Efficient Computation of LALR(1) Look-Ahead Sets

FRANK DeREMER and THOMAS PENNELLO
University of California, Santa Cruz, and MetaWare™ Incorporated

Two relations that capture the essential structure of the problem of computing LALR(1) look-ahead sets are defined, and an efficient algorithm is presented to compute the sets in time linear in the size of the relations. In particular, for a PASCAL grammar, the algorithm performs fewer than 15 percent of the set unions performed by the popular compiler-compiler YACC.

When a grammar is not LALR(1), the relations, represented explicitly, provide for printing user-oriented error messages that specifically indicate *how* the look-ahead problem arose. In addition, certain loops in the digraphs induced by these relations indicate that the grammar is not LR(k) for any k.

Finally, an oft-discovered and used but *incorrect* look-ahead set algorithm is similarly based on two other relations defined for the first time here. The formal presentation of this algorithm should help prevent its rediscovery.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax*; D.3.4 [**Programming Languages**]: Processors—*translator writing systems and compiler generators*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*parsing*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Context-free grammar, Backus-Naur form, strongly connected component, LALR(1), LR(k), grammar debugging,

1. INTRODUCTION

Since the invention of LALR(1) grammars by DeRemer [9], LALR grammar analysis and parsing techniques have been popular as a component of translator writing systems and compiler-compilers. However, DeRemer did not describe how to compute the needed look-ahead sets. Instead, LaLonde was the first to present an algorithm [20]. Since then, LaLonde's algorithm has been published by Anderson et al. [6, pp. 21–22], who also presented their own algorithm [6, p. 21]; Aho and Ullman [3, p. 238] have published the one used in YACC [16].

The work reported herein was supported in part by the Hewlett-Packard Company.
Portions of this paper were presented at the First SIGPLAN Symposium on Compiler Construction, Denver, Colo., Aug. 1979, and appear in *SIGPLAN Notices* (ACM) *14*, 8 (Aug. 1979), pp. 176–187.
Authors' address: Computer and Information Sciences, University of California, Santa Cruz, CA 95064.

Kristensen and Madsen [19] have improved LaLonde's algorithm, extending the results to LALR(k).

Various others have tried their hand at designing such an algorithm, often with the result of implementing a particular subset of LALR(1) that we dub "not quite" LALR(1) or NQLALR(1) [7, 11, 23, 25]. Subsequently, Watt attempted to repair his original approach [24], as did Chaney with DeRemer's approach [5]. Neither of these later attempts was correct, although both were more complex and worked in more cases than did the NQLALR method.

None of the correct LALR(1) algorithms, except the one developed by Kristensen and Madsen [19, Sec. 6.2], have been nearly as efficient as their NQLALR(1) counterparts. Later we describe the oversimplification that results in the simple, efficient algorithms that are not quite right. The purpose of the current paper is to provide an algorithm that efficiently exploits the essential structure of the problem.

1.1 Preview

When a grammar is not LR(0), one or more of the LR(0) parser's states must be "inconsistent", having either a "read-reduce" or a "reduce-reduce" conflict, or both. In the former case the parser cannot decide whether to read the next symbol of the input or to reduce a phrase on the stack. In the latter case the confusion is between distinct reductions. Looking ahead at the first symbol of the input may resolve the conflict, and DeRemer defined a grammar to be LALR(1) when each inconsistent state q can be augmented with look-ahead sets that resolve the conflict and result in a correct, deterministic or "consistent" parser [9].

More precisely, for each inconsistent state q and possible reduction $A \rightarrow \omega$ in q, let the "look-ahead set for $A \rightarrow \omega$ in q" be denoted by LA(q, $A \rightarrow \omega$). When the parser is in state q and the symbol at the head of the input is in LA(q, $A \rightarrow \omega$), ω must be reduced to A. Thus the look-ahead sets in q must be mutually disjoint and not contain any of the symbols that could be read from q.

Watt [24] has defined LA(q, $A \rightarrow \omega$) as $\{t \in T \mid S \Rightarrow^+ \alpha A t z$ and $\alpha\omega$ accesses $q\}$, where T is the set of terminals in the grammar. Intuitively, when the parser is in state q and $\alpha\omega$ is on the stack, reduction of ω to A is appropriate exactly when the input begins with some terminal t that can follow αA in a rightmost sentential form. Our purpose here is to investigate the underlying structure in this definition and to show how to compute LA efficiently.

The problem can be decomposed into four separate computations. In reverse order of computation they are as follows: LA is computed from "Follow" sets of nonterminal transitions; Follow sets are computed from "Read" sets of nonterminal transitions; Read sets are computed from "Direct Read" sets; and Direct Read sets are computed by inspecting the LR(0) parser.

A relation **includes** on nonterminal transitions relating the Follow sets is defined, along with a relation **reads** relating the Read sets. The Read sets are initialized to the Direct Read sets by inspection of the parser. Then their values are completed by a graph traversal algorithm for finding "strongly connected components" (SCCs), adapted to compute unions of the sets appropriately as it searches the digraph induced by the **reads** relation. If a nontrivial SCC is found, the grammar in question is not LR(k) for any k. Next the Read sets are used as initial values for the Follow sets, which are completed by the SCC algorithm applied to the digraph of the **includes** relation. Again, if a nontrivial SCC is encountered having a nonempty Read set in it, (we conjecture that) the grammar is not LR(k) for any k. In any case, the LALR(1) look-ahead sets are simply unions of appropriate Follow sets.

We now define terminology, define LALR(1), give theorems relating to look-ahead set computation, present the algorithm, discuss oversimplifications, give statistics for some practical grammars, show how to generate debugging diagnostics for grammars that are not LALR(1), and present conclusions.

2. TERMINOLOGY

The notions of *symbol* and *string* of symbols are assumed here. A *vocabulary* V is a set of symbols. V^* denotes the set of all strings of symbols from V. V^+ denotes $V^* - \{\epsilon\}$, where ϵ is the *empty string.* The *length* of any string α is denoted by $|\alpha|$. The first symbol of a nonempty string α is denoted by *First* α; the string following is denoted by *Rest* α; the last symbol is denoted by *Last* α. As just illustrated, arguments to functions are not parenthesized when the intent is clear.

If R is a relation, R^* denotes the reflexive, transitive closure of R, and R^+ denotes the transitive closure. We write $X =_s F(X)$ to mean that X is the smallest set satisfying $X = F(X)$. $\bigcup\{S_1, \ldots, S_n\}$, where the S_i are sets, denotes $S_1 \cup \cdots \cup S_n$.

2.1 CFGs

A *context-free grammar* (CFG) is a quadruple $G = \langle T, N, S, P\rangle$, where T is a finite set of *terminal* symbols, N is a finite set of *nonterminal* symbols such that $T \cap N = \emptyset$, $S \in N$ is the *start* symbol, and P is a finite subset of $N \times V^*$, where $V = T \cup N$ and each member (A, ω) is called a *production,* written $A \rightarrow \omega$. A is called the *left part* and ω the *right part.* We require a production $S \rightarrow S'\perp$ for some $S' \in N$ and $\perp \in T$ such that $\perp$ and S appear in no other production.

The following (usual) conventions hold in this paper:

$$\begin{array}{ll} S, A, B, C, \ldots & \in N \\ X & \in V \\ t, a, b, c, \ldots & \in T \\ \ldots, x, y, z & \in T^* \\ \alpha, \beta, \gamma, \ldots & \in V^* \end{array}$$

The relation $\Rightarrow_r$ is pronounced "directly (right) produces" and is defined on V^* such that $\alpha A y \Rightarrow_r \alpha\omega y$ for all $\alpha \in V^*$, $y \in T^*$, and $A \rightarrow \omega \in P$. The r subscript is dropped hereafter since we always mean *right* produces. Both $\Rightarrow^*$ and $\Rightarrow^+$ are pronounced "produces". A *nullable* nonterminal is one that produces ϵ. If $S \Rightarrow^* \alpha$, then α is called a *sentential form*; if $\alpha \in T^*$, then it is called a *sentence.* The *language* $L(G)$ generated by G is the set of sentences, that is, $\{x \in T^* \mid S \Rightarrow^+ x\}$. All grammars here are assumed to be *reduced,* that is, $S \Rightarrow^+ \alpha A \beta$ and $A \Rightarrow^* y$ for all $A \in N$ and some $\alpha, \beta \in V^*$ and $y \in T^*$.

Let G be a CFG and $k \geq 0$. G is *LR(k)* iff $S \Rightarrow^* \alpha A y \Rightarrow \alpha\omega y$ implies that, if $S \Rightarrow^* \gamma \Rightarrow \alpha\omega y'$, then $\gamma = \alpha A y'$ for all $\alpha, \gamma \in V^*$ and $y, y' \in T^*$ such that $\text{First}_k(y') = \text{First}_k(y)$ [17]. Here $\text{First}_k(y)$ is the prefix of y of length k, or just y if $|y| < k$.

2.2 LR parsers

Next we introduce a formalization of an LR parser, that is, any one-symbol look-ahead parser, such as an SLR(1), LALR(1), or LR(1) parser [1]. The generalization to multisymbol look-ahead is easy, but not relevant here. Given some tabular representation of the "LR automaton" defined below and the general LR parsing algorithm to interpret those tables, we have an "LR parser". The particular states, transitions, and look-ahead sets are determined by the grammar in question and by the construction technique. For example, the LALR(1) technique produces an "LALR(1) automaton".

An *LR automaton* for a CFG $G = \langle T, N, S, P\rangle$ is a sextuple $\text{LRA}(G) = \langle X, V, P, \text{Start}, \text{Next}, \text{Reduce}\rangle$, where K is a finite set of *states*, V and P are as in G, $\text{Start} \in K$ is the *start* state, $\text{Next}\colon K \times V \to K$ is called the *transition function*, and $\text{Reduce}\colon K \times T \to 2^P$ is called the *reduce function*. Next may be a partial function. Nondeterministic or "inconsistent" LR automata are allowed; the LALR(1) condition of Section 3 excludes such cases. A *transition* is a member of $K \times V$; it is a *terminal transition* if it is in $K \times T$ and a *nonterminal transition* if it is in $K \times N$. The transition (q, X) is represented by $q \xrightarrow{X} p$, where $p = \text{Next}(q, X)$, or by $q \xrightarrow{X}$ when p is irrelevant, and we define *Accessing_symbol* $p = X$; each state has a unique accessing symbol, except Start, which has none.

In the diagrams in text, LR automata are represented by state diagrams in which states are connected by transitions. For each state q in which Reduce indicates possible reductions, the productions are listed.

A *path H* is a sequence of states $q_0, \ldots, q_n$ such that

$$q_0 \xrightarrow{X_1} q_1 \longrightarrow \cdots \longrightarrow q_{n-1} \xrightarrow{X_n} q_n.$$

We say that *H spells* $\alpha = X_1 \cdots X_n$ and define *Spelling* $H = \alpha$ and *Top* $H = q_n$. H is denoted by $q_0 \overset{\alpha}{-\cdots\to} q_n$, pronounced "$q_0$ goes to q_n under α". An alternative notation for H is $[q_0\colon\alpha]$, given the automaton or its state diagram. The concatenation of $[q\colon\alpha]$ and $[q'\colon\alpha']$, where $\text{Top}\ [q\colon\alpha] = q'$, is written $[q\colon\alpha][q'\colon\alpha']$ and denotes $[q\colon\alpha\alpha']$. $[\text{Start}\colon\alpha]$ can be abbreviated $[\alpha]$; thus $[\ ]$ denotes Start alone. We say that α *accesses* q if $\text{Top}\ [\alpha] = q$.

A *configuration* is a member of $K^+ \times T^+$; its first part is called the *state stack* and its second the *input*. The relation $\vdash$ on configurations is pronounced "directly moves to" and is the union of $\vdash_{\text{read}}$ and $\vdash_{A\to\omega}$, for all $A \to \omega \in P$. $\vdash_{\text{read}}$ is pronounced "reads to": $[q\colon\alpha]tz \vdash_{\text{read}} [q\colon\alpha t]z$ if $\text{Next}(\text{Top}\ [q\colon\alpha], t)$ is defined. $\vdash_{A\to\omega}$ is pronounced "reduces ω to A in moving to": $[q\colon\alpha\omega]tz \vdash_{A\to\omega} [q\colon\alpha A]tz$ if $A \to \omega \in \text{Reduce}(\text{Top}\ [q\colon\alpha\omega], t)$ (and if $[q\colon\alpha A]$ is a path; but this additional constraint will always hold in the LR automata considered here). $\vdash^*$ and $\vdash^+$ are pronounced "moves to". The *language* L(LRA(G)) parsed by LRA(G) must be identically L(G) and is $\{z \in T^* \mid [\ ]z \vdash^+ [S'] \perp\}$.

A triple $(A, \alpha, \beta) \in N \times V^* \times V^*$ is called an *item*, written $A \rightarrow \alpha \cdot \beta$ if $A \rightarrow \alpha\beta$ is a production; if $\beta = \epsilon$, it is a *final item*. A set of items is called a (*parse*) *table*. The set of *LR*(0) *parse tables* PT(G) *for* a CFG G is

$$\begin{aligned}\mathrm{PT}(G) =_s {} & \{\text{Closure } \{S \rightarrow \cdot S' \perp\}\} \cup \\ & \{\text{Closure } IS \mid IS \in \text{Successors } IS' \text{ for } IS' \in \mathrm{PT}(G)\}\end{aligned}$$

where

$$\begin{aligned}\text{Closure } IS &= IS \cup \{A \rightarrow \cdot\omega \mid B \rightarrow \alpha \cdot A\beta \in IS \text{ and } A \rightarrow \omega \in P\}\\ \text{Successors } IS &= \{\text{Nucleus}(IS, X) \mid X \in V\}\\ \text{Nucleus}(IS, X) &= \{A \rightarrow \alpha X \cdot \beta \mid A \rightarrow \alpha \cdot X\beta \in IS\}.\end{aligned}$$

An *LR*(0) *automaton for* a CFG G is an LR automaton LRA(G) such that there exists a bijective function $F: K \rightarrow \mathrm{PT}(G) - \{\emptyset\}$ where

$$\text{Start} = F^{-1}(\text{Closure } \{S \rightarrow \cdot S' \perp\})$$

and for all $t \in T$, $X \in V$,

$$\begin{aligned}\text{Next}(q, X) &= F^{-1}(\text{Closure}(\text{Nucleus}(F\,q, X)))\\ \text{Reduce}(q, t) &= \{A \rightarrow \omega \mid A \rightarrow \omega \cdot \in F\,q\}.\end{aligned}$$

F simply establishes a one-to-one correspondence between tables (except $\emptyset$, the "trap table") and states and thus is an isomorphism between the parse tables and the parser. Hence, hereafter we elide all occurrences of F and F^{-1}, since context always determines whether q denotes a state or its corresponding parse table. The "LR(0)-ness" of the automaton is evident in that the definition of Reduce(q, t) is independent of t. Hereafter, "parser" is often used rather than automaton.

It is well known that the LR(0) automaton A is a correct parser for G, that is, L(A) = L(G); however, in general it is nondeterministic, due to the existence of "inconsistent" states. A state q is *inconsistent* iff there exists a $t \in T$ such that Next(q, t) is defined and Reduce(q, t) $\neq \emptyset$ (*read-reduce conflict*), or $|\text{Reduce}(q, t)| > 1$ (*reduce-reduce conflict*), or both.

A shorthand notation is useful for a certain sequence of moves:

$$[\alpha \mid]yz \vdash^* [\alpha \mid \beta]z \quad \text{iff} \quad (\text{Top } [\alpha], yz) \vdash^* ([\text{Top } [\alpha] : \beta], z).$$

This captures the notion that the parser reads y and reduces it to β, possibly including reductions on the empty string preceding y. The vertical bar is needed because $[\alpha]yz \vdash^* [\alpha\beta]z$ does *not* necessarily imply that y was reduced to β. For example, consider $[\gamma A]txz \vdash_{\text{read}} [\gamma At]xz \vdash_{A \rightarrow At} [\gamma A]xz \vdash^* [\gamma A\beta]z$, where tx was *not* reduced to β (here $y = tx$ and $\alpha = \gamma A$).

2.3 Graphs

A *directed graph* or *digraph* is a pair (V', E) where V' is a set of *vertices* and E is a subset of $V' \times V'$, each member of which is called an *edge*. In this paper V' is always finite. A *digraph-path*, or simply a *path* when the context is clear, is a sequence of vertices $v_1, \ldots, v_n$, $n > 1$, such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$; we say there is a path *from* v_1 *to* v_n. A *root* is a vertex having no paths to it. A *directed acyclic graph* (DAG) is a digraph in which there is no path from any vertex back to itself. A *forest* is a DAG in which there is at most one path to each vertex from a given vertex. A *tree* is a forest having exactly one root.

3. LALR LOOK-AHEAD SETS

"LALR(1) parser" can be defined by refining the definition of Reduce for an LR(0) parser. Intuitively, Reduce(q, t) should contain $A \rightarrow \omega$ only if there exist sentential $\alpha A t z$ and $\alpha \omega t z$ such that $\alpha\omega$ accesses q. The definition of LALR(1) parser is given after the definition of "look-ahead symbols" (LA):

Definition. For an LR(0) parser,

$$LA(q, A \rightarrow \omega) = \{t \in T \mid [\alpha\omega]tz \vdash_{A \rightarrow \omega} [\alpha A]tz \vdash^* [S' \perp], \text{ and } \alpha\omega \text{ access } q\}. \quad \square$$

Definition. An *LALR*(1) *parser for* a CFG G is like G's LR(0) parser, except that

$$\text{Reduce}(q, t) = \{A \rightarrow \omega \mid t \in \text{LA}(q, A \rightarrow \omega)\}. \quad \square$$

Definition. A CFG is *LALR*(1) iff its LALR(1) parser has no inconsistent states. □

The latter defines LALR(1) grammar in terms of LALR(1) parser; a grammar is LALR(1) iff its LALR(1) parser is deterministic. It is desirable to have a definition of LALR(1) grammar that does not involve the parser, but we know of no reasonable way to do this. We do, however, come a little closer in the following theorem, which Watt gave as a definition [24].

THEOREM. $LA(q, A \rightarrow \omega) = \{t \in T \mid S \Rightarrow^+ \alpha A t z$ *and* $\alpha\omega$ *access* $q\}$.

The proof depends essentially on the correctness of the LR(0) parser, that is, that the moves faithfully reflect the derivation; we do not prove this here.

The primary goal here is to show how to compute the LA sets. To do so we focus attention on nonterminal transitions and define "follow sets" for them.

Definition. For an LR(0) parser with nonterminal transition (p, A),

$$Follow(p, A) = \{t \in T \mid [\alpha A]tz \vdash^* [S'\perp] \text{ and } \alpha \text{ accesses } p\}. \quad \square$$

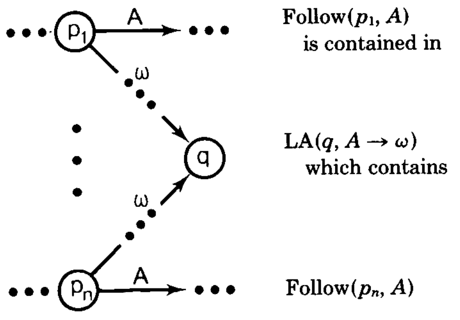
These are just the terminal symbols that can follow A in a sentential form whose prefix α, preceding A, accesses state p, given the correctness of the LR(0) parser. Stated in terms of derivations, Follow$(p, A) = \{t \in T \mid S \Rightarrow^+ \alpha A t z$ and α accesses $p\}$. Thus it is easy to see that each LA set is just the union of some related Follow sets.

THEOREM UNION

$$LA(q, A \rightarrow \omega) = \bigcup\{Follow(p, A) \mid (p, A) \text{ is a transition and } p \xrightarrow{\omega} q\}.$$

(Proof is given in the Appendix.) That is, the LA set for production $A \rightarrow \omega$ in state q is the union of the Follow sets for the A-transitions whose source state p has a path spelling ω that terminates at q. Intuitively, when the parser reduces ω to A in state q, each such p is a possible top state after ω is popped; then the parser must read A in p, all with some terminal t, the first of the input.

Diagrammatically,



The parser should reduce $A \to \omega$ when in state q if the next input symbol t is in any of the Follow(p_i, A) ($1 \le i \le n$), that is, if t can follow A in any of the left contexts "remembered" by states p_1 through p_n. The transitions (p_i, A) of concern can be captured via the following relation:

Definition [24]. $(q, A \to \omega)$ ***lookback*** (p, A) iff $p \overset{\omega}{\text{—}\cdots\text{→}} q$. □

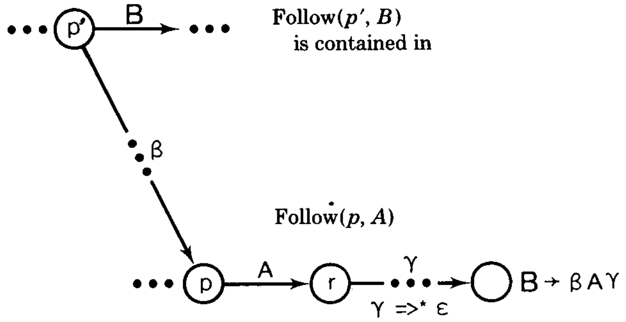
Thus LA$(q, A \to \omega) = \bigcup\{\text{Follow}(p, A) \mid (q, A \to \omega)$ **lookback** $(p, A)\}$.

The follow sets are, in turn, related to each other. In particular,

THEOREM. *Follow$(p', B) \subseteq$ Follow(p, A) if*

$$B \to \beta A \gamma, \quad \gamma \Rightarrow^* \epsilon, \quad \textit{and} \quad p' \overset{\beta}{\text{—}\cdots\text{→}} p.$$

Diagrammatically,



This is easy to see since, given some string α accessing p', we have $\alpha\beta$ accessing p, and in an appropriate right context, $\alpha\beta A$ can be reduced first to $\alpha\beta A\gamma$ via $\gamma \Rightarrow^* \epsilon$ and then to αB. Thus those symbols that can follow B in the left context remembered by p' can also follow A in the left context remembered by p. The above inclusion can be captured via a relation on nonterminal transitions.

Definition. (p, A) ***includes*** (p', B) iff

$$B \to \beta A \gamma, \quad \gamma \Rightarrow^* \epsilon, \quad \text{and} \quad p' \overset{\beta}{\text{—}\cdots\text{→}} p. \qquad \square$$

Thus, Follow$(p', B) \subseteq$ Follow(p, A) if (p, A) **includes** (p', B).

Next, observe that the symbols labeling terminal transitions "following" a nonterminal transition (p, A) are obviously in Follow(p, a).

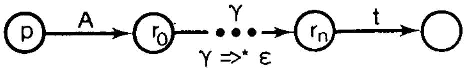
THEOREM. *Read$(p, A) \subseteq$ Follow(p, A).*

where

Definition. For an LR(0) parser with nonterminal transition (p, A),

$$Read(p, A) = \{t \in T \mid \alpha \text{ accesses } p \text{ and } [\alpha A \mid]tz \vdash^* [\alpha A \mid \gamma]tz \vdash_{\text{read}} [\alpha A \gamma t]z \vdash^* [S'\bot]\}. \quad \square$$

Diagrammatically,



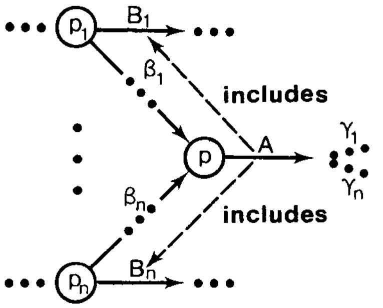
Read(p, A) is the set of terminals that can be read before any phrase including A is reduced. The definition is complicated by the possibility of numerous reductions of the empty string and nonterminals generating it, namely, $\gamma \Rightarrow^* \epsilon$, before the read move finally occurs. Read(p, A) is just the "direct read symbols" (DR, below) from r_0, if $\gamma = \epsilon$ in the above diagram (i.e., $n = 0$).

All contributions to the Follow sets have now been considered, and the results may be summarized in the following theorem, of which the previous two theorems are corollaries:

THEOREM UP

*Follow$(p, A) =_s$ Read$(p, A) \cup \bigcup\{$Follow$(p', B) \mid (p, A)$ **includes** $(p', B)\}$.*

(Proof is given in the Appendix.) That is, Follow(p, A) is exactly (1) the set of terminals that can be read, via the first read, after reducing a phrase to A in the left context "remembered" by p, before any phrase including A is reduced, together with (2) the Follow sets of the nonterminals to which some phrase containing A, followed at most by some nullable nonterminals, can be reduced before reading another symbol, each such nonterminal in the appropriate left context, of course. Diagrammatically, for n such nonterminals, B_1 through B_n,



(B_i, B_j; β_i, β_j; γ_i, γ_j are not necessarily distinct.) The dashed arrows indicate the **includes** relation. In a similar manner the Read sets can be decomposed.

THEOREM ACROSS

$$\mathit{Read}(p, A) =_s \mathit{DR}(p, A) \cup \bigcup\{\mathit{Read}(r, C) \mid (p, A) \textbf{\textit{ reads }} (r, C)\}.$$

(Proof is given in the Appendix.)

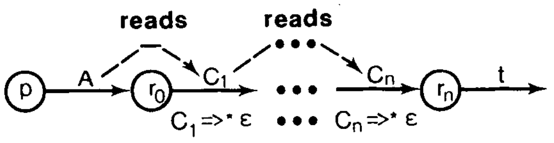
where

Definition. For an LR(0) parser with nonterminal transition (p, A),

$$\mathit{DR}(p, A) = \{t \in T \mid p \xrightarrow{A} r \xrightarrow{t} \}. \qquad \square$$

Definition. (p, A) ***reads*** (r, C) iff $p \xrightarrow{A} r \xrightarrow{C}$ and $C \Rightarrow^* \epsilon$. $\square$

The "direct read symbols" (DR) are simply those that label terminal transitions of the successor state r of the (p, A) transition. "Indirect read symbols" arise when nullable nonterminals can follow. Diagrammatically,



Here, (p, A) **reads** (r_0, C_1) **reads** $\cdots$ **reads** (r_{n-1}, C_n); thus, $\mathrm{DR}(r_{n-1}, C_n) \subseteq \mathrm{Read}(p, A)$ so that $t \in \mathrm{Read}(p, A)$.

In summary, to compute the LA sets, the Follow sets are needed, for which the Read sets are needed, for which the DR sets are needed. The Follow sets are interrelated as described by the **includes** relation, as are the Read sets by the **reads** relation. In the next section the computation of these sets is described by carrying information along the edges of the graphs induced by the **reads** and **includes** relations. A graph traversal algorithm is used to determine an optimum order for doing so, and simultaneously, to compute the sets.

4. GRAPH ALGORITHMS FOR LALR COMPUTATION

Theorems Up and Across relate Follow, Read, and DR in such a way that an appropriate graph traversal algorithm can be applied first to compute Read from DR, and then to compute Follow from Read. Note that DR is already directly available in the LR(0) parser. The two graphs of interest are those induced by the relations **reads** and **includes**, respectively. However, let us consider a more general problem first and then return to this specific LALR application.

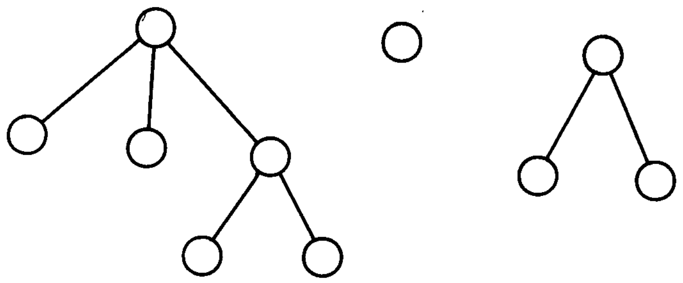
4.1 General Case

Let R be a relation on a set X. Let F be a set-valued function such that for all $x \in X$,

$$F\,x =_s F'\,x \cup \bigcup\{F\,y \mid xRy\} \tag{4.1}$$

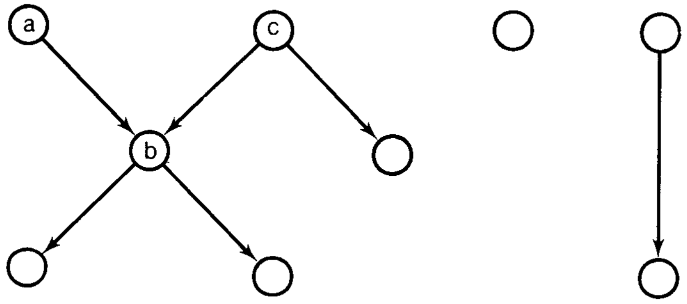
where $F'\,x$ is given for all $x \in X$. Let $G = (X, R)$ be the digraph induced by R, that is, G has vertex set X and (x, y) is an edge iff xRy. Then $F\,x$ can be efficiently computed from $F'\,x$ by traversing G appropriately, as we shall consider first when G is a forest, then a DAG, and finally a general digraph.

Suppose G is a forest, such as



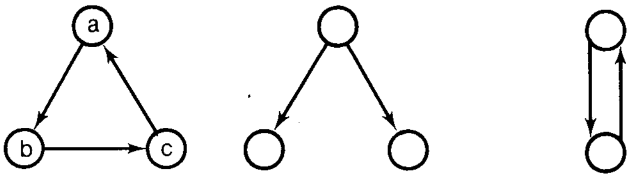
Each *leaf* x has no y such that xRy; thus $F\ x$ is simply $F'\ x$. Each *nonleaf* x is a *parent* of one or more children; a *child* of x is a vertex y such that xRy, and $F\ x$ in this case is $F'\ x$ unioned with the F-values of the children of x. Thus, a standard, recursive, tree-traversal algorithm T can be used to compute F in this case by carrying information from the leaves to the roots.

Now consider a DAG, such as



Vertices a and c "share" the child b. Algorithm T correctly computes F for all vertices, but it traverses b and its subtrees twice, once as a subtask of computing $F\ a$ and again as a subtask of $F\ c$. An algorithm D, based on T, can avoid such recomputation by marking each vertex on first encounter and never retraversing marked vertices.

Finally, consider the general case of a digraph with cycles and possibly no roots, for example,



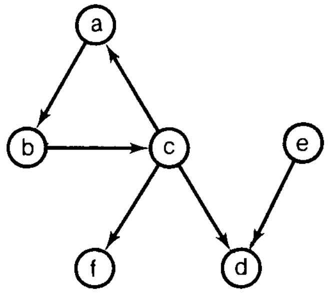
(4.2)

If algorithm D were to start its traversal at vertex a, it would visit a then b then c, incorrectly computing $F\ c = F'\ c \cup F'\ a$, although it would correctly compute $F\ a$ and $F\ b$. Worse yet, algorithm T would loop forever. Note that by the definition of F, $F\ a \subseteq F\ b \subseteq F\ c \subseteq F\ a$, so $F\ a = F\ b = F\ c = F'\ a \cup F'\ b \cup F'\ c$. A second trip around the cycle would solve the problem for simple loops such as the above, but in the general case there might be loops inside loops or shortcut paths between loops, so that a "second trip" is not so easy to define.

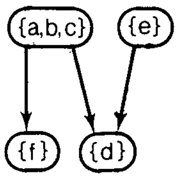
The generalization of such cycles is a *strongly connected component* (SCC), a maximal subset V'' of the vertices of a graph $G = (V', E)$ such that for all distinct $v_i, v_j \in V''$, there is a path from v_i to v_j (thus, vice versa). Call an SCC *trivial* iff it is a single vertex with no path to itself (recall that digraph paths must be of length one or more). This notion has importance in later theorems.

It is easy to see, as illustrated above, that if $V'' = \{x_1, \ldots, x_n\}$ then $F\,x_1 = \cdots = F\,x_n$ and this common value contains $F'\,x_1 \cup \cdots \cup F'\,x_n$. Thus, each SCC could be collapsed to a single vertex x'' with $F'\,x'' = F'\,x_1 \cup \cdots \cup F'\,x_n$, to form a new digraph G'. Then algorithm D could be applied and the results could be distributed from G' to G, for example, $F\,x_1 = \cdots F\,x_n = F\,x''$. It is well known that if all such SCCs were collapsed, G' would have no cycles, that is, it would be a DAG. Thus, algorithm D will work correctly on G'.

Note that each vertex v in G not involved in any cycle will be the only member of a trivial SCC and will thus appear unchanged in G'. For example, the digraph



reduces to the DAG

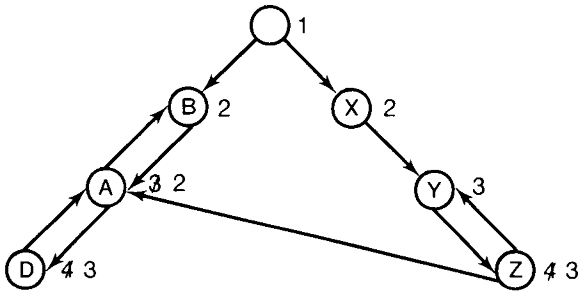


In practice the collapsed graph G' need not actually be constructed. Rather the computation of F can be effected while finding the SCCs. The following algorithm, Digraph, is an adaptation of one given by Eve and Kurki-Suonio [14]. We first modified the exposition of the algorithm to improve its readability and understandability. Then we added the three statements set off to the right to compute F. Further explication is given below the algorithm.

```
algorithm Digraph:
  input    $R$, a relation on $X$, and $F'$, a function from $X$ to sets.
  output   $F$, a function from $X$ to sets, such that $F\,x$ satisfies (4.1).
  let      $S$ be an initially empty stack of elements of $X$
  let      $N$ be an array of zeros indexed by elements of $X$
  for      $x \in X$ such that $N\,x = 0$ do call Traverse $x$ od
  where    recursive Traverse $x$ =                         # | Vertices |
    call    Push $x$ on $S$
    con     $d$: Depth of $S$
    assign  $N\,x \leftarrow d$                                ; $F\,x \leftarrow F'\,x$
    for     $y \in X$   such that $xRy$                      # | Edges |
      do    if       $N\,y = 0$ then call Traverse $y$ fi
            assign   $N\,x \leftarrow \mathrm{Min}(N\,x, N\,y)$          ; $F\,x \leftarrow F\,x \cup F\,y$
      od
    if      $N\,x = d$
      then repeat assign $N$(Top of $S$) $\leftarrow$ Infinity     ; $F$(Top of $S$) $\leftarrow F\,x$
                  until (Pop of $S$) = $x$                   # | Vertices |
      fi
    end Traverse
  end Digraph
```

The array N serves three purposes: (1) recording unmarked vertices ($N\,x = 0$), (2) associating a positive integer value with vertices that are under active consideration ($0 < N\,x <$ Infinity), and (3) marking with Infinity vertices whose SCCs have already been found. Each unmarked vertex is Traversed. Traverse pushes its argument on the stack S, marks it (via N) with its depth on S, and Traverses its "subtrees". If ever an edge is encountered from some descendent D to an ancestor A already on the stack (see diagram (4.3) below), then there exists a path from A to D to A and hence at least A and D and the intervening nodes on the stack belong to an SCC. The N-value of D is minimized to that of A to prevent D from being popped as the recursion unwinds. (In diagram (4.3) there is also an edge from A to B; thus A and B are in the same SCC.)

Finally, when all of the "subtrees" of some node B have been traversed and the N-value of B has not been reduced, B is recognized as the root of an SCC ($\{B, A, D\}$ in diagram (4.3)). As the SCC's members are popped, they are marked with Infinity. This prevents their interference in the discovery of other SCCs, for example, $\{Y, Z\}$. If, regarding diagram (4.3), $N\,A$ were not set to Infinity, $N\,Z$ would have been set to 2 and the algorithm would have incorrectly declared $\{X, Y, Z\}$ an SCC.



(4.3)

THEOREM. *Algorithm Digraph correctly determines SCCs.*

For a proof the reader is referred to that given by Eve and Kurki-Suonio [14], since Digraph is but a slight modification of their algorithm.

THEOREM LINEARITY. *Algorithm Digraph is order* $|Vertices| + |Edges|$ *of the digraph induced by relation R, that is,* linear *in the "size" of R.*

PROOF. Traverse is called once for each vertex v, due to the immediate marking and the avoidance of retraversing marked vertices. Inside Traverse, v is pushed on the stack once, and the **for**-loop body is executed once for each edge from v. The **repeat** loop executes only intermittently, when an SCC is determined, and simply pops vertices off the stack; ultimately |Vertices| are popped since that many were pushed. Thus, each vertex is pushed once and popped once, and each edge from it is traversed once. □

COROLLARY. *Algorithm Digraph performs one set union per edge of relation R, that is,* $F\,x \leftarrow F\,x \cup F\,y$.

In fact, it is possible to reduce the number of such unions inside each nontrivial SCC from the number of edges to the number of vertices in the SCC [14]. This improvement would become important in "highly connected" SCCs in a grammar with many terminal symbols, that is, in which set unions become expensive. We did not include the improvement here (nor in our own implementation [12]) because it would obscure the essential algorithm. In addition, nontrivial SCCs are infrequent in practice.

THEOREM. *Algorithm Digraph correctly computes F.*

PROOF. The theorem is based on the following facts. First, if $F\,x$ satisfies (4.1), then $F\,x = \bigcup\{F'\,y \mid xR^*y\}$; this is due to Theorem Equivalent, below. Second, Digraph implicitly computes R^* [14]. In fact, if $F'\,x = \{x\}$, for all $x \in X$, then $F\,x = \{y \in X \mid xR^*y\}$, the set of all vertices reachable from x in the digraph induced by R. □

THEOREM EQUIVALENT. *Suppose F and F′ are functions on* $X \times 2^Z$, *and R is a relation on* $X \times X$. *If* $\forall x \in X{:}F\,x =_s F'\,x \cup \bigcup\{F\,y \mid xRy\}$, *then* $\forall x \in X{:}F\,x = \bigcup\{F'\,y \mid xR^*y\}$.

(Proof is given in the Appendix.)

4.2 Application to LALR

Let the set X in algorithm Digraph be the set of nonterminal transitions of an LR(0) parser. First, let F' be DR and R be **reads**. Then the resulting F will be Read, that is, according to Theorem Across, the computed result will be

$$\text{Read}(p, A) = \text{DR}(p, A) \cup \bigcup\{\text{Read}(r, C) \mid (p, A)\ \textbf{reads}\ (r, C)\}.$$

Second, let F' be Read and R be **includes**. Then the resulting F will be Follow, that is, according to Theorem Up,

$$\text{Follow}(p, A) = \text{Read}(p, A) \cup \bigcup\{\text{Follow}(p', B) \mid (p, A)\ \textbf{includes}\ (p', B)\}.$$

Finally, compute

$$\text{LA}(q, A \rightarrow \omega) = \bigcup\{\text{Follow}(p, A) \mid (q, A \rightarrow \omega)\ \textbf{lookback}\ (p, A)\}$$

according to Theorem Union. Thus, the desired LALR(1) look-ahead sets result from two applications of algorithm Digraph and a final series of set unions.

From a relational point of view, $t \in \text{LA}(q, A \rightarrow \omega)$ iff $(q, A \rightarrow \omega)$ **lookback** (p, A) **includes*** (p', B) **reads** (r, C) **directly-reads** t, where

Definition. (r, C) ***directly-reads*** t iff $t \in \text{DR}(r, C)$. □

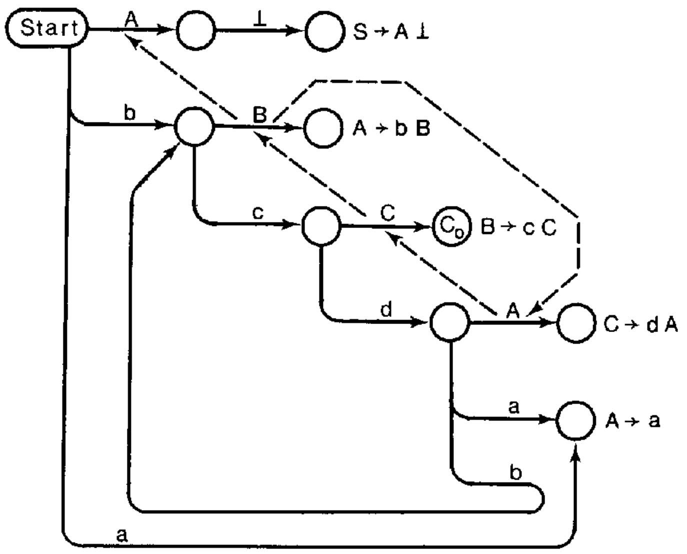
Watt proposed this formulation, although he (erroneously) omitted the **reads** relation altogether [24]. Watt's proposed bit matrix representations of the sparse relations **reads** and **includes** would be wasteful of space and time; for example, for a particular Ada grammar [13], each matrix contains almost five million bits (see Table I in Section 6.1). We have effectively provided an efficient way to compute $R =$ **reads*** ∘ **directly-reads** (Read), then $I =$ **includes*** ∘ R (Follow), and finally **lookback** ∘ I (LA).

4.3 Need for Digraph

Finally, we demonstrate that the generality of algorithm Digraph is needed because both the digraphs induced by **includes** and **reads** can, in general, be non-DAGs. In each case the existence of a nontrivial SCC implies that the grammar is not LR(k) for any k and may or may not be ambiguous.

4.3.1 SCCs in the **includes** Relation

Consider the LR(0) parser with the following state diagram:



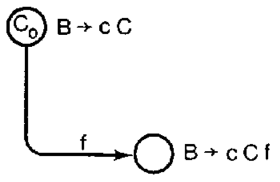
(4.4)

The dashed lines indicate the edges of the digraph induced by the **includes** relation. The digraph is not a DAG, since there is a cycle, and the corresponding grammar is not only LALR(1) but LR(0). (Adding the production $A \rightarrow b$ would make it non-LR(0), but still LALR(1).)

The above example, is, however, "dangerously close to being non-LR", in the sense that, if the Read set were nonempty for any of the A-, B-, or C-transitions involved in the loop, then the grammar would not be LR(k) for any k. It is our belief that the following generalization of this statement holds:

Conjecture Includes-SCC.[1] Let (p, A) be a nonterminal transition that is in a nontrivial SCC of the digraph induced by the **includes** relation. Then the corresponding grammar is not LR(k) for any k if Read$(p, A) \neq \varnothing$. □

Such a problem can be illustrated in the above parser by adding the production $B \rightarrow cCf$. This changes state C_0 to the following:



[1] A related, but simpler, result is proved by Pager [22, p. 41]. Unfortunately, Pager's result does not seem to help in proving this conjecture.

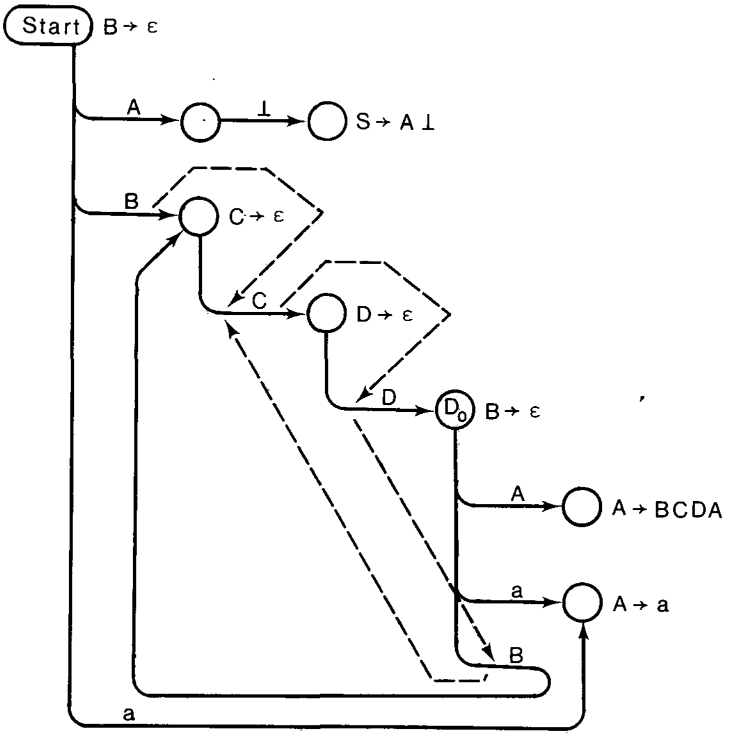
Thus f is in DR, Read, and Follow of the C-transition; thus it is in the Follow set of the A-transition in the loop; thus it is in Follow for the B-transition; thus it is in LA(C_0, $B \rightarrow cC$); and hence there is a read-reduce conflict, since f can also be read from state C_0. (The symbol $\perp$ is the only other symbol in each of these Follow sets, due to the A-transition from the start state.)

The grammar is not only non-LR but is also ambiguous. The ambiguity is evident in that $B \Rightarrow^* cdbB$ and distinctly $B \Rightarrow^* cdbBf$. This is essentially the classical "dangling **else** problem", where f is the **else** clause and cdb is the **if-then** clause: $B \Rightarrow^* cdbB \Rightarrow^* cdbcdbBf$ and distinctly $B \Rightarrow^* cdbBf \Rightarrow^* cdbcdbBf$.

This example can be made arbitrarily more subtle and complex by adding strings of nullable nonterminals to the ends of the various productions and prior to the f in the added production. Additionally, changing $B \rightarrow cC$ to $B \rightarrow cCX$, for example, where $X \rightarrow \epsilon$, still produces a read-reduce conflict on symbol f, but now the production involved is $X \rightarrow \epsilon$ in state C_0. If instead $B \rightarrow cCf$ is changed to $B \rightarrow cCXf$, then a reduce-reduce conflict results on symbol f in state C_0, since LA(C_0, $B \rightarrow cC$) = $\{f, \perp\}$ and LA(C_0, $X \rightarrow \epsilon$) = $\{f\}$.

4.3.2 SCCs in the **reads** relation

Now consider the LR(0) parser whose state diagram is



(4.5)

Here the dashed lines represent the edges of the **reads** relation. DR for the two B-transitions and the C-transition is empty, but for the D-transition it is $\{a\}$.

Thus Read for each is $\{a\}$ because the B's **read** C-, C- **reads** D-, and D- **reads** (the lower) B-transition. Hence a read–reduce conflict results on symbol a in state D_0, since LA(D_0, $B \rightarrow \epsilon$) contains Follow of the lower B-, which contains Read of the lower B-, which contains a.

In this particular case the grammar is ambiguous, since the empty string can be reduced to BCD many times prior to reducing to A, the only a in the only string in the language. However, by changing production $A \rightarrow BCDA$ to $A \rightarrow BCDAf$ the ambiguity is eliminated, while retaining the conflict. Now the grammar is unambiguous, since the number of f's fixes the number of reductions of empty to BCD, but it is still not LR(k) for any k, since the BCDs must be reduced before a is reduced to A, but the f's follow the a. In general,

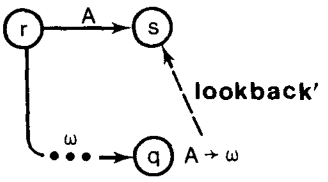
THEOREM READS-SCC. *If the digraph induced by the* **reads** *relation contains a nontrivial SCC, then the corresponding grammar is not LR(k) for any k.*

5. OVERSIMPLIFICATIONS

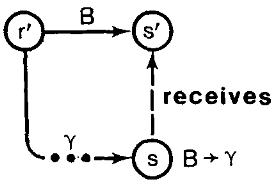
Two "clever ideas" come to mind, each of which is shown below to be inadequate.

5.1 NQLALR(1) Parsers

The most notable one, an oversimplification of the computation of LALR(1) look-ahead sets, has been invented independently by several researchers [7, 11, 23, 25] and continues to be reinvented. It involves defining another relation **receives** that is closely related to the union of **includes** and **reads** and leads to what we call "not quite LALR(1)" or NQLALR(1) parsers. The basic idea is to relate states rather than transitions. The reasoning is that LA(q, $A \rightarrow \omega$) must include all symbols that can be looked-ahead-at or read from any "restart" state s such that there is a "look-back" state r with an A-transition to s and path spelling ω to q:



If ω is reduced to A in q, r may be the top state after popping $|\omega|$ states. Then A is read and s entered, so any symbol looked-ahead-at or read by s would be in LA(q, $A \rightarrow \omega$), or really NQLA(q, $A \rightarrow \omega$). Any state s' reachable from s by reductions also contributes:



If we reduce γ to B in s, then we may enter state s', and any symbol valid to s' is in NQLA$(q, A \to \omega)$. Formally,

Definition. $(q, A \to \omega)$ ***lookback'*** s iff there is an $r \in K$ such that

$$r \xrightarrow{A} s \quad \text{and} \quad r \overset{\omega}{\cdots\to} q, \quad \text{where} \quad A \to \omega\cdot \in q. \qquad \square$$

Definition. s ***receives*** s' iff (s, p) **lookback'** s' for some production p. □

Definition

$$\text{NQLA}(q, A \to \omega) = \bigcup\{\text{NQFollow}(s) \mid (q, A \to \omega)\ \textbf{lookback}'\ s\}. \qquad \square$$

Definition

$$\text{NQFollow}(s) =_s \text{NQDR}(s) \cup \bigcup\{\text{NQFollow}(s') \mid s\ \textbf{receives}\ s'\}. \qquad \square$$

Definition. NQDR$(s) = \{t \in T \mid \text{Next}(s, t) \text{ is defined}\}$, that is, the same as DR except defined for states rather than transitions. □

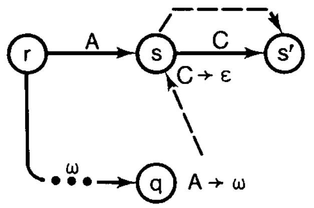
THEOREM. *NQFollow*$(s) = \bigcup\{NQDR(s') \mid s$ ***receives**** $s'\}$.

COROLLARY

$$NQLA(q, A \to \omega) = \bigcup\{NQDR(s) \mid (q, A \to \omega)\ \textbf{\textit{lookback}}' \circ \textbf{\textit{receives}}^*\ s\}.$$

The theorem follows from applying Theorem Equivalent to the definition of NQFollow, and the corollary follows from back-substitution into the definition of NQLA.

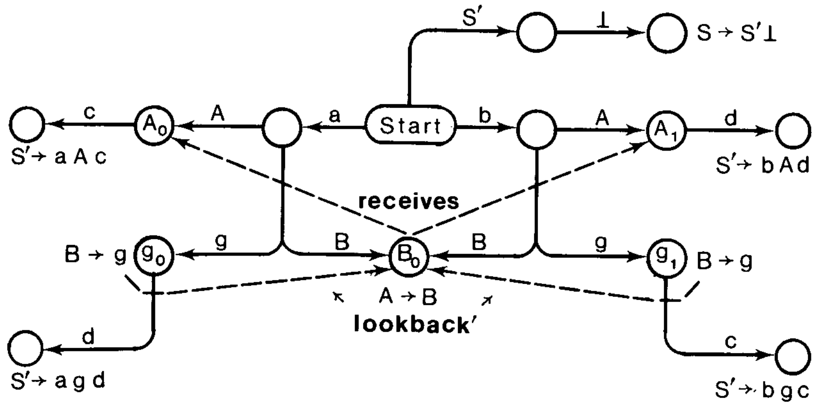
Note that the nullable nonterminals cause no problem here. For example,



The dashed arrow from s to s' comes directly from the definition of **receives** and obviously serves a purpose similar to the edges of **reads**. Thus, NQLA$(q, A \to \omega)$ is just the union for all the NQDRs of the states reachable via **lookback'**∘**receives***. Our old IBM 360 implementation [11] just computes **lookback'**∘**receives*** via bit matrix techniques, then unions the related read sets (NQDR) to get the NQLAs.

The inadequacy of NQLALR arises from the fact that inappropriate "paths of reductions" are traced through the parser, in effect, first reducing ω to A and landing in state s, but then reducing in s, say $B \to xA$, *without requiring that the same A-transition that led into state s be involved when leaving via reduction.*

The following LR(0) parser illustrates this point:



The relevant **receives** and **lookback**$'$ edges are indicated by dashed arrows.

In this case both NQLA(g_0, $B \rightarrow g$) and NQLA(g_1, $B \rightarrow g$) contain $\{c, d\}$, because the **receives** relation connects state B_0 with both states A_0 and A_1, which can read c and d, respectively. Hence the grammar is not NQLALR(1) because we have read-reduce conflicts in both states g_0 and g_1. The grammar is, however, LALR(1) and our correct approach (and our new implementation [12] results in LA(g_0, $B \rightarrow g$) = $\{c\}$ and LA(g_1, $B \rightarrow g$) = $\{d\}$, which are the correct LALR(1) sets. It would be instructive for the reader to draw in the two **lookback** and the two **includes** edges and observe how the two halves of the parser remain separated.

It is easy to see that LA(q, $A \rightarrow \omega$) $\subseteq$ NQLA(q, $A \rightarrow \omega$); this is because (q, $A \rightarrow \omega$) **lookback** (p, A) only if (g, $A \rightarrow \omega$) **lookback**$'$ Next(p, A), and (p, A) **includes** (p', B) only if p **receives** Next(p', B). NQLALR look-ahead sets are only "slightly larger" than LALR look-ahead sets. In practice, we have encountered only a few programming language grammars that are LALR but not NQLALR. NQLALR is a large improvement over SLR, however. In summary, SLR-LA(q, $A \rightarrow \omega$) $\supseteq$ NQLA(q, $A \rightarrow \omega$) $\supseteq$ LA(q, $A \rightarrow \omega$).

5.2 Combining **includes** and **reads**

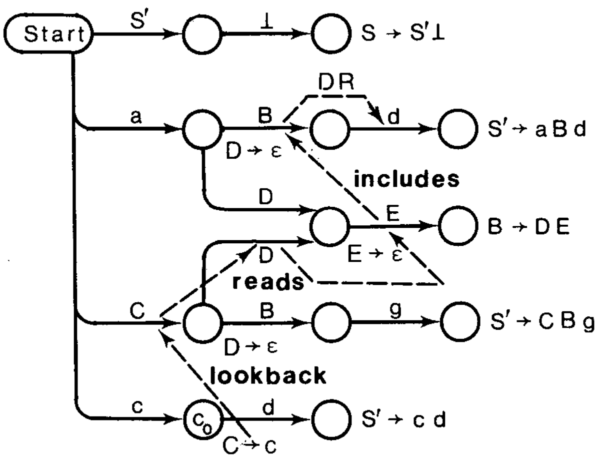
The second oversimplification consists of unioning **includes** and **reads** and running the Digraph algorithm only once. It implies that instead of computing

$$\text{Follow}(p, A) = \bigcup\{\text{DR}(r, C) \mid (p, A)\ \textbf{includes}^* \circ \textbf{reads}^* (r, C)\}$$

as described above (this formula can be obtained by applying Theorem Equivalent to rewrite the expressions for Follow and Read (see Theorems Up and Across) and back-substituting the rewritten Read in the rewritten Follow), we instead compute

$$\text{Follow}(p, A) = \bigcup\{\text{DR}(r, C) \mid (p, A)\ (\textbf{includes} \cup \textbf{reads})^* (r, C)\}.$$

The two equations are not equivalent in general, as indicated by the following counterexample:



LA(c_0, $C \rightarrow c$) contains Follow of the C-transition which **reads** the D-, which **reads** the E-, which **includes** the upper B-transition, whose DR set contains d, so a read-reduce conflict results in c_0. On the other hand, with the correct approach the relevant LA set contains g only. We have seen no one make this oversimplification, but it occurred to us when we tried to reduce the number of applications of Digraph from two to one.

6. LALR IMPLEMENTATION

A complete procedure to compute LALR(1) look-ahead sets from an LR(0) automaton is as follows:

A. Compute which nonterminals are nullable.
B. Initialize Read to DR: one set (bit vector of length the number of terminals) for each nonterminal transition, by inspection of the transition's successor state.
C. Compute **reads**: one list of nonterminal transitions per nonterminal transition, by inspection of the successor state of the latter transition.
D. Apply algorithm Digraph to **reads** to compute Read; if a cycle is detected, announce that the grammar is not LR(k) for any k.
E. Compute **includes** and **lookback**: one list of nonterminal transitions per nonterminal transition and reduction, respectively, by inspection of each nonterminal transition and associated production right parts, and by considering nullable nonterminals appropriately.
F. Apply algorithm Digraph to **includes** to compute Follow: use the same sets as initialized in part B and completed in part D, both as initial values and as workspace. If a cycle is detected in which a Read set is nonempty, announce that (as we conjecture) the grammar is not LR(k) for any k.
G. Union the Follow sets to form the LA sets according to the **lookback** links computed in part F.
H. Check for conflicts; if none, announce that the grammar is LALR(1)—we have a parser.

6.1 Efficiency

The number of bit vectors needed and the number of relation edges traversed may be minimized by following the strategy given below.

First, the assignment $F(\text{Top } \mathbf{of}\ S) \leftarrow F\ x$ in algorithm Digraph should only be done if $x \neq \text{Top } \mathbf{of}\ S$. This saves the expense of the set copy for trivial (singleton) SCCs, which in fact are in the majority. (Avoiding all set copy expense in nontrivial SCCs by doing the assignments by *reference*, that is, by having $F(\text{Top } \mathbf{of}\ S)$ and $F\ x$ *point* to the same bit vector, will not work; the scheme described here uses the same bit vector to represent the Read set and the Follow set of each nonterminal transition. Since the SCCs of the **reads** and **includes** relations are different, this would imply different "sharing" and would in fact invalidate the algorithm.)

Next compute the **includes** and **lookback** relations as described in step E above. Initially, allocate no sets to the nonterminal transitions. If the grammar contains any nullable nonterminals, then only for each nonterminal transition (p, A) involved in the **reads** relation, apply Digraph (with R = **reads**) to compute Read(p, A). Afterward, the only transitions having sets allocated to them will be those involved in the **reads** relation (typically, few or none at all). For each such transition (p, A), the set will be equal to Read(p, A).

Rather than specially detecting the transitions involved in the **reads** relation, the latter application of Digraph can be achieved by applying it to *all* nonterminal transitions where Digraph has been modified as follows: delete the assignment $F\ x \leftarrow F'\ x$; insert code to detect either fetching (F, x) or storing $(F\ x \leftarrow \cdots)$ Read(p, A) when (p, A) has not had a set allocated to it. In such a case Digraph should allocate a set to (p, A) and initialize it with DR(p, A) $(F'\ x)$.

Finally, note that the **reads** relation need not be precomputed (step C above) since it can be easily retrieved from the LR(0) automaton as Digraph needs it.

For each inconsistent state q and final item $A \rightarrow \omega\cdot$ in q, follow each **lookback** edge to a nonterminal transition (p, A). Invoke Digraph on (p, A) (with R = **includes**) to compute Follow(p, A). Take the union of all the Follow sets indicated by **lookback** to obtain LA$(q, A \rightarrow \omega)$. With the modifications made to Digraph and the Read set computation as described above, any transition (p, A) inspected by Digraph when computing Follow will either have a set allocated to it that contains Read(p, A), or Digraph will allocate a set to it and initialize it to DR(p, A), which must equal Read(p, A), because no set was allocated. Thus Read(p, A) will be either already computed or computed when needed by Digraph.

Due to this strategy, sets are needed only for those transitions involved in the **reads** relation or needed for the computation of the look-ahead set of final items in inconsistent states; in addition each such item needs a set to represent its look-ahead set. The only relation edges traversed will be the **reads** edges and that subset of the **includes** and **lookback** edges needed for the look-ahead set computation. Thus, fewer set unions and less set storage are needed, as described in Table I.

Table I lists the total number of unions performed by our implementation. This equals the sum of the number of **includes**, **lookback**, and **reads** edges plus a few more unions that occur when Digraph copies the F value of the root of an SCC to the values of the other vertices in the SCC (for Pascal, e.g., there were 4 SCCs with a total of 30 vertices, and $841 = 552 + 256 + 7 + (30 - 4)$). Under the

Table I. Relations Sizes, Sets Required, Timing Statistics, and Comparison with YACC

		Edges traversed/ actual edges				
Grammar	Unions	**includes**	**lookback**	**reads**	Sets	FI
PAL	1754	1160/1336	571/1565	0	555	25
XPL	660	423/ 613	233/ 978	0	279	26
PASCAL	841	552/ 579	256/1134	7	344	51
SL300	3678	2298/3226	1360/5845	27	1340	152
Ada	4534	3051/4739	1388/6501	69	1376	161
Ada′	2875	2641/4739	139/6501	69	1236	21

	SCC	NTX	LR(0)	LA	YACC	Reference
PAL	25	590	4.97	7.45	9503	[4]
XPL	5	420	3.43	4.09	5292	[21]
PASCAL	30	337	4.36	5.12	6096	[15]
SL300	23	1886	?	?	?	[8]
Ada	7	2257	?	?	?	[13]
Ada′	7	2257	?	?	?	[13]

heading Sets in the table is recorded the total number of bit vectors (Read, Follow, LA) that are required. By subtracting the number of actual look-ahead sets for final items (FI) in inconsistent states from the Sets column, one may determine how many sets were allocated solely to nonterminal transitions. The SCC column records the total number of vertices of the **includes** relation involved in nontrivial SCCs. NTX is the number of nonterminal transitions. The CPU time in seconds for the LR(0) computation and the look-ahead (LA) computation is for an HP-3000 computer with a 1.5 μs memory and includes checking for the LALR(1) condition. YACC tends to perform five to eight times as many set unions as does our algorithm (the YACC column).

Both SL300 and Ada were too large to run on YACC. Neither could they be run on the memory-limited HP 3000, and timing statistics for the machine they *were* run on are not available. The Ada grammar referenced is close to the grammar used to produce the statistics; the latter has not yet been published. The Ada′ entry is explained in a later section.

A slight time improvement could be made by only traversing necessary **reads** edges. Do not apply Digraph separately to compute Read sets. Rather, compute only the Follow sets. While computing a Follow set, if Digraph discovers a transition (p, A) for which Read(p, A) has not yet been computed, it calls itself to compute Read(p, A). To do this requires separate allocation of Read(p, A) and Follow(p, A); otherwise, the intermixing of the computation of Follow with that of Reads would be tantamount to the oversimplification described in Section 5.2. Since there are so few **reads** edges, the extra space needed is not worth the time saved.

To reduce storage consumption, an implementation should only store those **lookback** edges demanded by look-ahead computation. **Lookback** edges leading from final items in consistent states should be discarded. For Ada, for example, only 1388 **lookback** edges were needed, 21 percent of the total. In fact, more space and time can be saved by using SLR(1) look-ahead sets to resolve inconsistencies in as many states as possible. The Ada′ row in the table indicates the results if this is done; only 21 final items need LALR(1) look-ahead sets, and only 139 **lookback** edges are useful. Further, fewer **includes** edges are traversed, reducing the number of set unions and sets needed. Certainly, if a grammar is SLR(1) then none of the **includes/lookback** computation is necessary.

SLR(1) look-ahead sets may be computed from the LR(0) parser and the Read sets by an application of Digraph. SLR-LA$(q, A \rightarrow \omega)$ = SLR-Follow(A) = $\bigcup\{\text{Read}(p, B) \mid B \Rightarrow^* \beta A\gamma,\ \gamma \Rightarrow^* \epsilon,\ \beta \in V^*$, and (p, B) is a nonterminal transition$\}$. This was first observed by DeRemer [10], with minor errors regarding nullable nonterminals. Restated, SLR-Follow$(A) = F'\,A \cup \bigcup\{\text{SLR-Follow}(B) \mid A\ R\ B\}$, where $F'\,A = \{\text{Read}(p, A) \mid (p, A)$ is a nonterminal transition$\}$ and $A\ R\ B$ iff $B \Rightarrow^* \beta A\gamma$, $\gamma \Rightarrow^* \epsilon$, thus casting the SLR definition in a form suitable for computation by Digraph.

In fact Digraph is generally useful in computing other functions on context-free grammars, for example, whether a nonterminal can be derived from the start symbol or whether a nonterminal is both left and right recursive. This is essentially due to Digraph's relationship with the transitive-closure problem [14]. Replacing standard fast bit-matrix techniques by a variant of Digraph tripled the speed of the grammar-checking phase of the MetaWare™ translator writing system [12]. For sparse relations, Digraph does a much better job of computing transitive closure than do bit-matrix techniques.

6.2 Linearity

Algorithm Digraph is linear in the size of the relation to which it is applied, as established by Theorem Linearity of Section 4. For practical grammars, the size (number of edges) of the **includes** relation is about two to three times the number of nonterminal transitions in the parser. Each nonterminal transition (p, A) has one **includes** edge to it for each production for A that ends in a nonterminal B, or $B\gamma$ where $\gamma \Rightarrow^* \epsilon$, that is, usually only two or three at most. The **reads** relation is virtually nonexistent in practical cases, so it can be ignored. Thus, for practical LR(0) parsers, Digraph is about linear in the number of nonterminal transitions. These statements are substantiated by the statistics given in Table I.

In the worst case the size of the **includes** relation could be proportional to the square of the number of nonterminal transitions, since the relation could be nearly complete, that is, x **includes** y for *all* nonterminal transitions x and y. This worst case is illustrated by the grammar whose productions are $\{S \rightarrow S_1 \perp,\ S_i \rightarrow S_j,\ S_i \rightarrow t \mid 1 \leq i, j \leq n\}$. Ignoring the path $[S_1 \perp]$, the LR(0) parser for this grammar has n nonterminal transitions, one for each S_i, $1 \leq i \leq n$, and each has n **includes** edges to it; that is, each nonterminal transition has an edge to it from each of the others and from itself! Of course, this example is contrived, highly ambiguous, and has no redeeming virtue from a practical viewpoint.

6.3 Comparison with Other Algorithms

Both the algorithms of Aho and Ullman [3] and Anderson et al. [6] work by "propagating" look-ahead sets across the edges of an implicit (virtual) graph. When propagation causes the look-ahead information at a vertex to be increased, the vertex is queued up so that it may in turn cause propagation of the new information to other vertices. This process is iterated until the queue becomes empty. The order of propagation may not be optimal, in the sense that each edge is traversed only once. This causes the YACC algorithm to perform considerably more poorly than ours, as indicated by Table I. (YACC in fact does not even use a queue, but repeatedly scans *all* vertices until propagation ceases [3]. The author of YACC thought it might perform better on ambiguous grammars, which it accepts, but in fact it is even worse for them than for unambiguous grammars.)

LaLonde's algorithm [20] is essentially algorithm D of Section 4, but to avoid incorrect computation of look-ahead sets (see digram (4.2)), no information is retained at the vertices between the computation of the look-ahead sets for distinct reductions. Thus, edges may need to be traversed repeatedly as different look-ahead sets are computed.

In contrast to the abovementioned algorithms, ours traverses each edge exactly once.

Kristensen and Madsen [19] have improved LaLonde's algorithm so that intermediate results exactly analogous to our Follow sets are retained and used for future computation. To accomplish this, Kristensen and Madsen's algorithm detects SCCs by occasionally adding certain vertices to the sets $F\ x$ being computed. After traversing the successors of a vertex x, either $F\ x$ contains no vertices, or it contains exactly one vertex v. In the former case x is the root of a trivial SCC; in the latter, if $v = x$, then x is the root of a nontrivial SCC, and all occurrences of x in sets $F y$ are replaced by $F x - \{x\}$. The technique is clever but somewhat clumsy (all details have not been given here), and in fact the UNION-FIND algorithm [2] is necessary to keep the complexity of Kristensen and Madsen's algorithm to (*very* close to, but not exactly) linear in the size of the relation traversed. We believe that their algorithm incurs greater overhead than ours.

Kristensen and Madsen effectively use relations **item-includes** and **item-lookback** that are similar to **includes** and **lookback** but are defined on *items* instead of nonterminal transitions. Thus the vertices of the graph traversed are items. Since each nonterminal transition (p, A) in the parser in general is the result of one or more items $B \to \delta \cdot A\gamma$ in p, Kristensen and Madsen's approach incurs the cost of additional F sets. Since Kristensen and Madsen present no empirical data, it is difficult to determine the practicality of their algorithm.

In contrast, our algorithm nowhere uses the concept of item; thus all LR items may be discarded before look-ahead set computation beings. As a consequence, our algorithm may work on other than LR automata. For example, it may be possible to define an automaton using a precedence technique such that, if our algorithm is applied, it will compute some "precedence look-ahead sets" and thus determine whether the grammar is, say, simple precedence. We leave the exploration of such ideas for future research.

LR items are necessary, however, to produce the diagnostic debugging traces described in the next section.

7. DEBUGGING GRAMMARS THAT ARE NOT LALR(1)

As a pleasant by-product of our research into the area of look-ahead set computation, we found that the edges of the **includes** and **lookback** relations are just what are needed to produce helpful debugging information for grammars that are not LALR(1). These edges link back through the automaton from look-ahead sets, in particular those involved in conflicts, to the sources of the conflict symbols (CSs). This is exactly the trace through the grammar that the user usually has to find manually. For example, the MetaWare™ translator writing system [12] prints roughly the following trace for the conflict in state C_0 of a parser like that of diagram (4.4), to which the production $B \rightarrow cCf$ has been added to introduce the conflict:

```
A ⊥
b B
  c C f
    |
    d A                                          (7.1)
      b B
        c C reduce B → c C · {f} ?
            read   B → c C · f   ?
```

The diagram indicates that after the parser has read $bcdbcC$ and sees an f, it has two courses of action:

(1) reduce cC to B, reduce bB to A, reduce dA to C, read f; or
(2) read f.

Item $B \rightarrow cC \cdot f$ in state Top $[bcC]$ contributes an f to LA(C_0, $B \rightarrow cC$) by virtue of the productions $C \rightarrow dA$, $A \rightarrow bB$, and $B \rightarrow cC$ that trace a path from $p' =$ Top $[bc]$ to C_0. The trace consists of two derivations, and each right part in a derivation is positioned vertically beneath the nonterminal that derives it. The derivation above the line consisting of the single vertical line shows how the start state traces a path to p'. Immediately above the vertical line is the item that contributes the look-ahead symbol f. Below the vertical line is the derivation from the (nonterminal C in the) contributing item to the inconsistent state, ending in the final item with the conflict. The latter derivation induced the **lookback** and **includes** edges that relate (C_0, $B \rightarrow cC$) to transition (p', C), whose Read set contains f. Item $B \rightarrow cC \cdot f$ in C_0 causes f to be a read symbol also; hence the conflict.

Consider next the parser of diagram (4.5) and the following trace for state D_0:

```
A ⊥
B C D A
| D A
| A                                              (7.2)
| a
ε     reduce B → ε · {a} ?
      read   A → · a     ?
```

The parser's two choices of action are

(1) reduce ϵ to B, reduce ϵ to C, reduce ϵ to D, read a (after which a is reduced to A); or
(2) read a.

Item $A \rightarrow B \cdot CDA$ in state Top $[B]$ contributes a to LA(Start, $B \rightarrow \epsilon$) by virtue of the production $B \rightarrow \epsilon$ and the fact that $CDA \Rightarrow^+ A$. This latter derivation is presented just to the right of the "tower" of vertical lines. Notice how C and D "vanish" in the derivation because they are nullable; here $C \rightarrow \epsilon$ and $D \rightarrow \epsilon$ are productions. Were C or D not to directly derive ϵ, the additional derivation steps could be printed; however, we feel this would clutter the trace.

The general form of a trace is as follows:

$$\begin{array}{ll}
S' \perp & \\
\delta_1\ B_1\ \nu_1 & \\
\quad \delta_1\ B_2\ \nu_2 & \leftarrow \text{Derivation from the start state} \\
\qquad \vdots & \\
\qquad \delta_n\ B_n\ \nu_n & \\
\qquad\quad \alpha\ \ B\ \beta_1 & \leftarrow \text{to the contributing item.} \\
\qquad\qquad | \ \ \beta_2 & \\
\qquad\qquad | \ \ \vdots & \leftarrow \text{How the contributing item} \\
\qquad\qquad | \ \ \beta_{m-1} & \\
\qquad\qquad | \ \ t\ \ \beta_m & \leftarrow \text{contributes } t. \\
\qquad\qquad | & \\
\qquad\qquad \alpha_1\ A_1\ \gamma_1 & \\
\qquad\qquad\quad \alpha_2\ A_2\ \ \gamma_2 & \leftarrow \text{How the contributing item} \\
\qquad\qquad\qquad \vdots & \\
\qquad\qquad\qquad \alpha_{s-1}\ A_{s-1}\gamma_{s-1} & \leftarrow \text{relates to the item with the CS.} \\
\qquad\qquad\qquad\qquad \alpha_s & \textbf{reduce } A_{s-1} \rightarrow \alpha_s \cdot \{t\}\ ? \\
 & \text{(list of conflicting items here)}
\end{array} \tag{7.3}$$

Here, item $B_n \rightarrow \alpha B \cdot \beta_1$ in Top $[\delta_1 \cdots \delta_n \alpha B]$ contributes t to the look-ahead set of $A_{s-1} \rightarrow \alpha_s \cdot$ because $\beta_1 \Rightarrow \beta_2 \Rightarrow \cdots \Rightarrow t\beta_m$ as shown, and because the chain of productions below the vertical line causes $A_{s-1} \rightarrow \alpha_s \cdot$ to be related to the B-transition via **lookback** $\circ$ **includes***; here each $\gamma_i \Rightarrow^* \epsilon$ (but not necessarily so for each ν_i). Each A_i and B_i and the B produces the right part below it.

Traces are constructed by beginning with a final item $A_{s-1} \rightarrow \alpha_s \cdot$ in an inconsistent state q and traversing a **lookback**, then some **includes** edges until a nonterminal transition (p', B) is found whose Read set contains one of the CSs. There exist one or more items $B_n \rightarrow \alpha B \cdot \beta_1$ in Next(p', B) such that $\beta_1 \Rightarrow^* t\beta_m$ and t is a CS. A trace should be printed for each such item (barring redundant traces; see below). (Note that we do not inspect the Follow sets during the traversal from $A_{s-1} \rightarrow \alpha_s \cdot$. While Follow$(p', B)$ may contain a CS, there is no guarantee that Next(p', B) contains CS-contributing items; Follow(p', B) could have inherited its CS from another Follow set. CSs "originate" at Read sets, not at Follow sets.)

The trace consists of three components:

(a) the derivation from S' that gave rise to $B_n \rightarrow \alpha B \cdot \beta_1$ in Next(p', B);
(b) the derivation of $t\beta_m$ from β_1; and
(c) the productions that induced the **includes** and **lookback** edges from $(q, A_{s-1} \rightarrow \alpha_s)$ to (p', B).

For component (c) a breadth-first search should be employed during the original traversal of the **includes** edges. This keeps the size of the **includes** "chain" to a minimum, giving the "simplest" possible explanation to the reader. The production inducing an **includes** edge from (p_1, X_1) to (p_2, X_2) can be rediscovered by following the automaton transitions from state p_2 under the right parts of productions for X_2. Thus, inducing productions need not have been redundantly stored when the **includes** edges were originally computed.

For component (b), compute the set

$$\begin{aligned} E =_s \; & \{B_n \rightarrow \alpha B \cdot \beta_1\} \\ & \cup \{A \rightarrow \delta X \cdot \eta \quad | \, A \rightarrow \delta \cdot X\eta \in E \wedge X \Rightarrow^* \epsilon\} \\ & \cup \{C \rightarrow \cdot \alpha \quad | \, A \rightarrow \delta \cdot C\eta \in E \wedge C \rightarrow \alpha \in P\} \end{aligned}$$

linking additions to E back to the item that generated them. Items of the form $C \rightarrow \cdot t\beta_m$ with t a CS will be in E and can be traced back to $B_n \rightarrow \alpha B \cdot \beta_1$ by following the links. All derivations of CSs from β_1 can thus be produced.

Component (a) requires two computations. First, find the shortest path $[\xi]$ from the start state to Next(p', B). In our own implementation, this entails repeatedly asking for the lowest numbered state that has a transition to a given state, since the states are computed and numbered in a breadth-first manner. Then compute

$$\begin{aligned} E' =_s \; & \{(S \rightarrow \cdot S' \perp, 1)\} \\ & \cup \{(C \rightarrow \cdot \alpha, j) \quad | \, (A \rightarrow \delta \cdot C\eta, j) \in E' \wedge C \rightarrow \alpha \in P\} \\ & \cup \{(A \rightarrow \delta X \cdot \eta, j+1) \, | \, (A \rightarrow \delta \cdot X\eta, j) \in E' \wedge X = \xi_j \wedge j \leq |\xi|\} \end{aligned} \tag{7.4}$$

in a breadth-first fashion, linking additions to E' back to the pairs that generated them. Eventually $(B_n \rightarrow \alpha B \cdot \beta_1, |\xi| + 1)$ will appear in E', and the computation may stop: all of E' need not be computed. The desired production sequence may be obtained from inspecting the links. The breadth-first search and the fact that $[\xi]$ is the shortest path keep the size of the production sequence to a minimum. $[\xi]$ also serves to limit the size of E' by constraining the addition of pairs.

In practical grammars, an item $B_n \rightarrow \alpha B \cdot \beta_1$ may appear in several different left contexts, and in each may contribute First β_1 to the same particular LA set. Therefore, to prevent redundant trace output, only one trace per contributing item for an LA set should be printed. Additionally, if (referring to trace (7.3)) β_{m-1}

immediately derives more than one string of the form $t\beta_m$ with t a CS, then all of those strings can be listed on successive lines.

The following informal trace-printing algorithm summarizes the foregoing:

```
for   each inconsistent state $q$
  do let   Conflict-set = the set of symbols for which $q$ is LALR(1) inconsistent
     for   $A_{s-1} \rightarrow \alpha_s \cdot \in q$ such that LA($q$, $A_{s-1} \rightarrow \alpha_s$) $\cap$ Conflict-set $\neq \emptyset$
       do let   $EI = \emptyset$                                    # $EI$ = Explained items.
          for   ($p'$, $B$) found in a breadth-first manner such that
                ($q$, $A_{s-1} \rightarrow \alpha_s \cdot$) lookback$\circ$includes* ($p'$, $B$) $\wedge$ Read($p'$, $B$) $\cap$
                Conflict-set $\neq \emptyset$
            do  let   $T = \emptyset$
                for   $I = B_n \rightarrow \alpha B \cdot \beta_1 \in$ Nucleus(Next($p'$, $B$)) such that $I \notin EI$
                  do for   each derivation $\beta_1 \Rightarrow^* t\beta_m$ such that $t \in$ Conflict-set
                       do Print trace components (a), (b), and (c) above
                          assign $EI \leftarrow EI \cup \{I\}$; $T \leftarrow T \cup \{t\}$
                       od
                  od
                #    See "Read-item traces" below for next three lines:
                for   $I = D_{r-1} \rightarrow \xi_r \cdot t\eta_r \in q$ such that $t \in T$
                  do Print read-item trace for $I$
                  od
            od
       od
od
```

Read-Item Traces. While the tracing methods presented thus far indicate how terminals enter into look-ahead sets, they do not show how (conflicting) read-transitions arise in inconsistent states. Such information would be useful to the user trying to determine why the parser can either reduce *or* read in a particular inconsistent state.

Trace (7.3) indicates why the parser can reduce when the stack contains $\xi = \delta_1 \cdots \delta_n \alpha\alpha_1 \cdots \alpha_s$. A read trace *corresponding* to trace (7.3) should indicate why the parser can read when the stack contains ξ. Such a trace appears as follows:

$$\begin{array}{lllll}
S' \perp & & & & \\
\xi_1\ D_1\ \eta_1 & & & & \\
\quad \xi_2\ D_2\ \eta_2 & & & & \\
\qquad \vdots & & & & \\
\qquad \xi_{r-1}\ D_{r-1}\ \eta_{r-1} & & & & \\
\qquad\quad \xi_r \cdot t\ \ \eta_r & & \textbf{read}\ D_{r-1} \rightarrow \xi_r \cdot t\eta_r? & &
\end{array}$$

where $\xi_1 \cdots \xi_r = \xi$ and each D_i produces the right part on the next line. The constraint on the ξ_i's guarantees the correspondence; consequently, the inconsistent state q contains $D_{r-1} \rightarrow \xi_r \cdot t\eta_r$, the "conflicting" item.

For each conflicting read item $I = D_{r-1} \rightarrow \xi \cdot t\eta_r$ in q, the shortest possible trace can be constructed for I by the computation described in (7.4) above, except where ξ is replaced with $\delta_1 \cdots \delta_n \alpha\alpha_1 \cdots \alpha_s$, and the computation stops when I is produced in E'.

For trace (7.1), the MetaWare™ translator writing system produces the following read-conflict trace:

```
A ⊥
b B
  c C
    d A
      b B
        c C·f      read B → cC·f?
```

For trace (7.2), the corresponding trace produced is

```
A ⊥
·a          read A → ·a ?
```

The above traces are simple enough that the reader may not be convinced that they are generally useful. Consider, however, the following traces produced by the MetaWare™ translator writing system for an Ada grammar that was not LALR(1):

```
Ada-compilation ⊥
Compilation-unit
Program-unit
Subprogram-body
Subprogram-spec is Unit-body
                   Dclns begin Compound end
                   Declaration ;
                   Subprogram-spec is separate
                   |
                   procedure ⟨ID⟩ Params
                                  ϵ      reduce Params → ϵ·{is} ?
Ada-compilation ⊥
Compilation-unit
Program-unit
Subprogram-body
Subprogram-spec is Unit-body
                   Dclns begin Compound end
                   Declaration ;
                   Subprogram-dcln
                   procedure ⟨ID⟩·is new Name      read... ·is...?
```

Here, after the left context Subprogram-spec **is**, either another Subprogram-spec can produce a procedure header with an empty parameter list, or a Subprogram-dcln can produce a generic instantiation of a procedure. Clearly, the traces pinpoint the problem.

Reduce-Reduce Conflicts and LALR Versus LR. For read-reduce conflicts, we have advocated displaying corresponding traces that show how either the read or the reduction is legal, given a particular left context (parse stack) [ξ]

($\xi = \delta_1 \cdots \delta_n \alpha \alpha_1 \cdots \alpha_s$ in reduce trace (7.3)). However, for a reduce–reduce conflict, say between productions p_1 and p_2, our algorithm does not necessarily print *corresponding* traces for p_1 and p_2, that is, traces that share the same left context. Indeed, it is possible that no trace for p_1 will have the same left context as any trace for p_2.

This can happen in the following two circumstances: (1) the grammar is LR(1) but not LALR(1);[2] (2) an "accident" of our algorithm occurs. In either case, the person debugging the grammar may be confused: since the traces do not show why two different reduce moves are possible given a *single* left context, he may not understand why the conflict exists.

In the first case, when a reduce–reduce conflict exists both in the LALR(1) and the LR(1) automaton, a trace for each reduction exists (in each automaton) with the same left context. This is due to the LR(k) definition. But when the conflict is present in the LALR(1) automaton and *not* in the LR(1) automaton (call such a conflict *LALR-only*), then *no* such corresponding traces exist. The space-saving aspect of the LALR construction technique can cause a conflict by "merging" together two distinct left contexts.

In the second case, because we suggest printing only one reduce trace per contributing item and printing the shortest path from the start state to the item (these measures reduce the volume and size of the reduce traces), it is possible to construct a grammar that has an LALR-only and a distinct LR conflict, but for which no two printed reduce traces have the same left context. One might then erroneously guess that the conflict is LALR-only.

One way to avoid such confusion is as follows: when a reduce–reduce conflict occurs for productions $p_1, p_2, \ldots, p_n$, produce traces for p_1; then, for the other productions, produce only traces that correspond to the traces for p_1. If no corresponding trace can be found, then an LALR-only conflict has been pinpointed and should be reported as such (and noncorresponding traces should then be printed). But again, due to our limited selection of traces for p_1, even if corresponding traces for $p_2, \ldots, p_n$ are found, this does not necessarily mean that the grammar has no LALR-only conflicts. Some other (nonprinted) trace for p_1 might shed light on such a conflict.

However, if the user iteratively removes the traced conflicts from his grammar and resubmits the grammar to the generator, the omitted trace will eventually appear. Thus, while the generator does not necessarily indicate all conflicts in a single run, it will eventually pinpoint all conflicts and indicate whether they are LR or LALR only. This seems to be an acceptable solution, since users (and we) can typically only cope with a few conflicts at a time anyway. In practice, we have never encountered an LR reduce–reduce conflict for which the algorithm, as presented in the prior subsection, has not yielded corresponding traces. In addition, we have seen only one practical grammar containing an LALR-only conflict, and it *was* indicated by the lack of corresponding traces.

The best solution would be never to produce an automaton with LALR-only conflicts. The state-splitting approach suggested by Pager [22, p. 38] could be

[2] It is well known that any read–reduce conflict present in the LALR automaton is also present in the LR automaton. Thus, corresponding read-item traces may always be produced.

employed to expand the LALR(1) automaton so that it is locally LR(1) where reduce–reduce conflicts occur (and thus any LALR-only conflicts are eliminated). Then, corresponding reduce traces may always be found. LALR is too complicated a notion for the average translator writing system user to spend time unraveling—he needs to know what an LR item is, how states are constructed by merging item sets, etc.; consequently, correcting LALR-only conflicts is usually not easy.

In contrast, the traces presented in this section require *no* concept of item or of item-set merging. The user need only know that the parser's interpretation of the input text is restricted to one-symbol look-ahead; the traces pinpoint how two different interpretations of the same text (prefix) can arise. Anyone who can understand the LR(k) definition can understand the traces (assuming LALR-only conflicts have been removed by state-splitting and corresponding reduce traces are provided). The majority of MetaWare™ translator writing system users know little or nothing of LR automaton construction, and many do not understand the LR(k) definition; yet they have profitably used the traces. An important aspect of our traces is that they are well engineered for humans, that is, they relate to the grammar via derivations, not states or items.

Kristensen and Madsen [18] show how to determine when a grammar is LR(k) (versus LALR(k)) by inspecting only the LR(0) automaton. They present diagnostic traces that distinguish between LR and LALR-only conflicts. Although their results are theoretically interesting, the LALR-only traces are not very useful in practice for the general translator writing system user, as we have already argued. The example traces they present rely heavily on the reader's technical knowledge of LR and LALR automata, and as such are appropriate only for persons well trained in LR theory.

In summary, from the algorithms presented in this section we may produce diagnostic debugging traces that pinpoint both LALR-only and LR conflicts in the grammar. Such traces are easily read by even a novice grammar designer, and they are an essential component of any well-engineered translator writing system.

8. CONCLUSION

The two relations **includes** and **reads** have been defined to capture the essential structure of the problem of computing LALR(1) look-ahead sets. The look-ahead sets may be computed from information obtained by two successive applications of a graph traversal algorithm, one to each relation. The algorithm is linear in the size of the relation to which it is applied. Thus, barring minor and constant improvements in underlying representations, we suspect that this is the best possible algorithm for this problem. We leave any proof or disproof of this conjecture for future research.

A conjecture and a theorem relating the appearance of nontrivial SCCs in the **includes** and **reads** relations to properties of the grammar were presented. The relations were shown to be valuable for printing information to aid the grammar designer in debugging non-LALR(1) grammars.

Finally, the popular NQLALR algorithm was formalized and proved incorrect. This should help others avoid the same mistake.

APPENDIX. PROOFS OF THEOREMS

PROOF OF THEOREM UNION

$$\begin{aligned} &\mathrm{LA}(q, A \to \omega)\\ &\quad = \{t \in T \mid [\alpha\omega]tz \vdash_{A\to\omega} [\alpha A]tz \vdash^* [S'\bot] \wedge \alpha\omega \text{ accesses } q\}\\ &\quad = \{t \in T \mid [\alpha A]tz \vdash^* [S'\bot] \wedge \alpha\omega \text{ accesses } q\} \qquad \text{(by Lemma 1, below)}\\ &\quad = \{t \in T \mid [\alpha A]tz \vdash^* [S'\bot], \quad \alpha \text{ accesses } p \wedge p \xrightarrow{\omega} q\}\\ &\quad = \bigcup\{\mathrm{Follow}(p, A) \mid (q, A \to \omega)\,\mathbf{lookback}\,(p, A)\}. \qquad \square \end{aligned}$$

LEMMA 1

$$\begin{aligned} &\{t \in T \mid [\alpha\omega]tz \vdash^* [\alpha\omega \mid \gamma]tz \vdash_{A\to\omega\gamma} [\alpha A]tz \vdash^* [S'\bot]\\ &= \{t \in T \mid \qquad\qquad\qquad\qquad\qquad [\alpha A]tz \vdash^* [S'\bot]\}. \end{aligned}$$

PROOF. Due to the LR(0) parser construction, if $[\alpha A]$ is a path, so is $[\alpha\omega]$ and $[\alpha\omega\gamma]$ and in fact for every $z \in T^*$, $[\alpha\omega]z \vdash^* [\alpha\omega \mid \gamma]z \vdash_{A\to\omega} [\alpha A]z$. □

PROOF OF THEOREM UP. We prove the following equivalent to Theorem Up (see Theorem Equivalent): $\mathrm{Follow}(p, A) = \bigcup\{\mathrm{Read}(p', B) \mid (p, A)\ \mathbf{includes}^*\ (p', B)\}$.

$$\begin{aligned} \mathrm{Follow}(p, A) &= \{t \in T \mid [\alpha A]tz \vdash^* [S'\bot], \alpha \text{ accesses } p\}\\ &= \{t \in T \mid [\alpha A]tz \vdash^* [\alpha']tz \vdash_{\text{read}} [\alpha' t]z \vdash^* [S'\bot], \alpha \text{ accesses } p\}. \end{aligned}$$

That is, reductions may occur in configuration $[\alpha A]tz$ before t is read. These reductions are specified more fully next:

$$[\alpha A]tz \vdash^* [\alpha']tz \vdash_{\text{read}} [\alpha' t]z$$

iff there exist $n \geq 0$ and productions $B_1 \to \beta_1 A\gamma_1$, $B_2 \to \beta_2 B_1\gamma_1, \ldots, B_n \to \beta_n B_{n-1}\gamma_n$ with $\gamma_i \Rightarrow^* \epsilon$ such that

$$\begin{aligned} &[\alpha A]tz \;\,= [\alpha_1\beta_1 A \mid]tz \vdash^* [\alpha_1\beta_1 A \mid \gamma_1]tz \vdash_{P1} &&\text{where } P1 = B_1 \to \beta_1 A\gamma_1\\ &[\alpha_1 B_1]tz = [\alpha_2\beta_2 B_1 \mid]tz \vdash^* [\alpha_2\beta_2 B_1 \mid \gamma_2]tz \vdash_{P2} &&\text{where } P2 = B_2 \to \beta_2 B_1\gamma_2\\ &[\alpha_2 B_2]tz = [\alpha_3\beta_3 B_3 \mid]tz \vdash^* [\alpha_3\beta_3 B_2 \mid \gamma_3]tz \vdash_{P3} &&\text{where } P3 = B_3 \to \beta_3 B_2\gamma_3\\ &\quad\vdots\\ &[\alpha_n B_n \mid]tz \vdash^* [\alpha_n B_n \mid \gamma_{n+1}]tz \vdash_{\text{read}} [\alpha_n B_n \gamma_{n+1} t]z = [\alpha' t]z. \end{aligned}$$

That is, possible reductions of ϵ (to the γ_i) are interspersed between other stack reductions until finally t is read. Thus,

$$\begin{aligned} &\mathrm{Follow}(p, A)\\ &\quad = \bigcup_{n=0,\infty}\{t \in T \mid [\alpha A]tz = [\alpha_1\beta_1 A \mid]tz \vdash^* \cdots \text{(as above)} \vdash^* [S'\bot],\\ &\qquad\qquad \alpha \text{ accesses } p,\\ &\qquad\qquad B_1 \to \beta_1 A\gamma_1, \quad B_2 \to \beta_2 A\gamma_2, \ldots, B_n \to \beta_n B_{n-1}\gamma_n,\\ &\qquad\qquad \text{and} \quad \gamma_i \Rightarrow^* \epsilon\}. \end{aligned}$$

By Lemma 1, the steps $[\alpha A]tz \vdash^* [a_n B_n]tz$ can be "ignored" since they are all reductions, obtaining

$$\begin{aligned}&\text{Follow}(p, A)\\ &= \bigcup_{n=0,\infty} \{t \in T \mid [\alpha_n B_n \mid]tz \vdash^* [\alpha_n B_n \mid \gamma_{n+1}]tz \vdash_{\text{read}} [\alpha_n B_n \gamma_{n+1} t]z\\ &\qquad \vdash^* [S'\bot],\\ &\qquad \alpha \text{ accesses } p,\ \alpha = \alpha_n \beta_n \beta_{n-1} \cdots \beta_1,\\ &\qquad B_1 \to \beta_1 A \gamma_1,\ B_2 \to \beta_2 B_1 \gamma_2, \ldots, B_n \to \beta_n B_{n-1} \gamma_n,\\ &\qquad \text{and}\quad \gamma_i \Rightarrow^* \epsilon\}.\end{aligned}$$

Finally, observe that

$$\text{Top}[\alpha_n] \xrightarrow{\beta_n} \text{Top}[\alpha_{n-1}] \cdots \text{Top}\,[\alpha_2] \xrightarrow{\beta_2} \text{Top}[\alpha_1] \xrightarrow{\beta_1} \text{Top}[\alpha]$$

so that

$$(\text{Top}[\alpha_n], B_n)\ \textbf{includes}^{-1}(\text{Top}[\alpha_{n-1}], B_{n-1}) \cdots \textbf{includes}^{-1}(\text{Top}[\alpha], A).$$

Set $p' = \text{Top}[\alpha_n]$ to obtain

$$\begin{aligned}&\text{Follow}(p, A)\\ &= \bigcup_{n=0,\infty} \{t \in T \mid [\alpha_n B_n \mid]tz \vdash^* [\alpha_n B_n \mid \gamma_{n+1}]tz \vdash_{\text{read}} [\alpha_n B_n \gamma_{n+1} t]z \vdash^* [S'\bot],\\ &\qquad (p, A)\ \textbf{includes}^n\ (p', B_n),\ \alpha_n \text{ accesses } p'\}\\ &= \bigcup_{n=0,\infty} \bigcup \{\text{Read}(p', B_n) \mid (p, A)\ \textbf{includes}^n\ (p', B_n)\}\\ &= \bigcup \{\text{Read}(p', B) \mid (p, A)\ \textbf{includes}^*\ (p', B)\}.\end{aligned}$$ □

Now, to prove Theorem Across below, the following observation is needed:

LEMMA 2

$$\begin{aligned}&\{t \in T \mid [\alpha]tz \vdash_{\text{read}} [\alpha t]z \vdash^* [S'\bot]\}\\ &= \{t \in T \mid \textit{Next}(\textit{Top}[\alpha], t) \textit{ is defined } (= \textit{Top}[\alpha t])\}.\end{aligned}$$

This follows from the definition of $\vdash_{\text{read}}$ and the construction of the LR(0) parser.

PROOF OF THEOREM ACROSS. We prove the equivalent result: $\text{Read}(p, A) = \bigcup \{\text{DR}(q, C) \mid (p, A)\ \textbf{reads}^*\ (q, C)\}$.

$\text{Read}(p, A) = \{t \in T \mid [\alpha A \mid]tz \vdash^* [\alpha A \mid \gamma]tz \vdash_{\text{read}} [\alpha A \gamma t]z \vdash^* [S'\bot],\ \alpha \text{ accesses } p\}$.

Now γ is of the form $C_1 \cdots C_n$, where $C_i \in N$ and $C_i \Rightarrow^* \epsilon$, so

$$\begin{aligned}&\text{Read}(p, A)\\ &= \bigcup_{n=0,\infty} \{t \in T \mid [\alpha A \mid]tz \vdash^* [\alpha A \mid C_1 \cdots C_n]tz \vdash_{\text{read}} [\alpha A C_1 \cdots C_n t]z \vdash^* [S'\bot],\\ &\qquad \alpha \text{ accesses } p,\ C_i \in N,\ C_i \Rightarrow^* \epsilon,\ 1 \le i \le n\}.\end{aligned}$$

Due to Lemma 1, the steps $[\alpha A \mid]tz \vdash^* [\alpha A \mid C_1 \cdots C_n]tz$ can be eliminated:

$$\begin{aligned}&\text{Read}(p, A)\\ &= \bigcup_{n=0,\infty} \{t \in T \mid [\alpha A C_1 \cdots C_n]tz \vdash_{\text{read}} [\alpha A C_1 \cdots C_n t]z \vdash^* [S'\bot],\\ &\qquad \alpha \text{ accesses } p,\ C_i \in N,\ C_i \Rightarrow^* \epsilon,\ 1 \le i \le n\}.\end{aligned}$$

Let $\alpha A C_1 \cdots C_{n-1}$ access q. Then by Lemma 2,

$$\begin{aligned}\mathrm{Read}(p, A) &\\ = \textstyle\bigcup_{n=0,\infty} \{t \in T \mid & \mathrm{Next}(\mathrm{Next}(q, C_n), t) \text{ is defined}, \alpha \text{ accesses } p,\\ & \alpha A C_1 \cdots C_{n-1} \text{ accesses } q,\\ & C_i \in N,\ C_i \Rightarrow^* \epsilon,\ 1 \le i \le n\}.\end{aligned}$$

The definition of DR combined with the fact that (p, A) **reads** $(\mathrm{Top}[\alpha A], C_1)$ **reads** $(\mathrm{Top}[\alpha A C_1], C_2) \cdots$ **reads** $(\mathrm{Top}[\alpha A \cdots C_{n-1}], C_n)$ yields

$$\begin{aligned}\mathrm{Read}(p, A) &= \textstyle\bigcup_{n=0,\infty} \bigcup \{\mathrm{DR}(q, C_n) \mid (p, A) \textbf{ reads}^n\ (q, C_n)\}\\ &= \textstyle\bigcup \{\mathrm{DR}(q, C) \mid (p, A) \textbf{ reads}^*\ (q, C)\}.\end{aligned}$$ □

Proof of Theorem Equivalent. Let $G\,x = \bigcup\{F'\,y \mid xR^*y\}$. Let $P(F)$ be the predicate

$$\forall x : F\,x = F'\,x \cup \textstyle\bigcup\{F\,y \mid xRy\}.$$

We show that

(1) $P(G)$;
(2) if $P(F)$ holds, then $\forall x : G\,x \subseteq F\,x$

thus establishing that $\forall x : G\,x =_s F'\,x \cup \bigcup\{G\,y \mid xRy\}$.

(a)
$$\begin{aligned}\forall x: G\,x &= \textstyle\bigcup\{F'\,y \mid xR^*y\}\\ &= F'\,x \cup \textstyle\bigcup\{F'\,z \mid xR^+z\}\\ &= F'\,x \cup \textstyle\bigcup\{\bigcup\{F'\,z \mid yR^*z\} \mid xRy\}\\ &= F'\,x \cup \textstyle\bigcup\{G\,y \mid xRy\}.\end{aligned}$$

(b) Assume $P(F)$ holds for some F. Thus

$$\forall x: F\,x = F'\,x \cup \textstyle\bigcup\{F\,x_1 \mid xRx_1\}.$$

We can apply the expression itself to $F\,x_1$ to obtain

$$\begin{aligned}\forall x: F\,x &= F'\,x \cup \textstyle\bigcup\{F'\,x_1 \cup \bigcup\{F\,x_2 \mid x_1Rx_2\} \mid xRx_1\}\\ &= F'\,x \cup \textstyle\bigcup\{F'\,x_1 \mid xR^1x_1\} \cup \bigcup\{F\,x_2 \mid xR^2x_2\}.\end{aligned}$$

By repeating the process, for any $n \ge 0$ we can show that

$$\forall x: F\,x = \textstyle\bigcup\{F'\,y \mid xR^{0\cdots n}y\} \cup \bigcup\{F\,x_{n+1} \mid xR^{n+1}x_{n+1}\}$$

where $R^{0\cdots n} = R^0 \cup \cdots \cup R^n$. Now if $z \in \bigcup\{F'\,y \mid xR^*y\}$, then $z \in \{F'\,y \mid xR^ny\}$ for some $n \ge 0$. Hence $G\,x \subseteq F\,x$. □

Proof of Theorem Reads-SCC. Let $(p_1, A_1), \ldots, (p_n, A_n)$ be the vertices of the SCC. By the definition of **reads**, without loss of generality, the following loop exists in the LR (0) automaton:

$$p_1 \xrightarrow{A_1} p_2 \xrightarrow{A_2} \cdots p_n \xrightarrow{A_n} p_1.$$

Let α access p_1. Then $\alpha(A_1 \cdots A_n)^*$ is an infinite set of prefixes of rightmost sentential forms that trace paths from the parser's start state. As a special case, $[\alpha A_1 \cdots A_n]$ is such a path. It indicates that the parser, with $p_1 = \text{Top}[\alpha]$ on the stack top, can reduce ϵ successively to $A_1, \ldots, A_n$ until $p_1 = \text{Top}[\alpha A_1 \cdots A_n]$ again appears on the stack top. But the parser has followed a loop without reading any input, and therefore will do so forever. Thus the parser is incorrect, so that the grammar cannot be LR(1). Now all LR(k) parsers for the grammar must contain a loop, similar to the one above, in which some multiple of $A_1 \cdots A_n$ can be read, since $\alpha(A_1 \cdots A_n)^*$ are all valid prefixes of rightmost sentential forms. Thus no LR(k) parser can be correct, so the grammar is not LR(k) for any k. □

ACKNOWLEDGMENTS

Our thanks go to Martin Chaney for an essential insight along the way. Also, the papers of Watt and Wetherell had an inspirational effect on us. Thanks go to Xerox Palo Alto Research Center and Bill McKeeman for the tools by which this manuscript was prepared. We thank the referees for their careful review of the manuscript and their helpful comments.

REFERENCES

1. Aho, A.V., and Johnson, S.C. LR Parsing. *Comput. Surv. 6*, 2 (June 1974), 99–124.
2. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1976.
3. Aho, A.V., and Ullman, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
4. Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling,* vols. 1 and 2. Prentice-Hall, Englewood Cliffs, N.J., 1972.
5. Alpern, B., Chaney, M., Fay, M., Pennello, T., and Radin, R. Translator writing system for the Burroughs B5700. Computer and Information Sciences, Univ. California, Santa Cruz, Calif., 1976.
6. Anderson, T., Eve, J., and Horning, J. Efficient LR(1) parsers. *Acta Inf. 2* (1973), 12–39.
7. Barrett, W., and Couch, J. *Compiler Construction: Theory and Practice.* Science Research Associates, Chicago, Ill., 1979.
8. Barrett, W., Meyers, R., and Roberts, D.D. Systems programming language/300. General Systems Div., Hewlett-Packard Co., Cupertino, Calif., 1979.
9. DeRemer, F.L. Practical translators for LR(k) languages. Ph.D. dissertation, Dep. Electrical Engineering, Massachusetts Institute of Technology, Cambridge, 1969.
10. DeRemer, F.L. Simple LR(k) grammars. *Commun. ACM 14*, 7 (July 1971), 453–460.
11. DeRemer, F. XPL distribution tape containing NQLALR translator writing system. Computer and Information Sciences, Univ. California, Santa Cruz, 1972.
12. DeRemer, F., and Pennello, T.J. The MetaWare™ TWS User's Manual. MetaWare, 412 Liberty St., Santa Cruz, Calif., 1981.
13. DeRemer, F., Pennello, T.J., and Meyers, R. A syntax diagram for (preliminary) Ada. *SIGPLAN Notices* (ACM) *15*, 7, 8 (July–Aug. 1980), 36–47.
14. Eve, J., and Kurki-Suonio, R. On computing the transitive closure of a relation. *Acta Inf. 8* (1977), 303–314.
15. Jensen, K., and Wirth, N. *Pascal User Manual and Report,* 2nd ed. Springer-Verlag, New York, 1974.
16. Johnson, S.C. YACC—Yet another compiler compiler. Tech. Rep. CSTR 32, Bell Labs., Murray Hill, N.J., 1974.
17. Knuth, D.E. On the translation of languages from left to right. *Inf. Control 8* (1965), 607–639.
18. Kristensen, B.B., and Madsen, O.L. Diagnostics on LALR(k) conflicts based on a method for LR(k) testing. *BIT,* to appear.

19. KRISTENSEN, B.B., AND MADSEN, O.L. Methods for computing LALR(k) lookahead. *ACM Trans. Program. Lang. Syst. 3*, 1 (Jan. 1981), 60–82.
20. LALONDE, W.R., LEE, E.S., AND HORNING, J.J. An LALR(k) parser generator. In *Proc. IFIP Congress 71*. Elsevier Science, New York, pp. 151–153.
21. MCKEEMAN, W.M., HORNING, J.J., AND WORTMAN, D.B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
22. PAGER, D. The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency. *Inf. Sci. 12* (1977), 19–42.
23. WATT, D.A. Personal communication, 1974.
24. WATT, D.A. Personal communication (class notes), 1976.
25. WETHERELL, C. A correction to DeRemer's SLR(1) parser constructor algorithm. Unpublished manuscript. Lawrence Livermore Labs., Livermore, Calif., 1977.

Received April 1981; revised January 1982; Accepted February 1982