest powers of our non-traditional lexical precedence operators is to disable tokens in favor of other tokens with similar lexemes for the sake of sub-grammar reuse among different syntactic contexts.

Upon a syntax error, a pseudo-scanner could employ either grammar symbol namespaces or sub-language start symbols as a means to indicate the scope from which to select a match to return to the parser. That is, multiple fallback rows could be added to the *scanner_accepts* table, one per scope. When the pseudo-scanner is unable to select a match for a token from the current parser state's row, it normally looks in the fallback row that includes all tokens from the composite grammar. However, it could instead look for matches using the current scope's fallback row. Thus, the selected match might be more appropriate for the current sub-language, and so the parser might produce a more intuitive syntax error message and initiate a more successful syntax error recovery.

3.8 Summary

In this chapter, we have explained the design and implementation of our PSLR(1) generator tool, which generates a pseudo-scanner and a minimal LR(1) parser from a unified scanner and parser specification called a PSLR(1) specification. Unlike a traditional scanner, a pseudo-scanner examines the current parser state to determine what tokens are acceptable in the current syntactic left context, thus automating the tracking of sub-language transitions in a composite language. While this behavior automatically eliminates many of the scanner conflicts among tokens from different sub-languages, PSLR(1) specifications can employ a lexical precedence directive with which the user

can resolve the remaining scanner conflicts more carefully and declaratively than with a traditional scanner generator like Lex. PSLR(1) specifications can also employ a directive to declare that tokens, such as keywords and identifiers, are lexically tied and so should never be confused with one another regardless of where each is syntactically acceptable. We have explained how the generated pseudo-scanner selects a match to return to the parser when there are no syntactically acceptable tokens. We have introduced LAC, an LR(1) parsing algorithm extension that eliminates delayed syntax error detection and incorrect accepted token lists in syntax error messages for a PSLR(1) parser or traditional LR(1) parser that employs parser state merging, default reductions, or error actions for resolving S/R conflicts. We have described an extension to our original minimal LR(1) algorithm, IELR(1), to eliminate other incorrect PSLR(1) behavior induced by parser state merging. Finally, we have described mechanisms for handling the specification of whitespace and comments and the scoping of declarations to specific sub-languages. In this way, our PSLR(1) generator permits the user to specify the lexical and syntactic analysis of composite languages in a more careful, declarative, and modular fashion without the need to specify the start conditions and other ad-hoc code required by traditional scanner generators.

Chapter 4

Studies and Evaluation

In order to evaluate the benefits of PSLR(1), we examine four case studies. In section 4.1, we describe our first case study, PSLR(1) Bison's own internal parser. In section 4.2, we describe our second case study, an SQL parser written by John R. Levine for his text book, *flex & bison*. In section 4.3, we describe our third case study, ISO C99, which we choose as a representative of the C family of languages. In section 4.4, we describe our final case study, the template argument list specification from Figure 2.3c. In section 4.5, we present the results we have collected from our case studies.

4.1 PSLR(1) Bison

We have implemented our PSLR(1) generator as an extension of Bison. In this paper, we refer to our extended Bison as PSLR(1) Bison. Internally, Bison employs a traditional scanner-based LR(1) parser for analyzing its input parser specifications. Thus, we initially implemented PSLR(1) Bison to employ traditional scanner-based LR(1) internally as well. In both cases, the internal scanner is generated by Flex, and the internal parser is generated by Bison itself.

The language of Bison parser specifications is a challenging composition of multiple sub-languages as we described for Yacc in section 2.3.1. Because PSLR(1) Bison accepts a unified scanner and parser specification, PSLR(1) Bison must recognize regular expressions as yet another sub-language. Thus, both Bison's and PSLR(1) Bison's internal parsers are themselves candidates for PSLR(1).

For our first case study, we have implemented PSLR(1) Bison's internal parser a second time using PSLR(1) instead of traditional scanner-based LR(1). We then collected and compared readability and maintainability statistics for the two versions. We describe the results in section 4.5.

To ensure that our comparisons between the two versions of PSLR(1) Bison's internal parser are meaningful, it is vital to verify that the versions exhibit equivalent behavior. The Bison distribution contains a robust test suite with 240 test groups as of the latest release, Bison 2.4.1. The test suite for Bison 2.5, which is still under development, currently has 289 test groups. As we developed PSLR(1) Bison with a traditional scanner-based LR(1) parser internally, we extended the latter test suite to a total of 381 test groups in order to verify the ability to parse a user's PSLR(1) specification and to generate a correct PSLR(1) parser. After converting PSLR(1) Bison to use PSLR(1) to generate its own internal parser, we then re-ran this extended test suite in order to verify that equivalent behavior had been fully retained.

Based on the extended test suite, the PSLR(1) version of PSLR(1) Bison's internal parser successfully emulates the traditional scanner-based LR(1) version in almost all cases. All differences in behavior are related to syntax error handling. First, the PSLR(1) version employs sub-grammars involving many tokens in order to recognize some elements of the lexical syntax that the traditional scanner-based LR(1) version recognizes with single tokens. Thus, some of the behavioral differences are simply the names of the unexpected or expected tokens that the parsers report in syntax error messages. Second, the traditional scanner-based LR(1) version depends entirely on ad-hoc C code to handle syntax errors at the lexical level. In the PSLR(1) version, all levels of the syntax benefit from the same automated syntax error handling mechanisms, which we described in section 3.5, so syntax error reporting and recovery are generally more uniform. Nevertheless, both versions of the parser detect the same initial syntax error for every input parser specification in the test suite. Moreover, none of the behavioral differences detected by the test suite are evidence of incorrect behavior for either version.

4.2 Levine SQL

In his text, *flex & bison*, John R. Levine develops Flex and Bison specifications for translating a subset of SQL (Structured Query Language) into RPN (Reverse Polish Notation) [24]. In this paper, we refer to that subset of SQL as *Levine SQL*. As Levine explains in his text, he has made

his Flex and Bison specifications available online [25]. We downloaded our copy in January 2010. The download actually contains three versions of those specifications. The most robust version that does not employ GLR (Generalized LR) is contained in the Flex specification file `sql/lpmysql.l` and Bison specification file `sql/lpmysql.y`.

For our second case study, we used `sql/lpmysql.l` and `sql/lpmysql.y` as a basis to develop a traditional scanner-based LR(1) parser for Levine SQL. We then converted those specifications to PSLR(1). As for our first case study, we collected and compared readability and maintainability statistics for the traditional scanner-based LR(1) version versus the PSLR(1) version, and we describe the results in section 4.5.

The only differences between Levine's original specifications and the traditional scanner-based LR(1) parser that we employ in this case study are a few corrections that we found necessary to permit reasonable comparisons with our PSLR(1) version of the parser. First, in `sql/lpmysql.l`, the action for the scanner's `COUNT` token invokes Flex's `unput` function before accessing the Flex variable `yytext`. According to Flex's documentation, `unput` by default corrupts the contents of `yytext`, so `yytext` must be copied before invoking `unput`. Indeed, we found that this bug sometimes corrupts the diagnostics printed by a semantic value destructor defined in `sql/lpmysql.y`. Second, in `sql/lpmysql.l`, the action for newline does not correctly update `yylloc`, which is the global variable in which the parser's lookahead location is stored. As a result, the parser sometimes reports syntax errors at locations that do not exist in the input SQL. Third, both `sql/lpmysql.l` and `sql/lpmysql.y` contain a few superficial problems for the strict compiler settings we use. For example, there's an unused variable, a missing prototype, a missing standard include, and a use of the library function `strdup`, which is not defined by ISO C99.

As for our first case study, it is vital to verify that the traditional scanner-based LR(1) version and the PSLR(1) version of the parser exhibit equivalent behavior in order to ensure that our comparisons between the two versions are meaningful. Unlike our first case study, there is no existing test suite for the traditional scanner-based LR(1) version. Instead, we downloaded a copy of the SQL Test Suite, Version 6.0, from the website of NIST (U.S. National Institute of Standards and Technology) [7]. To form our Levine SQL test suite, we extracted from that download the 379 files whose names match the pattern `sql/*.sql`, each of which contains a series of SQL statements and SQL comments. We then wrote a script that compares the output of the traditional scanner-based LR(1) version and the PSLR(1) version of the Levine SQL parser for each of these files. That is, we

used the traditional scanner-based LR(1) version as a test oracle for the PSLR(1) version.

Because Levine SQL is only a subset of SQL, both versions of the parser report syntax errors for many of the SQL statements in our test suite. However, Bison's error recovery mechanism enables the parsers to continue parsing the remaining statements in each file so that the output from the parsers can be compared for those statements as well. For our entire Levine SQL test suite, both versions of the Levine SQL parser successfully parse 3675 SQL statements, report 11671 syntax errors, and produce identical output, which includes RPN translations and syntax error messages.

4.3 ISO C99

As we discuss in section 4.5, one of the goals of our first two case studies is to explore the effect of PSLR(1)'s IELR(1) extension. In our third case study, we explore the effect of that extension for the syntax that is common among the C family of languages, which includes C, C++, and Java. We ignore the newest complexities in the C family, such as the evolving C++0x template sub-language issue described in section 2.3.3.

Without the newest complexities, the C family syntax is unique among our case studies in that it is designed relatively well for traditional scanner-based LR(1). Specifically, unlike the language of PSLR(1) Bison parser specifications, sub-languages from the point of view of the scanner are usually just comments and literals, which are regular sub-languages. Unlike Levine SQL, all keywords are reserved words, and so their lexemes are recognized as keywords in all syntactic contexts except in comments and literals, where most characters are recognized as generic text. These differences mean that sub-languages in the C family syntax are less likely to share sub-grammars and, if they do share sub-grammars, they are less likely to require different tokenizations of the same character sequences when recognizing those sub-grammars. As a result, sub-languages are less likely to have parser states with common cores such that, once those states are merged by LALR(1), the pseudo-scanner can no longer adequately distinguish the sub-languages. Thus, by examining the C family syntax, the goal of this case study is to explore whether PSLR(1)'s IELR(1) extension splits parser states when there is no need to do so.

Rather than examining all languages from the C family individually, we choose ISO C99 as the most straight-forward representative. Thus, for our third case study, we converted the lexical grammar and the phrase structure grammar from Annex A of the ISO C99 standard into a PSLR(1)

specification [10]. However, unlike our first two case studies, we have not attempted to develop this PSLR(1) specification into a fully functional application. We have developed it only far enough to demonstrate the effects of PSLR(1)'s IELR(1) extension. We assert that the resulting specification represents a realistic stage of development that any PSLR(1) user might encounter while developing an ISO C99 parser, and so it is important to show that unnecessary state splitting and the conflict duplication that can result are avoided in order to facilitate further development. In the remainder of this section, we support our assertion by detailing the steps we took to convert the ISO C99 grammars to PSLR(1).

Our first step in the conversion was to reproduce in PSLR(1) form a faithful copy of the ISO C99 lexical grammar and phrase structure grammar. The only elements of these grammars that we omitted were the keyword *_Imaginary*, which is never used in the phrase structure grammar, and the preprocessor, which is typically handled by a separate parser. Other than those omissions, we copied the phrase structure grammar as our PSLR(1) grammar, and we defined tokens for the keywords listed in section A.1.2 of the standard, for the punctuators listed in section A.1.7, and for the remaining symbols that the lexical grammar defines and that the phrase structure grammar uses: *identifier*, *constant*, *string-literal*, and *enumeration-constant*. We defined named regular expressions for all other symbols from the lexical grammar. Finally, we defined layout tokens for whitespace, multiline comments, and single-line comments based on section 6.4 of the standard. For the resulting specification, PSLR(1) Bison reported many lexical tie candidates and pseudo-scanner conflicts. Because the way in which such errors are resolved can have a significant influence on the behavior of PSLR(1)'s IELR(1) extension, the purpose of the remaining conversion steps was to find a reasonable way to resolve those errors.

Our second step in the conversion to PSLR(1) was to simplify the task of resolving the lexical tie candidates and pseudo-scanner conflicts by eliminating the token *enumeration-constant*. The ISO standard's lexical grammar defines the syntax of *enumeration-constant* as exactly the syntax of the token *identifier*, and the occurrences of *enumeration-constant* in the phrase structure grammar are in syntactic contexts where *identifier* can never appear. Thus, we were able to simplify our PSLR(1) specification without affecting the C99 syntax by merely substituting these occurrences of *enumeration-constant* with *identifier*. Also, the syntax of the token *constant* explicitly includes the syntax of *enumeration-constant* and thus of *identifier*, but, according to the phrase structure grammar, *identifier* can be recognized in every syntactic context in which

constant can be recognized. This creates a syntactic ambiguity that must be resolved via semantics. Thus, without affecting the C99 syntax, we removed *enumeration-constant* from the definition of *constant*, and we assumed that semantic actions would later be added to perform a symbol table lookup to recognize when an *identifier* should be treated semantically as a *constant* in these syntactic contexts. However, actually implementing semantics is beyond the scope of this case study.

Together, the above modifications left *enumeration-constant* completely unused, so we removed its definition from the specification. Interestingly, *enumeration-constant* is the only symbol that occurs in a production RHS in the ISO standard's phrase structure grammar, that never occurs as a production LHS in the phrase structure grammar, that is defined by the lexical grammar, but that is not listed as a token in section A.1. That is, the ISO standard uses *enumeration-constant* not as a real token but as an alias for *identifier* in certain syntactic contexts. As described above, our second step in the conversion to PSLR(1) simply eliminated this aliasing. By doing so, it also eliminated the need to resolve many redundant errors. That is, before the second step, PSLR(1) Bison reported many of the same lexical tie candidates and pseudo-scanner conflicts for *enumeration-constant* as for *identifier*. After the second step, it only reported them for *identifier*.

Our final step in converting the ISO C99 grammars to PSLR(1) was to resolve all remaining pseudo-scanner conflicts and lexical tie candidates by adding a set of lexical declarations. For conciseness, we declared symbol sets for keywords and punctuators. We lexically tied all pairs of conflicting punctuators. The punctuator `'.'` conflicts with *constant*, and *identifier* conflicts with *constant*, *string-literal*, and all keywords, so we lexically tied all of those tokens. The punctuator `'/'` conflicts with the layout tokens for multiline and single-line comments, so we lexically tied them as well. Using the rule of longest match, we resolved all length conflicts among all tokens mentioned in this paragraph. Finally, we resolved the identity conflict between *identifier* and every keyword by giving the keyword higher precedence.

4.4 Template Argument Lists

Our final case study is the PSLR(1) specification from Figure 2.3c. We refer to the language defined by this specification as the Template Argument Lists language. The purpose of this case study is to evaluate the success of PSLR(1)'s IELR(1) extension in overcoming the PSLR(1)-relative inadequacy that we described in section 3.4.

4.5 Results

In this section, we present the results of our cases studies. In the tables in this section, we identify each case study by the language it examines: PSLR(1) Bison, Levine SQL, ISO C99, or Template Argument Lists. As explained in the previous sections, all of our case studies involve PSLR(1) parsers, but only our first two case studies compare their PSLR(1) parsers with equivalent traditional scanner-based LR(1) parsers. For conciseness, we frequently abbreviate the phrase "traditional scanner-based LR(1)" as merely "traditional". In section 4.5.1, we examine the grammars and parser tables for our case studies, and we discuss the parser table changes made by PSLR(1)'s IELR(1) extension. In section 4.5.2, we assess the readability and maintainability of the traditional specifications versus the PSLR(1) specifications.

4.5.1 Grammars and Parser Tables

Table 4.1 measures the sizes of the traditional and PSLR(1) grammars for each of our case studies. The PSLR(1) grammar for PSLR(1) Bison's internal parser is larger than its traditional grammar because many of the tokens from the traditional scanner specification do not have a regular syntax, and so we converted those tokens to nonterminals and defined them in context-free form in the PSLR(1) grammar. The grammars for Levine SQL are the largest among our case studies, but the lexical syntax is so simple that no new nonterminals are required for PSLR(1). Interestingly, the PSLR(1) grammar for Levine SQL has three less productions than the traditional grammar. The reason is that those three productions depend on tokens that the traditional parser specification defines but that the traditional scanner specification does not define. While Flex and Bison do not communicate in order to detect such inconsistencies between traditional scanner and parser specifications, PSLR(1) Bison reports errors for tokens that appear in the grammar but that have not been assigned a regular expression. Thus, we were forced to remove the useless productions from the PSLR(1) grammar. PSLR(1)'s ability to enforce such strictness is one advantage of its unification of the scanner and parser specification formalisms.

In Table 4.2, we count the parser states and parser conflicts for the traditional and PSLR(1) specifications for each of our case studies. As expected based on the grammar sizes, converting the traditional specification to the PSLR(1) specification without extending IELR(1) for PSLR(1) increases the size of the parser tables for PSLR(1) Bison's internal parser, but it slightly decreases

Language	Specification	$\|T\|$	$\|V\|$	$\|T \cup V\|$	$\|P\|$
PSLR(1) Bison	Traditional	71	35	106	122
	PSLR(1)	117	115	232	294
Levine SQL	Traditional	252	73	325	303
	PSLR(1)	277	73	350	300
ISO C99	PSLR(1)	90	86	176	235
Tmplt. Arg. Lists	PSLR(1)	6	5	11	7

Table 4.1: Grammar Sizes. These counts measure the size of each case study's grammar $G = (V, T, P, S)$, such that V is the set of nonterminals, T is the set of terminals or tokens, P is the set of productions, and S is the start symbol. These counts include the productions and nonterminals that Bison generates implicitly for mid-rule actions.

Traditional Specification with IELR(1)

Language	States	S/R	R/R
PSLR(1) Bison	187	0-0	0-0
no prec/assoc	187	0-0	0-0
Levine SQL	626	0-0	0-0
no prec/assoc	626	480-0	0-0

PSLR(1) Specification

Language	States			S/R			R/R		
	IE	PS	Canon	IE	PS	Canon	IE	PS	Canon
PSLR(1) Bison	383	383	1356	0-0	0-0	0-0	0-0	0-0	0-0
no prec/assoc	383	383	1356	2-0	2-0	2-0	0-0	0-0	0-0
Levine SQL	610	663	8009	0-0	0-0	0-0	0-0	0-0	0-0
no prec/assoc	610	663	8009	480-0	960-480	21600-21120	0-0	0-0	0-0
ISO C99	391	391	1773	10-0	10-0	17-7	10-0	10-0	12-2
Tmplt. Arg. Lists	18	19	22	0-0	0-0	0-0	0-0	0-0	0-0

Table 4.2: Parser Tables. In the first table, we describe the parser tables that IELR(1) generates for the traditional specifications for our first two case studies. In the second table, we describe the parser tables that IELR(1), IELR(1) with its extension for PSLR(1), and canonical LR(1) generate for the PSLR(1) specifications for all our case studies. We report the number of states and the number of parser conflicts left unresolved by the user. We also show adjustments to account for such unresolved parser conflicts that are perfectly duplicated among states with identical cores.

Language	Reveals	States	Tokens
PSLR(1) Bison	0	0	0
Levine SQL	25	25	1
ISO C99	0	0	0
Tmplt. Arg. Lists	1	1	1

Table 4.3: Lexical Reveals. For each of our case studies, this table reports the number of lexical reveals caused by extending IELR(1) for PSLR(1), the number of parser states containing lexical reveals, and the number of unique tokens for which there are lexical reveals.

the size of the parser tables for Levine SQL. Because some of our case studies include precedence and associativity declarations that resolve their parser conflicts, we also describe their parser tables when generated without these declarations in order to better represent the complexity of the specification. For example, in the table for traditional specifications, the "no prec/assoc" row beneath the "Levine SQL" row reveals that 480 S/R conflicts are resolved by the user.

For PSLR(1) specifications, Table 4.2 also compares the parser tables generated by IELR(1), by IELR(1) with its extension for PSLR(1), and by canonical LR(1). Only Levine SQL and Template Argument Lists experience state splitting when adding PSLR(1)'s IELR(1) extension. For Template Argument Lists, the IELR(1) tables are identical to the LALR(1) parser tables, which are shown in the second column of Table 2.1, and canonical LR(1) requires 4 more states, as shown in the first column of Table 2.1. However, PSLR(1)'s IELR(1) extension requires only 1 additional state, which is equivalent to canonical LR(1) state 18. For Levine SQL, when PSLR(1)'s IELR(1) extension splits states and duplicates conflicts as a result, the number of states increases by a factor of 1.09, and the number of S/R conflicts increases by a factor of 2. However, when canonical LR(1) is used instead of the extended IELR(1), the number of states increases by a factor of 13, and the number of S/R conflicts increases by a factor of 45. Thus, our previous conclusion comparing IELR(1) and canonical LR(1) continues to hold when IELR(1) is extended for PSLR(1): canonical LR(1) can severely worsen the developer's burden of investigating parser conflicts in order to resolve them [16].

Table 4.3 describes the benefit of PSLR(1)'s IELR(1) extension for each of our case studies. As discussed above, PSLR(1)'s IELR(1) extension does not split any states for PSLR(1) Bison's internal parser or for ISO C99, so these case studies show no benefit. The lack of benefit for ISO C99 comes as no surprise given the reasons we discussed in section 4.3. The reasons that PSLR(1) Bison's internal parser does not benefit are less obvious but are similar. Stated simply, in the PSLR(1) specification language, among sub-languages that share sub-grammars, character sequences tend to be tokenized in the same manner when recognizing those sub-grammars, so there is little chance that LALR(1) state merging can create PSLR(1)-relative inadequacies. While neither the PSLR(1) Bison nor the ISO C99 case study proves that, in general, PSLR(1)'s IELR(1) extension avoids state splitting for cases when state splitting would offer no benefit, they do provide evidence of its ability to recognize such cases.

As expected based on the state splitting shown in Table 4.2, Levine SQL and Template Argument Lists do benefit from PSLR(1)'s IELR(1) extension. In order to measure the benefit, we

define the concept of a ***lexical reveal***, and in Table 4.3 we count the lexical reveals. For example, for Template Argument Lists, we can see the sole lexical reveal by examining the parser tables in Table 2.1. For LALR(1) state 1, assuming the rule of longest match, the match (“>”,'>') is never recognized for the input character sequence “>>” because the match (“>>”,'>>') has higher precedence. However, PSLR(1)'s IELR(1) extension splits LALR(1) state 1 into canonical LR(1) states 1 and 18, so, from the perspective of the viable prefixes of canonical LR(1) state 18, the match (“>”,'>') is revealed because the token '>>' is no longer accepted. Because a token can match an infinite number of lexemes, some parser tables can experience an infinite number of revealed matches, so we define the number of lexical reveals as the number of tokens for which at least one match is revealed instead of as the number of revealed matches.

If matches are revealed for the same token in more than one parser state after state splitting, we count the lexical reveals once per state because each state represents a different set of viable prefixes. For example, for Levine SQL, PSLR(1)'s IELR(1) extension reveals matches for only one token. However, those matches are revealed for that token in 25 different parser states after state splitting, so we count 25 lexical reveals. To explain the source of these lexical reveals, we start by examining the traditional scanner and parser. In the grammar, the tokens `BETWEEN` and `AND` each appear in only one production, and it is the same production:

```
expr:  expr BETWEEN expr AND expr %prec BETWEEN
```

The token `AND` matches the lexeme “`AND`”, but the sub-grammar for the nonterminal `expr` contains the token `ANDOP`, which can also match the lexeme “`AND`”. While the parser is recognizing the sub-grammar of `expr`, the scanner must have some way to decide whether to recognize “`AND`” as `AND` or as `ANDOP`. The traditional scanner's solution is to enter a new start condition after the token `BETWEEN` and to leave that start condition upon reaching `AND`. While in that start condition, it recognizes “`AND`” as `AND`. Outside of that start condition, it recognizes “`AND`” as `ANDOP`. Thus, from the point of view of the scanner, `BETWEEN` and `AND` delimit a sub-language. PSLR(1)'s equivalent solution is (1) to depend on the basic behavior of the pseudo-scanner to avoid recognizing `AND` anywhere except in the `BETWEEN` sub-language because that's the only place `AND` is syntactically acceptable and (2) to declare `AND` with higher lexical identity precedence than `ANDOP`. However, because the sub-grammar for `expr` is shared by the main sub-language and the `BETWEEN` sub-language, some parser states with common cores are shared as well. When those parser states and

thus their lookahead sets are merged, the two sub-languages become indistinguishable, and so the match ("AND",ANDOP) becomes hidden by ("AND",AND) in all uses of expr including 25 different syntactic contexts within the main sub-language. PSLR(1)'s IELR(1) extension splits states so that the pseudo-scanner can distinguish between the sub-languages and reveal the match for ANDOP in those 25 syntactic contexts.

4.5.2 Readability and Maintainability

Tables 4.4, 4.5, and 4.6 present our readability and maintainability statistics for the traditional versus PSLR(1) specifications of our first two case studies. The first column in the tables of Table 4.4 identifies the specification examined by each row. Each of the remaining columns presents a different assessment of the size of the specifications. For example, the second column simply counts the number of lines of code, but the third column counts characters instead. We explain the meaning of each size assessment in detail later in this section. For every size assessment, the first two rows of each table give the sizes for the traditional parser and scanner specifications. Except in the last column, the traditional scanner specification sizes for PSLR(1) Bison's internal parser include two header files written in C that we were able to discard when converting to PSLR(1). No files generated by Flex, Bison, or PSLR(1) Bison are included in any of the sizes for any of the specifications because those files are not maintained directly by the specification developer.

The third row gives the size for the unified PSLR(1) specification. The goal of PSLR(1) is to simplify the specification of the scanner, so we wish to compare the traditional scanner specification with the scanner portion of the unified PSLR(1) specification. However, when converting to PSLR(1), the merging of scanner and parser specifications does not leave exact boundaries between these specifications. Fortunately for our analysis, for both case studies, there were only a few changes to the code from the traditional parser specifications when converting to PSLR(1). Thus, the fourth row estimates the size of the PSLR(1) scanner specification by subtracting the size of the traditional parser specification from the size of the unified PSLR(1) specification. In each of the first four rows, we also show the C content of each specification. Thus, the fourth row estimates the C content in the PSLR(1) scanner specification by subtracting the size of the C code in the traditional parser specification from the size of the C code in the unified PSLR(1) specification and then dividing by the estimate of the total size of the PSLR(1) scanner specification. The fifth row presents the ratio of the estimated PSLR(1) scanner specification size to the traditional scanner specification size, and

PSLR(1) Bison

Specification	Lines	Chars	Norm. Chars	Norm. Chars (No External C)
Traditional Parser	836, 45% C	23276, 61% C	16190, 61% C	13514, 54% C
Traditional Scanner	1382, 65% C	39483, 71% C	21234, 83% C	14682, 76% C
PSLR(1) Scanner & Parser	1503, 44% C	46610, 56% C	32424, 55% C	26143, 44% C
PSLR(1) Scanner	667, 43% C	23334, 50% C	16234, 48% C	12629, 33% C
Scanner Ratio	48%	59%	76%	86%
Scanner C Ratio	31%	41%	44%	37%

Levine SQL

Specification	Lines	Chars	Norm. Chars	Norm. Chars (No External C)
Traditional Parser	1005, 13% C	27812, 39% C	24060, 41% C	22187, 36% C
Traditional Scanner	368, 7.6% C	10302, 58% C	9001, 60% C	8672, 59% C
PSLR(1) Scanner & Parser	1259, 13% C	36347, 33% C	28599, 39% C	26512, 34% C
PSLR(1) Scanner	254, 15% C	8535, 16% C	4539, 27% C	4325, 23% C
Scanner Ratio	69%	83%	50%	50%
Scanner C Ratio	140%	22%	23%	20%

Table 4.4: Specification Sizes. For our first two case studies, these tables show specification sizes and C content. To estimate the PSLR(1) scanner, we subtract sizes for the traditional parser from the unified PSLR(1) parser and scanner. The ratios in the last two rows show the PSLR(1) scanner divided by the traditional scanner. To normalize character counts, we removed all comments, and then we reduced contiguous whitespace to a single space. External C is C code that is not embedded in action declarations and that could have been moved to an external C library or include.

---

the final row presents the same ratio for the size of the C code.

We include line counts in our results because line counts are the most common first assessment of size that software engineers tend to make. However, this assessment can be very unrealistic. The first problem we address is that line counts are too dependent on the number of characters that the developer of each specification tended to write per line. The line counts for the C code extracted from these specifications are impacted especially severely by this problem because passages of C code are often contained between braces within a single line. When those passages are concatenated together to compute the full size of the C code, no newline is seen until a passage of C code that contains multiple lines is encountered, so lines of C code tend to be very long. Of course, we could have appended a newline after the last passage of C code from each line, but this would have produced the opposite problem of extremely short lines of C code. A better assessment of code size that avoids this problem is simply a count of characters as presented in the third column of each

table in Table 4.4. In some cases, this makes a dramatic difference in the numbers. For example, the scanner C ratio for Levine SQL falls from the unrealistic 140% to the realistic 22%.

A simple character count is still not the most realistic assessment of code size. Like line counts, it is too dependent on code formatting and on the amount of comments the developer of each specification tended to write. Thus, for our next assessment of code size, we normalized our character counts in two steps. First, we removed all Flex, Bison, PSLR(1) Bison, and C comments from the specifications. Second, a single whitespace character can be syntactically significant in the specifications, but there is no syntactic difference between multiple contiguous whitespace characters and a single whitespace character except in literals. Thus, we reduced all passages of contiguous whitespace not appearing in literals to a single whitespace character. The fourth column of each table in Table 4.4 presents the resulting sizes.

For our final assessment of specification sizes, we consider that a developer's choice of when to place C code in a specification file and when to place C code in an external include or library is sometimes arbitrary. For example, the specifications for PSLR(1) Bison's internal parser are a small portion of the total volume of code in PSLR(1) Bison, and much of the additional code is used as a library by those specifications. However, some of the C code placed within those specifications could have just as easily been placed in an external include or library so that it would not dilute our measurements. We use the term *external C* to refer to all C code that is placed or could have been placed in an external file and that thus does not appear in any kind of scanner or parser action. For the last column of each table in Table 4.4, we ignore all external C. In this way, the last column focuses on the C code that is inherently mixed within the scanner and parser formalisms and that thus can complicate the task of comprehending the language specified by those formalisms.

For producing a realistic assessment of specification sizes, there are merits to counting normalized characters both with and without external C. Keeping external C has the advantage of detecting changes in the amount of external C when converting to PSLR(1), but external C that does not change dilutes the ratios. Fortunately, the difference between these assessments is not large for our case studies. For PSLR(1) Bison's internal parser, converting from traditional to PSLR(1) reduces the total size of the scanner specification by nearly one fourth of its size when external C code is included in the assessment. For Levine SQL, the total size of the scanner specification is reduced by half under either assessment. More interestingly, for both case studies, we have clearly achieved our goal of reducing the need to write ad-hoc code in a general-purpose programming language

like C by converting that code to formalisms like grammars, regular expressions, and precedence declarations, all of which are tailored for specifying scanners, parsers, or, more generally, languages. For PSLR(1) Bison, the C code in actions embedded within such language formalisms was reduced to 37% of its original size, and for Levine SQL it was reduced to 20% of its original size.

So far, we have assessed the maintainability and readability of traditional versus PSLR(1) specifications by comparing only total code size and amount of C code. However, we consider that a reduction in C code to 20% of the original size is pointless if that C code's complexity increases dramatically. Thus, in Table 4.5, we assess the complexity of the C code in the actions embedded in the specifications. We do not consider external C code because only its interface need be understood when attempting to understand the language or semantics being specified.

To assess the complexity of C code in the embedded actions, Table 4.5 counts the non-linear control structures, variables, and conditions that code contains. By non-linear, we mean we don't count control structures appearing in idioms with linear control flow, such as:

```
#define FUNCTION        \
  do {                  \
    /* function body */ \
  } while (0)
```

The first column in each table in Table 4.5 identifies the specification examined by each row. The second column shows that no specification contains a goto statement. For the traditional scanner specifications, the third column counts start conditions, which we assert are analogous to goto labels while invocations of Flex's `BEGIN` macro, which specifies start condition transitions, are analogous to gotos. The fourth and fifth columns count global variables and variables local to the scanner function, `yylex`. In both cases, we count variables defined by the user, but we do not count variables whose definitions are generated automatically. In the sixth column, we count `for`, `while`, and `do-while` loops. In the seventh column, we count each `if`, `else if`, and `else`. However, we also count the ternary operator as an `if` and an `else`. All `switch` statements in the specifications happen to have a simple structure that could easily have been rewritten as a single chain of `if`, `else if`, and `else`. Thus, we count any group of `case` labels associated with a single block of code as either an `if` or `else if`. We count the `default` label as an `else`.

In the final column of each table in Table 4.5, we count the number of conditions appearing in control structures in embedded C code. For example, the following `if` contains two conditions, `A` and `B`:

PSLR(1) Bison

Specification	Gotos	Start Conds.	Global User Vars.	`yylex` User Vars.	Loops	`if`, `else if`, `else`	conds.
Traditional Parser	0	–	6	–	8	8	17
Traditional Scanner	0	21	10	8	0	48	40
PSLR(1) Scanner & Parser	0	–	9	–	8	23	32
PSLR(1) Scanner	0	–	3	–	0	15	15
Scanner Ratio	nan	0%	30%	0%	nan	31%	38%

Levine SQL

Specification	Gotos	Start Conds.	Global User Vars.	`yylex` User Vars.	Loops	`if`, `else if`, `else`	conds.
Traditional Parser	0	–	0	–	0	21	20
Traditional Scanner	0	3	2	0	0	1	1
PSLR(1) Scanner & Parser	0	–	0	–	0	21	20
PSLR(1) Scanner	0	–	0	–	0	0	0
Scanner Ratio	nan	0%	0%	nan	nan	0%	0%

Table 4.5: Complexity Counts. These tables count non-linear control structures and variables not appearing in external C (as defined for Table 4.4). Similar to McCabe's cyclomatic complexity, the last column counts conditions appearing in those control structures.

PSLR(1) Bison

Specification	Gotos	Start Conds.	Global User Vars.	`yylex` User Vars.	Loops	`if`, `else if`, `else`	conds.
Traditional Parser	0	–	0.444	–	0.592	0.592	1.26
Traditional Scanner	0	1.43	0.681	0.545	0	3.27	2.72
PSLR(1) Scanner & Parser	0	–	0.344	–	0.306	0.880	1.22
PSLR(1) Scanner	0	–	0.238	–	0	1.19	1.19
Scanner Ratio	nan	0%	35%	0%	nan	36%	44%

Levine SQL

Specification	Gotos	Start Conds.	Global User Vars.	`yylex` User Vars.	Loops	`if`, `else if`, `else`	conds.
Traditional Parser	0	–	0	–	0	.947	0.901
Traditional Scanner	0	0.346	0.231	0	0	0.115	0.115
PSLR(1) Scanner & Parser	0	–	0	–	0	0.792	0.754
PSLR(1) Scanner	0	–	0	–	0	0	0
Scanner Ratio	nan	0%	0%	nan	nan	0%	0%

Table 4.6: Complexity Frequences. These tables give the ratio of the control structure, variable, and condition counts from Table 4.5 to the character counts from the last column of Table 4.4.

```
if (A && B) { /*code*/ }
```

Counting conditions in this manner is a first step in computing McCabe's widely used measure of cyclomatic complexity [26]. The only other step is to add 1 to the condition count. However, McCabe's measure is only appropriate for assessing control flow throughout a single cohesive passage of code. Instead, we are measuring the cumulative complexity of separate passages of C code that we have extracted from various locations in a larger specification. That is, we don't wish to include language formalisms like grammars, regular expressions, and precedence declarations, which define control flow among those passages of C code, because our assumption is that such language formalisms are inherently less complex to comprehend in the specification of a language than passages of a general-purpose programming language like C. Thus, we simply count conditions from passages of C embedded throughout each specification, and we observe that those conditions are the embedded C's *contributions* to the cyclomatic complexity of the full specification.

Table 4.6 repeats the counts from Table 4.5 but it divides them by the specification sizes we presented in the last column of Table 4.4. That is, rather than examining the total number of control structures and conditions as in Table 4.5, Table 4.6 examines the frequency with which they appear in the portion of the specifications that are not external C. In both Tables 4.5 and 4.6, the final row of each table gives the ratio of each count or frequency for the PSLR(1) scanner specification to that of the traditional scanner specification. Regardless of whether we examine counts or frequencies, the PSLR(1) scanner specification for PSLR(1) Bison's internal parser exhibits a significant drop relative to the traditional scanner specification for every assessment of embedded C complexity that we include in our tables. For Levine SQL, the traditional scanner specification's counts and frequencies are relatively small already, and the PSLR(1) scanner specification's counts and frequencies are all zero. Thus, instead of finding evidence that the complexity of the embedded C was forced to increase in order to permit the significant reduction in the size of the embedded C when switching from traditional scanner-based LR(1) to PSLR(1), we found evidence that the complexity of the embedded C actually reduced significantly as well.

Chapter 5

Related Work

Although terminology and implementations vary, several other researchers have independently discovered the basic premise of what we call a pseudo-scanner. The earliest description we have found is a 1991 publication by Nawrocki, which we discuss in section 5.1. In section 5.2, we discuss an unpublished paper from Keynes, which is the only paper we have reviewed that acknowledges that LR(1) state merging can induce incorrect pseudo-scanner behavior. The most recent publication is from 2007 from Van Wyk and Schwerdfeger, and we discuss it in section 5.3. In section 5.4, we discuss scannerless GLR, a popular nondeterministic alternative to PSLR(1) that abandons the scanner altogether in order to parse composite languages.

5.1 Nawrocki

The earliest mention of the premise of the pseudo-scanner we have found is a 1991 publication by Nawrocki [27]. Nawrocki categorizes scanner conflicts and explains how the current parser state can sometimes be examined to resolve them automatically. However, he does not address the resulting problems with tokens that should be lexically tied as we discussed in section 3.3. He assumes an LALR(1) parser, but he does not mention that LR(1) state merging can induce incorrect pseudo-scanner behavior as we explained in section 3.4. He demonstrates his techniques with examples from Modula-2 and Ada, but he does not discuss the usefulness of his techniques for composite languages in general.

Nawrocki defines two types of scanner conflicts, which are similar to the two types of pairwise

scanner conflicts we defined in Definition 2.2.5. His definition for an identity conflict, which he calls an *I-conflict*, is equivalent to ours. His definition for an *LM-conflict* (longest match conflict) is similar to our definition for a length conflict. We now state Nawrocki's definition for an LM-conflict formally in terms of our scanner conflict model.

Definition 5.1.1 (LM-conflict)

As an extension of Definition 2.2.5, if the pair (λ, t) and (λ', t') is an *LM-conflict* for ξ over T, then $|\lambda| \neq |\lambda'|$. Thus, an LM-conflict is a type of length conflict. Without loss of generality, we assume that $|\lambda| > |\lambda'|$. Nawrocki denotes such an LM-conflict as $t \succ t'$. Unlike a length conflict, an LM-conflict requires that $\exists(\lambda'', t'') \in (\Xi^+, T) : (t'' \cong \lambda'') \wedge (\lambda' \cdot \lambda''[1] \preceq \lambda)$. In other words, a length conflict is an LM-conflict iff the shorter lexeme is followed by one lexically acceptable character. □

We assert that a length conflict for which the pair (λ'', t'') from Definition 5.1.1 does not exist should not be handled differently than any other length conflict, but Nawrocki excludes this case from his definitions without justification and does not explain how it should be handled.

Nawrocki's first technique for resolving scanner conflicts is LC (left context). LC resolves an LM-conflict in a manner similar to the way the basic behavior of a pseudo-scanner resolves a length conflict. That is, given the current parser state s_p and the remaining input character sequence ξ, s_p indicates the syntactic left context of ξ. If only one of the tokens t and t' is in $acc(s_p)$, then LC resolves $t \succ t'$ for s_p by rejecting the unacceptable token. If both t and t' are in $acc(s_p)$, then LC fails to resolve $t \succ t'$ for s_p.

Nawrocki's second technique for resolving scanner conflicts is ELC (extended left context). For the LM-conflict $t \succ t'$ with conflicting matches (λ, t) and (λ', t'), ELC is like LC in that it rejects whichever of t or t' is not acceptable according to the syntactic left context. However, ELC is more powerful than LC because ELC also rejects t' if it is not acceptable according to one character of syntactic right context. That is, if there does not exist any token that meets the conditions for t'' given in Definition 5.1.1 and that is also syntactically acceptable after t', then ELC rejects t'. Nawrocki does not explain how to handle the case where ELC cannot resolve $t \succ t'$ because LC fails while such a t'' does exist. In contrast, our PSLR(1) generator usually employs the rule of longest match and thus rejects t' when LC fails regardless of the existence of such a t''. That is, our generator's effective resolution of $t \succ t'$ disagrees with ELC's resolution only in two cases: (1) where ELC's resolution is indeterminate and thus useless, and (2) where the user has specified

non-traditional lexical precedence rather than longest match.

Nawrocki does not explain how to resolve identity conflicts. We assume he intends to employ LC as our pseudo-scanner does. Like the rule of longest match, ELC is not defined for identity conflicts because both matches in an identity conflict are of the same length.

In this section, we have explained a generalized version of Nawrocki's scanner conflict resolution techniques in order facilitate the comparison with our PSLR(1) generator. Given the token t and the set of all parser states Σ_p, Nawrocki defines $\mathrm{LC}(t) = \{s_p \in \Sigma_p : t \in acc(s_p)\}$. For any pair of tokens t and t', we have stated that the LC technique resolves each occurrence of the conflict $t \succ t'$ in favour of t' if the current parser state is a member of $\mathrm{LC}(t')$ but not of $\mathrm{LC}(t)$. However, according to Nawrocki's exact description, only membership in $\mathrm{LC}(t')$ need be tested. Thus, Nawrocki actually requires that $\mathrm{LC}(t) \cap \mathrm{LC}(t') = \emptyset$. In other words, Nawrocki requires that LC be able to resolve $t \succ t'$ for every possible syntactic left context in order to resolve it for any of them. Our generalization removes this restriction. Nawrocki imposes a similarly unnecessary but more cryptic restriction for ELC.

5.2 Keynes

The only paper we have reviewed that acknowledges that LR(1) state merging can induce new conflicts in the pseudo-scanner is a paper by Keynes [20]. Unfortunately, Keynes' paper appears to be unpublished. Our analysis is based on a copy that we downloaded in September 2008 from the URL mentioned in our bibliography. The front page bears the date November 2, 2007.

Keynes cites Nawrocki's definitions of identity conflicts and LM-conflicts. He then reformulates those definitions more clearly in terms of transitions in the scanner's FSA, but he maintains the limitation that we noted in Nawrocki's LM-conflict definition relative to our length conflict definition. Keynes also presents an interesting algorithm for resolving scanner conflicts by modifying actions in the scanner at the time of scanner and parser generation. In order to compare his approach with ours, we now extend our formal model to express Keynes' algorithm.

Definition 5.2.1 (Accepts Extended)

Given the LR(1) parser state s_p, the scanner state s_s, and the character $c \in \Xi$, we extend Definitions 2.2.12 and 3.2.12 to further overload *acc* as follows:

1. $acc(s_s, c) = \{t : \exists s'_s : \exists \rho \in \Xi^+ : \rho[1] = c \wedge \delta^*(s_s, \rho) = s'_s \wedge t \in acc(s'_s)\}$.

2. $acc(s_p, s_s, c) = acc(s_p) \cap acc(s_s, c)$.

□

Definition 5.2.2 (Accepts-2)

The model for LR(2) parser states is different than the model for LR(1) parser states in that each item in a lookahead set is a sequence of two tokens instead of a single token. Given the LR(2) parser state s_p, the scanner state s_s, the scanner's start state s_{s0}, and the character $c \in \Xi$, we extend Definitions 2.2.12, 3.2.12, and 5.2.1 to overload *acc* further for the LR(2) case:

1. $acc(s_p) = \{t : \exists((\ell \rightarrow \varrho), d, K, (a_t, a_p, a_s)) \in s_p : (a_t = \text{“S”} \wedge \varrho[d] = t) \vee (a_t = \text{“R”} \wedge \exists t' : (t, t') \in K)\}$

2. $acc(s_p, s_s) = acc(s_p) \cap acc(s_s)$ as for LR(1).

3. $acc(s_p, s_s, c) = acc(s_p) \cap acc(s_s, c)$ as for LR(1).

4. $acc_2(s_p) = \{(t, t') : \exists((\ell \rightarrow \varrho), d, K, (a_t, a_p, a_s)) \in s_p : (a_t = \text{“S”} \wedge \varrho[d] = t \wedge t' \in acc(a_s)) \vee (a_t = \text{“R”} \wedge (t, t') \in K)\}$.

5. $acc_2(s_p, s_s, s_{s0}, c) = acc_2(s_p) \cap \{(t, t') : t \in acc(s_s) \wedge t' \in acc(s_{s0}, c)\}$.

□

Definition 5.2.3 (Keynes Scanner Conflict Resolution)

Given the set of parser states Σ_p and the set of scanner states Σ_s with start state s_{s0}, then, for every combination $(s_p, s_s, c) \in (\Sigma_p, \Sigma_s, \Xi)$, Keynes records a separate action on c in s_s as follows:

1. If $acc(s_p, s_s) = \emptyset$, then transition on c.

2. If $|acc(s_p, s_s)| = 1$ and $acc(s_p, s_s, c) = \emptyset$, then accept $acc(s_p, s_s)[1]$ on c.

3. If $|acc(s_p, s_s)| > 1$ and $acc(s_p, s_s, c) = \emptyset$, then there's an unresolvable identity conflict. (Keynes actually specifies an accept action in this case but does not specify which token to accept.)

4. If $acc(s_p, s_s) \neq \emptyset$, $acc(s_p, s_s, c) \neq \emptyset$, and $acc_2(s_p, s_s, s_{s0}, c) = \emptyset$, then transition on c.

5. If $acc(s_p, s_s) \neq \emptyset$, $acc(s_p, s_s, c) \neq \emptyset$, and $acc_2(s_p, s_s, s_{s0}, c) \neq \emptyset$, then there's a potential LM-conflict that is unresolvable.

□

Keynes' algorithm from Definition 5.2.3 almost exhibits the basic pseudo-scanner behavior we described in Definition 3.1.1. That is, if every match that is accepted by the current scanner state is not accepted by the current parser state, rule 1 specifies that such matches be ignored and that the transition to the next scanner state be taken in search of a longer match. Otherwise, if every longer match that might be accepted by some later scanner state is not accepted by the current parser state, rules 2 and 3 specify that one of the current matches be accepted instead and that the next scanner transition not be taken. When both current matches and longer matches are possible, rules 4 and 5 employ a technique similar to Nawrocki's ELC. Like Nawrocki's ELC technique, rules 4 and 5 disagree with longest match only when they are indeterminate and thus useless. Other than rule 4, Keynes' algorithm does not attempt to resolve pseudo-scanner conflicts. In order for Keynes' algorithm to fully support basic pseudo-scanner behavior, rules 3 and 5 would need to select some specific match whose token appears in $acc(s_p)$, thus resolving pseudo-scanner conflicts instead of simply identifying them.

Another problem with Keynes' algorithm is that it sometimes falsely detects LM-conflicts. The trouble is that the set $acc(s_p, s_s, c)$ is capable of identifying only the possibility of a longer match and thus only the possibility of an LM-conflict based on the next character. It does not examine the remaining character sequence to be sure a longer match and LM-conflict are actually present. To overcome this problem, the run-time scanning algorithm could implement the rule of longest match as follows. The scanner would record the most recently seen scanner state at which it identified a potential LM-conflict. If the scanner later encountered an error action before finding a longer match, it would return to the last recorded state as the final accepting state and rewind the input character sequence accordingly. In this way, the scanner would accept the shorter match when there is no actual longer match. Keynes suggests that, when his algorithm fails to resolve scanner conflicts, the scanner could fall back to Lex conflict resolution rules, so such a longest match implementation might indeed be his intention. However, implementing the rule of longest match renders rules 4 and 5 from Definition 5.2.3 useless. Fortunately, eliminating rules 4 and 5 would be beneficial because of their use of acc_2, which requires LR(2) computation (specifically, he uses LALR(2)), which he notes

is "one of the slower parts of the system" [20].

Because scanner transitions might need to be modified differently for each parser state, there must be some means to store the numerous FSA's that result. Keynes proposes two possibilities. First, the generator can construct the original FSA plus an auxiliary table that records, per parser state, what changes must be made to the FSA. Second, the multiple FSA's can be constructed and merged into a single FSA with multiple start states. Our PSLR(1) generator constructs the scanner FSA plus the *scanner_accepts* table, so it most resembles Keynes' first proposal.

Unlike Nawrocki, Keynes' algorithm does handle length conflicts that are not LM-conflicts. Specifically, when s_s from Definition 5.2.3 is a state that accepts the shorter lexeme in such a conflict, then $acc(s_{s0}, c) = \emptyset$ by Definition 5.1.1, thus $acc_2(s_p, s_s, s_{s0}, c) = \emptyset$ by Definition 5.2.2, and so rule 4 in Definition 5.2.3 chooses the transition action on c. Also, Keynes does not impose Nawrocki's restriction that a scanner conflict between a pair of tokens must be resolvable for every parser state in order for it to be resolvable for any parser state.

Keynes discusses how the pseudo-scanner should behave when it cannot match any token acceptable by the current parser state. He defines two main approaches. An *exclusive scanner* returns an error instead of a specific token. As Keynes notes, an exclusive scanner is not appropriate for a parser whose error recovery strategy involves discarding syntactically unacceptable tokens as they are returned by the scanner. Moreover, error messages are usually more succinct and thus meaningful after erroneous character sequences have been tokenized. In contrast, when an *inclusive scanner* cannot match any token acceptable by the current parser state, it then attempts to select a token without regard to the current parser state. This approach avoids the difficulties of the exclusive scanner, but there is still no guarantee that it chooses the best possible token for error recovery. In section 3.5, we explained the fallback row of our *scanner_accepts* table, which enables our pseudo-scanners to behave like Keynes' inclusive scanner.

Keynes' paper is the only work we have reviewed that acknowledges that LR(1) state merging can induce incorrect behavior in the pseudo-scanner. In his terminology, the scanner cannot always determine the left context from the current parser state because LR(1) state merging can lose left context. Keynes also states that "these problem states fortunately seem to be relatively rare in practice" [20], but he offers no citation or statistical evidence to support this statement. As we described in section 4.5.1, the results from our Levine SQL case study suggest that this statement is not accurate. He suggests that a full LR(1) parser could be employed instead of LALR(1), but

he cites the work of Spector who proposes a minimal LR(1) algorithm [33]. As we explained in section 3.4, minimal LR(1) algorithms like IELR(1) and Spector's are designed to avoid merging parser states when doing so would induce incorrect parser behavior. In order to avoid incorrect pseudo-scanner behavior as well, such an algorithm requires an extension like our IELR(1) extension for PSLR(1), but Keynes does not mention this requirement.

Keynes's techniques appear to be based mainly on Nawrocki's work. Like Nawrocki, Keynes does not discuss the usefulness of his techniques for composite languages in general. Unlike Nawrocki, he does discuss the value of a unified scanner and parser specification. Keynes also briefly addresses issues relative to traditional scanner-based parsing, such as how to specify whitespace and how to distinguish between keywords and identifiers.

5.3 Van Wyk and Schwerdfeger

A 2007 paper from Van Wyk and Schwerdfeger is the most recent publication we have found that discusses the premise of the pseudo-scanner, which they refer to as a *context-aware scanner* [40]. Instead of discussing the usefulness of the pseudo-scanner for composite languages in general as we do, they focus on its usefulness specifically for extensible languages. However, because extensible languages are one of the most complex forms of composite languages, the set of issues that they address is similar to the set of issues we have addressed in this dissertation. Many of our solutions are similar as well. In this section, rather than enumerating the similarities, we discuss the solutions from their paper that have interesting differences from ours.

Unlike the other papers we have reviewed, Van Wyk and Schwerdfeger describe an explicit lexical precedence declaration for resolving pseudo-scanner identity conflicts. Specifically, $t \succ t'$ specifies that any match for t has higher precedence than a match with the same lexeme for t'. For reasons similar to those we gave in section 3.2.1, the lexical precedence relation specified by "$\succ$" is not implicitly transitive. However, unlike our scanning algorithm, their scanning algorithm always employs the rule of longest match to resolve pseudo-scanner length conflicts, for which they discuss no other resolution mechanism. In this way, their "$\succ$" operator merges the functionality of the "$<-$" and "$<\sim$" operators from our `%lex-prec` directive, and "$-\sim$" is implicit for pseudo-scanner length conflicts between other pairs of tokens. Thus, unlike our PSLR(1) generator, their generator does not warn the user when new pseudo-scanner length conflicts arise.

The "$\succ$" operator also merges lexical precedence with lexical tying. That is, in addition to a lexical precedence relationship, $t \succ t'$ declares t and t' to be lexically tied, and Van Wyk and Schwerdfeger do not provide any declarative means to specify lexical precedence or lexical ties separately. The need to declare lexical ties separately is revealed by Van Wyk and Schwerdfeger's example of the keywords `'for'` and `'foreach'`. These keywords should be declared lexically tied so that, in a context in which only the keyword `'for'` is syntactically acceptable, the pseudo-scanner selects the keyword `'foreach'` for the input character sequence "`foreach`" instead of selecting the keyword `'for'` and leaving the trailing "`each`" to be recognized as an identifier by the subsequent pseudo-scanner invocation. However, the "$\succ$" operator is not appropriate because there is no identity conflict between `'for'` and `'foreach'`. Van Wyk and Schwerdfeger's unfortunate solution here is that the user rewrite the grammar to specify that whitespace is required immediately following any occurrence of `'for'` that precedes an identifier. We observe that a distinct lexical tie declaration using our `%lex-tie` directive would be simpler, clearer, and more maintainable.

For some pairs of tokens, combining lexical precedence and lexical tying is not a problem. As we explained in section 3.3, lexical precedence and symmetric lexical ties are needed for reserved keywords and identifiers. However, "$\succ$" can only declare asymmetric lexical ties. For example, if the user declares `'while'` $\succ$ `ID`, then `'while'` is lexically tied to every occurrence of `ID` that appears in a parser state accept set, but `ID` is not lexically tied to occurrences of `'while'`. Thus, in the example C statement in Figure 3.6b, which we discussed in section 3.3, the pseudo-scanner would not recognize the character sequence "`whiles`" as an `ID` token because only the `'while'` token is syntactically acceptable here. Instead, it would recognize "`while`" as a `'while'` token, leaving the trailing "`s`" for a subsequent pseudo-scanner invocation. If the C grammar were to accept an `ID` token following a `'while'` token, the pseudo-scanner would then recognize the trailing "`s`" as the `ID` token instead of reporting the expected syntax error. The "$\succ$" operator's behavior in this example would obviously not be correct even if `'while'` were a non-reserved keyword, for which asymmetric lexical tying is usually appropriate as we discussed in section 3.3. The trouble is that the lexical tie declared by "$\succ$" is the reverse of what is needed for non-reserved keywords given the lexical precedence relationship declared by "$\succ$". We assume that Van Wyk and Schwerdfeger's solution in these examples would be the same as in their `'for'` versus `'foreach'` example. That is, the user must rewrite the grammar to specify that whitespace is required immediately following each occurrence of `'while'` that precedes `ID`. Again, we observe that our `%lex-tie` directive is

simpler, clearer, and more maintainable.

For any set of tokens with pseudo-scanner identity conflicts for which "≻" is not appropriate, Van Wyk and Schwerdfeger allow the user to provide a *disambiguation function*. In contrast to the declarative nature of the "≻" operator, this disambiguation function can contain arbitrary user code, and so it usually cannot be evaluated until run time. As a result, generation-time pseudo-scanner FSA optimizations that might make use of declarative pseudo-scanner conflict resolution mechanisms like "≻" would not always be possible.

Van Wyk and Schwerdfeger describe two scenarios in which disambiguation functions prove useful. First, the user can write a disambiguation function that always returns a specific token and thus performs exactly the task of the "≻" operator but without the implicit lexical tie. Again, we prefer that this scenario be handled declaratively as permitted by our `%lex-prec` directive. Second, a disambiguation function can resolve a conflict using semantics. For example, it can look up an identifier in a symbol table in order to determine whether the pseudo-scanner should return a variable name token or a type name token.

Unfortunately, Van Wyk and Schwerdfeger's disambiguation functions can sometimes produce confusing behavior. The token returned by a disambiguation function might have been requested as a lookahead by the parser. After the parser receives the token, the parser might perform a series of reduce actions and their associated semantic actions before finally shifting the token. Those semantic actions might alter semantic properties, such as symbol table scope, and thus change which token the disambiguation function should have returned. Moreover, as the reduce actions are performed, the current parser state might change, thus the set of conflicting tokens might change, and thus which disambiguation function should have been invoked might change. To address this problem, after every such reduce action, Van Wyk and Schwerdfeger's pseudo-scanner rescans the input text and invokes whatever disambiguation function then looks appropriate. In other words, their pseudo-scanner relies on potentially incorrect interpretations of the input text to select the series of reduce actions that then alter those interpretations. Van Wyk and Schwerdfeger provide no proof that this approach results in the selection of the correct token.

We observe that a less confusing way to employ semantics to resolve conflicts is provided by the Anagram parser generator [1], which is also mentioned by Keynes [20]. For example, instead of declaring separate identifier tokens for variable names and type names, the user can declare a single generic identifier token, a variable name nonterminal, and a type name nonterminal. Each of the two

nonterminals appears on the LHS of a production whose RHS contains only the generic identifier token. In any context where both a variable name and a type name are acceptable, the parser encounters a conflict between the two productions immediately after shifting the generic identifier token. The key to Anagram's solution is that the user can provide a disambiguation function that resolves this parser conflict instead of the corresponding pseudo-scanner conflict from Van Wyk and Schwerdfeger's approach. The advantage is that the generic identifier token remains the lookahead throughout the series of reduce actions before the parser shifts it. Thus, the identifier must be interpreted as a variable name or type name only after the series of reduce actions is finished and thus after semantic properties like symbol table scope are finalized. As part of our future work, we are considering extending our PSLR(1) generator to support Anagram's solution.

Van Wyk and Schwerdfeger employ LALR(1) and overlook the trouble that LR(1) state merging can cause. Moreover, they specifically make the claim that, once a scanner returns a lookahead token that is acceptable by the current parser state, the following series of parser reduce actions on that token cannot push a parser state that does not accept that token. They use this claim to reach the conclusion that the input need not be rescanned after every reduce action except when disambiguation functions must be employed. Their claim is actually true for canonical LR(1) parser tables. However, because of the merging of LR(1) states and sets of acceptable tokens, their claim cannot be guaranteed for LALR(1) parser tables even when the grammar is LALR(1). Also, LR(1) state merging can cause the selection of an incorrect disambiguation function because it can add invalid tokens to a pseudo-scanner conflict. In section 3.4, we described PSLR(1)'s IELR(1) extension, which guarantees that LR(1) state merging never causes the pseudo-scanner to select an incorrect token.

One other interesting aspect of Van Wyk and Schwerdfeger's paper is that they optimize the pseudo-scanner by using functions like our *acc* function in a manner similar to the way Keynes does. Most interestingly, they compute $acc(s_p, s_s, c)$ to avoid useless scanner transitions. However, they do not use this function to compute a separate pseudo-scanner FSA for each parser state at generation time as Keynes does. Instead, for every scanner state s_s, they compute a function $poss(s_s) = acc(s_s) \cup \{t : \exists c \in \Xi : t \in acc(s_s, c)\}$ at generation time. Notice that, given any pair of scanner states s_s and s'_s and character c such that $\delta(s_s, c) = s'_s$, then $acc(s_s, c) = poss(s'_s)$ by Definition 5.2.1. Thus, at run time, their pseudo-scanner follows the transition on the next input character c from the current scanner state s_s to discover the next scanner state s'_s and then

compute the intersection $acc(s_p) \cap poss(s'_s)$, which is equal to $acc(s_p, s_s, c)$ by Definition 5.2.1. If $acc(s_p, s_s, c) = \emptyset$, their pseudo-scanner does not bother to scan further. Along with Keynes' approach, we are considering extending our PSLR(1) implementation with this approach as part of our future work. If *poss* requires significant storage when computed for every scanner state, we might try Keynes' approach of evaluating $acc(s_p, s_s, c)$ only at each scanner state s_s for which $acc(s_p, s_s) \neq \emptyset$, as we showed in Definition 5.2.3.

5.4 Scannerless GLR

The key issue addressed by the premise of the pseudo-scanner is that different sub-languages within a composite language often require different scanner specifications. While a pseudo-scanner handles this issue by automatically communicating with the parser as we explained in section 3.1, a traditional scanner generator requires the user to manually specify start conditions for the sub-languages as we explained in section 2.3. However, there exists another common solution that does not require a pseudo-scanner or start conditions. That is, the user can extract from the scanner specification all parts of the lexical syntax that are specific to only a proper subset of the sub-languages and then express those parts in the sub-languages' grammars instead. Thus, token definitions in the scanner specification are reduced to simple building blocks of lexical syntax that are generic enough to be shared by all the sub-languages' grammars. In the extreme case where the sub-languages have no common lexical syntax, every token must be reduced to only a single character. The composite grammar is then called a *character-level grammar*.

For character-level grammars, the scanner's functionality is reduced to merely buffering the input character sequence. Thus, the scanner's functionality can easily be incorporated into the parser. Salomon and Cormack introduced the term *scannerless parsing* to identify this architecture [31]. However, rather than viewing character-level grammars and scannerless parsers as an extreme case of a common approach to handling language composition, Salomon and Cormack instead employ them as the basis for unified scanner and parser specifications that completely eliminate the complexity of scanner and parser communication.

Consider a scannerless LR(1) parser, p, and a scanner-based LR(1) parser, p', for the same language. Because every token in p is reduced to only a single character, the tokens that trigger different parser actions in any parser state in p easily become indistinguishable. As a result, p

usually has significantly more conflicts than p'. One naive approach to solving this problem is to generate LR(k) parser tables for p where k is the length of the longest lexeme from the tokens in p'. This approach would allow p to see at least as far ahead in the input character stream as p' when attempting to choose a parser action. However, for many tokens like identifiers, the length of lexemes is usually unbounded and thus would require $k = \infty$. Unfortunately, LR(∞) parser tables would require infinite storage.

Visser combines scannerless parsing with GLR (Generalized LR) [38, 39], which was first described by Tomita [34]. A GLR parser usually employs parser tables from the LR(1) family. However, upon encountering a conflict in those parser tables during a parse, the GLR parser branches and explores all possible parses to which the conflicting parser actions lead. For this reason, GLR is called a *non-deterministic* parsing algorithm. An incorrect parser action leads to a syntax error, which kills the corresponding parsing branch. If all parsing branches die, the GLR parser reports a syntax error. In this way, a GLR parser is able to look as far into the input character stream as necessary to determine which parser action, if any, leads to a syntactically correct parse. This effective LR(∞) behavior solves the unbounded lookahead problem for scannerless parsing.

If multiple parsing branches survive in a GLR parser, then the grammar is ambiguous. That is, while a deterministic LR parser can only compute one parse tree for a given input, a GLR parser handles ambiguous grammars by computing all possible parse trees, each of which might represent a different semantic interpretation of the input. Visser employs this ability of GLR in order to implement *reject productions*. For example, if a keyword is a reserved word and thus should never be mistaken for an identifier, the user can write a production for the identifier that rejects the keyword's lexeme. When the keyword's lexeme appears in the input, the GLR parser must branch to recognize all possible interpretations, potentially resulting in multiple parse trees. The reject production specifies that the parse tree that represents the lexeme's interpretation as an identifier be pruned. If the interpretation of the lexeme as the keyword is not possible in the current parsing context, then no parse trees remain, and the parser reports a syntax error. Notice that reject productions are similar to Van Wyk and Schwerdfeger's "$\succ$" operator, which combines asymmetric lexical tying with lexical precedence for identity conflicts. That is, in any context in which the identifier is syntactically acceptable, the keyword has precedence even if the keyword is not syntactically acceptable.

In section 5.2, we described a common and simple implementation of the rule of longest

match for resolving scanner length conflicts. Our *pseudo_scan* function from Definition 3.2.16 also implements the rule of longest match. For a scannerless parser, an implementation of the rule of longest match is not as obvious. Visser's solution is *follow restrictions*. For example, the user can declare that an identifier or keyword should never be followed immediately by a letter. Thus, in Figure 3.6b, which we described in section 3.3, the parser cannot then recognize the character sequence "`whiles`" as the keyword `'while'` followed by a trailing "`s`". Moreover, if the identifier were syntactically acceptable here, the parser could only recognize the complete "`whiles`" as the identifier. Because the identifier is not actually syntactically acceptable, the parser reports a syntax error as expected. Recall that, in order for the rule of longest match in a pseudo-scanner to be useful for this example, a token like the identifier must be lexically tied to the `'while'` keyword so that the pseudo-scanner selects the longest matching lexeme, "`whiles`", even though it is not syntactically acceptable in this context. Thus, a follow restriction is similar to combining asymmetric lexical tying with the rule of longest match. However, for a token with a complex regular expression, the correct regular expression for the follow restriction is not always straightforward.

In this section, we have summarized Visser's scannerless GLR system and compared some of its mechanisms with our PSLR(1) mechanisms. Because of the popularity of Visser's system, we conclude our dissertation in section 6 with a high-level comparison of the two systems' relative advantages.

Chapter 6

Merits of the Work

In chapter 5, we compared PSLR(1) with several other parser generator systems that employ a pseudo-scanner, and we explained how PSLR(1) overcomes those systems' shortcomings. We also compared PSLR(1) to Visser's popular scannerless GLR system, which employs non-deterministic parsing in order to abandon the scanner altogether. Because of the wide recognition of scannerless GLR as a robust system for parsing composite languages, we conclude our dissertation by explaining how PSLR(1) achieves Visser's original goals for scannerless GLR even though PSLR(1) does not require the complexity of non-deterministic parsing.

In section 2.1 of Visser's original scannerless GLR paper, he provides a list of advantages of scannerless GLR parsing over traditional scanner-based LR parsing [38]. This list originally inspired us to conceive of the pseudo-scanner as a deterministic alternative. In other words, we intend for (pseudo-scanner)-based LR(1) parsing to be pseudo-(scannerless), and we abbreviate this as PSLR(1). We now explain how our PSLR(1) system addresses each such advantage:

1. "No Scanner." Unlike scannerless GLR, PSLR(1) does require scanner generation and thus does not eliminate the "complicated interface between scanner and parser" [38]. Instead, PSLR(1) automates this interface by generating the type of scanner that we call a pseudo-scanner.

2. "Integrated Uniform Grammar Formalism." Our PSLR(1) generator accepts an integrated scanner and parser specification. As a result, if a token outgrows its regular syntax, then the user can more easily convert it into a nonterminal so that its syntax can be specified via a

context-free grammar instead, and vice-versa. Moreover, an integrated specification is easier to divide into modules, one per sub-language. However, our lexical and syntactic formalisms are not as uniform as in Visser's system. For example, for nonterminals, we have not implemented any equivalent of lexical precedence rules like longest match. It is not clear to us that such lexical declarations would actually prove useful for the kinds of constructs that are usually expressed using a context-free grammar given that traditional parser generators also do not permit any equivalent of these declarations in the grammars they accept. We expect that it will prove sufficient to apply these lexical declarations and other syntactic declarations to the tokens within a nonterminal's grammar instead.

3. "Disambiguation by Context." In describing scannerless GLR, Visser says that "lexical analysis is guided by context-free analysis. If a token does not make sense at some position, it will not even be considered" [38]. This advantage is precisely the purpose of the basic behavior of the pseudo-scanner, which we described in section 3.1.

4. "Conservation of Lexical Structure." At run time, it is often useful to construct a representation similar to a parse tree for the lexical structure of the input so that the lexical structure can be further analyzed in later phases of the application. Specifying this construction using context-free grammar productions and associated semantic actions is often easier than using a traditional scanner specification. Thus, even if a token's syntax is regular, the user may find it useful to convert the token into a nonterminal. Unfortunately, as Visser points out, whitespace and comments specified using a mechanism like PSLR(1)'s layout tokens would then be permitted within the token's syntax. However, in section 3.6, we explained how lexical precedence declarations can be used to prevent the recognition of layout tokens in such syntactic contexts. Moreover, in section 3.7, we discussed a mechanism by which the user can declaratively scope directives to specific nonterminals' grammars, and this mechanism could be used to restrict the scope of layout tokens.

5. "Conservation of Layout." In section 3.6, we explained how some parsing applications require whitespace and comments not to be discarded, and we explained PSLR(1)'s mechanism for attaching whitespace and comments to other tokens so that they can be returned to the parser.

6. "Expressive Lexical Syntax." Visser is here referring to the ability to express tokens, whitespace, and comments using a context-free grammar. Again, tokens including layout tokens can

be converted to nonterminals or lexical nonterminals, which we discussed in section 3.6.

Visser alludes to the obvious disadvantages of his system relative to traditional scanner-based LR as follows:

> Since ambiguity of a context-free grammar is undecidable (Floyd, 1962), it is also undecidable whether a conflict is due to an ambiguity or to a lack of lookahead. Because complete character level grammars frequently need arbitrary length lookahead, methods to solve conflicts in the table will not always succeed. [38]

In other words, it is the user's responsibility to determine whether each conflict in the parser tables produced from the user's grammar is due to insufficient lookahead or to an ambiguity. If the user overlooks an ambiguity, a GLR parser can unexpectedly produce multiple parse trees at run time. As Van Wyk and Schwerdfeger state, "for building extensible languages we prefer a guarantee of determinism since languages may be composed at the direction of the programmer, not a language expert who can resolve syntactic or lexical ambiguities" [40]. The task of debugging parser tables is already especially difficult because of the character-level grammars required by the scannerless architecture. The character-level grammars expand the number of conflicts and the number of parser states, which are polluted with the extra symbols and productions needed to define the lexical syntax. In this way, the separation of concerns permitted by scanner-based parsing is completely lost. Because PSLR(1) employs a scanner and does not employ GLR, it does not suffer from these disadvantages.

The application domain of our PSLR(1) system is the deterministic parsing of composite languages. Outside of this domain, our PSLR(1) system lacks some of the features of Visser's scannerless GLR system. Specifically, we have not proposed to provide the full capabilities of follow restrictions and GLR, which enables unbounded lookahead, the discovery all possible parse trees in the case of ambiguous grammars, and reject productions. However, there is no aspect of our PSLR(1) system that would prevent the incorporation of these mechanisms. If we were to extend our PSLR(1) generator to provide these mechanisms, the user could employ them only for the portions of the grammar that require them. Other portions of the grammar would still retain PSLR(1)'s advantages over scannerless GLR. For this reason, when scannerless GLR features are required, we predict that users can more easily develop parsers using a hybrid of PSLR(1) and scannerless GLR rather than using scannerless GLR or PSLR(1) alone.

In summary, PSLR(1) is a more robust system for generating deterministic parsers for composite languages than either traditional scanner-based LR(1) parser generation, existing pseudo-scanner-based parser generation systems, or scannerless GLR. To achieve this result, PSLR(1) employs novel techniques that combine the most useful aspects of all of these systems and that overcome each of their shortcomings.

Bibliography

[1] Anagram. Parsifal Software. http://www.parsifalsoft.com/.

[2] Bison. The GNU Project. http://www.gnu.org/software/bison/.

[3] Flex. SourceForge. http://flex.sourceforge.net/.

[4] Gawk. The GNU Project. http://www.gnu.org/software/gawk/.

[5] GCC. The GNU Project. http://gcc.gnu.org/.

[6] Groff. The GNU Project. http://www.gnu.org/software/groff/.

[7] The SQL Test Suite, Version 6.0. NIST. http://www.itl.nist.gov/div897/ctg/sql_form.htm.

[8] *International Standard, Programming Languages – C++.* Number ISO/IEC 14882:2003(E). American National Standards Institute, October 2003.

[9] Single UNIX Specification, Version 3. The IEEE and the Open Group, April 2004. http://www.opengroup.org/bookstore/catalog/t041.htm.

[10] *International Standard, Programming Languages – C.* Number ISO/IEC 9899:1999/TC3. ISO and IEC, September 2007.

[11] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[12] Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for aspectj. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 209–228, New York, NY, USA, 2006. ACM.

[13] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, DEC SRC, 1994.

[14] Ralph Corderoy. troff.org, The Text Processor for Typesetters. http://troff.org.

[15] Joel E. Denny and Brian A. Malloy. IELR(1): Practical LR(1) Parser Tables for Non-LR(1) Grammars with Conflict Resolution. In *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 240–245, New York, NY, USA, 2008. ACM.

[16] Joel E. Denny and Brian A. Malloy. The IELR(1) Algorithm for Generating Minimal LR(1) Parser Tables for Non-LR(1) Grammars with Conflict Resolution. *Science of Computer Programming*, In Press, Corrected Proof, 2009.

[17] The GNU Project. *Bison*, 2.4.1 edition, December 2008.

[18] Robert Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM.

[19] S. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[20] Nathan Keynes. Better Parsing Through Lexical Conflict Resolution, November 2007. `http://www.deadcoderemoval.net/files/honours-thesis.pdf`.

[21] P. Klint, R. Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[22] R. Lämmel. Grammar Testing. In *FASE*, pages 201–216, 2001.

[23] M. E. Lesk and E. Schmidt. Lex: A Lexical Analyzer Generator. In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[24] J. R. Levine. *flex & bison*. O'Reilly Media, Inc., Sebastopol, CA 95472, 2009.

[25] J. R. Levine. *flex & bison* examples code, 2009. `ftp://ftp.iecc.com/pub/file/flexbison.zip`.

[26] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[27] Jerzy R. Nawrocki. Conflict detection and resolution in a lexical analyzer generator. *Inf. Process. Lett.*, 38(6):323–328, 1991.

[28] D. Pager. The lane tracing algorithm for constructing LR(k) parsers. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 172–181, New York, NY, USA, 1973. ACM Press.

[29] D. Pager. A Practical General Method for Constructing LR(k) Parsers. *Acta Informatica*, 7(3):249–268, September 1977.

[30] D. Pager. The Lane-Tracing Algorithm for Constructing LR(k) Parsers and Ways of Enhancing Its Efficiency. *Information Sciences*, 12(1):19–42, 1977.

[31] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. *SIGPLAN Not.*, 24(7):170–178, 1989.

[32] D. Spector. Full LR(1) parser generation. *SIGPLAN Not.*, 16(8):58–66, 1981.

[33] D. Spector. Efficient full LR(1) parser generation. *SIGPLAN Not.*, 23(12):143–150, 1988.

[34] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[35] M. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5(1):1–41, 1996.

[36] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. In *IWPC*, page 108, Washington, DC, USA, 1998.

[37] Daveed Vandevoorde. Right Angle Brackets, January 2005. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html`.

[38] Eelco Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, August 1997.

[39] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[40] Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 63–72, New York, NY, USA, 2007. ACM.