Science of Computer Programming 75 (2010) 943–979

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution

Joel E. Denny*, Brian A. Malloy

School of Computing, Clemson University, Clemson, SC 29634, USA

ARTICLE INFO

Article history:
Received 17 July 2008
Received in revised form 31 March 2009
Accepted 12 August 2009
Available online 10 September 2009

Keywords:
Grammarware
Canonical LR
LALR
Minimal LR
Yacc
Bison

ABSTRACT

There has been a recent effort in the literature to reconsider grammar-dependent software development from an engineering point of view. As part of that effort, we examine a deficiency in the state of the art of practical LR parser table generation. Specifically, LALR sometimes generates parser tables that do not accept the full language that the grammar developer expects, but canonical LR is too inefficient to be practical particularly during grammar development. In response, many researchers have attempted to develop minimal LR parser table generation algorithms. In this paper, we demonstrate that a well known algorithm described by David Pager and implemented in Menhir, the most robust minimal LR(1) implementation we have discovered, does not always achieve the full power of canonical LR(1) when the given grammar is non-LR(1) coupled with a specification for resolving conflicts. We also detail an original minimal LR(1) algorithm, IELR(1) (Inadequacy Elimination LR(1)), which we have implemented as an extension of GNU Bison and which does not exhibit this deficiency. Using our implementation, we demonstrate the relevance of this deficiency for several real-world parser specifications, and we demonstrate the feasibility of IELR(1). Finally, we demonstrate that, if canonical LR(1) were employed instead, grammar development would be severely impeded regardless of the power of the computer hardware.

1. Introduction

Grammar-dependent software is omnipresent in software development [20]. 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 [20,23]. 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 modern grammars such as the C++ and C# grammars. Thus, industrial strength parser development requires considerable effort, and many approaches to parser generation are ad hoc [33,34].

One source of difficulty in parser development stems from a deficiency in the state of the art of practical LR parser table generation. LR parsing is the most general deterministic shift-reduce parsing method known, and canonical LR is the

* Corresponding author.
E-mail addresses: jdenny@cs.clemson.edu (J.E. Denny), malloy@cs.clemson.edu (B.A. Malloy).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.08.001

most general technique for generating LR parser tables from a given grammar [13]. As a result, when a grammar developer considers the general behavior of deterministic shift-reduce parsing without the complexity of any further restriction, the language he then expects his grammar to define is accepted in full by the grammar's canonical LR parser tables. In previous decades, canonical LR parser tables required "too much space and time to be useful in practice" [13], but the validity of this statement is fading with the increasing memory capacity and processing speed of modern computer hardware. However, as our results in Section 4.2 demonstrate, the size of canonical LR parser tables also severely impedes the debugging process during grammar development regardless of the power of the computer hardware.

In contrast, LALR parser tables are practical because they merge canonical LR parser states. Thus, they are employed by widely used tools like Yacc [11,19] and its GNU implementation Bison [1]. Unfortunately, as Bison's manual points out, this parser state merging causes LALR parser tables to contain "mysterious conflicts" that "don't look warranted" [17]. These conflicts cause the parser to encounter "unnatural errors" because LALR "is not powerful enough to match the intuition of the grammar writer" [30,31]. The reason LALR behavior seems unintuitive to a grammar developer is because, in order to understand the language that an LALR parser accepts for his grammar, he must consider not only the behavior of deterministic shift-reduce parsing but also the complex details of LALR parser state construction and merging. In this way, LALR can worsen the difficulty of developing a correct grammar. Even for an existing correct LALR grammar, LALR can interfere with incremental changes [30,31], which are inevitable in the face of software evolution.

In response, many researchers have developed *minimal LR*[1] algorithms, which attempt to generate parser tables with the power of canonical LR but with nearly the efficiency of LALR [6,8,25–28,30,31]. Menhir [7] is an implementation of an algorithm described by David Pager [26]. Initially released in 2005 [12], Menhir is the most robust minimal LR(1) implementation we have discovered available.

In this paper, we show that Pager's algorithm and Menhir are not always able to generate parser tables with the full power of canonical LR(1) if the given grammar is non-LR(1) coupled with a specification for resolving conflicts. We also describe an original minimal LR(1) algorithm, IELR(1), which does not suffer from this deficiency. In Section 2, we establish a formal theoretical foundation for our IELR(1) algorithm, and we present several non-LR(1) grammar examples that demonstrate the deficiency of Pager's algorithm. In Section 3, we employ the formal models introduced in Section 2 in order to detail our IELR(1) algorithm, which we have implemented as an extension of Bison. We also describe how to modify IELR(1) to generate canonical LR(1) parser tables. In Section 4, we compare the Bison LALR(1) implementation with our IELR(1) and canonical LR(1) implementations using our example grammars plus five parser specifications for popular languages as case studies. In Sections 5 and 6, we review related work and discuss future work. In Section 7, we use the results of our case studies to conclude that (1) Menhir's deficiency does affect real-world parsers, (2) it can create bugs relative to the intended design of such parsers, (3) IELR(1) is feasible for generating minimal LR(1) parsers for sophisticated real-world LR(1) parser specifications, and (4) if canonical LR(1) were employed instead of IELR(1), grammar development would be severely impeded regardless of the power of the computer hardware.

2. Formal IELR(1) foundation

In this section, we establish a formal theoretical foundation for our IELR(1) algorithm by introducing original concepts and terminology related to LR parsing and by defining a set of formal models, which we employ throughout the rest of the paper. To facilitate this discussion, we also explore several original non-LR(1) grammar examples. In Section 2.1, we clarify a few aspects of the notation that we employ. In Section 2.2, we analyze the parse trees and languages generated by our first two example grammars without the restrictions of any particular parsing technique, and we state our formal model for context-free grammars in general as well as for the parse trees and languages they generate. In Section 2.3, we explore the effects of a specification for resolving the example grammars' LR(1) conflicts, and we then provide an intuitive definition for our LR(1) parser specification model. In Sections 2.4 and 2.5, we describe the example grammars' canonical LR(1) and LALR(1) parser tables, we formalize our LR(1) parser specification model in terms of canonical LR(1), and we introduce a formal categorization of inadequacies and conflicts in LALR(1) and minimal LR(1) parser tables. In Section 2.6, we use our examples and formal models to demonstrate the deficiency from which Pager's algorithm and Menhir suffer but IELR(1) does not.

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".

[1] In practice, the word "minimal" in "minimal LR" means "very small" or "locally minimum" rather than "globally minimum". See Section 3.8 for details.

1. $S \rightarrow aAa$
2. $S \rightarrow bAb$
3. $A \rightarrow a$
4. $A \rightarrow aa$

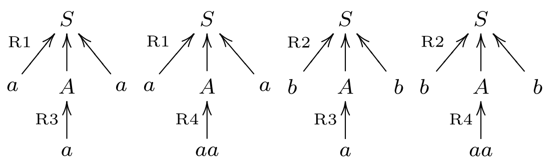


Fig. 1. An unambiguous grammar. This grammar generates a language consisting of 4 sentences, each of which corresponds to one parse tree: *aaa*, *aaaa*, *bab*, and *baab*.

1. $S \rightarrow aAa$	4. $S \rightarrow bAb$	7. $A \rightarrow aa$
2. $S \rightarrow aBb$	5. $S \rightarrow bBa$	8. $B \rightarrow aa$
3. $S \rightarrow aCc$	6. $S \rightarrow bCa$	9. $C \rightarrow aa$

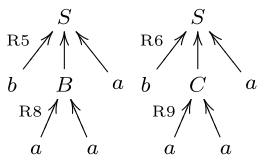


Fig. 2. An ambiguous grammar. This grammar is ambiguous as it can generate two parse trees for the sentence *baaa*.

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".
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.

To express grammars, string rewriting, languages, and automaton state transitions, we combine the notation of Hopcroft and Ullman [18] and Sippu and Soisalon-Soininen [29].

2.2. Context-free

We begin this section by stating our formal model for context-free grammars, their associated parse trees, and the languages they generate without the restrictions of any particular parsing technique.

Definition 2.1 (*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* or *tokens*, 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 $\forall p \in P, \exists \ell \in V : \exists \varrho \in \{V \cup T\}^* : p = (\ell \rightarrow \varrho)$. □

Definition 2.2 (*Parse Trees*)**.** Given a context-free grammar G, then we denote the set of *parse trees* generated by G as $\mathscr{T}(G)$. □

Definition 2.3 (*Context-Free Language*)**.** Given a context-free grammar $G = (V, T, P, S)$, then we denote the *context-free language* generated by G as the set of *sentences* $L(G) \subseteq T^*$ such that, $\forall \tau \in L(G)$, there exists at least one parse tree in $\mathscr{T}(G)$ that derives τ. □

Definition 2.4 (*Ambiguous Context-Free Grammar*)**.** A context-free grammar G is *ambiguous* iff $\exists \tau \in L(G)$ such that there exists more than one parse tree in $\mathscr{T}(G)$ that derives τ. □

As written, the context-free grammar in Fig. 1 generates a context-free language consisting of 4 sentences: *aaa*, *aaaa*, *bab*, and *baab*. The grammar is unambiguous as there does not exist any input sentence for which the grammar can generate more than one parse tree. However, the grammar in Fig. 2 is ambiguous as it can generate two parse trees for the input sentence *baaa*.

2.3. LR(1) parser specifications

Where a dot merely indicates a point in the parse of a sentence, consider how an LR parser for the grammar of Fig. 1 behaves when it reaches the dot in either of the sentences, $ba \cdot b$ and $ba \cdot ab$. These sentences look the same before the dot and so the parser performs identical actions until this point. If the parser then looks ahead one token and sees b, it knows it must *reduce* the previous a to A as in the third parse tree in Fig. 1. If it sees a instead, it must *shift* the a so that it can then reduce the previous aa to A as in the fourth parse tree. Similarly, at the dot in $aa \cdot a$ and $aa \cdot aa$, the parser must choose to reduce in order to accept *aaa* as in the first parse tree, but it must choose to shift in order to accept *aaaa* as in the second parse tree. However, in this pair of sentences, the first token of *lookahead* is the same, a, and so does not distinguish between the possible parser actions. Thus, this grammar is *non-LR(1)*, and these actions are *conflicting actions* in an LR(1) parser.

Assume that the grammar developer has designed the grammar in Fig. 1 with LR(1) in mind and has declared a as *left-associative*. For the sake of the examples in this paper, we employ the definitions of token and production precedence and associativity traditionally implemented by parser generators like Yacc, Bison, and Menhir. Thus, the parser chooses to reduce in the cases of both $aa \cdot a$ and $aa \cdot aa$. In this way, the grammar developer has prevented the parser from generating the only parse tree for the sentence *aaaa* and has thus specified a new language consisting of only 3 sentences: *aaa*, *bab*, and *baab*.

Notice that we were able to determine the parse trees and thus the language for the grammar in Fig. 1 while assuming an LR(1) parser but without examining any parser tables. We merely examined the general behavior of deterministic shift-reduce parsing with one token of lookahead when given: (1) a grammar and (2) an associativity specification to resolve the conflict. Based on this example, we now provide an intuitive definition of the *LR(1) parser specification* model that we employ in this paper. We rewrite this definition formally in Section 2.4.

Definition 2.5 (*LR(1) Parser Specification, Informal*)**.** An *LR(1) parser specification* is a tuple (G, Δ) such that G is a context-free grammar and Δ is a function that selects a unique parser action from any set of conflicting actions encountered by an LR(1) parser for G at any point in any input sentence. We denote the set of parse trees generated by an LR(1) parser that fulfills this specification as $\mathscr{T}(G, \Delta)$. We denote the language accepted by such an LR(1) parser as $L(G, \Delta)$. □

In theory, a Δ could be employed that implements one or more of Klint and Visser's disambiguation filters, which they define independently of the parsing method [21]. In the case of the offside rule, for example, Δ would need to examine token location information gathered at the run time of the parser. One could also imagine a Δ that examines semantic information at run time. Such Δ's can expand the class of languages beyond LR(1). However, in this paper, we are interested only in Δ's that can be evaluated at the time of LR(1) parser generation so that the generated parser is always a deterministic PDA (pushdown automaton). Because any deterministic PDA can be specified by some LR(1) grammar [18], LR(1) parser specifications with such Δ's define exactly the same class of languages as LR(1) grammars. The advantage of LR(1) parser specifications is that Δ can significantly improve the expressiveness and conciseness of a grammar.

In practice, for parser generators like Yacc, Bison, and Menhir, Δ includes a user portion and a parser generator portion. The user portion usually specifies token and production precedence and associativity. The parser generator portion specifies a default mechanism to select a unique parser action from a set of conflicting actions when the user portion fails to do so. We discuss Yacc and Bison's mechanisms for Δ in greater detail in Section 2.4.

Observation 2.6 ($\mathscr{T}$ *and* L *for* (G, Δ) *vs* G)**.** In Definition 2.5, Δ eliminates parser actions from sets of conflicting actions, and thus it eliminates potential parse trees generated by the LR(1) parser:

1. If G is LR(1), there are no conflicting actions, so $\mathscr{T}(G, \Delta) = \mathscr{T}(G)$ and $L(G, \Delta) = L(G)$.
2. If G is non-LR(1) and unambiguous, there are conflicting actions but only one parse tree per sentence in $L(G)$, so $\mathscr{T}(G, \Delta) \subset \mathscr{T}(G)$ and $L(G, \Delta) \subset L(G)$.
3. If G is ambiguous, there are conflicting actions, and Δ may eliminate some or all parse trees for some sentences. Thus, $\mathscr{T}(G, \Delta) \subset \mathscr{T}(G)$ and $L(G, \Delta) \subseteq L(G)$. □

As we have demonstrated in this section, the developer of an LR(1) parser specification (G, Δ) can determine $L(G, \Delta)$ by examining which parse trees Δ removes from $\mathscr{T}(G)$ to form $\mathscr{T}(G, \Delta)$. However, he often has other reasons to determine $\mathscr{T}(G, \Delta)$. For example, he may wish to associate semantic actions with the productions of G, and a reverse-rightmost derivation of each parse tree reveals one possible sequence in which the parser might perform those semantic actions. Thus, in this paper, we are concerned with both the language that an LR(1) parser accepts and the set of parse trees that it generates.

2.4. Canonical LR(1) as a base model

Definition 2.5 does not include the type of the LR(1) parser tables as part of our LR(1) parser specification model because, according to Sections 1 and 2.3, it is more intuitive and convenient for an LR(1) parser specification developer to be able to ignore the restrictions of any particular type of LR(1) parser tables. However, canonical LR(1) is the most general technique for generating LR(1) parser tables [13]. As a result, canonical LR(1) parser tables exhibit the full power that the developer of an LR(1) parser specification (G, Δ) expects from an LR(1) parser when he attempts to determine $\mathscr{T}(G, \Delta)$ and $L(G, \Delta)$ without actually examining any parser tables. Thus, in this section, we are able to use canonical LR(1) as a basis to formalize our LR(1) parser specification model without loss of generality. In this section, we also develop formal models for many related concepts in order to facilitate later comparisons between different types of LR(1) parser tables and canonical LR(1).

In order to compute parser tables for a given grammar, parser generators often *augment* the grammar with a new production. We number the new production 0. When the parser reaches a reduce action by production 0, it has successfully parsed and accepted the given input sentence. We denote this reduce action with `Acc`.

Definition 2.7 (*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 a special token marking the end of any input sentence. □

Definition 2.8 (*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. As in our example sentences in Section 2.3, the dot indicates a position in a parse.
3. The tuple (p, d) is the *core* of m.
4. K is a set of tokens called the *lookahead set*. □

Definition 2.9 (*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 in Definition 2.10 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 $a_p = (\ell \to \varrho)$, 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. $a_t =$ "" to indicate no action. This is used below when resolving conflicts. Both a_p and a_s are left undefined. $\square$

Definition 2.10 (*LR(1) State*)**.** Given a context-free grammar $G : \mathscr{G}(G) = (V', T', P', S')$, then we model an *LR(1) state* s for G as a set of LR(1) items augmented with associated LR(1) actions. Thus, $\forall m = (p, d, K, a) \in s$, given that $p = (\ell \to \varrho) \wedge a = (a_t, a_p, a_s)$, then:

1. (p, d, K) is an LR(1) item for G.
2. $\forall m' = (p', d', K', a') \in s : m \neq m', (p, d) \neq (p', d')$.
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, 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, then the set $\{(p, d) : \exists (p, d, K, a) \in s\}$ is the *core* of s. $\square$

Definition 2.11 (*Canonical LR(1)* Parser)**.** Given a context-free grammar G, then $\mathscr{C}(G)$ is the set of LR(1) states for G's *canonical LR(1) parser* generated using $\mathscr{G}(G)$ and using Knuth's original LR parser table generation algorithm [13,22]. $\square$

The first column of Table 1 shows the canonical LR(1) parser tables for the grammar of Fig. 1. Each numbered cell represents a parser state. Each row within a state represents an augmented item. The table also shows the effect of declaring a as left-associative, but we do not build conflict resolution into our formal model until later in this section.

The first example parse in Table 2 is the parse of the sentence *baab* using the canonical LR(1) parser tables of Table 1. The parser stack is initialized by pushing the *start state*, state 0. The parser's next action is always dictated by the actions recorded in the state at the top of the stack and the next symbol in the input. Whenever the top state has no action on the next symbol, the parser reports a *syntax error*, denoted with `Err`.

Definition 2.12 (*Isocore Relation*)**.** We introduce the operator $\doteq$ to indicate the ***isocore*** relation, which is overloaded to operate on augmented LR(1) items or LR(1) states:

1. Given any two augmented LR(1) items $m = (p, d, K, a)$ and $m' = (p', d', K', a')$, then $m \doteq m'$ iff $(p, d) = (p', d')$.
2. Given any two LR(1) states s and s', then $s \doteq s'$ iff $|s| = |s'|$ and $\forall m \in s, \exists m' \in s' : m \doteq m'$. $\square$

Definition 2.13 (*Isocore Set*)**.** Given a set of LR(1) states Σ and an LR(1) state s, then the set of LR(1) states $\mathscr{I}(s, \Sigma) = \{s' \in \Sigma : s' \doteq s\}$. $\square$

For example, in the canonical LR(1) parser tables of Table 1, state 9 is an isocore of state 11, but their lookahead sets and actions are different.

Between reduce actions, an LR(1) parser acts as a DFA (deterministic finite automaton) such that the state at the top of the parser stack is always the *current state* in the DFA. This view establishes some convenient terminology and notation that we employ when, for conciseness, we do not wish to discuss parser stack behavior or the relationship between actions and items. We employ δ as a function that can examine the *transition* actions recorded in any LR(1) state. The statement $\delta(s, y) = s'$ holds iff there is a transition from state s on symbol y to another state s'. By Definition 2.9, the transition is a shift if y is a token, and it is a goto if y is a nonterminal. Thus, in the canonical LR(1) parser tables of Table 1, state 4 is a *predecessor* of states 6 and 11, which are thus *successors* of state 4.

Definition 2.14 (*Accessing Symbol*)**.** Given an LR(1) state s, then $\dagger(s)$ is the *accessing symbol* of s. Because of the way LR(1) parser tables are generated, the following statements are always true:

1. $\forall ((\ell \to \varrho), d, K, a) \in s : d > 1, \varrho[d-1] = \dagger(s)$.
2. Given any LR(1) state s', if $\exists y : \delta(s', y) = s$, then $y = \dagger(s)$.
3. Iff s is the start state, $\nexists \dagger(s)$ because (1) throughout the start state, the dot in every item is at the leftmost position and (2) no state has a transition to the start state. Throughout this paper, we formally identify start states using this property. $\square$

Table 1
Parser tables for the unambiguous grammar. These are the canonical LR(1) and LALR(1) parser tables for the grammar of Fig. 1. Differences between canonical LR(1) and LALR(1) are shown in bold. In both cases, state 9 has a S/R conflict on a resolved as a reduce because a is left-associative.

Canonical LR(1)			LALR(1)		
0. $S' \to \cdot S\#$,	$\{\}$	G1	0. $S' \to \cdot S\#$,	$\{\}$	G1
$S \to \cdot aAa$,	$\{\#\}$	S3	$S \to \cdot aAa$,	$\{\#\}$	S3
$S \to \cdot bAb$,	$\{\#\}$	S4	$S \to \cdot bAb$,	$\{\#\}$	S4
1. $S' \to S \cdot \#$,	$\{\}$	S2	1. $S' \to S \cdot \#$,	$\{\}$	S2
2. $S' \to S\# \cdot$,	$\{\}$	Acc	2. $S' \to S\# \cdot$,	$\{\}$	Acc
3. $S \to a \cdot Aa$,	$\{\#\}$	G5	3. $S \to a \cdot Aa$,	$\{\#\}$	G5
$A \to \cdot a$,	$\{a\}$	S9	$A \to \cdot a$,	$\{a\}$	S9
$A \to \cdot aa$,	$\{a\}$	S9	$A \to \cdot aa$,	$\{a\}$	S9
4. $S \to b \cdot Ab$,	$\{\#\}$	G6	4. $S \to b \cdot Ab$,	$\{\#\}$	G6
$A \to \cdot a$,	$\{b\}$	**S11**	$A \to \cdot a$,	$\{b\}$	**S9**
$A \to \cdot aa$,	$\{b\}$	**S11**	$A \to \cdot aa$,	$\{b\}$	**S9**
5. $S \to aA \cdot a$,	$\{\#\}$	S7	5. $S \to aA \cdot a$,	$\{\#\}$	S7
6. $S \to bA \cdot b$,	$\{\#\}$	S8	6. $S \to bA \cdot b$,	$\{\#\}$	S8
7. $S \to aAa \cdot$,	$\{\#\}$	R1	7. $S \to aAa \cdot$,	$\{\#\}$	R1
8. $S \to bAb \cdot$,	$\{\#\}$	R2	8. $S \to bAb \cdot$,	$\{\#\}$	R2
*9. $A \to a \cdot$,	$\boldsymbol{\{a\}}$	R3	*9. $A \to a \cdot$,	$\boldsymbol{\{ab\}}$	R3
$A \to a \cdot a$,	$\boldsymbol{\{a\}}$	~~S10~~	$A \to a \cdot a$,	$\boldsymbol{\{ab\}}$	~~S10~~
10. $A \to aa \cdot$,	$\boldsymbol{\{a\}}$	R4	10. $A \to aa \cdot$,	$\boldsymbol{\{ab\}}$	R4
11. $\boldsymbol{A \to a \cdot}$,	$\boldsymbol{\{b\}}$	**R3**			
$\boldsymbol{A \to a \cdot a}$,	$\boldsymbol{\{b\}}$	**S12**			
12. $\boldsymbol{A \to aa \cdot}$,	$\boldsymbol{\{b\}}$	**R4**			

Table 2
A parse for the unambiguous grammar. This table shows how the canonical LR(1) and LALR(1) parser tables of Table 1 parse the sentence *baab*. Because LALR(1) merges state 11 into state 9, it rejects the sentence even though the canonical LR(1) parser accepts it.

Canonical LR(1)			LALR(1)		
Stack	Input	Action	Stack	Input	Action
0	*baab#*	S4	0	*baab#*	S4
0, 4	*aab#*	S11	0, 4	*aab#*	S9
0, 4, 11	*ab#*	S12	0, 4, 9	*ab#*	R3
0, 4, 11, 12	*b#*	R4	0, 4	*Aab#*	G6
0, 4	*Ab#*	G6	0, 4, 6	*ab#*	Err
0, 4, 6	*b#*	S8			
0, 4, 6, 8	*#*	R2			
0	*S#*	G1			
0, 1	*#*	S2			
0, 1, 2		Acc			

For example, in the canonical LR(1) parser tables of Table 1 the accessing symbol for states 3, 9, and 11 is a, but for state 4 it is b. State 0 has no accessing symbol because it is the start state.

Definition 2.15 (*Lanes*). Given a set of LR(1) states Σ and an LR(1) state $s \in \Sigma$, then the set of ***lanes*** in Σ for s is $\mathscr{L}(s, \Sigma) = \{\lambda \in \Sigma^+ : \nexists\dagger(\lambda[1]) \wedge \lambda[|\lambda|] = s \wedge \forall i : 1 < i \le |\lambda|, \delta(\lambda[i-1], \dagger(\lambda[i])) = \lambda[i]\}$. By Definition 2.14, if s is the start state, $\mathscr{L}(s, \Sigma) = \{(s)\}$. □

Definition 2.16 (*Viable Prefixes*). Given a set of LR(1) states Σ and an LR(1) state $s \in \Sigma$, then the set of *viable prefixes* of s within Σ is $\mathscr{V}(s, \Sigma) = \{\upsilon : \exists s' \in \Sigma : \nexists\dagger(s') \wedge \delta^*(s', \upsilon) = s\}$. In other words, a viable prefix is the sequence of accessing symbols corresponding to a lane. Thus, if s is the start state, $\mathscr{V}(s, \Sigma) = \{\epsilon\}$. □

For example, in the canonical LR(1) parser tables of Table 1, state 9's only lane[2] is (0, 3, 9), and its only viable prefix is *aa*. State 11's only lane is (0, 4, 11) and its only viable prefix is *ba*. State 0's only lane is (0) and its only viable prefix is ϵ.

Definition 2.17 (*LR(1) Conflict Contributions*)**.** Given an LR(1) state s and token t, then the ***contributions*** to a conflict on t in s are $\Gamma(t, s) = \{(a_t, a_p) : \exists \ell : \exists \varrho : \exists d : \exists K : \exists a_s : ((\ell \to \varrho), d, K, (a_t, a_p, a_s)) \in s : (a_t = \text{“S”} \wedge \varrho[d] = t) \vee (a_t = \text{“R”} \wedge t \in K)\}$. □

Definition 2.18 (*LR(1) Conflict*)**.** Given an LR(1) state s and token t, then s has an *LR(1) conflict* on t iff $|\Gamma(t, s)| \geq 2$. In that case, s is said to be a *conflicted state*, and t is said to be a *conflicted token*. □

Definition 2.19 (*Dominant Contribution Function*)**.** Iff a function Δ is a ***dominant contribution function*** for a set of LR(1) states Σ, then $\forall s \forall t : s \in \Sigma \wedge |\Gamma(t, s)| \geq 1, \Delta(t, \Gamma(t, s)) \in \Gamma(t, s)$. □

Definition 2.20 (*LR(1) Parser, Conflicts Resolved*)**.** Given an LR(1) parser table generation function $\mathscr{F}$ (such as $\mathscr{C}$), then there is an LR(1) parser table generation function $\mathscr{F}_r$ (such as $\mathscr{C}_r$) such that, for any context-free grammar G and any dominant contribution function Δ for $\mathscr{F}(G)$, $\mathscr{F}_r(G, \Delta)$ is the set $\mathscr{F}(G)$ but with any LR(1) conflicts resolved. In order to resolve those conflicts, for every LR(1) state $s \in \mathscr{F}(G)$ and for every augmented LR(1) item $((\ell \to \varrho), d, K, (a_t, a_p, a_s)) \in s$, $\mathscr{F}_r$ performs the following modifications:

1. If $a_t = \text{“R”}$, then $\mathscr{F}_r$ rewrites K as $\{t : \Delta(t, \Gamma(t, s)) = (a_t, a_p)\}$.
2. If $a_t = \text{“S”}$ and $\Delta(\varrho[d], \Gamma(\varrho[d], s)) \neq (a_t, a_p)$, then $\mathscr{F}_r$ rewrites a_t as “”. □

Because canonical LR(1) is the most general technique for generating LR(1) parser tables for a grammar, a conflict in a grammar's canonical LR(1) parser tables indicates that the grammar is non-LR(1). For example, state 9 in Table 1 has both a shift and a reduce action that contribute to the conflict on token *a*. This is a *S/R conflict*. Because *a* is declared left-associative, the reduce action is the dominant contribution, so the conflict is *resolved* as the reduce action. If *a* had no declared associativity, a parser generator like Yacc or Bison would usually report the conflict as unresolved by the user and would then resolve it automatically by choosing the shift action as the dominant contribution.

A conflict whose contributions are all reduce actions is a *R/R conflict*. For example, the first column of Table 3 shows the canonical LR(1) parser tables for the grammar of Fig. 2. State 20 has a R/R conflict on *a*. Yacc and Bison (without GLR (Generalized LR) mode [32]) provide no mechanism for the user to resolve or even suppress warnings about R/R conflicts, which are considered by many grammar developers to be a sign of a grammar design problem [17]. Instead, the user must eliminate them by restructuring the grammar. Until he does so, the parser generator resolves each R/R conflict by choosing to reduce by the production that appears earliest in the grammar. Thus, R8 is the dominant contribution on *a* in state 20.

If a conflict left unresolved by the user has a shift contribution and multiple reduce contributions, Bison reports it twice: once as a S/R conflict and once as a R/R conflict. Nevertheless, the parser generator must choose just one dominant contribution from all of the contributions in order to resolve the conflict. Thus, our model treats such a conflict as a single S/R conflict.

We are now ready to formalize Definition Definition 2.5 using the model we have developed so far.

Definition 2.21 (*LR(1) Parser Specification, Formal*)**.** An *LR(1) parser specification* is a tuple (G, Δ) such that G is a context-free grammar and Δ is a dominant contribution function for $\mathscr{C}(G)$. The set of parse trees generated by an LR(1) parser that fulfills this specification is the set of parse trees generated by the $\mathscr{C}_r(G, \Delta)$ parser: $\mathscr{T}(G, \Delta) \subseteq \mathscr{T}(G)$. The language accepted by such an LR(1) parser is thus the language accepted by the $\mathscr{C}_r(G, \Delta)$ parser: $L(G, \Delta) \subseteq L(G)$. □

The term *inadequacy* is often used synonymously with conflict when discussing LR parsing. However, in this paper, we draw important distinctions between the two terms. Moreover, we identify two different kinds of inadequacies that an LR(1) parser may exhibit. We define the first kind here, and we define the second kind in Section 2.5.4. In both definitions, it is important to notice that a parser whose conflicts have all been resolved still contains inadequacies.

Definition 2.22 (*Grammar-Relative Inadequacy*)**.** Given an LR(1) parser table generation function $\mathscr{F}$, an LR(1) parser specification (G, Δ), an LR(1) state $s \in \mathscr{F}(G)$, and a token t, then, iff s has a conflict on t, we say that both $\mathscr{F}(G)$ and $\mathscr{F}_r(G, \Delta)$ have a ***grammar-relative inadequacy*** that ***manifests*** as that conflict in s. That is, the parsers defined by $\mathscr{F}(G)$ and $\mathscr{F}_r(G, \Delta)$ are not adequate to generate every parse tree in $\mathscr{T}(G)$. □

In this section, we have established canonical LR(1) as a formal basis against which to compare the behavior of any type of LR(1) parser. In doing so, we have formalized our LR(1) parser specification model, and we have defined formal models for many related concepts in order to facilitate such comparisons. These models are also vital to the definition of our IELR(1) algorithm in Section 3.

[2] Pager defines the term *lane* with a significantly different but related meaning [25–27].

Table 3
Parser tables for the ambiguous grammar. These are the canonical LR(1) and LALR(1) parser tables for the grammar of Fig. 2. We have omitted states 5 through 16, which are the same between canonical LR(1) and LALR(1).

Canonical LR(1)			LALR(1)		
0. $S' \rightarrow \cdot S\#$,	$\{\}$	G1	0. $S' \rightarrow \cdot S\#$,	$\{\}$	G1
$S \rightarrow \cdot aAa$,	$\{\#\}$	S3	$S \rightarrow \cdot aAa$,	$\{\#\}$	S3
$S \rightarrow \cdot aBb$,	$\{\#\}$	S3	$S \rightarrow \cdot aBb$,	$\{\#\}$	S3
$S \rightarrow \cdot aCc$,	$\{\#\}$	S3	$S \rightarrow \cdot aCc$,	$\{\#\}$	S3
$S \rightarrow \cdot bAb$,	$\{\#\}$	S4	$S \rightarrow \cdot bAb$,	$\{\#\}$	S4
$S \rightarrow \cdot bBa$,	$\{\#\}$	S4	$S \rightarrow \cdot bBa$,	$\{\#\}$	S4
$S \rightarrow \cdot bCa$,	$\{\#\}$	S4	$S \rightarrow \cdot bCa$,	$\{\#\}$	S4
1. $S' \rightarrow S \cdot \#$,	$\{\}$	S2	1. $S' \rightarrow S \cdot \#$,	$\{\}$	S2
2. $S' \rightarrow S\# \cdot$,	$\{\}$	Acc	2. $S' \rightarrow S\# \cdot$,	$\{\}$	Acc
3. $S \rightarrow a \cdot Aa$,	$\{\#\}$	G5	3. $S \rightarrow a \cdot Aa$,	$\{\#\}$	G5
$S \rightarrow a \cdot Bb$,	$\{\#\}$	G9	$S \rightarrow a \cdot Bb$,	$\{\#\}$	G9
$S \rightarrow a \cdot Cc$,	$\{\#\}$	G13	$S \rightarrow a \cdot Cc$,	$\{\#\}$	G13
$A \rightarrow \cdot aa$,	$\{a\}$	S17	$A \rightarrow \cdot aa$,	$\{a\}$	S17
$B \rightarrow \cdot aa$,	$\{b\}$	S17	$B \rightarrow \cdot aa$,	$\{b\}$	S17
$C \rightarrow \cdot aa$,	$\{c\}$	S17	$C \rightarrow \cdot aa$,	$\{c\}$	S17
4. $S \rightarrow b \cdot Ab$,	$\{\#\}$	G7	4. $S \rightarrow b \cdot Ab$,	$\{\#\}$	G7
$S \rightarrow b \cdot Ba$,	$\{\#\}$	G11	$S \rightarrow b \cdot Ba$,	$\{\#\}$	G11
$S \rightarrow b \cdot Ca$,	$\{\#\}$	G15	$S \rightarrow b \cdot Ca$,	$\{\#\}$	G15
$A \rightarrow \cdot aa$,	$\{b\}$	**S19**	$A \rightarrow \cdot aa$,	$\{b\}$	**S17**
$B \rightarrow \cdot aa$,	$\{a\}$	**S19**	$B \rightarrow \cdot aa$,	$\{a\}$	**S17**
$C \rightarrow \cdot aa$,	$\{a\}$	**S19**	$C \rightarrow \cdot aa$,	$\{a\}$	**S17**
⋮			⋮		
17. $A \rightarrow a \cdot a$,	$\boldsymbol{\{a\}}$	S18	17. $A \rightarrow a \cdot a$,	$\boldsymbol{\{ab\}}$	S18
$B \rightarrow a \cdot a$,	$\boldsymbol{\{b\}}$	S18	$B \rightarrow a \cdot a$,	$\boldsymbol{\{ba\}}$	S18
$C \rightarrow a \cdot a$,	$\boldsymbol{\{c\}}$	S18	$C \rightarrow a \cdot a$,	$\boldsymbol{\{ca\}}$	S18
18. $A \rightarrow aa \cdot$,	$\boldsymbol{\{a\}}$	R7	*18. $A \rightarrow aa \cdot$,	$\boldsymbol{\{ab\}}$	R7
$B \rightarrow aa \cdot$,	$\boldsymbol{\{b\}}$	R8	$B \rightarrow aa \cdot$,	~~$\boldsymbol{\{ba\}}$~~	~~R8~~
$C \rightarrow aa \cdot$,	$\boldsymbol{\{c\}}$	R9	$C \rightarrow aa \cdot$,	$\boldsymbol{\{c}$~~$\boldsymbol{a}$~~$\boldsymbol{\}}$	R9
19. $\boldsymbol{A \rightarrow a \cdot a}$,	$\{\boldsymbol{b}\}$	**S20**			
$\boldsymbol{B \rightarrow a \cdot a}$,	$\{\boldsymbol{a}\}$	**S20**			
$\boldsymbol{C \rightarrow a \cdot a}$,	$\{\boldsymbol{a}\}$	**S20**			
***20.** $\boldsymbol{A \rightarrow aa \cdot}$,	$\{\boldsymbol{b}\}$	**R7**			
$\boldsymbol{B \rightarrow aa \cdot}$,	$\{\boldsymbol{a}\}$	**R8**			
$\boldsymbol{C \rightarrow aa \cdot}$,	~~$\{\boldsymbol{a}\}$~~	~~**R9**~~			

2.5. LALR(1) conflicts reconsidered

Using the model we have developed so far, we now show how LALR(1) parser tables are constructed using a *merge* function that merges the lookahead sets of canonical LR(1) isocore sets.

Definition 2.23 (*LALR(1) Parser*)**.** Given a context-free grammar G and an LR(1) state s, then $merge(s, G)$ is the state s except that, for every augmented LR(1) item $(p, d, K, (a_t, a_p, a_s)) \in s$, *merge* performs the following modifications:

1. *merge* rewrites K as $\{t : \exists s' \in \mathscr{I}(s, \mathscr{C}(G)) : \exists K' : \exists a'_s : (p, d, K', (a_t, a_p, a'_s)) \in s' \wedge t \in K'\}$.
2. If $a_t =$ "S" or $a_t =$ "G", then *merge* replaces a_s with $merge(a_s, G)$.

Given a context-free grammar G, then $\mathscr{A}(G) = \{merge(s, G) : s \in \mathscr{C}(G)\}$ is the set of LR(1) states for G's *LALR(1) parser*. □

The second column of Table 1 shows the LALR(1) parser tables for the grammar in Fig. 1. They are nearly identical to the canonical LR(1) parser tables but every isocore set is fully merged. For example, there is no longer a state 11. Each lookahead set of LALR(1) state 9 is the union of the corresponding lookahead sets from canonical LR(1) states 9 and 11. Canonical LR(1) states 10 and 12 are also isocores and are merged in a similar fashion to form LALR(1) state 10.

In this section, we discuss the various categories of conflicts that can appear in LALR(1) parser tables relative to canonical LR(1) parser tables as a result of state merging. We also generalize these categories for minimal LR(1) parser tables.

2.5.1. Mysterious new conflicts

Merging canonical LR(1) isocores to create LALR(1) parser tables can create new conflicts that are not present in the canonical LR(1) parser tables. The Bison manual calls these *mysterious conflicts* because they can be a source of confusion for parser specification developers [17]. The trouble is that a simple grammar analysis as we performed in Section 2.3 does not easily reveal mysterious conflicts. To discover them, the parser specification developer is forced to examine parser tables.

In this paper, we rename what the Bison manual calls mysterious conflicts to ***mysterious new conflicts*** because we wish to emphasize that this term always refers to conflicts that do not exist in the canonical LR(1) parser tables. As the Bison manual points out, such conflicts are always R/R conflicts [17] because, as Aho et al. point out, merging isocores can never create new S/R conflicts [13].

For example, the second column of Table 3 shows the LALR(1) parser tables for the grammar in Fig. 2. Canonical LR(1) states 18 and 20 are isocores and are merged to form LALR(1) state 18, which has a R/R conflict on b. This conflict has two contributions: R8, which is contributed by canonical LR(1) state 18, and R7, which is contributed by canonical LR(1) state 20. Because canonical LR(1) states 18 and 20 each make just one contribution, neither alone has the conflict, so the conflict is new in the LALR(1) parser tables.

Canonical LR(1) state 20 has one viable prefix, *baa*. After that viable prefix and upon seeing a lookahead of b, both the canonical LR(1) and LALR(1) parsers perform R7 and ultimately accept the sentence *baab*. In other words, from the perspective of the viable prefix of canonical LR(1) state 20, LALR(1) state 18's conflict on b is new, but it causes no mysterious parser behavior because the dominant contribution stays the same.

Canonical LR(1) state 18 has one viable prefix, *aaa*. After that viable prefix and upon seeing a lookahead of b, the canonical LR(1) parser performs R8. Unfortunately for input sentence $aaa \cdot b$, which starts with that viable prefix and thus requires R8 on b, the LALR(1) parser performs R7 instead. That is, the dominant contribution from canonical LR(1) state 18 to the conflict on b is not the same as the dominant contribution from the merged LALR(1) state 18. This leads the LALR(1) parse of $aaa \cdot b$ to a syntax error even though canonical LR(1) is able to parse it successfully. Thus, we say that LALR(1) state 18's conflict on b is a mysterious new conflict from the perspective of the viable prefix of canonical LR(1) state 18.

2.5.2. Mysterious invasive conflicts

In both sets of parser tables in Table 1, state 9 contains the S/R conflict whose resolution as reduce (1) permits a successful completion of the parse of $aa \cdot a$ but (2) leads the parse of $aa \cdot aa$ to a syntax error as discussed in Section 2.3. The corresponding state in the parse of $ba \cdot b$ and $ba \cdot ab$ is canonical LR(1) state 11, which has no conflict and thus permits a successful completion of the parse of both sentences.

Because the LALR(1) algorithm merges state 11 into state 9, the LALR(1) parser encounters the conflict of state 9 even after the viable prefix of state 11. Unfortunately for $ba \cdot ab$, which starts with that viable prefix and requires a shift on a, the conflict is resolved as a reduce. That is, the dominant contribution from canonical LR(1) state 11 to this conflict is not the same as the dominant contribution from the corresponding merged LALR(1) state. As Table 2 demonstrates, this leads the LALR(1) parse of $ba \cdot ab$ to a syntax error even though canonical LR(1) is able to parse it successfully.

Because this conflict already exists in the canonical LR(1) parser tables, it is not a mysterious new conflict. Instead, we have identified a second category of mysterious conflicts: ***mysterious invasive conflicts***. We say that this conflict is a mysterious invasive conflict from the perspective of the viable prefix of canonical LR(1) state 11 because state 11's parser actions were altered by merging with the existing conflict of canonical LR(1) state 9.

2.5.3. Mysterious mutated conflicts

In the canonical LR(1) parser tables of Table 3, state 20 contains a R/R conflict on a with two contributions: R8 and R9. Its canonical LR(1) isocore, state 18, does not contain a conflict on a but does contain one contribution: R7. The corresponding merged state, LALR(1) state 18, contains all three contributions.

As a result, the LALR(1) parser encounters an expanded version of canonical LR(1) state 20's conflict on a even after the viable prefix of canonical LR(1) state 18. In that sense, the conflict is invasive from the perspective of the viable prefix of canonical LR(1) state 18. However, also from that perspective, the conflict causes no mysterious parser behavior because the dominant contribution from canonical LR(1) state 18 is the same as that from the merged LALR(1) state 18.

Instead, we have identified a third category of mysterious conflicts: ***mysterious mutated conflicts***. We say this conflict is a mysterious mutated conflict from the perspective of the viable prefix of canonical LR(1) state 20 because state 20 already had the conflict but its dominant contribution is not the same as that from the merged LALR(1) state 18. Whereas the canonical LR(1) parser successfully completes the parse of $baa \cdot a$, this change in parser actions leads the LALR(1) parser to a syntax error.

2.5.4. Formal categorization of conflicts

Like $\mathscr{A}$, minimal LR(1) parser table generators usually merge the lookahead sets of a set of isocores I to form s. In the case of $\mathscr{A}$, $I = \mathscr{I}(s, \mathscr{C}(G))$. However, for minimal LR(1), $I \subseteq \mathscr{I}(s, \mathscr{C}(G))$. We can now formalize our categorization of parser table conflicts and make some important observations about some of those categories.

Definition 2.24 (*Conflict Categorization*). Given an LR(1) parser specification (G, Δ), an LR(1) state s, and a token t, then s has a conflict on t iff $|\Gamma(t, s)| \geq 2$. Let I be the set of states whose lookahead sets were merged to form s such that

$I \subseteq \mathscr{I}(s, \mathscr{C}(G))$. Given some $i \in I$, then the conflict on t in s is, from the perspective of $\mathscr{V}(i, \mathscr{C}(G))$:

1. ***Irrelevant*** iff $|\Gamma(t, i)| = 0$.
2. ***Relevant*** otherwise. In that case, from the same perspective, the conflict is exactly one of:
   (a) ***New*** iff $\forall i' \in I, |\Gamma(t, i')| < 2$.
   (b) ***Invasive*** iff $|\Gamma(t, i)| = 1$ and $\exists i' \in I : |\Gamma(t, i')| \geq 2$.
   (c) ***Mutated*** iff $|\Gamma(t, i)| \geq 2$ and $\Gamma(t, i) \neq \Gamma(t, s)$.
   (d) ***Stable*** iff $|\Gamma(t, i)| \geq 2$ and $\Gamma(t, i) = \Gamma(t, s)$.

Finally, from the same perspective, the conflict is ***mysterious*** iff it is relevant and $\Delta(t, \Gamma(t, i)) \neq \Delta(t, \Gamma(t, s))$. □

Observation 2.25 (*Conflict Restrictions*)**.** Given an LR(1) parser specification (G, Δ), an LR(1) state s with a conflict on token t, the set of states $I \subseteq \mathscr{I}(s, \mathscr{C}(G))$ whose lookahead sets were merged to form s, and some $i \in I$, then, from the perspective of $\mathscr{V}(i, \mathscr{C}(G))$:

1. The conflict can be new only if it is a R/R conflict. We discussed this in Section 2.5.1.
2. The conflict can be invasive, mutated, or stable only if G is non-LR(1) because these categories require a version of the conflict to exist in the canonical LR(1) tables.
3. The conflict can be mysterious only if it is new, invasive, or mutated. Because a stable conflict does not experience any change in its set of contributions, it cannot experience a change in dominant contribution and so cannot be mysterious.
4. The conflict can be a mutated conflict only if it has multiple reduce contributions. The proof is as follows. By Definition 2.17, because $i \in I$, $\Gamma(t, i) \subseteq \Gamma(t, s)$. By the definition of mutated, $\Gamma(t, i) \neq \Gamma(t, s)$, so $\Gamma(t, i) \subset \Gamma(t, s)$ and $|\Gamma(t, i)| < |\Gamma(t, s)|$. By the definition of mutated, $|\Gamma(t, i)| \geq 2$, so $|\Gamma(t, s)| \geq 3$. Only one of the at least 3 contributions in s can be a shift, so the at least 2 remaining contributions must be reduces.
5. By the observations above, if the conflict is mysterious and has only one reduce contribution, it must be an invasive S/R conflict. □

In Section 4.2, we analyze the results of our case studies using Observation 2.25 in order to explain the special significance of mysterious invasive S/R conflicts for real-world LR(1) parser specifications.

In Section 2.4, we defined the first of two kinds of inadequacies that an LR(1) parser may exhibit. We are now ready to define the second kind for LALR(1) parsers and minimal LR(1) parsers. The discussion in Section 2.6 further refines this definition.

Definition 2.26 (*LR(1)-Relative Inadequacy*)**.** Given an LR(1) parser table generation function $\mathscr{F}$, an LR(1) parser specification (G, Δ), an LR(1) state $s \in \mathscr{F}(G)$, and a token t, then, iff s has a mysterious conflict on t, we say that both $\mathscr{F}(G)$ and $\mathscr{F}_r(G, \Delta)$ have an ***LR(1)-relative inadequacy*** that ***manifests*** as that conflict in s. That is, the parsers defined by $\mathscr{F}(G)$ and $\mathscr{F}_r(G, \Delta)$ are not adequate to generate every parse tree in $\mathscr{T}(G, \Delta)$. □

Observation 2.27 (*Inadequacy Subtype*)**.** Comparing Definitions 2.22 and 2.26 reveals that every LR(1)-relative inadequacy is a grammar-relative inadequacy because every mysterious conflict is a conflict. □

2.5.5. Summary

Given the grammar of Fig. 1, given that a is left-associative, and assuming an LR(1) parser, the language so specified is a set of 3 sentences: *aaa*, *bab*, and *baab*. As expected, the canonical LR(1) parser tables for this specification accept exactly that language. However, LALR(1) diminishes the language to only 2 sentences: *aaa* and *bab*. Thus, by Definition 2.21, LALR(1) is not able to fulfill this LR(1) parser specification. The culprit is not what the Bison manual calls a mysterious conflict, which we call a mysterious *new* conflict. The culprit belongs to a new category of mysterious conflicts that we call mysterious *invasive* conflicts. That is, switching from canonical LR(1) to LALR(1) can cause a parser to encounter existing conflicts after viable prefixes for which it never would have encountered those conflicts before and, as a result, to perform incorrect actions. This switch can also cause a parser to perform incorrect actions after viable prefixes for which it would have encountered an existing conflict before. We call this is a mysterious *mutated* conflict.

2.6. The deficiency of Pager's algorithm

Many algorithms for generating LR(1) parser tables merge states only if they pass some sort of *compatibility test*. Canonical LR(1) parser tables are relatively large because the compatibility test is relatively restrictive: states must be isocores and their lookahead sets must be identical. LALR(1) parser tables are sometimes too small to maintain the full power of LR(1) because the compatibility test is too lenient: states must simply be isocores. David Pager describes an algorithm to generate LR(1) parser tables while employing tests for two kinds of compatibility, *weak compatibility* and *strong compatibility*, both of which lie somewhere in between [26]. Throughout this paper, we refer to this algorithm as *Pager's algorithm*. While Pager does suggest a generalization of his algorithm to LR(k) with $k > 1$, we assert that such a generalization does not resolve the fundamental issues we address in this section, so we restrict our discussion to the $k = 1$ case for simplicity.

Given an LR(1) grammar, neither Pager's weak nor his strong compatibility test ever permits a merge that would induce a mysterious new conflict in any state. The advantage of Pager's strong compatibility test is that it never rejects a merge that cannot induce a mysterious new conflict. However, Pager's weak compatibility test may. One advantage of Pager's weak compatibility test is that it can be performed faster than Pager's strong compatibility test. Thus, Pager recommends that his strong compatibility test be performed only if his weak compatibility test rejects a merge. Moreover, Pager's weak

1. $S \rightarrow aAa$	4. $S \rightarrow bAb$	7. $A \rightarrow aa$
2. $S \rightarrow aBa$	5. $S \rightarrow bBa$	8. $B \rightarrow aa$
3. $S \rightarrow aCa$	6. $S \rightarrow bCa$	9. $C \rightarrow aa$

Fig. 3. Another ambiguous grammar. This grammar is the grammar of Fig. 2 with the last token in each of productions 2 and 3 replaced with a. This affects the lookahead sets in canonical LR(1) states 17 and 18 in Table 3 such that they now pass Pager's weak compatibility test with states 19 and 20.

compatibility test is conceptually simpler to implement, and Pager suggests that omitting his strong compatibility test does not usually increase parser table size significantly, if at all, for practical grammars.

The correctness of Pager's weak compatibility test relies on the assumption that the given grammar is LR(1). Pager defines a *potential conflict* as the occurrence of the same token in two items' lookahead sets within the same state. If a grammar is LR(1), a potential conflict in a canonical LR(1) state cannot be a real conflict or lead to a real conflict in a successor state because that would require a non-LR(1) grammar. Because it cannot, Pager shows that the same potential conflict in the merged state cannot either. Thus, stated simply, the weak compatibility test does not permit two isocores to be merged if the merged state would have a potential conflict unless one of the isocores alone already has a similar potential conflict.

We now state Pager's definition of weak compatibility formally in terms of our model.

Definition 2.28 (*Pager's Weak Compatibility*). Given two LR(1) states s and s', those states pass *Pager's weak compatibility* test iff both of the following are true:

1. $s \overset{\circ}{=} s'$.
2. $\forall (p_1, d_1) : \exists (p_1, d_1, K_1, a_1) \in s \wedge \exists (p_1, d_1, K_1', a_1') \in s'$,
   $\forall (p_2, d_2) : \exists (p_2, d_2, K_2, a_2) \in s \wedge \exists (p_2, d_2, K_2', a_2') \in s'$,
   at least one of the following is true:
   (a) $K_1 \cap K_2' = \emptyset \wedge K_2 \cap K_1' = \emptyset$.
   (b) $K_1 \cap K_2 \neq \emptyset$.
   (c) $K_1' \cap K_2' \neq \emptyset$. □

For example, in Table 3 consider the mysterious new conflict on *b* in LALR(1) state 18, which is the merge of canonical LR(1) states 18 and 20. Canonical LR(1) states 18 and 20 fail Pager's weak compatibility test because the intersection of the first two lookahead sets is non-empty for the merged state but empty for each isocore. The same is true for the first and third lookahead sets, but one such failure is enough to fail the test. However, this one application of the test is not enough to prevent the merge. Consider their only predecessors, states 17 and 19, which transition on *a* to states 18 and 20. If states 17 and 19 were merged to form LALR(1) state 17, there could be only one successor on *a*, so states 18 and 20 would then also have to be merged to form LALR(1) state 18.

By Definition 2.26, the conflict on *b* in LALR(1) state 18 is the manifestation of an LR(1)-relative inadequacy of the LALR(1) parser. However, as we have just shown, the inadequacy is not caused by LALR(1) state 18 alone. In this paper, we call both LALR(1) states 17 and 18 ***LR(1)-relative inadequate*** states because they both must be split in order to eliminate the mysterious new conflict of LALR(1) state 18. That is, when collectively considering all inadequate states that together result in an individual conflict on an individual token, we refer to that collection as an individual inadequacy of the parser tables. Moreover, even though LALR(1) state 17 is not the conflicted state, we still say it makes two contributions to the inadequacy. That is, it has *b* in the lookahead sets of its first two items and those lookaheads *propagate* to the first two lookahead sets of LALR(1) state 18 where they manifest as the two reduce contributions.

Unfortunately, once conflict resolution has been applied to a mysterious conflict, the inadequacy then manifests only as an incorrect parser action, which is more subtle to detect. Fortunately, unlike LALR(1), Pager's algorithm successfully avoids the inadequacy that manifests as a mysterious conflict on *b* in LALR(1) state 18: canonical LR(1) states 17 and 19 fail Pager's weak compatibility test just as canonical LR(1) states 18 and 20 do. By avoiding all mysterious conflicts and thus not allowing the parser actions to differ from canonical LR(1) for any viable prefix, Pager's algorithm generates parser tables for the grammar of Fig. 2 that generate the same parse trees and accept the same language as canonical LR(1).

We identify two problems with Pager's algorithm for our purposes. Both stem from the fact that Pager's algorithm is not designed for non-LR(1) grammars. First, Pager's weak compatibility test does not always correctly handle R/R conflicts that are already present in the canonical LR(1) states. For example, in the grammar of Fig. 2, if we replace the last token in each of productions 2 and 3 with *a* to produce the grammar of Fig. 3, all lookahead sets then contain only *a* in canonical LR(1) states 17 and 18 in Table 3. Thus, when merging canonical LR(1) states 17 with 19 and 18 with 20, there are no new potential conflicts, so Pager's algorithm does not avoid the resulting mysterious mutated conflict. Second, neither Pager's weak nor his strong compatibility test considers S/R conflicts. For example, the strong compatibility test ignores the S/R conflict in canonical LR state 9 in Table 1 and so fails to avoid the mysterious invasive conflict that results from the merge with state 11. While the weak compatibility test occasionally rejects additional merges, that does not manage to help in this example. Each of the potential conflicts that appears in the merged state 9 already appears in one of its canonical LR(1) isocores.

Initially released in 2005, Menhir is an implementation of Pager's algorithm [12]. In this paper, we examine Menhir version 20070322 [7]. At the time of this writing, Menhir is the most robust minimal LR(1) implementation we have reviewed, so we have selected it for comparison with our own work. Most importantly, Menhir is the only minimal LR(1) implementation that we have reviewed that is able to generate parser tables without conflicts for every LR(1) grammar we have tested. For avoiding mysterious conflicts for non-LR(1) grammars, Menhir is more robust than Pager's original

$$\begin{array}{llllll} 1. & S \to aAa & 4. & S \to bAa & 6. & A \to a \\ 2. & S \to aAb & 5. & S \to bBb & 7. & B \to a \\ 3. & S \to aBa & & & & \end{array}$$

Fig. 4. Pager vs Menhir. For this grammar, Pager's algorithm generates parser tables that contain a mysterious new conflict on *b* between reductions 6 and 7. However, Menhir rejects the isocore merge that creates this conflict.

algorithm because, as revealed by Menhir's source comments, Menhir applies Pager's weak compatibility "token-wise" in order to "potentially make conflict explanations easier". We now state this revised definition of weak compatibility formally in terms of our model.

Definition 2.29 (*Menhir's Weak Compatibility*). Given two LR(1) states s and s', those states pass *Menhir's weak compatibility* test iff both of the following are true:

1. $s \stackrel{\circ}{=} s'$.
2. $\forall(p_1, d_1) : \exists(p_1, d_1, K_1, a_1) \in s \wedge \exists(p_1, d_1, K_1', a_1') \in s'$,
   $\forall(p_2, d_2) : \exists(p_2, d_2, K_2, a_2) \in s \wedge \exists(p_2, d_2, K_2', a_2') \in s'$,
   $\forall t$, at least one of the following is true:
   (a) $t \notin \{K_1 \cap K_2'\} \wedge t \notin \{K_2 \cap K_1'\}$.
   (b) $t \in \{K_1 \cap K_2\}$.
   (c) $t \in \{K_1' \cap K_2'\}$. □

Fig. 4 presents a non-LR(1) grammar for which Pager's weak compatibility test permits the creation of a mysterious new conflict[3] on token *b* because of an existing conflict on *a*. Because Menhir's weak compatibility test is token-wise, it rejects the merge that creates the conflict on *b* despite the existing conflict on *a*. We have not discovered any context-free grammar for which Menhir allows the creation of a mysterious new conflict.

Like Bison, Menhir is designed to accept a parser specification consisting of a non-LR(1) grammar coupled with a specification for resolving conflicts. Even though Menhir employs Pager's algorithm with an improved weak compatibility test, it fails to achieve the power of canonical LR(1) for some such parser specifications. For example, we have confirmed that, given the grammar of Fig. 1 combined with the declaration of *a* as left-associative, Menhir generates a parser that does not accept the sentence *baab*. We have also confirmed that, given the grammar of Fig. 3, Menhir generates a parser that does not accept the sentence *baaa*. Because canonical LR(1) parsers do accept these sentences, by Definition 2.21 Menhir is unable to fulfill these LR(1) parser specifications.

3. The IELR(1) algorithm

In this section, we describe our IELR(1) algorithm and our Bison implementation of it in 6 phases. We provide a high-level overview of these phases in Section 3.1, and we describe them in detail in Sections 3.2–3.7. In Section 3.8, we discuss some theoretical shortcomings of IELR(1) parser table efficiency. In Section 3.9, we explain how we parameterize the IELR(1) algorithm to generate full canonical LR(1) parser tables. Throughout, we assume an LR(1) parser specification (G, Δ) such that $G = (V, T, P, S)$ and $\mathscr{G}(G) = (V', T', P', S')$.

3.1. Overview

Pager's algorithm attempts to ***avoid*** LR(1)-relative inadequate states by refusing to merge isocores if its compatibility tests ***predict*** that the merge would induce a mysterious conflict somewhere in the parser tables. We use the term predict because the inadequacy may not manifest as a mysterious conflict until a successor state that the algorithm computes later. As we demonstrated in Section 2.6, Pager's compatibility tests fail to predict some mysterious conflicts for non-LR(1) grammars.

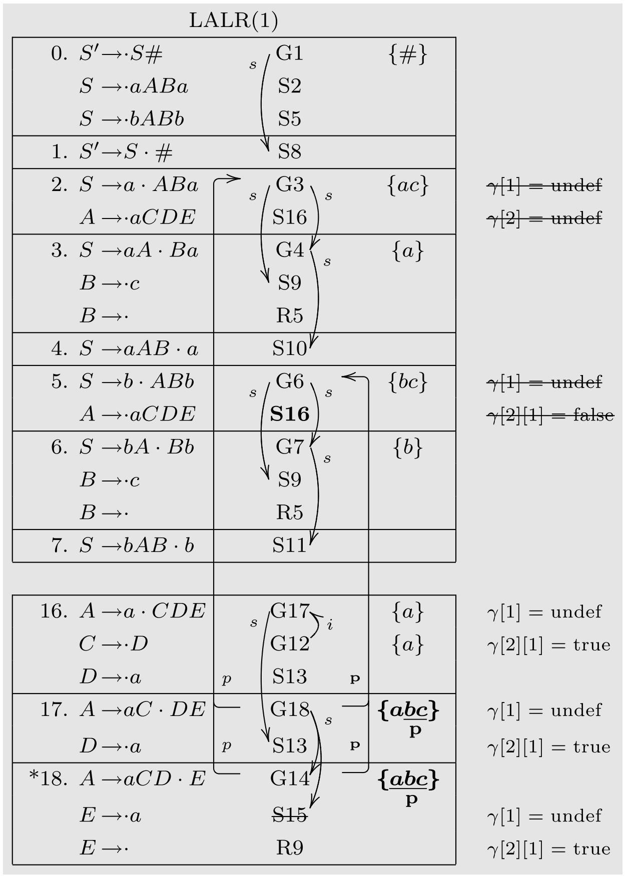
In contrast, our IELR(1) algorithm does not try to make predictions in order to avoid inadequate states. Instead, its first phase computes LALR(1) parser tables, which fully merge every isocore set and thus contain some form of every possible grammar-relative inadequacy that can exist after any possible combination of isocore merges. With Observation 2.27 in mind, IELR(1) then computes which of those grammar-relative inadequacies are also LR(1)-relative inadequacies, and it ***eliminates*** those that are by splitting inadequate states.

The IELR(1) algorithm consists of 6 phases, which we label phase 0 through phase 5:

- Phase 0: LALR(1). This phase computes the LALR(1) parser tables for *G*. To do so, it first computes LR(0) parser tables, and it then employs the algorithm of DeRemer and Pennello [16] to compute reduction lookahead sets from *goto follow sets*, which are sets of tokens that can follow goto actions during a syntactically correct parse. For example, Table 4 shows the cores and actions of interesting LALR(1) states for the grammar in Fig. 5. It also shows goto follow sets and diagrams their dependencies, some of which must be recomputed in later phases while splitting states. While neither DeRemer and

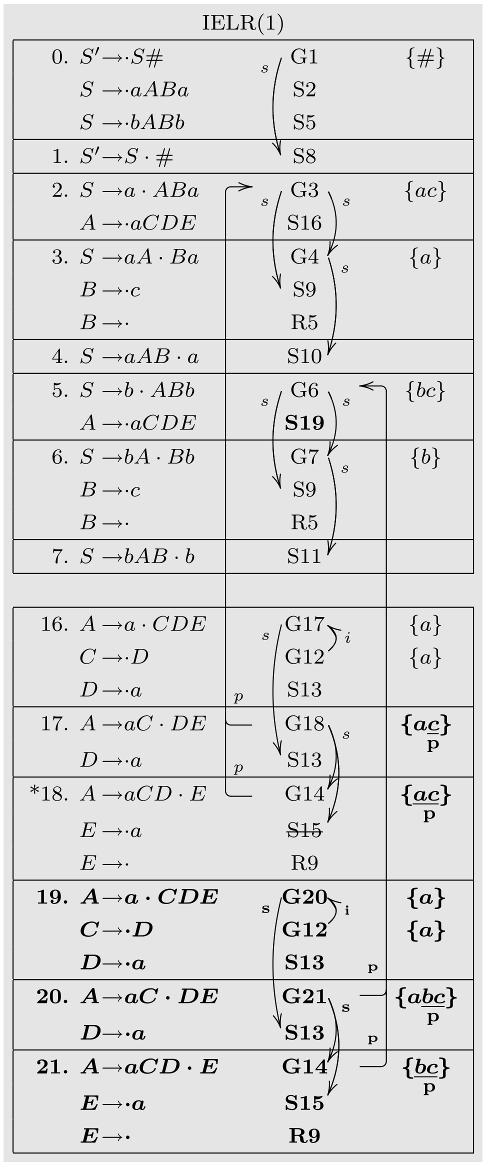
[3] Previously, we stated that Pager's weak compatibility test can always avoid mysterious new conflicts [15], but we did not note that this does not hold true for non-LR(1) grammars.

Table 4
LALR(1) parser tables with goto follows. This table shows Fig. 5's LALR(1) goto follow edges and sets, underlining follows that are only from predecessors. It also shows the inadequacy contribution matrices from the inadequacy annotations that phase 2 computes.



LALR(1)			
0. $S'\to\cdot S\#$	G1	$\{\#\}$	
$S\to\cdot aABa$	S2		
$S\to\cdot bABb$	S5		
1. $S'\to S\cdot\#$	S8		
2. $S\to a\cdot ABa$	G3	$\{ac\}$	~~$\gamma[1] = \text{undef}$~~
$A\to\cdot aCDE$	S16		~~$\gamma[2] = \text{undef}$~~
3. $S\to aA\cdot Ba$	G4	$\{a\}$	
$B\to\cdot c$	S9		
$B\to\cdot$	R5		
4. $S\to aAB\cdot a$	S10		
5. $S\to b\cdot ABb$	G6	$\{bc\}$	~~$\gamma[1] = \text{undef}$~~
$A\to\cdot aCDE$	**S16**		~~$\gamma[2][1] = \text{false}$~~
6. $S\to bA\cdot Bb$	G7	$\{b\}$	
$B\to\cdot c$	S9		
$B\to\cdot$	R5		
7. $S\to bAB\cdot b$	S11		
16. $A\to a\cdot CDE$	G17	$\{a\}$	$\gamma[1] = \text{undef}$
$C\to\cdot D$	G12	$\{a\}$	$\gamma[2][1] = \text{true}$
$D\to\cdot a$	S13		
17. $A\to aC\cdot DE$	G18	$\boldsymbol{\{a\underline{bc}\}}$ **p**	$\gamma[1] = \text{undef}$
$D\to\cdot a$	S13		$\gamma[2][1] = \text{true}$
*18. $A\to aCD\cdot E$	G14	$\boldsymbol{\{a\underline{bc}\}}$ **p**	
$E\to\cdot a$	~~S15~~		$\gamma[1] = \text{undef}$
$E\to\cdot$	R9		$\gamma[2][1] = \text{true}$

Table 5
IELR(1) parser tables with goto follows. This table shows the IELR(1) version of the parser tables shown in Table 4. Differences between the parser tables are shown in bold.



IELR(1)		
0. $S'\to\cdot S\#$	G1	$\{\#\}$
$S\to\cdot aABa$	S2	
$S\to\cdot bABb$	S5	
1. $S'\to S\cdot\#$	S8	
2. $S\to a\cdot ABa$	G3	$\{ac\}$
$A\to\cdot aCDE$	S16	
3. $S\to aA\cdot Ba$	G4	$\{a\}$
$B\to\cdot c$	S9	
$B\to\cdot$	R5	
4. $S\to aAB\cdot a$	S10	
5. $S\to b\cdot ABb$	G6	$\{bc\}$
$A\to\cdot aCDE$	**S19**	
6. $S\to bA\cdot Bb$	G7	$\{b\}$
$B\to\cdot c$	S9	
$B\to\cdot$	R5	
7. $S\to bAB\cdot b$	S11	
16. $A\to a\cdot CDE$	G17	$\{a\}$
$C\to\cdot D$	G12	$\{a\}$
$D\to\cdot a$	S13	
17. $A\to aC\cdot DE$	G18	$\boldsymbol{\{a\underline{c}\}}$ **p**
$D\to\cdot a$	S13	
*18. $A\to aCD\cdot E$	G14	$\boldsymbol{\{a\underline{c}\}}$ **p**
$E\to\cdot a$	~~S15~~	
$E\to\cdot$	R9	
19. $\boldsymbol{A\to a\cdot CDE}$	**G20**	$\boldsymbol{\{a\}}$
$\boldsymbol{C\to\cdot D}$	**G12**	$\boldsymbol{\{a\}}$
$\boldsymbol{D\to\cdot a}$	**S13**	
20. $\boldsymbol{A\to aC\cdot DE}$	**G21**	$\boldsymbol{\{a\underline{bc}\}}$ **p**
$\boldsymbol{D\to\cdot a}$	**S13**	
21. $\boldsymbol{A\to aCD\cdot E}$	**G14**	$\boldsymbol{\{\underline{bc}\}}$ **p**
$\boldsymbol{E\to\cdot a}$	**S15**	
$\boldsymbol{E\to\cdot}$	**R9**	

1. $S\to aABa$	4. $B\to c$	7. $D\to a$
2. $S\to bABb$	5. $B\to$	8. $E\to a$
3. $A\to aCDE$	6. $C\to D$	9. $E\to$

Fig. 5. Grammar demonstrating goto follows. We assume production 9 is declared with higher precedence than *a*.

Pennello's algorithm nor its existing Bison implementation is our own work, we analyze their algorithm in Section 3.2 in order to reach original conclusions that prove fundamental to the remaining phases of IELR(1).

- Phase 1: Compute Auxiliary Tables. From the LALR(1) parser tables, this phase computes a few additional tables that are employed by later phases to identify dependencies that remain stable during state splitting. In Section 3.3, we continue our analysis of DeRemer and Pennello's algorithm from phase 0 in order to define these tables.
- Phase 2: Compute Annotations. Using the information computed in phases 0 and 1, this phase identifies each 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 those states' contributions. In Table 4, a portion of these annotations is shown to the right of the parser tables. In the matrix γ, rows correspond to the possible contributions to the conflict, and columns correspond to the *kernel* items in the annotated state. The Boolean stored in a cell is true iff the

conflicted token is present in the corresponding kernel item's lookahead set and, as a result, the annotated state makes the corresponding contribution. Any isocore that might be split from the annotated state and retain the conflicted token in that lookahead set would make that contribution as well. An entire row is undefined only if every isocore that might be split from the annotated state would make that contribution regardless of kernel item lookahead sets. We explain the details of how these annotations are computed in Section 3.4.

- 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's LR(0) construction. The main difference is that, when considering whether to merge isocores, it employs a stricter state compatibility test based on the LALR(1) states' annotations from phase 2. For example, Table 5 shows the IELR(1) parser states constructed from the LALR(1) parser states of Table 4. We define the state compatibility test and the algorithm for recomputing the states in Section 3.5.
- 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 table conflicts.

3.2. Phase 0: LALR(1)

Phase 0 computes LALR(1) parser tables. It performs the computation in two steps:

1. Compute LR(0) parser tables.
2. Compute reduction lookahead sets using the technique described by DeRemer and Pennello [16].

In Section 3.2.1, we define a formal model for the efficient LR(1) state data structure that phase 0 employs. In Section 3.2.2, we illustrate briefly how DeRemer and Pennello's algorithm computes reduction lookahead sets from *goto follow sets* despite stateinformation missing from this data structure. In Section 3.2.3, we analyze several properties of goto follow sets that are useful in later phases of IELR(1). We conclude with a few notes on our Bison implementation of IELR(1) phase 0 in Section 3.2.4.

3.2.1. An efficient LR(1) state model

The LR(1) state model we defined in Section 2.4 has been convenient for our discussions so far as it reflects the way we illustrate LR(1) parser tables in, for example, Table 1. It was especially useful when analyzing Pager's and Menhir's compatibility tests in terms of item lookahead sets. However, we now define an alternative model that reflects the efficient data structures used by IELR(1) phase 0's LALR(1) algorithm. The more efficient model represents actions separately from items and without duplicate actions, it drops all *non-kernel items*, and it drops item lookahead sets.

Definition 3.1 (*Efficient LR(1) State*). Phase 0 initializes Σ as a set of efficient LR(1) states such that $|\Sigma| = |\mathscr{A}(G)|$ and, $\forall s : 1 \le s \le |\Sigma|, \Sigma[s] = (C, A_t, A_r)$ such that:

1. $C = \{((\ell \to \varrho), d) : \exists((\ell \to \varrho), d, K, a) \in \mathscr{A}(G)[s] : \ell = S' \vee d > 1\}$ is the set of *kernel item* cores in $\mathscr{A}(G)[s]$.
2. $A_t = \{(\varrho[d], s') : \exists((\ell \to \varrho), d, K, (a_t, a_p, a_s)) \in \mathscr{A}(G)[s] : d < |\varrho|+1 \wedge a_s = \mathscr{A}(G)[s']\}$ is a set describing the transition actions in $\mathscr{A}(G)[s]$.
3. $A_r = \{(K, a_p) : \exists((\ell \to \varrho), d, K, (a_t, a_p, a_s)) \in \mathscr{A}(G)[s] : d = |\varrho| + 1\}$ is a set describing the reduce actions in $\mathscr{A}(G)[s]$. $\square$

For example, in the LALR(1) parser tables of Table 1, state 4 has two items with an S9 action on token *a*. This is the same action and need be represented only once in the A_t from Definition 3.1. Both of these items are also non-kernel items because the LHS is not the start symbol of the augmented grammar and the dot is at the beginning of the RHS. Thus, only the state's first item has an entry in C. Even though item lookahead sets are helpful while analyzing parser tables, only those lookahead sets associated with reduce actions are useful to an LR(1) parser. Thus, C contains only cores, and the reduction lookahead sets are stored in A_r. However, because state 4 has no reduce actions, A_r is empty. Moreover, as we see in this section, the non-kernel items and the item lookahead sets need not be stored in order to compute LALR(1) parser tables.

Most of the relations we have already defined in terms of our original model have an equivalent and straightforward counterpart for our more efficient model because they do not utilize any state information that we have removed. Thus, we utilize such relations for our efficient model without providing formal extensions to their definitions.

3.2.2. Reduction lookaheads from goto follows

The result of phase 0 step 1 is exactly Σ except that, for every state s, given that $s = (C, A_t, A_r) \in \Sigma$, and, for every reduce action r, given that $r = (K, p) \in A_r$, the reduction lookahead set K is not yet computed. Step 2's job is to compute each such reduction lookahead set. However, each reduction lookahead set is a copy of the corresponding item's lookahead set, which neither step computes. Moreover, step 1 discards non-kernel items as it computes states even though some reductions come from non-kernel items.

Despite the information missing from the efficient LR(1) state data structure, step 2 manages to compute reduction lookahead sets. In order to understand how, it is useful to examine the detailed path of lookahead propagation.

Observation 3.2 (*Lookahead Propagation Path*). We use the LALR(1) parser tables of Table 1 as an example.

1. A reduction lookahead set is generated from its associated item's lookahead set. For example, the lookahead set for state 9's R3 is the lookahead set of state 9's first item, which happens to be a kernel item.
2. A kernel item lookahead set is generated from the lookahead sets of the same item in every predecessor except that the dot is one symbol to the left. For example, the lookahead set of state 9's first item is generated from the lookahead sets of state 3's and 4's second items, which are non-kernel items.
3. A non-kernel item lookahead set is generated from the *follow set* of the same state's goto on the non-kernel item's LHS. A goto follow set is the set of tokens that can appear next in a syntactically correct input sentence after the goto. For example, in each of states 3 and 4, the non-kernel item lookahead sets are generated from the follow set of the goto on A.
4. A goto follow set is generated from potentially two sources:
   (a) In each RHS in which the goto appears, the remainder of the RHS after the goto's nonterminal. For example, the follow set for the goto on A in state 3 contains a, and it contains b in state 4.
   (b) The lookahead set for each item in which the goto appears if the remainder of the RHS after the goto's nonterminal is nullable. That lookahead set can be traced recursively to the follow sets of the gotos that eventually generate the item's core. For example, in the first item of state 3, if all symbols following A were nullable nonterminals, the follow set for the goto on A would be generated from the first item's lookahead set, which is eventually generated from the follow set of the goto on S in state 0: $\{\#\}$. □

Observation 3.3 (*Lookback Dependencies*). We can summarize points 1, 2, and 3 from Observation 3.2 as follows:

1. The reduction lookahead set of the accept action is always empty because it is ultimately generated from the lookahead set of the kernel item in the start state, which has no predecessors. For example, the reduction lookahead set in state 2 of Table 1 is empty.
2. Before conflict resolution, the start state is the only state with no predecessors, so every reduction lookahead set except that of the accept action is generated from one or more goto follow sets. DeRemer and Pennello call this dependency a *lookback*. For example, in the LALR(1) parser tables of Table 1, the follow set for the goto on A in state 3 is $\{a\}$, and it is $\{b\}$ in state 4. Thus, the lookahead set of state 9's R3 is $\{ab\}$. □

Using Observation 3.3, goto follow sets are the key component of DeRemer and Pennello's algorithm. For this purpose, phase 0 step 2 computes the following three goto tables that the remaining IELR(1) phases also require: *from_state*, *to_state*, and *goto_follows*.

Definition 3.4 (*Goto Tables*). *ngotos* is the number of distinct gotos that appear in Σ. That is, $ngotos = |\{(s, n) : s \in \Sigma \wedge n \in V' \wedge \exists\delta(s, n)\}|$. $\forall g : 1 \le g \le ngotos$, all of the following are true:

1. $1 \le from\_state[g] \le |\Sigma|$.
2. $1 \le to\_state[g] \le |\Sigma|$.
3. Goto g is the transition $\delta(\Sigma[from\_state[g]], \dagger(\Sigma[to\_state[g]])) = \Sigma[to\_state[g]]$.
4. $goto\_follows[g]$ is the follow set for goto g. □

3.2.3. Computing goto follows

In order to understand how DeRemer and Pennello's algorithm computes *goto_follows*, we must identify the dependencies of goto follow sets by examining point 4 in Observation 3.2. Consider the grammar in Fig. 5. Table 4 shows the cores of several interesting states from that grammar's LALR(1) parser tables. It also shows the goto follow sets and their dependencies. In the next several paragraphs, we use this example to demonstrate the various types of goto follow set dependencies.

Point 4a in Observation 3.2 can be reworded as: a goto is followed by the transitions in the successor state, so its follow set depends on them. DeRemer and Pennello call these dependencies *reads*. For clarity, we call them ***successor dependencies*** and label their edges in Table 4 with an *s*. For example, in Table 4:

1. State 3 contains S9 on c. Thus, state 2's G3 can be followed by c. DeRemer and Pennello call this a *direct read*.
2. State 3 also contains G4 on B, and $B \Rightarrow^* \epsilon$. Any of the tokens that can follow state 3's G4 can thus also follow state 2's G3. DeRemer and Pennello call this an *indirect read*. As we discuss later in this section, there are some cases where a successor goto follow set is only partially inherited in this way.
3. State 17 contains G18 on D, and $D \not\Rightarrow^* \epsilon$. Thus, the follow set of state 16's G17 does not depend on the follow set of state 17's G18.

We now state DeRemer and Pennello's definition of read dependencies in terms of our model, and we define the follow set generated by the transitive closure. For any relation F, by $F^*(g, g')$, we mean $g = g' \vee F^+(g, g')$.

Definition 3.5 (*Goto Follows Successor Relation*). $\forall g \forall g' : 1 \le g \le ngotos \wedge 1 \le g' \le ngotos$, the relation $GF_s(g, g')$ holds iff $to\_state[g] = from\_state[g'] \wedge \dagger(\Sigma[to\_state[g']]) \Rightarrow^* \epsilon$. □

Definition 3.6 (*successor_follows*). $\forall g : 1 \le g \le ngotos$, the ***successor-generated*** follow set for goto g is $successor\_follows[g] = \{t \in T' : \exists g' : GF_s{}^*(g, g') \wedge \exists\delta(\Sigma[to\_state[g']], t)\}$. □

Point 4b in Observation 3.2 can be summarized as: a goto's follow set may also depend upon the follow sets of the gotos that generate its items' cores. DeRemer and Pennello call these dependencies *includes*. We divide these dependencies into two categories. If an includes dependency can be reached from a goto's follow set via a path involving only points 4b and 3 in Observation 3.2, the dependency must appear in the same state as the goto, so we call it an ***internal dependency***,[4] and we label its edge in Table 4 with an *i*. Otherwise, the includes dependency must involve point 2 as well, so we call it a ***predecessor dependency***, and we label its edge in Table 4 with a *p*. A predecessor dependency always appears in an eventual predecessor of the state containing the goto, but this predecessor might actually be the state containing the goto if the parser tables contain a transition loop. For example, in Table 4:

1. In state 16, G12's only item core is generated from state 16's G17. In G12's item core, the remainder of the RHS after G12's nonterminal is ϵ. Thus, any token that can follow G17 can also follow G12. Because the path from G12 to G17 involves only points 4b and 3 in Observation 3.2, we call this an internal dependency, which is possible only because G12's nonterminal is at the beginning of the RHS.
2. In state 16, G17's only item core is in turn generated from state 2's G3 and state 5's G6. In G17's item core, the remainder of the RHS after G17's nonterminal is *DE*, and $DE \not\Rightarrow^* \epsilon$. Thus, the follow set of state 16's G17 does not depend on the follow set of state 2's G3 or of state 5's G6.
3. In state 17, G18's only item core is also generated from state 2's G3 and state 5's G6. In G18's item core, the remainder of the RHS after G18's nonterminal is *E*, and $E \Rightarrow^* \epsilon$. Thus, any token that can follow state 2's G3 or state 5's G6 can also follow state 17's G18. Because the paths from state 17's G18 to state 2's G3 and to state 5's G6 involve points 4b, 2, and 3 in Observation 3.2, we call each of these a predecessor dependency, which is possible only because the nonterminal of state 17's G18 is not at the beginning of the RHS.

We now state DeRemer and Pennello's definition of includes dependencies in terms of our model. We then modify it to formally define internal dependencies and predecessor dependencies.

Definition 3.7 (*Goto Follows Includes Relation*). $\forall g \forall g' : 1 \le g \le ngotos \wedge 1 \le g' \le ngotos$, the relation $GF_{ip}(g, g')$ holds iff $\exists \alpha \in \{V' \cup T'\}^* : \exists \beta \in V'^*$ such that all of the following are true:

1. $\exists(\ell \to \varrho) \in P' : \ell = \dagger(\Sigma[to\_state[g']]) \wedge \varrho = (\alpha, \dagger(\Sigma[to\_state[g]]), \beta)$.
2. $\delta^*(\Sigma[from\_state[g']], \alpha) = \Sigma[from\_state[g]]$.
3. $\beta \Rightarrow^* \epsilon$. □

Definition 3.8 (*Goto Follows Internal Relation*). In Definition 3.7, iff $GF_{ip}(g, g') \wedge \alpha = \epsilon$, then the relation $GF_i(g, g')$ holds. In this case, $from\_state[g'] = from\_state[g]$. □

Definition 3.9 (*Goto Follows Predecessor Relation*). In Definition 3.7, iff $GF_{ip}(g, g') \wedge \alpha \neq \epsilon$, then the relation $GF_p(g, g')$ holds. In this case, $\Sigma[from\_state[g']]$ is an eventual predecessor of $\Sigma[from\_state[g]]$. □

Observation 3.10 ($GF_{ip} \Leftrightarrow GF_i \vee GF_p$). $\forall g \forall g' : 1 \le g \le ngotos \wedge 1 \le g' \le ngotos$, $GF_{ip}(g, g') \Leftrightarrow GF_i(g, g') \vee GF_p(g, g')$. □

Successor, internal, and predecessor dependencies all must be considered in order to compute complete goto follow sets. As DeRemer and Pennello point out, it is thus tempting to compute *goto_follows* using a definition such as Definition 3.12, given below.

Definition 3.11 (*Goto Follows Dependency Relation*). $\forall g \forall g' : 1 \le g \le ngotos \wedge 1 \le g' \le ngotos$, $GF_{sip}(g, g') \Leftrightarrow GF_s(g, g') \vee GF_i(g, g') \vee GF_p(g, g')$. □

Definition 3.12 (*goto_follows, Oversimplified*). $\forall g : 1 \le g \le ngotos$, the complete follow set for goto g is $goto\_follows[g] = \{t \in T' : \exists g' : GF_{sip}{}^*(g, g') \wedge \exists \delta(\Sigma[to\_state[g']], t)\}$. □

Definition 3.12 has the nice property of requiring only one closure computation. However, as we mentioned in our successor dependency examples at the beginning of this section, there are some goto follow sets that inherit some successor goto follow sets only partially. Definition 3.12 does not take this into account.

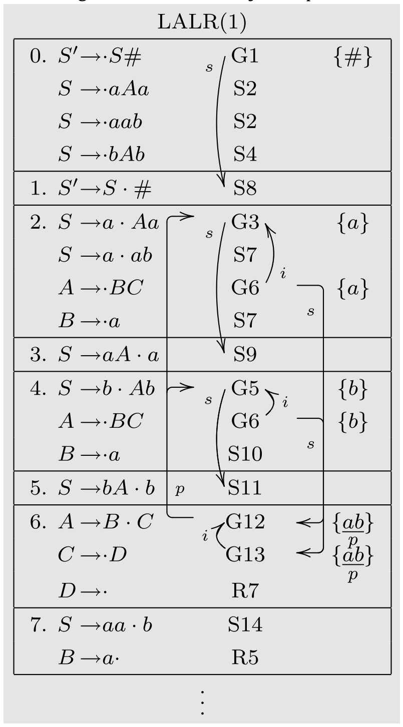
For example, consider the grammar in Fig. 6. Table 6 shows the goto follow sets and their dependencies in that grammar's LALR(1) parser tables. State 6 is the point of convergence of two lanes in the parser tables. That is, state 2 and state 4 both have transitions to state 6. State 6 has a predecessor dependency on each of these two lanes and combines follow sets that are distinct in each. That is, state 2's G3 has $\{a\}$, state 4's G5 has $\{b\}$, and state 6's G12 and G13 then have $\{ab\}$. In turn, both of these lanes have successor dependencies on the follow sets in state 6. However, if the lanes were to inherit all tokens from state 6's follow sets, the lanes would be indirectly inheriting tokens from each other, and thus the lanes would effectively be partially merged earlier than state 6. LALR(1) demands that only isocores are merged. That is, *b* cannot follow G6 in state 2 and *a* cannot follow G6 in state 4 despite their dependencies on state 6's G12 and G13 follow sets.

Observation 3.13 (*Successor Not Before Predecessor*). For each token in a goto follow set, there must be a path along goto follow set dependencies to a shift action on that token such that, after the path traverses any successor dependency, it never traverses another predecessor dependency. □

1. $S \rightarrow aAa$
2. $S \rightarrow aab$
3. $S \rightarrow bAb$
4. $A \rightarrow BC$
5. $B \rightarrow a$
6. $C \rightarrow D$
7. $D \rightarrow$

Fig. 6. Grammar for goto follows caveats.

Table 6
Parser tables with goto follows caveats. This table shows Fig. 6's LALR(1) goto follow edges and sets, underlining follows that are only from predecessors.



LALR(1)		
0. $S' \rightarrow \cdot S\#$	G1	$\{\#\}$
$S \rightarrow \cdot aAa$	S2	
$S \rightarrow \cdot aab$	S2	
$S \rightarrow \cdot bAb$	S4	
1. $S' \rightarrow S \cdot \#$	S8	
2. $S \rightarrow a \cdot Aa$	G3	$\{a\}$
$S \rightarrow a \cdot ab$	S7	
$A \rightarrow \cdot BC$	G6	$\{a\}$
$B \rightarrow \cdot a$	S7	
3. $S \rightarrow aA \cdot a$	S9	
4. $S \rightarrow b \cdot Ab$	G5	$\{b\}$
$A \rightarrow \cdot BC$	G6	$\{b\}$
$B \rightarrow \cdot a$	S10	
5. $S \rightarrow bA \cdot b$	S11	
6. $A \rightarrow B \cdot C$	G12	$\{\underset{p}{\underline{ab}}\}$
$C \rightarrow \cdot D$	G13	$\{\underset{p}{\underline{ab}}\}$
$D \rightarrow \cdot$	R7	
7. $S \rightarrow aa \cdot b$	S14	
$B \rightarrow a \cdot$	R5	
⋮		

We now state DeRemer and Pennello's definition for goto follow sets in terms of our model. Unlike Definition 3.12, which requires only one closure computation, DeRemer and Pennello's definition obeys the restriction of Observation 3.13. As a result, it requires two closure computations: $GF_{ip}{}^*$ and the $GF_s{}^*$ for *successor_follows*.

Definition 3.14 (*goto_follows, via successor_follows*)**.** $\forall g : 1 \le g \le ngotos$, the complete follow set for goto g is $goto\_follows[g] = \{t \in T' : \exists g' : GF_{ip}{}^*(g, g') \wedge t \in successor\_follows[g']\}$. □

Definition 3.14 raises additional questions about correct paths along goto follow set dependencies:

1. Because the computation of *successor_follows* does not include GF_i, it does not allow a successor dependency to be traversed before any internal dependencies. Do we miss some goto follows as a result?
2. This definition allows internal and predecessor dependencies to be traversed in any order. Do we gain invalid goto follows as a result?

While it is not our objective to prove the correctness of DeRemer and Pennello's algorithm, the above two questions prove relevant to IELR(1) phase 1, so we address them in Section 3.3.4.

3.2.4. Bison implementation

In our implementation of IELR(1) phase 0, we use Bison's existing LALR(1) implementation, which already computes all the goto tables mentioned in Definition 3.4. Our only change to that implementation is an interface change: we expose the goto tables to the remaining phases of IELR(1).

4 Internal dependencies are closely related to what Pager calls an *internal connecting lane* [26,27].

3.3. Phase 1: Compute auxiliary tables

Because IELR(1) splits LALR(1) states in order to eliminate LR(1)-relative inadequacies, it must have some means to analyze how state splitting affects those inadequacies. Before conflicts are resolved, conflicts are the most obvious manifestation of inadequacies, so IELR(1) starts by analyzing how state splitting affects conflicts. By Definitions 2.17 and 2.10, conflicts have contributions both from shift actions, as computed entirely from the core of a state, and from reduce actions, as computed from the core and the lookahead sets. Because states are split into isocores, state splitting does not affect cores and thus does not affect shift actions. However, lookahead sets and thus reduce actions are affected by changes in the paths of lookahead propagation described in Observation 3.2. Therefore, in order to analyze how state splitting affects inadequacies, IELR(1) must analyze how state splitting affects lookahead propagation paths.

Phase 0's dependency relations are one way to simplify the details of lookahead propagation paths. However, some of these dependency relations are not stable during state splitting. Instead of attempting to reuse the unstable dependency relations while analyzing state splitting, IELR(1) traces in greater detail the portions of lookahead propagation paths that the unstable dependency relations simplified for phase 0. Nevertheless, phase 1 is able to offer some stable simplifications for this trace. To do so, phase 1 computes three tables: *predecessors*, *follow_kernel_items*, and *always_follows*. We define these tables in Sections 3.3.1–3.3.3, respectively. In Section 3.3.4, we reconsider phase 0's computation of *goto_follows* in terms of phase 1's *always_follows*. In Section 3.3.5, we discuss our Bison implementation of IELR(1) phase 1 and some optimizations that might be made in conjunction with phase 0.

3.3.1. Predecessors

The key to analyzing how state splitting affects lookahead propagation paths is recognizing that each isocore split from a state has a different set of predecessors than the original state. State splitting thus alters the phase 0 dependency relations that involve the portion of lookahead propagation paths described by point 2 in Observation 3.2.

As explained in Observation 3.3, lookback dependencies involve the portion of lookahead propagation paths described by point 2 in Observation 3.2. Thus, lookback dependencies are not stable during state splitting. For example, in the LALR(1) parser tables of Table 1, the lookahead set of the first item in state 9 is generated from the lookahead set of the second item in each of states 3 and 4, which are the predecessors of state 9. As a result, the lookahead set for state 9's R3 has a lookback dependency on the follow set of the goto on *A* in each of states 3 and 4. However, in the canonical LR(1) tables, LALR(1) state 9 is split into states 9 and 11. State 9 retains only state 3 as a predecessor, and state 11 retains only state 4 as a predecessor. Thus, state 9's R3 retains a lookback dependency only in state 3, and state 11's R3 retains a lookback dependency only in state 4. Thus, the viable prefix of state 11 no longer encounters *a* in the R3 lookahead set, and so it no longer encounters the conflict on *a*.

Goto follow set predecessor dependencies, as the name implies, also involve the portion of lookahead propagation paths described by point 2 in Observation 3.2 via the recursion mentioned in point 4b. Thus, goto follow set predecessor dependencies are not stable during state splitting. For example, Table 5 shows the IELR(1) version of the LALR(1) parser tables from Table 4. In LALR(1) state 18, the lookahead set for R9 is the follow set of G14, so there's a conflict on *a*. However, LALR(1) state 18 is split into IELR(1) states 18 and 21. As a result, the viable prefix of state 21 no longer encounters *a* in the G14 follow set, so it no longer encounters the conflict on *a*. The reason it does not encounter *a* in the G14 follow set is that it no longer encounters the G14 follow set's predecessor dependency on state 2's G3 follow set because it no longer sees state 2 as an eventual predecessor.

In the LALR(1) tables in the above examples, notice that we can trace lookahead propagation paths along multiple lanes by starting at each conflicted state and iterating in reverse towards the start state. In other words, in order to determine how state splitting can affect the unstable lookback dependencies and goto follow set predecessor dependencies from phase 0, we can start by examining states' existing predecessors. For example, in Table 4, state 18 is the conflicted state. By examining predecessor states, we can iterate what appears to be a single lane back through states 17 and 16. However, at this point we see two lanes diverging into states 2 and 5. Then they converge again into state 0, the start state, where all lanes must ultimately converge according to Definition 2.15 and Observation 2.14. Thus, the two lanes are $\lambda_1 = (0, 2, 16, 17, 18)$ and $\lambda_2 = (0, 5, 16, 17, 18)$. In λ_1, $\{ac\}$ propagates from state 2's G3 follow set to state 18's R9 lookahead set. In λ_2, $\{bc\}$ propagates from state 5's G6 follow set also to state 18's R9 lookahead set. IELR(1) has to split LALR(1) states 16, 17, and 18 in order to split this lookahead set and thus eliminate the LR(1)-relative inadequacy that manifests as state 18's conflict on *a*.

Phase 2 iterates in reverse the lanes for conflicted LALR(1) states and, along the way, annotates states that phase 3 may need to split in order to eliminate LR(1)-relative inadequacies. To facilitate phase 2's iteration of lanes, phase 1 computes a *predecessors* table for the LALR(1) parser tables.

Definition 3.15 (*Predecessors*). $\forall s : 1 \leq s \leq |\Sigma|, predecessors[s] = \{s' : \exists y : \delta(\Sigma[s'], y) = \Sigma[s]\}$. □

While phase 3 employs phase 2's annotations during state splitting, *predecessors* itself is discarded at the end of phase 2.

3.3.2. Goto follows from kernel items

As phase 2 iterates lanes for conflicted states in the LALR(1) tables, it must also trace lookahead propagation paths in greater detail in order to leave useful annotations on the states it visits. Consider portions of that trace that start with a goto follow set, recurse on points 4b and 3 in Observation 3.2, and end on kernel item lookahead sets, which are necessarily of the same state as the goto follow set because points 2 and 4a are never employed. Phase 1 computes a *follow_kernel_items* table to simplify these portions. That is, *follow_kernel_items* stores goto follow sets' dependencies on kernel item lookahead sets of the same state. Because this involves points 4b and 3 without point 2's predecessor relation, the goto follow set internal dependency relation from phase 0 can be employed.

Definition 3.16 (*follow_kernel_items*). $\forall g : 1 \le g \le ngotos$ given that $\Sigma[from\_state[g]] = (C, A_t, A_r)$, $\forall k : 1 \le k \le |C|$ given that $C[k] = ((\ell \to \varrho), d)$, $follow\_kernel\_items[g][k]$ is true iff all of the following are true:

1. $\exists g' : GF_i{}^*(g, g') \wedge \varrho[d] = \dagger(\Sigma[to\_state[g']])$.
2. $\varrho[d+1..|\varrho|] \Rightarrow^* \epsilon$. $\square$

For example, consider state 6 in Table 6. Let g be G13, a goto on D. Let g' be G12, a goto on C. Let $k = 1$ for the first and only kernel item core, $A \to B \cdot C$. It is true that $GF_i{}^*(g, g')$ and that the dot in the first kernel item is in front of C, so condition 1 in Definition 3.16 holds. The remainder of the RHS after C in that kernel item is ϵ, and it is certainly true that $\epsilon \Rightarrow^* \epsilon$, so condition 2 in Definition 3.16 holds. That is, $follow\_kernel\_items[g][k]$ is true because G13 does depend on the first kernel item's lookahead set. Moreover, it is true that $GF_i{}^*(g', g')$. That is, $follow\_kernel\_items[g'][k]$ is true because G12 also depends on the first kernel item's lookahead set.

3.3.3. Always-generated follows

In Section 3.3.1, we showed that lookback dependencies and goto follow set predecessor dependencies are unstable as state splitting changes state predecessors. The only remaining mutually exclusive dependencies from phase 0 are goto follow set successor and internal dependencies. In this section, we explore the effect of state splitting on these remaining dependencies.

For any state s, the core of s generates the cores of the successors of s, thus the transition actions of the successors of s, and thus the successor dependencies of the goto follow sets of s. The internal dependencies of goto follow sets of s are, as their name implies, generated from the core of s as well. When s is split into isocores s' and s'', each of s' and s'' must then retain every successor and internal dependency from s except that each such dependency may lie in an isocore of the state in which it previously lay. For this reason, we call successor and internal dependencies ***always dependencies***.

Definition 3.17 (*Goto Follows Always Relation*). $\forall g \forall g' : 1 \le g \le ngotos \wedge 1 \le g' \le ngotos, GF_{si}(g, g') \Leftrightarrow GF_s(g, g') \vee GF_i(g, g')$. $\square$

For example, consider the LALR(1) parser tables of Table 4 and the corresponding IELR(1) parser tables of Table 5. Let s be LALR(1) state 16, and let s' and s'' be its isocores, IELR(1) states 16 and 19. From s, each of s' and s'' retains the successor dependency of the follow set of the goto on C. The states in which that successor dependency lies for s, s', and s'' are LALR(1) state 17, IELR(1) state 17, and IELR(1) state 20, which are isocores as expected. From s, each of s' and s'' also retains the internal dependency of the follow set of the goto on D. The states in which that internal dependency lies are just s, s', and s'', which are of course isocores.

Let p be any goto follow set dependency path starting in s, traversing only always dependencies, and terminating at a shift action, where all goto follow set dependency paths terminate according to Definitions 3.14 and 3.6. Applying the always dependency concept recursively, there must be paths p' for s' and p'' for s'' that are isocoric with p. Any follows inherited via these paths must then be the same for s, s', and s''. That is, follows that are inherited only via always dependencies form an ***always-generated*** follow set, which phase 1 computes from the LALR(1) parser tables and which later phases need not recompute for isocores produced by state splitting.

Definition 3.18 (*always_follows, two closure*). $\forall g : 1 \le g \le ngotos$, the ***always-generated*** follow set for goto g is $always\_follows[g] = \{t \in T' : \exists g' : GF_i{}^*(g, g') \wedge t \in successor\_follows[g']\}$. $\square$

The only difference between Definition 3.14 for *goto_follows* and Definition 3.18 for *always_follows* is the use of $GF_{ip}{}^*$ versus $GF_i{}^*$. For any goto g, it is clear that $\{g' : GF_i{}^*(g, g')\} \subseteq \{g' : GF_{ip}{}^*(g, g')\}$, so Theorem 3.19 follows.

Theorem 3.19 (*Always Follows are Goto Follows*). $\forall g : 1 \le g \le ngotos, always\_follows[g] \subseteq goto\_follows[g]$. $\square$

One similarity between Definitions 3.14 and 3.18 is the number of closure computations required. Each definition requires one closure computation for $GF_{ip}{}^*$ or $GF_i{}^*$. Each definition depends on *successor_follows*, which requires a second closure computation for $GF_s{}^*$. In Definition 3.12, we tried to define *goto_follows* to require only one closure computation, but we found that we gained invalid follows as a result. However, we now provide a definition for *always_follows* that is correct even though it requires only one closure computation.

Definition 3.20 (*always_follows, one closure*). $\forall g : 1 \le g \le ngotos$, the ***always-generated*** follow set for goto g is $always\_follows[g] = \{t \in T' : \exists g' : GF_{si}{}^*(g, g') \wedge \exists \delta(\Sigma[to\_state[g']], t)\}$. $\square$

In the remainder of this section, we prove that our two definitions for *always_follows* are equivalent. In order to do so, we first need to prove a few theorems about our goto follows relations.

Theorem 3.21 (*Equivalent always_follows Definitions*). $\forall g : 1 \le g \le ngotos$, *Definitions 3.18 and 3.20 define always_follows$[g]$ as the same set.* □

Consider Table 6. In states 2 and 4, G6's follow set has a successor dependency on state 6's G13 follow set, which has an internal dependency on state 6's G12 follow set. As a result, state 2's and state 4's G6 follow sets indirectly depend on state 6's G12 follow set. However, they also have a direct successor dependency on it. According to Theorem 3.22, this is no coincidence.

Theorem 3.22 ($GF_sGF_i \Rightarrow GF_s$). $\forall g \forall g' \forall g'' : 1 \le g \le ngotos \wedge 1 \le g' \le ngotos \wedge 1 \le g'' \le ngotos, GF_s(g, g') \wedge GF_i(g', g'') \Rightarrow GF_s(g, g'')$. □

Proof (*Theorem* 3.22). By Definitions 3.5 and 3.8, because $GF_s(g, g')$ and $GF_i(g', g'')$, then:

1. $to\_state[g] = from\_state[g']$.
2. $from\_state[g'] = from\_state[g'']$.
3. $\exists(\ell \to \varrho) \in P' : \ell = \dagger(\Sigma[to\_state[g'']]) \wedge \exists \beta \in V'^* : \varrho = (\dagger(\Sigma[to\_state[g']]), \beta)$.
4. $\dagger(\Sigma[to\_state[g']]) \Rightarrow^* \epsilon$.
5. $\beta \Rightarrow^* \epsilon$.

Thus:

1. $to\_state[g] = from\_state[g'']$ by points 1 and 2.
2. $\dagger(\Sigma[to\_state[g'']]) \Rightarrow^* \epsilon$ by points 3, 4, and 5.

Thus, $GF_s(g, g'')$. □

Corollary 3.23 ($GF_{si}{}^* \Leftrightarrow GF_i{}^*GF_s{}^*$). $\forall g \forall g' : 1 \le g \le ngotos \wedge 1 \le g' \le ngotos, GF_{si}{}^*(g, g') \Leftrightarrow \exists g'' : GF_i{}^*(g, g'') \wedge GF_s{}^*(g,'' g')$. □

Proof (*Corollary* 3.23). $GF_{si}{}^*(g, g') \Leftarrow \exists g'' : GF_i{}^*(g, g'') \wedge GF_s{}^*(g,'' g')$ because the RHS is merely a special case of the LHS. We prove the reverse implication by construction:

1. $GF_{si}{}^*(g, g')$ is shorthand for $\exists \gamma \in \{1..ngotos\}^* : g = \gamma[1] \wedge g' = \gamma[|\gamma|] \wedge \forall i : 1 \le i < |\gamma|, GF_{si}(\gamma[i], \gamma[i+1])$.
2. $\forall i : 1 < i < |\gamma| \wedge GF_s(\gamma[i-1], \gamma[i]) \wedge GF_i(\gamma[i], \gamma[i+1])$, we remove $\gamma[i]$ from γ recognizing that, by Theorem 3.22, $GF_s(\gamma[i-1], \gamma[i+1])$.
3. In other words, without altering the first or last goto, we have transformed γ into another valid dependency chain, γ', in which a successor dependency never appears before an internal dependency. Thus, when iterating γ' from left to right, at some goto g'' the internal dependencies, if any, must end and the successor dependencies, if any, must begin.

We have identified a $g'' : GF_i{}^*(g, g'') \wedge GF_s{}^*(g,'' g')$. □

Proof (*Theorem* 3.21). Substitute Definition 3.6 into Definition 3.18 to yield this equivalent definition:

$$\forall g : 1 \le g \le ngotos, always\_follows[g] = \{t \in T' : \exists g' : GF_i{}^*(g, g') \wedge t \in \{t' \in T' : \exists g'' : GF_s{}^*(g', g'') \wedge \exists \delta(\Sigma[to\_state[g'']], t')\}\}$$

which can be rewritten as:

$$\forall g : 1 \le g \le ngotos, always\_follows[g] = \{t \in T' : \exists g' : \exists g'' : GF_i{}^*(g, g') \wedge GF_s{}^*(g', g'') \wedge \exists \delta(\Sigma[to\_state[g'']], t)\}.$$

By Corollary 3.23, this is equivalent to Definition 3.20. □

3.3.4. *Goto follows reconsidered*

We now provide an alternative definition for *goto_follows*. The only difference from Definition 3.14 is the substitution of *always_follows* for *successor_follows*.

Definition 3.24 (*goto_follows, via always_follows*). $\forall g : 1 \le g \le ngotos$, the complete follow set for goto g is $goto\_follows[g] = \{t \in T' : \exists g' : GF_{ip}{}^*(g, g') \wedge t \in always\_follows[g']\}$. □

Theorem 3.25 (*Equivalent goto_follows Definitions*). $\forall g : 1 \le g \le ngotos$, *Definitions 3.14 and 3.24 define goto_follows$[g]$ as the same set.*

Proof (*Theorem* 3.25). Substitute Definition 3.18 into Definition 3.24 to yield this equivalent definition:

$$\forall g : 1 \le g \le ngotos, goto\_follows[g] = \{t \in T' : \exists g' : GF_{ip}{}^*(g, g') \wedge t \in \{t' \in T' : \exists g'' : GF_i{}^*(g', g'') \wedge t' \in successor\_follows[g'']\}\}$$

which can be rewritten as:

$$\forall g : 1 \le g \le ngotos, goto\_follows[g] = \{t \in T' : \exists g' : GF_{ip}{}^*(g, g') \wedge \exists g'' : GF_i{}^*(g', g'') \wedge t \in successor\_follows[g'']\}.$$

By Observation 3.10, this is equivalent to Definition 3.14. □

We can now answer one of the questions we asked at the end of Section 3.2.3: disallowing a successor dependency to be traversed before an internal dependency does not affect the computation of *goto_follows*. That is, it is correct to use either *always_follows* or *successor_follows* in the definition of *goto_follows*.

We do not fully answer the second question at the end of Section 3.2.3, but we now address one aspect of it that is interesting for our purposes. In Definition 3.24, internal dependencies are required both by $GF_{ip}{}^*$ and by $GF_{si}{}^*$ from Definition 3.20 for *always_follows*. It is tempting to eliminate this sharing and thus simplify the algorithm by substituting $GF_p{}^*$ for $GF_{ip}{}^*$ in Definition 3.24. That is, do we really need to consider internal dependencies twice? However, this change would disallow traversing an internal dependency before a predecessor dependency.

For example, consider state 6 in Table 6. G13's follow set has an internal dependency on G12's follow set, which has a predecessor dependency on the follow sets of state 2's G3 and state 4's G5. If we were to disallow the follow set of state 6's G13 to inherit from those predecessors through G12's follow set, then G13 would have an empty follow set. As a result, R7 would have an empty lookahead set and thus never be performed by the parser. Thus, there exist cases where disallowing the traversal of an internal dependency before a predecessor dependency would lose correct follows, so $GF_p{}^*$ cannot be substituted for $GF_{ip}{}^*$ in Definition 3.24.

3.3.5. Bison implementation

In order to compute *goto_follows*, Bison's existing LALR(1) implementation starts by computing successor dependencies followed by *successor_follows*, which requires one closure computation. Next, it computes includes dependencies, and then it overwrites *successor_follows* as it computes *goto_follows* using Definition 3.14. This requires another closure computation. Based on this observation and on the analysis in Sections 3.3.3 and 3.3.4, we discuss two possible implementations for the computation of *follow_kernel_items* and *always_follows* in Bison:

1. Minimize modifications to the existing Bison LALR(1) implementation. In phase 1, compute internal dependencies followed by *follow_kernel_items*, which requires one closure computation. Next, extend the internal dependency table with successor dependencies, and then compute *always_follows* using the one-closure form from Definition 3.20. The total number of closure computations for phases 0 and 1 is then 4. Successor and internal dependencies are computed twice.
2. Maximize code reuse and performance. That is, move phase 1's computation of internal dependencies, *follow_kernel_items*, successor dependencies, and *always_follows* between the two steps of phase 0. These phase 1 computations still require two closure computations. However, while computing internal dependencies, also compute a separate table of all includes dependencies.That is, because the algorithm for includes dependencies is a more general version of the algorithm for internal dependencies, they can be merged. In phase 0 step 2, compute *goto_follows* using the includes dependencies and using *always_follows* as in Definition 3.24. This requires only one additional closure computation. The total number of closure computations for phases 0 and 1 is then 3. Successor and internal dependencies are computed only once, and *successor_follows* is never computed.

Due to its improved code reuse, implementation 2 is the more obvious choice. We recommend it for any new parser generator without an existing LALR(1) implementation. As IELR(1) becomes accepted as a permanent feature in Bison, we plan to adopt implementation 2. However, for our initial introduction of IELR(1) as an experimental feature in Bison, we chose implementation 1 in order to maximize the modularity of the IELR(1) code with respect to the existing LALR(1) code. For our case studies, phase 1 usually took less than 1% of Bison's total run time, so the theoretical performance gains from implementation 2 might not actually be significant in practice.

3.4. Phase 2: Compute annotations

As we mentioned in the previous section, phase 2's job is to iterate in reverse the lanes[5] for conflicted LALR(1) states and, along the way, to annotate states that phase 3 might need to split in order to eliminate LR(1)-relative inadequacies. So that phase 3 can decide whether it is actually useful to split each state, a state's annotations must describe whether and how any isocores that might be split from it can contribute to LR(1)-relative inadequacies. Phase 2 computes this contribution information by tracing the conflicted tokens' propagation paths through kernel item lookahead sets and through goto follow sets along the conflicted states' lanes. Phase 2 caches the lookahead sets in an *item_lookahead_sets* table, which we describe in Section 3.4.1. We explain in detail how phase 2 computes inadequacy annotations using that table in Section 3.4.2. In Section 3.4.3, we explain some important phase 2 optimizations.

3.4.1. Item lookahead sets

The efficient LR(1) state model of Definition 3.1 does not store item lookahead sets. However, phase 2 must trace conflicted tokens within the LALR(1) tables along the detailed lookahead propagation paths described in Observation 3.2. Fortunately, doing so does not require computing all item lookahead sets for all states. Instead, phase 2 computes lookahead

5 This iteration is similar to Pager's lane-tracing algorithm [25,27].

sets only for kernel items that fall in the propagation paths of conflicted tokens. Moreover, it computes them only when needed and then caches them in an *item_lookahead_sets* table.

Definition 3.26 provides a recursive *item_lookahead_sets* definition that describes phase 2's computation of this table. Because the recursion can terminate at goto follow sets computed in phase 0, phase 2 does not require the full time that it would require if it were to compute the desired lookahead sets solely from LR(0) states.

Definition 3.26 (*item_lookahead_sets*)**.** Given $s : 1 \le s \le |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$, and given $k : 1 \le k \le |C| \wedge C[k] = ((\ell \to \varrho), d)$, then the lookahead set for kernel item $C[k]$ in state $\Sigma[s]$ is *item_lookahead_sets*$[s][k]$ such that:

1. If $d > 2$, then $\mathit{item\_lookahead\_sets}[s][k] = \{t \in T' : \exists s' \in \mathit{predecessors}[s] : \exists (C', A'_t, A'_r) = \Sigma[s'] : \exists k' : C'[k'] = ((\ell \to \varrho), d-1) \wedge t \in \mathit{item\_lookahead\_sets}[s'][k']\}$. This set is derived from point 2 of the lookahead propagation path of Observation 3.2.
2. If $d = 2$, then $\mathit{item\_lookahead\_sets}[s][k] = \{t \in T' : \exists g : \mathit{from\_state}[g] \in \mathit{predecessors}[s] \wedge \dagger(\Sigma[\mathit{to\_state}[g]]) = \ell \wedge t \in \mathit{goto\_follows}[g]\}$. This set is derived from points 2 and 3 of the lookahead propagation path of Observation 3.2.
3. If $d = 1$, then $\mathit{item\_lookahead\_sets}[s][k] = \emptyset$. That is, by Definition 3.1, $d = 1 \Rightarrow \ell = S'$. By Definition 2.7, $\ell = S' \Rightarrow \varrho = S\#$, and # marks the end of the input, so there are no lookaheads. □

In Definition 3.26, we provide the $d = 1$ case only for conceptual completeness. Phase 2 never actually encounters this case because no conflicted token's propagation path is ever traced to the empty lookahead set that appears after the token marking the end of the input. This assertion can be proven using Observation 3.2. However, this assertion's validity is not vital to IELR(1), so we omit the proof.

3.4.2. *Annotation lists*

In phase 2, IELR(1) is not yet always able to compute whether any given grammar-relative inadequacy in the LALR(1) tables is an LR(1)-relative inadequacy. Thus, in order to compute how isocores that phase 3 might split from LALR(1) states might contribute to LR(1)-relative inadequacies, phase 2 must first compute a list of all grammar-relative inadequacies in the LALR(1) parser tables. Phase 2 identifies each inadequacy uniquely by the conflict by which the inadequacy manifests. For each inadequacy, phase 2 records the conflicted state, the conflicted token, and all of the conflict's contributions within the conflicted state.

Definition 3.27 (*inadequacy_lists*)**.** Given $s : 1 \le s \le |\Sigma|$, then the list of ***inadequacy manifestation descriptions*** for state $\Sigma[s]$ is $\mathit{inadequacy\_lists}[s] = \{(s, t, \Gamma(t, \Sigma[s])) : t \in T' \wedge \Gamma(t, \Sigma[s]) > 1\}$. □

Definition 3.28 specifies a general model for the annotations that phase 2 adds to LALR(1) states. Each annotation references the manifestation description for the inadequacy from which phase 2 computed the annotation.

Definition 3.28 (*Inadequacy Annotation*)**.** Given $s : 1 \le s \le |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$, then an ***inadequacy annotation*** for $\Sigma[s]$ is a tuple (n_i, γ) such that:

1. $\exists s' : 1 \le s' \le |\Sigma| \wedge \exists t \in T' : n_i = (s', t, \Gamma(t, \Sigma[s'])) \in \mathit{inadequacy\_lists}[s']$.
2. γ is an ***inadequacy contribution matrix***, which describes whether and how any isocore split from $\Sigma[s]$ can make each of the contributions of $\Gamma(t, \Sigma[s'])$. That is, $|\gamma| = |\Gamma(t, \Sigma[s'])|$, and, $\forall i : 1 \le i \le |\gamma|, \gamma[i]$ is either:
   (a) Undefined because contribution $\Gamma(t, \Sigma[s'])[i]$ is an ***always contribution*** from the isocores that may be split from $\Sigma[s]$. That is, any isocore that may be split from $\Sigma[s]$ is guaranteed to make contribution $\Gamma(t, \Sigma[s'])[i]$.
   (b) A Boolean sequence such that $|\gamma[i]| = |C|$. For any isocore that may be split from $\Sigma[s]$, that isocore makes contribution $\Gamma(t, \Sigma[s'])[i]$ iff $\exists j$ such that both $\gamma[i][j]$ is true and t appears in the lookahead set of kernel item $C[j]$ in that isocore. This gives rise to the following two terms:
      i. Iff $\exists j : \gamma[i][j]$, then $\Gamma(t, \Sigma[s'])[i]$ is a ***potential contribution*** from the isocores that may be split from $\Sigma[s]$.
      ii. Iff $\nexists j : \gamma[i][j]$, then $\Gamma(t, \Sigma[s'])[i]$ is a ***never contribution*** from the isocores that may be split from $\Sigma[s]$. That is, any isocore that may be split from $\Sigma[s]$ is guaranteed not to make contribution $\Gamma(t, \Sigma[s'])[i]$. □

Definition 3.29 specifies how phase 2 computes inadequacy annotations, which it stores in the sequence *annotation_lists*. This definition employs an *annotate_manifestation* function to compute annotations on conflicted states, and it employs an *annotate_predecessor* function to compute annotations on predecessors of annotated states. Thus, this definition is recursive via *annotate_predecessor* and implies that phase 2 should perform a reverse iteration along conflicted states' lanes. This definition also implies that phase 2 may terminate an iteration along a lane upon encountering either of two conditions: (1) it annotates a state that has no predecessors, or (2) it computes an annotation for a state but discovers the state already has an identical annotation, so iterating further would fruitlessly replicate more annotations already computed. Condition 2 is important to avoid an infinite loop. That is, because Definition 3.28 describes a finite number of possible annotations for a given set of LALR(1) parser tables, condition 2 means that phase 2 is guaranteed to terminate.

Definition 3.29 (*annotation_lists*)**.** Given $s : 1 \le s \le |\Sigma|$, then the list of inadequacy annotations for state $\Sigma[s]$ is *annotation_lists*$[s]$, which is the union of the following two sets:

1. $\{n_a : \exists n_i \in \mathit{inadequacy\_lists}[s] : n_a = \mathit{annotate\_manifestation}(s, n_i)\}$.
2. $\{n_a : \exists s' : s \in \mathit{predecessors}[s'] \wedge \exists n'_a \in \mathit{annotation\_lists}[s'] : n_a = \mathit{annotate\_predecessor}(s, s', n'_a)\}$. □

Definition 3.30 (*annotate_manifestation*)**.** Given:

1. $s : 1 \leq s \leq |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$.
2. $n_i = (s, t, \Gamma(t, \Sigma[s])) \in inadequacy\_lists[s]$.

Then $annotate\_manifestation(s, n_i) = (n_i, \gamma)$ is an inadequacy annotation such that $|\gamma| = |\Gamma(t, \Sigma[s])|$ and, $\forall i : 1 \leq i \leq |\gamma|$ given that $\Gamma(t, \Sigma[s])[i] = (a_t, a_p)$, both of the following are true:

1. If $a_t =$ "S", then $\gamma[i]$ is undefined.
2. If $a_t =$ "R", then, given that $a_p = (\ell \to \varrho)$, both of the following are true:
   (a) If $\varrho \neq \epsilon$, then $|\gamma[i]| = |C|$ and, $\forall j : 1 \leq j \leq |C|, \gamma[i][j]$ is true iff $C[j] = ((\ell \to \varrho), |\varrho| + 1)$.
   (b) If $\varrho = \epsilon$, then $\gamma[i] = compute\_lhs\_contributions(s, \ell, t)$. □

Definition 3.31 (*compute_lhs_contributions*)**.** Given:

1. $s : 1 \leq s \leq |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$.
2. $\ell \in V' : \exists g : from\_state[g] = s \wedge \dagger(\Sigma[to\_state[g]]) = \ell$. This g is unique because a state cannot have more than one goto on the same nonterminal.
3. $t \in T'$.

Then $compute\_lhs\_contributions(s, \ell, t) =$ either:

1. Undefined if $t \in always\_follows[g]$.
2. A Boolean sequence κ if $t \notin always\_follows[g]$. $|\kappa| = |C|$. $\forall i : 1 \leq i \leq |\kappa|, \kappa[i]$ is true iff $follow\_kernel\_items[g][i] \wedge t \in item\_lookahead\_sets[s][i]$. □

For example, consider the LALR(1) parser tables in Table 4. Because state 18 is conflicted, Definition 3.29 demands that phase 2 compute an annotation for state 18 using *annotate_manifestation*. The conflict is on *a* and has 2 contributions. Thus, in Definition 3.30, $s = 18$, $t = a$, and $|\gamma| = 2$. Contribution 1 is S15, so $\gamma[1]$ is undefined. Contribution 2 is R9. R9's LHS is *E*. R9's RHS is ϵ. Thus, $\gamma[2]$ is computed from $compute\_lhs\_contributions(18, E, a)$. Let g be the goto index for state 18's G14, the goto on *E*. Because $goto\_follows[g]$ inherits *a* only via a predecessor dependency, $a \notin always\_follows[g]$ by Definition 3.20. For this reason and because there is only one kernel item in state 18, $|\gamma[2]| = 1$ by Definition 3.31. The one kernel item's RHS is $aCD \cdot E$, so $follow\_kernel\_items[g][1]$ is then true by Definition 3.16. $a \in item\_lookahead\_sets[18][1]$ by Definition 3.26. By Definition 3.31, $\gamma[2][1]$ is then true.

Consider the corresponding IELR(1) parser tables in Table 5. The isocores split from LALR(1) state 18 are IELR(1) states 18 and 21. Based on $\gamma[1]$, S15 is an always contribution to LALR(1) state 18's conflict on *a* because any isocore split from state 18 must make that contribution. Both IELR(1) isocores indeed make the S15 contribution. Based on $\gamma[2]$, R9 is a potential contribution to that conflict because such an isocore makes that contribution iff the first kernel item's lookahead set contains *a*. IELR(1) state 18 thus makes the R9 contribution, but IELR(1) state 21 does not.

Definition 3.32 (*annotate_predecessor*)**.** Given:

1. $s : 1 \leq s \leq |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$.
2. $s' : 1 \leq s' \leq |\Sigma| \wedge \Sigma[s'] = (C', A'_t, A'_r) \wedge s \in predecessors[s']$.
3. $n'_a = (n_i, \gamma') \in annotation\_lists[s'] : \exists s'' : \exists t : n_i = (s,'' t, \Gamma(t, \Sigma[s''])) \in inadequacy\_lists[s''] \wedge |\gamma'| = |\Gamma(t, \Sigma[s''])|$.

Then $annotate\_predecessor(s, s', n'_a) = (n_i, \gamma)$ is an inadequacy annotation such that $|\gamma| = |\gamma'|$ and, $\forall i : 1 \leq i \leq |\gamma|$, all of the following are true:

1. $\gamma[i]$ is undefined iff either $\gamma'[i]$ is undefined or $\exists j : 1 \leq j \leq |C'|$ such that both of the following are true:
   (a) $\gamma'[i][j]$.
   (b) Given that $C'[j] = ((\ell' \to \varrho'), d')$, both of the following are true:
      i. $d' = 2$.
      ii. $compute\_lhs\_contributions(s, \ell', t)$ is undefined.
2. Otherwise, $|\gamma[i]| = |C|$ and, $\forall j : 1 \leq j \leq |C|, \gamma[i][j]$ is true iff $\exists k : 1 \leq k \leq |C'|$ such that both of the following are true:
   (a) $\gamma'[i][k]$.
   (b) Given that $C[j] = ((\ell \to \varrho), d) \wedge C'[k] = ((\ell' \to \varrho'), d')$, either of the following is true:
      i. $((\ell \to \varrho), d) = ((\ell' \to \varrho'), d' - 1) \wedge t \in item\_lookahead\_sets[s][j]$.
      ii. $d' = 2 \wedge compute\_lhs\_contributions(s, \ell', t)[j]$. □

For example, consider LALR(1) state 17 in Table 4. Because state 17 is a predecessor of state 18, Definition 3.29 demands that phase 2 compute an annotation for state 17 using *annotate_predecessor* with the state 18 annotation we described above. Thus, in Definition 3.32, $s = 17$, $s' = s'' = 18$, $t = a$, the contribution matrix from state 18's annotation is now renamed to γ', and $|\gamma| = |\gamma'| = 2$. Because $\gamma'[1]$ is undefined, $\gamma[1]$ is undefined. $\gamma'[2][j]$ is true only for $j = 1$, but the dot in kernel item 1 of state 18 is not in position 2, so $\gamma[2]$ is defined. Because state 17 has only one kernel item, $|\gamma[2]| = 1$. The core of kernel item 1 in state 17 is the same as the core of kernel item 1 in state 18 except the dot is one position to the left. Definition 3.26 demanded that phase 2 compute $item\_lookahead\_sets[17][1]$ in order to compute $item\_lookahead\_sets[18][1]$ previously, and $a \in item\_lookahead\_sets[17][1]$. Thus, $\gamma[2][1]$ is true. Notice that $\gamma = \gamma'$.

Consider the isocores split from LALR(1) state 17: IELR(1) states 17 and 20 in Table 5. Based on $\gamma[1]$, both isocores make contribution 1 to the inadequacy that manifests as LALR(1) state 18's conflict on a. Based on $\gamma[2]$, contribution 2 is a potential contribution because an isocore split from LALR(1) state 17 makes contribution 2 iff the first kernel item's lookahead set contains a. IELR(1) state 17 thus makes contribution 2, but IELR(1) state 20 does not.[6] This result is intuitive because IELR(1) state 20 is the predecessor of IELR(1) state 21, which has a shift on a but no reduce on a and thus no conflict.

Because LALR(1) state 16 is a predecessor of state 17, Definition 3.29 demands that phase 2 compute an annotation for state 16 using *annotate_predecessor* with the state 17 annotation we just described. In Definition 3.32, $s = 16$, $s' = 17$, $s'' = 18$, $t = a$, and the contribution matrix from state 17's annotation is now renamed to γ'. The computation is similar to the computation for the annotation on state 17 and again yields $\gamma = \gamma'$. LALR(1) state 16 is split into IELR(1) states 16 and 19. Both IELR(1) isocores make contribution 1 to the inadequacy that manifests as LALR(1) state 18's conflict on a, but only IELR(1) state 16 makes contribution 2. Again, this result is intuitive because only IELR(1) state 16 remains in a lane of the conflict from LALR(1) state 18.

3.4.3. Split-stable dominant contributions

We continue our example from the previous section using Fig. 5's grammar and its LALR(1) and IELR(1) parser tables from Tables 4 and 5. Because LALR(1) state 2 is a predecessor of state 16, Definition 3.29 demands that phase 2 compute an annotation for state 2 using *annotate_predecessor* with the state 16 annotation we described in the previous section. In Definition 3.32, $s = 2$, $s' = 16$, $s'' = 18$, $t = a$, the contribution matrix from state 16's annotation is now renamed to γ', and $|\gamma| = |\gamma'| = 2$. Because $\gamma'[1]$ is undefined, $\gamma[1]$ is undefined. $\gamma'[2][j]$ is true only for $j = 1$, the dot in kernel item 1 of state 16 is in position 2, and the LHS is A, so phase 2 must compute $compute\_lhs\_contributions(2, A, a)$. Let g be the goto index for state 2's G3, the goto on A. Because $goto\_follows[g]$ inherits a via successor dependencies alone, $a \in always\_follows[g]$ by Definition 3.20. By Definition 3.31, $\gamma[2]$ is then undefined.

Notice that both of LALR(1) state 2's contributions to the inadequacy that manifests as LALR(1) state 18's conflict on a are identified as always contributions. Thus, for any isocores split from states 2 and 18, the state 18 isocore that is reachable from a state 2 isocore must contain both contributions. Because the set of contributions is exactly the same for all isocores, the dominant contribution must be the same as well. We thus say that the dominant contribution from state 2 is ***split-stable***. Because the dominant contribution from state 2 is split-stable, splitting state 2 cannot help eliminate this inadequacy, so this annotation on state 2 is useless.

Because LALR(1) state 0 is a predecessor of state 2, Definition 3.29 demands that phase 2 compute an annotation for state 0 using *annotate_predecessor* with the state 2 annotation we just described. By Definition 3.32, an always contribution in a contribution matrix remains an always contribution in the predecessor's contribution matrix. Thus, the dominant contribution specified by state 0's contribution matrix is split-stable, and so state 0's annotation is useless. If state 0 had a predecessor, its annotation would be identical and useless. And so on.

LALR(1) state 5 is another predecessor of state 16. Thus, Definition 3.29 demands that phase 2 also compute an annotation for state 5 using *annotate_predecessor* with the state 16 annotation we described in the previous section. In Definition 3.32, $s = 5$, $s' = 16$, $s'' = 18$, $t = a$, the contribution matrix from state 16's annotation is renamed to γ', and $|\gamma| = |\gamma'| = 2$. Because $\gamma'[1]$ is undefined, $\gamma[1]$ is undefined. $\gamma'[2][j]$ is true only for $j = 1$, the dot in kernel item 1 of state 16 is in position 2, and the LHS is A, so phase 2 must compute $compute\_lhs\_contributions(5, A, a)$. Let g be the goto index for state 5's G6, the goto on A. $a \notin goto\_follows[g]$, so $a \notin always\_follows[g]$ by Theorem 3.19. For this reason and because there is only one kernel item in state 5, $|\gamma[2]| = 1$ by Definition 3.31. The one kernel item's RHS is $b \cdot ABb$, and $Bb \not\Rightarrow^* \epsilon$, so $follow\_kernel\_items[g][1]$ is then false by Definition 3.16. By Definition 3.31, $\gamma[2][1]$ is then false.

Contribution 1 to the inadequacy that manifests as LALR(1) state 18's conflict on a is an always contribution from the isocores that may be split from state 5, but contribution 2 is a never contribution. Thus, for any isocores split from states 5 and 18, the state 18 isocore that is reachable from a state 5 isocore must contain contribution 1. As long as the lane in which the state 5 isocore appears does not converge at some eventual successor state with a lane in which a state 2 isocore appears, then the state 18 isocore cannot contain contribution 2. Phase 3 does not let such lanes converge because of their differing dominant contributions. Because all isocores that can be split from state 5 make exactly the same contributions as state 5, the dominant contribution from state 5 is split-stable and thus always dominates in the reachable state 18 isocore, so this annotation on state 5 is useless.

Like an always contribution, a never contribution in a contribution matrix remains a never contribution in the predecessor's contribution matrix by Definition 3.32. Thus, Definition 3.29 demands that phase 2 compute an annotation for state 0 that is identical to state 5's annotation and thus is also useless. If state 0 had a predecessor, its annotation would be identical and useless. And so on.

Observation 3.33 (*Simple Split-Stable Dominance*). Given an inadequacy annotation $n_a = (n_i, \gamma)$, then γ specifies a split-stable dominant contribution if (but not only if, as we explain below), $\forall i : 1 \leq i \leq |\gamma|, \gamma[i] = \text{undefined} \vee \forall j : 1 \leq j \leq |\gamma[i]|, \neg\gamma[i][j]$. In other words, when all contributions are always or never contributions, γ specifies a split-stable dominant contribution. Split-stable dominance is completely defined in Definition 3.35. □

[6] Be careful here. The token sets shown in these tables are goto follow sets not item lookahead sets. To compute item lookahead sets, use Definition 3.26 or Observation 3.2.

Observation 3.34 (*Useless Inadequacy Annotations*). An inadequacy annotation $n_a = (n_i, \gamma)$ is ***useless*** iff γ specifies a split-stable dominant contribution. In this case, IELR(1) phase 2 can discard n_a and can skip all *annotate_predecessor* invocations on n_a specified by Definition 3.29. That is, phase 2 can terminate its iteration along the current LALR(1) lane. □

As we mentioned in Section 2.4, if the user of a parser generator like Yacc or Bison provides no precedence or associativity declarations that resolve some S/R conflict, the resulting dominant contribution function returns the shift action for this conflict no matter what reduce contributions are present. In light of this observation, consider how the examples we have been discussing change in the case of Bison if we assume no precedence or associativity declarations for the grammar of Fig. 5. The dominant contribution for the conflict on a in LALR(1) state 18 in Table 4 would be split-stable because any isocore split from state 18 would shift on a. In other words, the conflict on a would not be mysterious from the perspective of any viable prefix of state 18. Thus, the annotation on state 18 would be useless as would be all the annotations that phase 2 computes from that annotation along the lanes of state 18. Phase 3 would have no reason to split any LALR(1) states.

Definition 3.35 (*Split-Stable Dominant Contribution*). Given an inadequacy annotation $n_a = (n_i, \gamma) : \exists s : \exists t : n_i = (s, t, \Gamma(t, \Sigma[s])) \in$ *inadequacy_lists*$[s]$, then γ specifies a ***split-stable dominant contribution*** iff $\Delta(t, \Gamma') = \Delta(t, \Gamma'')$ for the one Γ' and for every Γ'' that meet the following conditions:

1. $\Gamma'' \subset \Gamma' \subseteq \Gamma(t, \Sigma[s])$.
2. $\forall i : 1 \le i \le |\gamma|, \Gamma(t, \Sigma[s])[i] \in \Gamma'$ iff either $\gamma[i]$ is undefined or $\exists j : \gamma[i][j]$. In other words, Γ' is formed by removing the never contributions from $\Gamma(t, \Sigma[s])$ according to γ.
3. $\forall i : 1 \le i \le |\gamma| \land \Gamma(t, \Sigma[s])[i] \in \Gamma' \land \Gamma(t, \Sigma[s])[i] \notin \Gamma,'' \exists j : \gamma[i][j]$. In other words, all contributions removed from Γ' to form any Γ'' must be potential contributions according to γ. □

While Observation 3.34 is an important optimization of phase 2, it is not necessary for correct behavior. Moreover, the exact details of implementing the computation specified by Definition 3.35 depend upon a parser generator's exact rules for Δ. Our IELR(1) implementation for Bison includes this computation, but we found it to be non-trivial. An implementation could instead rely solely on the special case of Definition 3.35 identified by Observation 3.33. This special case is independent of Δ because it requires that there is no subset of Γ' that meets the Γ'' conditions in Definition 3.35.

As in our last example above, whenever phase 2 computes a useless annotation with the function *annotate_manifestation*, the associated grammar-relative inadequacy is not an LR(1)-relative inadequacy. However, the reverse is not true. In general, the fact that an annotation is useful does not indicate that splitting the annotated state necessarily proves useful. The trouble is that, while phase 2 determines an annotation's usefulness by examining its contribution matrix, sometimes the remainder of a conflicted state's lanes must be examined to determine the usefulness of state splitting. Thus, these cases must wait for phase 3, which can examine all the annotations that phase 2 has computed in the lanes.

3.5. Phase 3: Split states

Phase 3 behaves similarly to canonical LR(1), Pager's algorithm, and phase 0 step 1. That is, each of these algorithms computes a new set ofparser tables by computing successor states recursively starting from the start state. Along the way, each algorithm merges each new state with some existing state if the two states pass that algorithm's state compatibility test. In the final parser tables computed by each of these algorithms, each state is an isocore of one LALR(1) state, and their transition successors are also isocores. Thus, each lane in the parser tables is isocoric with one lane in the LALR(1) parser tables. However, while LALR(1) fully merges all isocore sets, the stricter state compatibility tests of the other algorithms may cause some sections of some lanes to branch into multiple isocores with different lookahead sets.

Phase 0 step 1 already computed LALR(1) state cores, the transitions among them, and thus the cores of all possible lanes in the new parser tables, so phase 3 need not recompute them. Phase 3 must retain LALR(1) state cores anyway in order not to lose their associations with inadequacy annotations, which phase 3's state compatibility test depends upon. However, phase 3 must compute new lookahead sets. In order to do so, it need not recompute non-kernel items, which phase 0 step 1 discarded. Instead, like phase 2, phase 3 can employ *follow_kernel_items*, *always_follows*, and phase 0's goto tables. Thus, in order to compute successor states recursively from the start state as we described in the previous paragraph, phase 3 merely propagates new lookaheads along the existing LALR(1) state transitions and, as dictated by inadequacy annotations, splits successors that have no existing isocores that are compatible with the new lookaheads.

Throughout this section, let Σ be the set of states computed so far at any point in time during phase 3. At the beginning of phase 3, Σ is the set of LALR(1) states computed by phase 0 except that, for every state s, given that $s = (C, A_t, A_r) \in \Sigma$, and, for every reduce action r, given that $r = (K, p) \in A_r$, phase 3 discards the LALR(1) reduction lookahead set K to be replaced in phase 4. Phase 3 assigns each new state a higher index than the index of any existing state, so each original LALR(1) state core retains its original index $s : 1 \le s \le |\mathscr{A}(G)|$. In Section 3.5.1, we define the tables that phase 3 employs in order to keep track of isocores in Σ. In Section 3.5.2, we define how phase 3 propagates lookaheads from each state to its successor states. In Section 3.5.3, we define phase 3's compatibility test for merging states in order to minimize $|\Sigma|$. Using these definitions in Section 3.5.4, we define phase 3's algorithm for computing the final Σ before phase 4 computes full reduction lookahead sets.

3.5.1. *Tracking isocores*

Every new state that phase 3 computes is a product of splitting one of the original LALR(1) states. We refer to that LALR(1) state as the new state's ***LALR(1) isocore***. For completeness, we say that the LALR(1) isocore for an original LALR(1) state is itself.

As we explained in Section 3.4, the inadequacy annotations on the LALR(1) isocore of a state describe whether and how that state can contribute to inadequacies. Moreover, phase 0 constructed its goto tables for the original LALR(1) states, but valuable information for any state can be found in the goto tables' entries for that state's LALR(1) isocore. Thus, when phase 3 computes a state, it records the index of its LALR(1) isocore.

Definition 3.36 (*lalr1_isocores*)**.** Given $s : 1 \le s \le |\Sigma|$, then *lalr1_isocores*[*s*] is the index of the ***LALR(1) isocore*** of $\Sigma[s]$. That is, $1 \le lalr1\_isocores[s] \le |\mathscr{A}(G)|$ and $\Sigma[s] \doteq \Sigma[lalr1\_isocores[s]]$. Thus, $s \le |\mathscr{A}(G)| \Leftrightarrow s = lalr1\_isocores[s]$. □

Each time phase 3 propagates new lookaheads from a state to one of its successors, phase 3 may need to check the successor's isocores to see if the new lookaheads are compatible with any of them. Thus, phase 3 must maintain a record of all isocore sets within Σ.

Definition 3.37 (*isocore_nexts*)**.** Given $s : 1 \le s \le |\Sigma|$, then *isocore_nexts*[*s*] is the next member after *s* in a circularly linked list whose members form the set of state indices $\{i : \Sigma[i] \in \mathscr{I}(\Sigma[s], \Sigma)\}$. □

3.5.2. *Item lookahead sets*

Because phase 3 may split states, it cannot use the LALR(1) data that phase 2 originally computed in the table *item_lookahead_sets*, so phase 3 must recompute this table. Like phase 2, phase 3 need not compute all lookaheads. Phase 3 need compute only the lookaheads mentioned in inadequacy annotations because those are the only lookaheads that influence phase 3's state compatibility test. Thus, Definition 3.38 defines a *lookahead_set_filters* function that examines inadequacy annotations. Using *lookahead_set_filters*, Definition 3.40 defines a *propagate_lookaheads* function to propagate lookaheads from a state through these filters to a specified successor. *propagate_lookaheads* returns the propagated lookaheads so that phase 3 can then decide whether to merge them into the existing successor's *item_lookahead_sets* entry or to merge them into the entry for some isocore of that successor.

Definition 3.38 (*lookahead_set_filters*)**.** Given $s : 1 \le s \le |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$, then both of the following are true:

1. $|lookahead\_set\_filters(s)| = |C|$.
2. $\forall j : 1 \le j \le |C|, lookahead\_set\_filters(s)[j] = \{t \in T' : \exists(n_i, \gamma)$
   $\in annotation\_lists[lalr1\_isocores[s]] : \exists s' : n_i = (s', t, \Gamma(t, \Sigma[s'])) \wedge \exists i : \gamma[i][j]\}$. □

Definition 3.39 (*compute_goto_follow_set*)**.** Given:

1. $s : 1 \le s \le |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$.
2. $n \in V'$.

Then $compute\_goto\_follow\_set(s, n) = \{t \in T' : \exists g : from\_state[g] = lalr1\_isocores[s] \wedge \dagger(\Sigma[to\_state[g]]) = n \wedge (t \in always\_follows[g] \vee \exists k : follow\_kernel\_items[g][k] \wedge t \in item\_lookahead\_sets[s][k])\}$. □

Definition 3.40 (*propagate_lookaheads*)**.** Given:

1. $s : 1 \le s \le |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$.
2. $s' : 1 \le s' \le |\Sigma| \wedge \Sigma[s'] = (C', A'_t, A'_r) \wedge \exists y \in \{V' \cup T'\} : \delta(\Sigma[s], y) = \Sigma[s']$.

Then $|propagate\_lookaheads(s, s')| = |C'|$ and, $\forall k' : 1 \le k' \le |C'|$, given that $C'[k'] = ((\ell' \to \varrho'), d')$, both of the following are true:

1. If $d' > 2$, then $propagate\_lookaheads(s, s')[k'] = \{t \in lookahead\_set\_filters(s')[k'] : \exists k : C[k] = ((\ell' \to \varrho'), d' - 1) \wedge t \in item\_lookahead\_sets[s][k]\}$.
2. If $d' = 2$, then $propagate\_lookaheads(s, s')[k'] = lookahead\_set\_filters(s')[k']$
   $\cap\, compute\_goto\_follow\_set(s, \ell')$. □

In Definition 3.40, the $d' = 1$ case is impossible. That is, by Definition 3.1, $d' = 1$ requires that $\ell' = S'$ and thus that $\Sigma[s']$ be the start state. Thus, by Definition 2.14, $\nexists y \in \{V' \cup T'\} : \delta(\Sigma[s], y) = \Sigma[s']$, but this contradicts the given.

At the beginning of phase 3, $\forall s : 1 \le s \le |\Sigma|$, $\Sigma[s]$ is one of the original LALR(1) states, and so phase 3 has not yet recomputed *item_lookahead_sets*[*s*] from any predecessor of $\Sigma[s]$. That is, *s* essentially marks a place holder for a new state. Thus, for the first predecessor of $\Sigma[s]$ from which phase 3 computes the lookaheads to propagate to *item_lookahead_sets*[*s*], phase 3 can consider those lookaheads to be compatible with *item_lookahead_sets*[*s*]. Because some inadequacy annotations in *annotation_lists*[*s*] might specify always contributions, phase 3 cannot merely initialize *item_lookahead_sets*[*s*] to empty sets in order to guarantee that *item_lookahead_sets*[*s*] appears compatible with lookaheads from the first predecessor. Instead, phase 3 records whether it has yet computed lookaheads from any predecessor, and phase 3's state compatibility test must check this record before bothering to examine the contents of *item_lookahead_sets*[*s*].

Definition 3.41 (*lookaheads_recomputed*)**.** Given $s : 1 \le s \le |\Sigma|$, *lookaheads_recomputed*[*s*] is true iff phase 3 has computed the lookaheads that at least one predecessor of $\Sigma[s]$ propagates to *item_lookahead_sets*[*s*]. Thus, $\forall s : 1 \le s \le |\mathscr{A}(G)|$, phase 3 initializes *lookaheads_recomputed*[*s*] to false. □

3.5.3. State compatibility

Definition 3.42 defines a *dominant_contribution* function, which phase 3 employs while testing the compatibility of two states. This function accepts a state s, an inadequacy annotation n_a on the LALR(1) isocore of s, and lookahead sets K. It then computes the dominant contribution that s would make to the inadequacy referenced by n_a if the lookahead sets of s were replaced by K. In the special case that s would make no contributions, *dominant_contribution* returns undefined.

Definition 3.42 (*dominant_contribution*)**.** Given:

1. $s : 1 \le s \le |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$.
2. $n_a = (n_i, \gamma) \in annotation\_lists[lalr1\_isocores[s]] : \exists s' : \exists t \in T' : n_i = (s', t, \Gamma(t, \Sigma[s']))$ $\in inadequacy\_lists[s']$.
3. $K : |K| = |C| \wedge \forall k : 1 \le k \le |C|, K[k] \subseteq T'$.

Let $\Gamma' = \{c : \exists i : 1 \le i \le |\Gamma(t, \Sigma[s'])| \wedge c = \Gamma(t, \Sigma[s'])[i] \wedge (\gamma[i] = \text{undefined} \vee (\exists j : \gamma[i][j] \wedge t \in K[j]))\}$, then $dominant\_contribution(s, n_a, K)$ is either:

1. $\Delta(t, \Gamma')$ iff $|\Gamma'| > 0$.
2. Undefined iff $|\Gamma'| = 0$. □

Phase 3 considers two states to be compatible only if they are isocores and thus appear in the same isocore list recorded in *isocore_nexts*. To test the compatibility of two isocores, phase 3 employs an *is_compatible* function, which Definition 3.43 below defines using *dominant_contribution*. *is_compatible* does not actually examine two complete states. Instead, it examines one complete state, which includes both a core and lookahead sets, and it examines lookahead sets computed by *propagate_lookaheads*. The combination of the complete state's core with the latter lookahead sets defines the second state implicitly.

Definition 3.43 (*is_compatible*)**.** Given:

1. $s : 1 \le s \le |\Sigma| \wedge \Sigma[s] = (C, A_t, A_r)$.
2. $K : |K| = |C| \wedge \forall k : 1 \le k \le |C|, K[k] \subseteq T'$.

$is\_compatible(s, K)$ is true iff $\neg lookaheads\_recomputed[s]$ or, $\forall n_a \in annotation\_lists[lalr1\_isocores[s]]$, any of the following are true:

1. $dominant\_contribution(s, n_a, item\_lookahead\_sets[s]) = dominant\_contribution(s, n_a, K)$.
2. $dominant\_contribution(s, n_a, item\_lookahead\_sets[s])$ is undefined.
3. $dominant\_contribution(s, n_a, K)$ is undefined. □

As mentioned at the end of the previous section and as reflected in Definition 3.43, *is_compatible* must consider two isocores to be compatible if one of them is an LALR(1) isocore whose lookahead sets phase 3 has not yet recomputed. Otherwise, *is_compatible* considers two isocores to be compatible iff, for every grammar-relative inadequacy from the LALR(1) parser tables,the two isocores make compatible sets of contributions. *is_compatible* considers two such sets of contributions to be compatible iff either (1) they have the same dominant contribution or (2) one or both sets are empty. We assume for now that Δ is ***merge-stable***. That is, for case 1, if the two isocores are merged, then the merged state is guaranteed to make the same dominant contribution as each of the original isocores. Thus, from the perspective of each viable prefix of the conflicted LALR(1) state by which the inadequacy manifests, either (1) there is no change in parser action on the conflicted token when the two isocores are merged or (2) the conflict is irrelevant. By Definitions 2.24 and 2.26, merging two states that pass phase 3's compatibility test cannot then induce a mysterious conflict and thus an LR(1)-relative inadequacy in the new parser tables.

In order to simplify our discussion so far, we have been ignoring a potential flaw in phases 2 and 3 of the IELR(1) algorithm. That is, there might exist Δ's that are not merge-stable. Thus, even when phase 3 deems two states to be compatible based on phase 2's inadequacy annotations, merging those states might induce an LR(1)-relative inadequacy in the parser tables. Fortunately, Bison never computes such a Δ regardless of what the user specifies for G or for the user portion of Δ. The flaw is possible however if an alternate IELR(1) implementation chooses to resolve conflicts in a different manner than Bison. In the remainder of this section, we formalize the concept of merge stability and explain how an IELR(1) implementation can fix the flaws in phases 2 and 3 when it cannot guarantee Δ to be merge-stable.

Consider the example depicted in Fig. 7. By Observation 3.34, phase 2 discards any inadequacy annotation that specifies a split-stable dominant contribution and that is thus useless. Let p be a state for whose LALR(1) isocore phase 2 discarded such an inadequacy annotation. Let i be the associated inadequacy. Let d be the dominant contribution from p to i. By Definitions 3.38 and 3.40, phase 3 records a particular lookahead in p only if some inadequacy annotation on the LALR(1) isocore of p specifies a potential contribution that depends on whether that lookahead appears in p. By Definition 3.35, a useless inadequacy annotation can specify potential contributions. Thus, phase 3 might not record some lookaheads that determine whether p makes such contributions to i. Recording these lookaheads seems useless anyway because, by Definition 3.35, the dominant contribution to i from all possible isocores of p is guaranteed to be d, and so the presence of these lookaheads in p does not affect whether the isocores can be merged safely. However, in general, phase 3 records lookaheads in a state for

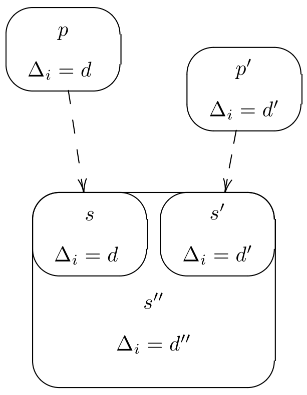


Fig. 7. Merge stability example. This figure depicts states p, p', s, s', and s'' from the example in Section 3.5.3. In this figure, we use the shorthand Δ_i to refer to the dominant contribution that each state makes to the inadequacy i according to Δ. If Δ is merge-stable, then $d = d' \Rightarrow d = d''$.

more than just testing the compatibility of that state with its isocores. Phase 3 also propagates the lookaheads to the state's successors throughout the rest of the lanes where the lookaheads might affect the compatibility of other states.

Continuing the example in Fig. 7, let s be a state such that phase 2 eventually computed the useless inadequacy annotation on the LALR(1) isocore of p from an inadequacy annotation on the LALR(1) isocore of s via the recursion in *annotate_predecessor*. Thus, p propagates its contributions for i through s. Assume no other state propagates contributions for i through s, so the dominant contribution from s to i is also d. Let s' and p' be two states with the same relationship we have described so far for s and p, and let d' be the dominant contribution to i from p' and thus from s'. However, assume $p \not\doteq p'$ while $s \doteq s'$. Thus, p and p' reside in separate lanes that converge at s and s' if phase 3 decides to merge s and s' to form a new state, s''. Let d'' be the dominant contribution from s'' to i. Although s and all isocores of p make dominant contribution d to i, s' and s'' can pick up other contributions to i propagated from p'. Assume phase 2 did not discard the annotation for i on the LALR(1) isocore of s, s', and s'', so its dominant contribution is not split-stable. Thus, the contributions propagated from p' might yield d' and d'' that are not equal to d.

Now assume $d = d'$. By Definition 3.43, phase 3 should then merge s and s' to form s''. We must consider two questions:

1. Can the dominant contribution still change because of the merge? Phase 3 should not actually merge s and s' if $d \neq d''$. To handle this issue, *is_compatible* could be adjusted to tentatively merge s and s' and then compare d and d''.
2. What if phase 3 never propagated some lookaheads from p to s because phase 2 discarded the useless annotation on the LALR(1) isocore of p? Thus, phase 3 might not be aware of all possible contributions to i from s. Phase 3 can be sure these contributions do not affect d, but they might interact with contributions propagated from p' and thus affect d''. In this case, the previous adjustment to *is_compatible* is not sufficient. This issue can be handled in any of the following ways:
   (a) Adjust *lookahead_set_filters* so that phase 3 would propagate all lookaheads in the parser tables without regard to inadequacy annotations.
   (b) Adjust phase 2 to replace Definition 3.35 with Observation 3.33. That is, phase 2 would never discard inadequacy annotations that specify potential contributions, and thus phase 3 would propagate all lookaheads that might influence state compatibility.
   (c) Merge s and s' regardless of d''. Add a new kind of annotation to s'' stating that d is the dominant contribution from s'' to i regardless of what contributions s'' contains. Propagate this annotation to successors of s'' alongside any contributions to i. In the remainder of phase 3 and in phase 5, ignore actual contributions to i from any state with such an annotation.

For our IELR(1) implementation in Bison, we resolved this issue by verifying that the answer to question 1 above is always no. That is, we verified that Δ is always merge-stable no matter what the user specifies for G or for the user portion of Δ.

Definition 3.44 (*Merge Stability*). Δ is ***merge-stable*** iff, $\forall s : 1 \le s \le |\mathscr{A}(G)|, \forall t \in T', \forall \Gamma' \subseteq \Gamma(t, \Sigma[s]) : |\Gamma'| > 0$, $\forall \Gamma'' \subseteq \Gamma(t, \Sigma[s]) : |\Gamma''| > 0$, it holds true that if $\Delta(t, \Gamma') = \Delta(t, \Gamma'')$ then $\Delta(t, \Gamma') = \Delta(t, \{\Gamma' \cup \Gamma''\})$. □

We found verifying merge stability for Bison to be non-trivial. Moreover, it may be interesting to define a Δ that implements one or more of Klint and Visser's disambiguation filters [21] completely at parser-generation time. It may also be interesting to permit the parser specification developer to provide explicit code to be evaluated for Δ at parser-generation time. If such a Δ proves not to be merge-stable or if the implementer wishes to avoid verifying the merge stability of each new Δ, it would be necessary to make adjustments to phases 2, 3, and 5 as outlined above.

3.5.4. Algorithm

In this section, we outline phase 3's algorithm in pseudo-code using the functions, tables, and initializations we described in the preceding sections.

Phase 3's entry point is the routine *split_states*, defined in Definition 3.45. Conceptually, the *split_states* routine propagates lookaheads to successor states recursively starting from the start state. However, the pseudo-code on lines 5–9 reflects our Bison implementation of *split_states*, which performs a breadth-first iteration. Bison's existing implementation of phase 0 step 1 also computes states in a breadth-first order. Thus, the states with indices $> |\mathscr{A}(G)|$ in Σ are ordered consistently with the states with indices $\le |\mathscr{A}(G)|$ to make the generated parser tables easier to understand.

On line 9, *split_states* invokes *compute_state*, defined in Definition 3.47, to propagate lookaheads from a specified state, $\Sigma[s]$, to some compatible isocore of the specified successor, $\Sigma[s']$, which is reached by the specified transition, x. On lines 1–10, *compute_state* computes the lookaheads to propagate, K, and tries to select a compatible isocore, $\Sigma[i]$. On lines 13–20, if there is no compatible isocore, *compute_state* appends a new isocore to Σ, sets its lookaheads to K, and updates transition x, making the new isocore the successor. If, instead, phase 3 has not yet computed the lookaheads from any predecessor of $\Sigma[i]$, then $i = s'$ and *compute_state* sets the lookaheads of $\Sigma[s']$ to K on lines 21–23. Otherwise, on lines 24–26, *compute_state* updates transition x, making $\Sigma[i]$ the successor, and invokes *merge_lookaheads*, defined in Definition 3.46, to merge K with the existing lookaheads in $\Sigma[i]$.

Definition 3.45 (*split_states*).

```
 1 for (let $s = 1$; $s \le |\Sigma|$; $s = s + 1$) do:
 2     set $lalr1\_isocores[s] = s$.
 3     set $isocores\_nexts[s] = s$.
 4     set $lookaheads\_recomputed[s] = $ false.
 5 for (let $s = 1$; $s \le |\Sigma|$; $s = s + 1$) do:
 6     let $(C, A_t, A_r) = \Sigma[s]$.
 7     for (let $x = 1$; $x \le |A_t|$; $x = x + 1$) do:
 8         let $(y, s') = A_t[x]$.
 9         $compute\_state(s, s', x)$.  □
```

Definition 3.46 (*merge_lookaheads*(i, K)).

```
 1 let $(C, A_t, A_r) = \Sigma[i]$.
 2 let $new\_lookaheads = $ false.
 3 for (let $k = 1$; $k \le |C|$; $k = k + 1$) do:
 4     set $K[k] =$
 5         $K[k] - \{K[k] \cap item\_lookahead\_sets[i][k]\}$.
 6     if ($|K[k]| > 0$) do:
 7         set $new\_lookaheads = $ true.
 8         set $item\_lookahead\_sets[i][k] =$
 9             $K[k] \cup item\_lookahead\_sets[i][k]$.
10 if ($new\_lookaheads$) do:
11     for (let $x = 1$; $x \le |A_t|$; $x = x + 1$) do:
12         let $(y, i') = A_t[x]$.
13         if ($\neg lookaheads\_recomputed[i']$) do:
14             break.
15         $compute\_state(i, i', x)$.  □
```

Definition 3.47 (*compute_state*(s, s', x)).

```
 1 let $K = propagate\_lookaheads(s, s')$.
 2 let $found = $ false.
 3 let $i = s'$.
 4 while (true) do:
 5     if ($is\_compatible(i, K)$) do:
 6         set $found = $ true.
 7         break.
 8     if ($isocore\_nexts[i] = s'$) do:
 9         break.
10     set $i = isocore\_nexts[i]$.
11 let $(C, A_t, A_r)$ be an alias for $\Sigma[s]$.
12 let $(y, r)$ be an alias for $A_t[x]$.
13 if ($\neg found$) do:
14     append $\Sigma[i]$ to $\Sigma$.
15     append $lalr1\_isocores[i]$ to $lalr1\_isocores$.
16     append $isocore\_nexts[i]$ to $isocore\_nexts$.
17     set $isocore\_nexts[i]$ to $|isocore\_nexts|$.
18     append true to $lookaheads\_recomputed$.
19     append $K$ to $item\_lookahead\_sets$.
20     set $r = |\Sigma|$.
21 else if ($\neg lookaheads\_recomputed[i]$) do:
```

```
22      set $item\_lookahead\_sets[i] = K$.
23      set $lookaheads\_recomputed[i] =$ true.
24  else do:
25      set $r = i$.
26      $merge\_lookaheads(i, K)$.   □
```

If $\Sigma[i]$ does not already contain some of the lookaheads specified in K, *merge_lookaheads* immediately propagates the updated lookaheads from $\Sigma[i]$ to each successor, $\Sigma[i']$, by recursive invocations of *compute_state* on line 15.[7] On line 13, upon encountering a transition of index x from $\Sigma[i]$ to a $\Sigma[i']$ such that $\neg lookaheads\_recomputed[i']$, *merge_lookaheads* can be sure that *split_states* has not yet propagated lookaheads from $\Sigma[i]$ along all transitions of indices x and above. Thus, *split_states* propagates lookaheads along these transitions later, and so *merge_lookaheads* need not recurse any deeper.

3.6. Phase 4: Compute reduction lookaheads

Phase 4 runs step 2 of phase 0 again without modification. That is, it computes the full lookahead sets on reductions in all IELR(1) parse states.

3.7. Phase 5: Resolve remaining conflicts

All that is left is to resolve the remaining conflicts in the parser tables. Our IELR(1) implementation uses Bison's existing conflict resolution algorithm without modification.

3.8. Suboptimum state merging

IELR(1) is a minimal LR(1) algorithm in that it generates a minimally sized set of states, Σ, that fulfills a given LR(1) parser specification, (G, Δ), as described in Definition 2.21. For example, for every practical (G, Δ) we discuss in Section 4, our IELR(1) implementation generates a Σ that is nearly as small as $\mathscr{A}_r(G, \Delta)$. However, like previous minimal LR(1) algorithms, IELR(1) is not guaranteed to always merge states in a manner that yields the absolute minimum $|\Sigma|$ necessary to fulfill (G, Δ). We refer to this shortcoming as ***suboptimum state merging***. In Sections 3.8.1–3.8.3, we discuss three separate causes of suboptimum state merging for IELR(1) and some ways to reduce their effects.

3.8.1. Phase 3 orphans

Consider the recursive invocation of *compute_state* in Definition 3.46 for *merge_lookaheads*. The purpose of this invocation is to propagate new lookaheads from $\Sigma[i]$ along transition x to the successor $\Sigma[i']$. If $\Sigma[i']$ is not compatible with the new lookaheads, *compute_state* finds or constructs an isocore of $\Sigma[i']$ to replace $\Sigma[i']$ as the successor on transition x. However, *compute_state* does not remove from $\Sigma[i']$ any lookaheads that phase 3 has already propagated from $\Sigma[i]$ but that phase 3 has not propagated from any other predecessor of $\Sigma[i']$. We say such lookaheads are ***phase 3 orphans***.

Phase 3 orphans can cause future state compatibility tests to fail unnecessarily. The final Σ might then contain states that are compatible with one another but not merged, unnecessarily increasing $|\Sigma|$. We call these states ***remergeable states***. Moreover, phase 3 orphans can cause a state to lose every predecessor if, for the lookaheads that phase 3 tries to propagate from each predecessor, the state fails the compatibility test. That state and possibly its successors are then *unreachable states*, which are useless and unnecessarily increase $|\Sigma|$.

Definition 3.48 (*Unreachable State*). Let s be the index of the start state in Σ. $\forall s' : 1 \le s' \le |\Sigma|$, $\Sigma[s']$ is an *unreachable state* iff $\nexists \sigma \in \{V' \cup T'\}^* : \delta^*(\Sigma[s], \sigma) = \Sigma[s']$. □

An algorithm to remove any state that matches Definition 3.48 is straightforward. The first author previously contributed an implementation for such an algorithm to Bison 2.3b in order to remove states rendered unreachable by conflict resolution in LALR(1) parser tables. Thus, our IELR(1) implementation also contains this implementation, which we think of as an optional phase 6.

One way to avoid all effects of phase 3 orphans would be to remove phase 3 orphans immediately during phase 3. That is, phase 3 would maintain a record of every state's predecessors. Every time *compute_state* would remove a predecessor from a state, *compute_state* would then recompute the state's lookaheads from its remaining predecessors. However, our IELR(1) implementation does not remove phase 3 orphans or identify remergeable states. As we mentioned previously, our implementation still manages to generate a Σ that is nearly as small as $\mathscr{A}_r(G, \Delta)$ for every practical (G, Δ) we have tried.

7 The recursion that *merge_lookaheads* invokes is similar to Pager's *context-propagation procedure* [26].

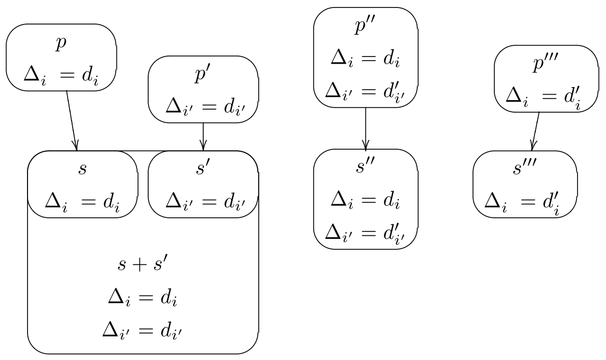


Fig. 8. Greedy merging example. This figure depicts the states from the example in Section 3.8.3. In this figure, we use the shorthand Δ_i to refer to the dominant contribution that each state makes to the inadequacy i according to Δ.

3.8.2. Phase 5 orphans

When phase 5 resolves conflicts, it removes parser actions as described in Definition 2.20. When phase 5 removes a reduce action, some associated gotos might become useless. When a goto becomes useless or when phase 5 removes a shift action, some lookahead propagation paths might be severed. Phases 2 and 3 do not predict what lookahead propagation paths phase 5 might sever. Thus, severed lookahead propagation paths can result in ***phase 5 orphans*** of two different types:

1. Phase 2 might invoke *annotate_predecessor* on a pair of states between which phase 5 removes the transition that establishes the predecessor relationship. We call the result an ***orphaned inadequacy annotation***.
2. Phase 3 might propagate lookaheads to a state along a transition that phase 5 removes. If those lookaheads have no other path to reach that state, we call them ***orphaned lookaheads***.

Both types of phase 5 orphans have the same effect as phase 3 orphans. That is, they can cause state compatibility tests to fail unnecessarily, producing remergeable and unreachable states, which increase $|\Sigma|$. Again, unreachable state removal is straightforward to implement as IELR(1) phase 6. Remergeable states are now more complicated.

Definition 3.49 (*Remergeable State*)**.** $\forall s : 1 \le s \le |\Sigma|$, $\Sigma[s]$ is a ***remergeable state*** iff, after removing all orphaned lookaheads from *item_lookahead_sets* and removing all orphaned inadequacy annotations from *annotation_lists*, $\exists s' : \Sigma[s] \doteq \Sigma[s'] \wedge$ *is_compatible*$(s,$ *item_lookahead_sets*$[s'])$. □

We propose that all remergeable states resulting from phase 5 orphans could be remerged by running an altered version of phases 2 and 3 after phase 5. These altered phases 2 and 3 would recompute annotation lists and lookaheads with full knowledge of which lookahead propagation paths are severed by phase 5. However, we have not attempted to formulate the details of such an algorithm as we have not discovered a practical (G, Δ) that would benefit significantly.

3.8.3. Greedy merging

Even when the final Σ contains no unreachable or remergeable states, it is possible that $|\Sigma|$ might not be the absolute minimum necessary to fulfill (G, Δ). The trouble is that phase 3's algorithm to find a minimal $|\Sigma|$ is greedy. The crucial choices come as *compute_state* searches for compatible states with which to merge new lookaheads. If no compatible state exists, *compute_state* has no choice but to create a new state. If one or more compatible states exist, *compute_state* could merge the new lookaheads with any of those states without increasing $|\Sigma|$, so it chooses the first compatible state it finds. Thus, in all cases, *compute_state* makes a locally optimum choice. However, if *compute_state* were to consider the effect the current merge has on every future merge, it might be able to reduce the final $|\Sigma|$ further by making a different choice. That is, the locally optimum choice that *compute_states* makes might not permit a globally optimum solution.

Consider the example depicted in Fig. 8. Let p be a state that makes dominant contribution d_i to inadequacy i. Let p' be a state that makes dominant contribution $d_{i'}$ to inadequacy i'. Let p'' be a state that makes dominant contribution d_i to inadequacy i and dominant contribution $d'_{i'}$ to inadequacy i'. Let p''' be a state that makes dominant contribution d'_i to inadequacy i. Assume these states make no contributions to any other inadequacies. Let s be a state such that s has no predecessor but p and p propagates its contributions for i through s. Let s', s'', and s''' be states that have the same relationship with p', p'', and p''', respectively, that s has with p. Assume no states in the set $\{p, p', p,'' p'''\}$ are isocores while all states in the set $\{s, s', s,'' s'''\}$ are isocores. Assume there are no other isocores of s.

Assume *split_states* computes the successors of p' after computing the successors of p but before computing the successors of p'' or p'''. That is, when *compute_state* computes the lookaheads for s', the only existing isocore of s' is s. In this case, *compute_state* makes the locally optimum choice of merging the lookaheads for s' with s to form the state $s + s'$. When *split_states* later computes s'' and s''' from p'' and p''', no isocores are compatible resulting in the isocore set $\{s + s', s,'' s'''\}$. However, if *compute_state* were instead to make the locally suboptimum choice of immediately increasing $|\Sigma|$ by creating s' separate from s, it could later merge the lookaheads for s'' with s and the lookaheads for s''' with s' resulting in the smaller isocore set $\{s + s,'' s' + s'''\}$ and thus a smaller $|\Sigma|$.

Table 7
Grammar characteristics. 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.

Grammar	Version	$\lvert T\rvert$	$\lvert V\rvert$	$\lvert T \cup V\rvert$	$\lvert P\rvert$
Fig. 1		2	2	4	4
Fig. 2		3	4	7	9
Fig. 3		2	4	6	9
Fig. 4		2	3	5	7
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

Any algorithm that might be devised to search for a globally optimum merging of states cannot merely examine each isocore set in Σ in isolation. Such an algorithm must also consider the effect that merging any group of states has on merging their successor states and on merging their predecessor states. That is, such an algorithm must consider merging of isocoric lanes. Again, we have not discovered a practical (G, Δ) that would benefit significantly from such a complex algorithm, so we have not attempted to devise one.

3.9. *Canonical LR(1) via IELR(1)*

We have parameterized our IELR(1) implementation so that a user can request LALR(1), IELR(1), or canonical LR(1) tables. The modification needed to generate LALR(1) tables is trivial: skip phases 1, 2, 3, and 4. To generate canonical LR(1) tables:

- Phase 0: It is not necessary to compute the *goto_follows* table or any reduction lookaheads sets.
- Phase 1: It is not necessary to compute the *predecessors* table.
- Phase 2: Skip entirely.
- Phase 3: Propagate all lookaheads, and use the canonical LR(1) state compatibility test. That is:
  1. In Definition 3.38, change condition 2 to: $\forall j : 1 \leq j \leq |C|$, $lookahead\_set\_filters(s)[j] = T'$.
  2. Adjust Definition 3.43 so that $is\_compatible(s, K)$ is true iff $\neg lookaheads\_recomputed[s] \lor K = item\_lookahead\_sets[s]$.

  Because of the canonical LR(1) state compatibility test, *new_lookaheads* can never be true in Definition 3.46. Thus, *merge_lookaheads* never recursively invokes *compute_state*. We find this is a nice point to insert a run-time assertion check.
- Phase 4: Perform either of the following:
  1. Perform IELR(1) phase 4 without modification. That is, repeat phase 0 step 2 as when generating IELR(1) tables. While this approach has the advantage of reusing existing code, it is less efficient than the next approach.
  2. Instead of computing *goto_follows* and then reduction lookahead sets, use the lookahead sets already computed in phase 3. That is, $\forall s : 1 \leq s \leq |\Sigma|$, given that $\Sigma[s] = (C, A_t, A_r)$, and, for every reduce action $r \in A_r$, given that $r = (K, (\ell \rightarrow \varrho))$:
     (a) If $\varrho \neq \epsilon$, then set $K = \{t \in T' : \exists k : C[k] = ((\ell \rightarrow \varrho), |\varrho| + 1) \land t \in item\_lookahead\_sets[s][k]\}$.
     (b) If $\varrho = \epsilon$, set $K = compute\_goto\_follow\_set(s, \ell)$.

     However, our implementation does require that *from_state* and *to_state* be recomputed because Bison's table compression phase depends on them.
- Phase 5: Perform without modification.
- Phase 6: Perform without modification.

4. Results of using IELR(1)

In this section, we compare the Bison LALR(1) implementation with our IELR(1) implementation using nine LR(1) parser specifications as case studies. In Section 4.1, we describe the case studies in detail. In Section 4.2, we compare the parser tables that the implementations generate for each of the case studies. In Section 4.3, we compare the performance of the implementations in terms of time and space.

Table 8
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) via IELR(1) implementation generate for our case studies. We report the number of states and the number of conflicts left unresolved by the user. For canonical LR(1) and IELR(1), we show adjustments to account for such unresolved conflicts that are perfectly duplicated among isocores.

Grammar	States			S/R			R/R		
	LA	IE	Canon	LA	IE	Canon	LA	IE	Canon
Fig. 1	10	12	12	0	0–0	0–0	0	0–0	0–0
no prec/assoc	11	11	13	1	1–0	1–0	0	0–0	0–0
Fig. 2	19	21	21	0	0–0	0–0	2	1–0	1–0
Fig. 3	19	21	21	0	0–0	0–0	1	2–0	2–0
Fig. 4	15	16	16	0	0–0	0–0	2	1–0	1–0
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

4.1. Case studies

Table 7 characterizes the grammar, G, of each of our case studies. For some case studies, the grammar is coupled with precedence and associativity declarations to form the user portion of a dominant contribution function, Δ. Bison's default mechanisms for resolving conflicts, as described in Section 2.4, complete Δ and thus the LR(1) parser specification (G, Δ).

The grammars of our first four case studies are the example grammars of Figs. 1–4. With the grammar of Fig. 1, we include the declaration of a as left-associative.

Our next 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) [2,11]. 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 [4,14]. 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 [3].

The original parser specifications for our Gawk, Gpic, C, and Java case studies contain more code than just the grammars and the precedence and associativity declarations required for our comparison between LALR(1) and IELR(1). In some cases, Bison requires so much memory to process such extraneous code that performance differences between the LALR(1) and IELR(1) implementations are difficult to discern. Thus, the results we report are based on versions of these parser specifications from which we have removed the extraneous code.

The latest version of the C++ programming language is C++ 2003. Annex A of the C++ 2003 specification presents a formal C++ grammar [10]. 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 with tokens, and (2) removed all productions that only those nonterminals depend upon.

4.2. LALR(1) vs. IELR(1) parser tables

Table 8 describes the parser tables that the Bison LALR(1) implementation, our IELR(1) implementation, and our canonical LR(1) via IELR(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 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 conflicts are ultimately resolved by user declarations.

When IELR(1) splits an LALR(1) state into isocores, conflicts in the LALR(1) state might be completely eliminated. For example, as Table 8 shows, IELR(1) eliminates a R/R conflict for each of the Figs. 2 and 4 case studies. In some cases, different mutated versions of a conflict may end up in different isocores instead. For example, for the Fig. 3 case study, LALR(1) identifies only 1 R/R conflict, but IELR(1) discovers that the parser specification is actually more complex: when that conflict

Table 9
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 isocores.

Grammar	Actions		States		Tokens	
	IE	Canon	IE	Canon	IE	Canon
Fig. 1	1–0	1–0	1–0	1–0	1	1
Fig. 2	2–0	2–0	2–0	2–0	2	2
Fig. 3	1–0	1–0	1–0	1–0	1	1
Fig. 4	1–0	1–0	1–0	1–0	1	1
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

is split into 2 conflicts in different isocores, they have different dominant contributions. Finally, in other cases, a conflict might be perfectly duplicated among several isocores. 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) unresolved conflict count in Table 8, we subtract all but one copy of each unresolved conflict that is perfectly duplicated among isocores.

Table 9 describes the parser actions that are corrected in the parser tables by switching from LALR(1) to IELR(1). For the Fig. 1 case study, the LALR(1) and IELR(1) parser tables generated by Bison are similar to the LALR(1) and canonical LR(1) parser tables shown in Table 1.[8] Thus, when a is declared left-associative, LALR(1) accepts only 2 input sentences, but IELR(1) accepts the same set of 3 input sentences that canonical LR(1) accepts. Similarly, the LALR(1) and IELR(1) parser tables generated by Bison for the Figs. 2–4 case studies are similar to the LALR(1) and canonical LR(1) parser tables discussed in Section 2, and they accept the same sentences.

For the Gawk and Gpic case studies, every corrected action originates from a mysterious invasive S/R conflict from the LALR(1) parser tables. This is not surprising. As discussed in Section 2.4, when using an LALR(1) parser generator like Yacc or Bison, the parser specification developer usually must restructure a grammar to eliminate warnings about multiple reduce contributions in a conflict. As a result, well evolved parser specifications using such parser generators are less likely to generate conflicts containing multiple reduce contributions. Thus, by Observation 2.25, their mysterious conflicts are most likely to be invasive S/R conflicts.

The Gawk 3.1.0 parser specification turns out to be flawed, so not even canonical LR(1) can actually generate a parser that reflects the SUSv3 specification of AWK. Gawk 3.1.2 and future versions fix this flaw in such a way that LALR(1) and IELR(1) then manage to generate identical parsers.

After exploring the Gpic parser specification's source comments, we conclude that IELR(1) corrects a bug in a Gpic feature designed by the author of Gpic's parser specification. However, we also contacted the current Gpic developers for their opinion. They seem to have been previously unaware that the affected feature even exists, and some members expressed an interest in seeing the feature removed regardless of whether the bug could be fixed. For the full discussion, see the Groff mailing list archives [5].

We have confirmed that Menhir is unable to recognize the need to split off any corrected isocore for the Gawk or Gpic case study and so leaves the incorrect actions. In this way, LALR(1) and Menhir fail to generate parsers that accurately fulfill these LR(1) parser specifications, but IELR(1) is successful.

Our canonical LR(1) via IELR(1) results lead to an interesting insight. As has 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 [13]. The state counts in Table 8 are consistent with this observation. Because of the increased splitting of states, the number of conflicts that are perfectly duplicated among isocores 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 every one of our case studies, every new conflict is a perfect duplicate, so the adjusted conflict counts are the same as for IELR(1).

Also because of the increased splitting of states in canonical LR(1), the action correction counts shown in Table 9 are often higher than for IELR(1). However, the number of tokens on which parser actions are corrected cannot change because every new action correction must be a perfect duplicate of an action correction from IELR(1). That is, every viable prefix that encounters an action correction in the canonical LR(1) parser tables must also encounter that same action correction in the

[8] The number of states is different because of unreachable state removal.

Table 10
Parser tables computation. This table describes the performance of the Bison LALR(1) implementation and our IELR(1) implementation for each of our case studies. The real run time measures the full run time for Bison. However, we report peak memory usage only for the duration of the LALR(1) or IELR(1) algorithm. We also report the total number of inadequacy annotations attached to the LALR(1) states during phase 2 of IELR(1).

Grammar	Real run time (s)		Peak mem usage (KB)		Annotations
	LA	IE	LA	IE	
Fig. 1	0.035	0.035	57	60	1
Fig. 2	0.035	0.035	58	61	4
Fig. 3	0.035	0.035	56	61	2
Fig. 4	0.035	0.035	58	61	2
Gawk	0.058	0.076	100	250	2537
Gpic	0.108	0.153	100	440	6123
C	0.107	0.139	210	620	6573
Java	0.134	0.237	375	900	4636
C++	0.069	0.248	410	950	5346

IELR(1) parser tables. This must be true because canonical LR(1) parsers and IELR(1) parsers generate exactly the same parse trees and thus accept exactly the same language.

4.3. *LALR(1) vs. IELR(1) performance*

In Table 10, we report the performance of the Bison LALR(1) implementation and our IELR(1) implementation for each of our case studies.[9] We performed these measurements on a Toshiba Tecra M4-S435 with an Intel Pentium M Processor 740 [1.73 GHz, 2 MB L2, 533 MHz FSB] with 1024 MB DDR2 SDRAM running the Slackware 10.2.0 distribution of GNU/Linux with Linux kernel version 2.6.13. Using GNU Bash's (version 3.0.16) built-in time command[10] we measured the full real run time of Bison, and we report the average for 3 trial runs. We estimated peak memory usage during the execution of the LALR(1) and IELR(1) algorithms based on diagrams generated by Massif from Valgrind 3.2.3 [9].

5. Related work

The need to improve the state of the art in grammar-dependent software engineering has been discussed recently in the literature [20,23,33,34]. Many attempts to develop minimal LR algorithms have been made [6,8,25–28,30,31]. Menhir [7] is an implementation of an algorithm described by David Pager [26]. It is the most robust minimal LR(1) implementation we have discovered available, but it is not always able to generate parser tables with the full power of canonical LR(1) if the given grammar is non-LR(1) coupled with a specification for resolving conflicts. We previously introduced IELR(1) as a new minimal LR(1) algorithm that does not suffer from this deficiency [15], but we have not previously defined the algorithm in detail.

6. Future work

There are many facets of IELR(1) remaining to be explored. Most importantly, while the results in this paper suggest that IELR(1) can correctbugs in parsers whose specifications are already well evolved with LALR(1), we are especially interested in how IELR(1) facilitates the development of new parser specifications and new formal languages. Thus, we have contributed our IELR(1) implementation to the Bison project and plan to collect feedback from the Bison community. IELR(1) is currently scheduled to appear in Bison 2.5.

In Section 3, we discussed a number of efficiency concepts for IELR(1): split-stable dominant contributions, merge-stable dominant contribution functions, and suboptimum state merging. Results gathered from further exploration of these concepts might help us to improve our IELR(1) implementation and might aid developers of alternative implementations of IELR(1).

We are also interested in generalizing IELR(1) to IELR(k) for $k \geq 1$. Because the IELR(1) algorithm employs LALR(1), IELR(k) should employ LALR(k). We also suspect that the modifications we described in Section 3.9 for generating canonical LR(1) parser tables via IELR(1) are also appropriate for generating canonical LR(k) parser tables via IELR(k).

Because of the current popularity of GLR (Generalized LR) parsing [32,35], it would be useful to explore the benefit of coupling IELR(1) with GLR. A GLR parser can employ parser tables generated by any algorithm in the LR family. Upon encountering an unresolved conflict in those parser tables during a parse, the GLR parser branches and explores all parses to which the conflicting parser actions lead. Thus, for any context-free grammar, if no conflicts in the parser tables are resolved,

[9] The performance is slightly different than we previously reported [15] as we use a newer build of Bison in this paper.
[10] Previously, we mistakenly reported using GNU time instead [15].

then all parser actions remain viable, and the GLR parser is able to construct any parse tree specified by the grammar. Even if the parser tables are generated by LALR(1), there is no chance of mysterious behavior because the isocores in any merge are never forced to select potentially different dominant contributions for any conflict. Instead, the isocores always agree that all conflicting actions should be performed.

If a grammar is ambiguous, the GLR parser constructs multiple parse trees for some input sentences. In some cases, it is possible to eliminate some of these parse trees statically simply by resolving the associated conflicts in favour of the desired parse trees. Unfortunately, during LALR(1) parser table generation, the isocores in any merge can then disagree on which contribution to a conflict is dominant. Upon encountering such a resolved conflict, the GLR parser does not branch but behaves in exactly the same mysterious manner as the deterministic LALR(1) parser. In general, when any conflicts in the parser tables used by GLR need to be resolved statically, we conclude that those parser tables should be generated by IELR(1). One potential improvement to the IELR(1) algorithm presented in this paper would be to ignore unresolved conflicts when annotating states in phase 2 because, again, unresolved conflicts cannot cause mysterious behavior in a GLR parser.

7. Concluding remarks

We are 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 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. Moreover, the Gpic case study shows that such incorrect actions can create actual bugs relative to the intended design of a real-world LALR(1)-generated parser. Pager's algorithm and Menhir do not address the incorrect actions in either of these case studies. IELR(1) corrects them for both.

Because of the maturity of Gpic, we are not surprised that the only bug IELR(1) corrects for Gpic belongs to an obscure feature. Otherwise, we would have expected this bug to have been discovered by now and resolved by other means, such as grammar restructuring or an ad hoc solution. However, the obscurity of this feature does call for further investigation into the general nature of the bugs that IELR(1) corrects for practical parsers.

Because of the maturity of the GCC project, we are not surprised that its C and Java parser specifications suffer from no incorrect parser actions that result from a misuse of the LALR(1) algorithm. Nevertheless, these case studies demonstrate IELR(1)'s ability to recognize when LALR(1) parser tables are sufficient without unnecessarily splitting its states into additional states.

The importance of IELR(1) for our Gawk and Gpic case studies versus our C and Java case studies also suggests that IELR(1) might typically be more relevant to the developer of a DSL (domain-specific language) parser specification rather than to the developer of a GPL (general-purpose programming language) parser specification. This conclusion seems plausible given that the developer of a DSL parser specification is often a domain expert but not a language development expert [24]. By employing IELR(1) instead of LALR(1), Pager's algorithm, or Menhir, a DSL parser specification developer need not struggle with grammar restructuring, ad hoc solutions, and subsequent maintenance in order to manually eliminate mysterious conflicts.

The performance measurements from our case studies show that IELR(1) is feasible for generating minimal LR(1) parsers for sophisticated real-world LR(1) parser specifications. Specifically, for each of our case studies when using the IELR(1) algorithm, Bison did not run longer than 0.3 s, and the IELR(1) algorithm never required as much as 1 MB of memory at any point in time.

Finally, our results demonstrate that canonical LR(1) would severely impede the development of an LR(1) parser specification regardless of the power of the computer hardware. For the practical grammars in our case studies, IELR(1) simplifies the process of debugging parser table conflicts by merging compatible canonical LR(1) states and thus reducing the number of conflicts by an order of magnitude without altering the accepted language.

References

[1] Bison, The GNU project, http://www.gnu.org/software/bison/.
[2] Gawk, The GNU project, http://www.gnu.org/software/gawk/.
[3] GCC, The GNU project, http://gcc.gnu.org/.
[4] Groff, The GNU project, http://www.gnu.org/software/groff/.
[5] [Groff] parsing a corner specification, Groff mailing list archives, http://lists.gnu.org/archive/html/groff/2007-08/msg00051.html.
[6] LRGen P.B. Mann, http://www.paulbmann.com/lrgen/.
[7] Menhir F. Pottier, Y. Régis-Gianas, http://caml.inria.fr/pub/ml-archives/caml-list/2005/12/a879494ee13a0389b692cf2b27da8254.en.html.
[8] The Honalee LR(k) Algorithm D.R. Tribble, http://david.tribble.com/text/honalee.html.
[9] Valgrind, http://www.valgrind.org/.
[10] International Standard, Programming Languages — C++, No. ISO/IEC 14882:2003(E), American National Standards Institute, 2003.
[11] Single UNIX Specification, Version 3, The IEEE and the Open Group, http://www.opengroup.org/bookstore/catalog/t041.htm (April 2004).
[12] Menhir release announcement F. Pottier, http://caml.inria.fr/pub/ml-archives/caml-list/2005/12/a879494ee13a0389b692cf2b27da8254.en.html (December 2005).
[13] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.
[14] R. Corderoy, troff.org, The text processor for typesetters, http://troff.org.
[15] J.E. Denny, B.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, ACM, New York, NY, USA, 2008.
[16] F. DeRemer, T. Pennello, Efficient computation of LALR(1) look-ahead sets, ACM Trans. Program. Languages Syst. 4 (4) (1982) 615–649.
[17] The GNU project, Bison, 2.4.1 ed., December 2008.

[18] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.
[19] S. Johnson, Yacc: Yet another compiler compiler, in: Holt, Rinehart, and Winston (Eds.), UNIX Programmer's Manual, vol. 2, New York, NY, USA, 1979, pp. 353–387.
[20] P. Klint, R. Lämmel, C. Verhoef, Toward an engineering discipline for grammarware, ACM Trans. Softw. Eng. Methodol. 14 (3) (2005) 331–380.
[21] P. Klint, E. Visser, Using filters for the disambiguation of context-free grammars, in: Proc. ASMICS Workshop on Parsing Theory, 1994.
[22] D.E. Knuth, On the translation of languages from left to right, Inform. Control 8 (6) (1965) 607–639.
[23] R. Lämmel, Grammar testing, in: FASE, 2001.
[24] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344.
[25] 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, ACM Press, New York, NY, USA, 1973.
[26] D. Pager, A practical general method for constructing LR(k) parsers, Acta Inform. 7 (3) (1977) 249–268.
[27] D. Pager, The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency, Inform. Sci. 12 (1) (1977) 19–42.
[28] P. Pepper, LR parsing = grammar transformation + LL parsing, making LR parsing more understandable and more efficient, Tech. Rep. 99-05, TU Berlin, 1999.
[29] S. Sippu, E. Soisalon-Soininen, Parsing Theory. Vol. 1: Languages and Parsing, Springer-Verlag New York, Inc., New York, NY, USA, 1988.
[30] D. Spector, Full LR(1) parser generation, SIGPLAN Not. 16 (8) (1981) 58–66.
[31] D. Spector, Efficient full LR(1) parser generation, SIGPLAN Not. 23 (12) (1988) 143–150.
[32] M. Tomita, Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems, Kluwer Academic Publishers, Norwell, MA, USA, 1985.
[33] M. van den Brand, E. Visser, Generation of formatters for context-free languages, ACM Trans. Softw. Eng. Methodol. 5 (1) (1996) 1–41.
[34] M.G.J. van den Brand, A. Sellink, C. Verhoef, Current parsing techniques in software renovation considered harmful, in: IWPC, Washington, DC, USA, 1998.
[35] E. Visser, Syntax definition for language prototyping, Ph.D. Thesis, University of Amsterdam, September 1997.