Noncanonical SLR(1) Grammars

KUO-CHUNG TAI
North Carolina State University

Two noncanonical extensions of the simple LR(1) (SLR(1)) method are presented, which reduce not only handles but also other phrases of sentential forms. A class of context-free grammars called *leftmost* SLR(1) (LSLR(1)) is defined by using lookahead symbols which appear in leftmost derivations. This class includes the SLR(1), reflected SMSP, and total precedence grammars as proper subclasses. The class of LSLR(1) languages properly includes the deterministic context-free languages, their reflections, and total precedence languages. By requiring that phrases which have been scanned be reduced as early as possible, a larger class of context-free grammars called *noncanonical* SLR(1) (NSLR(1)) is defined. The NSLR(1) languages can be recognized deterministically in linear time using a two-stack pushdown automaton. An NSLR(1) parser generator has been implemented. Empirical results show that efficient NSLR(1) parsers can be constructed for some non-LR grammars which generate nondeterministic languages. Applications of the NSLR(1) method to improve the parsing and translation of programming languages are discussed.

Key Words and Phrases: context-free grammars, SLR(k) grammars, LR(k) grammars, noncanonical parsing, two-stack pushdown automata, deterministic context-free languages, nondeterministic languages, parsers, compilers
CR Categories: 4.12, 5.22, 5.23

1. INTRODUCTION

A bottom-up parsing method is said to be *canonical* if it reduces only handles (leftmost phrases) of sentential forms. A canonical bottom-up parsing method can be extended to a *noncanonical* method by allowing it to reduce handles as well as other phrases of sentential forms. Such an extension may enlarge the class of parsable grammars and possibly the class of recognizable languages. A formal study of noncanonical extensions of bottom-up parsing methods has been done by Szymanski and Williams [14].

This paper presents two noncanonical extensions of DeRemer's simple LR(1) (SLR(1)) method [5]. First, a class of context-free grammars called *leftmost* SLR(1) (LSLR(1)) is defined by using lookahead symbols which appear in leftmost derivations. This class includes the SLR(1), reflected SMSP (simple mixed strategy precedence) [1], and Colmerauer's total precedence (a noncanonical extension of Wirth and Weber's simple precedence method) [3] grammars as

This research was supported in part by the National Science Foundation under Grant MCS77-24582.
Author's address: Department of Computer Science, North Carolina State University, P.O. Box 5972, Raleigh, NC 27650.

proper subclasses. The class of LSLR(1) languages properly includes the deterministic context-free languages, their reflections, and total precedence languages. Then by requiring that phrases which have been scanned be reduced as early as possible, a larger class of context-free grammars called *noncanonical* SLR(1) (NSLR(1)) is defined. The NSLR(1) languages can be recognized deterministically in linear time using a two-stack pushdown automaton. With these two noncanonical extensions, some nondeterministic languages can be recognized deterministically in linear time.

The paper is organized as follows. Section 2 presents terminology and some basic definitions. A survey of previous work on noncanonical parsing is presented in Section 3. Section 4 contains a summary of the SLR(1) method. In Section 5, the class of LSLR(1) grammars is defined. In Section 6, the class of NSLR(1) grammars is defined and an algorithm for constructing NSLR(1) parsers is given. Section 7 shows the properties of LSLR(1) and NSLR(1) grammars. Section 8 discusses the implementation of an NSLR(1) parser generator and presents some experimental results. Finally, the applications of the NSLR(1) method to improve the parsing and translation of programming languages are discussed.

2. TERMINOLOGY

A *context-free* (CF) *grammar* is a quadruple (V_N, V_T, P, S), where V_N is a finite nonempty set of symbols called *nonterminals*, V_T is a finite set of symbols distinct from those in V_N called *terminals*, P is a finite set of pairs called *productions*, and S is a distinguished symbol in V_N called the *start symbol*. Each production is written $A \rightarrow x$ and has a left part A in V_N and a right part x in V^*, where $V = V_N \cup V_T$. V^* denotes the set of all strings composed of symbols in V, including the empty string ϵ. An ϵ-production is a production of the form $A \rightarrow \epsilon$.

The capital letters $A, B, \ldots, F, S$ denote nonterminals, X, Y, Z denote nonterminals or terminals, lowercase letters $a, b, c, \ldots, h$ denote terminals, and $u, v, \ldots, z$ denote strings in V^*. $|u|$ denotes the length of (number of symbols in) the string u; therefore $|\epsilon| = 0$. u_i denotes the ith symbol of u. ϕ denotes the empty set.

If $A \rightarrow x$ is a production and uAv is a string in V^+, then $uAv \Rightarrow uxv$. The transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^+$, and the reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. A *derivation* is a sequence of strings $w_0, w_1, \ldots, w_n$, where $n \geq 0$, such that $w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n$; it is written $w_0 \Rightarrow^* w_n$. A *rightmost* (or *canonical*) *derivation* is a derivation in which the rightmost nonterminal of each string is replaced to form the next; it is written $w_0 \underset{\text{rm}}{\Rightarrow}^* w_n$. Similarly, a *leftmost derivation* is written $w_0 \underset{\text{lm}}{\Rightarrow}^* w_n$. A string w is called a *sentential form* of G if $S \Rightarrow^* w$. A sentential form w is called a *left-sentential* (*right-sentential*) *form* if $S \underset{\text{lm}}{\Rightarrow}^* (\underset{\text{rm}}{\Rightarrow}^*)\ w$. A *sentence* is a sentential form containing only terminal symbols. The *language* $L(G)$ generated by G is the set of sentences, i.e., $L(G) = \{w \text{ in } V_T^* \mid S \Rightarrow^* w\}$.

A grammar G is said to be *ambiguous* if there is a sentence in $L(G)$ with two or more distinct leftmost (or rightmost) derivations; otherwise, G is said to be *unambiguous*. A symbol X in V $(X \neq S)$ is said to be *useless* if there does not exist a derivation of the form $S \Rightarrow^* uXv \Rightarrow^* uxv$, where u, v, and x are in V_T^*. G is said to be *ϵ-free* if either there is no ϵ-production in P, or there is exactly one

ϵ-production $S \rightarrow \epsilon$ and S does not appear in the right part of any production in P. G is said to be *cycle-free* if there is no derivation of the form $A \Rightarrow^+ A$ for any A in V_N. G is said to be *proper* if it is cycle-free, is ϵ-free, and has no useless symbols. For every CF grammar G, a pushdown automaton (PDA) can be constructed to recognize $L(G)$ [2]. A context-free language is said to be *deterministic* if it can be recognized by a deterministic PDA.

A string x is said to be a *phrase* of a sentential form uxv, where u and v are in V^*, if $S \Rightarrow^* uAv \Rightarrow uxv$. String uxv may be *reduced* to uAv by replacing x with A. This replacement is called a *reduction*. A leftmost phrase of a sentential form is called a *handle*.[1] If G is unambiguous, every sentential form of G has a unique handle. A *bottom-up parsing method* is an algorithm which constructs the reverse of a derivation by making a sequence of reductions from the input, a string in V_T^*, to S. *Canonical* bottom-up parsing methods reduce only handles of sentential forms, while *noncanonical* bottom-up parsing methods reduce handles as well as other phrases of sentential forms.

3. SURVEY OF EXISTING NONCANONICAL PARSING METHODS

Compared with canonical parsing, noncanonical parsing allows greater freedom to select a phrase for reduction and therefore has several advantages. First, the set of grammars which are deterministically parsable by noncanonical parsing is larger. Second, the set of languages defined by noncanonically parsable grammars may contain some nondeterministic languages. Some of the existing canonical parsing methods can be extended for noncanonical parsing without loss of parsing efficiency. A brief discussion of existing noncanonical parsing methods follows.

Noncanonical Extension of Simple Precedence (SP)

Colmerauer [3] defined *total precedence* relations, which are generalizations of the Wirth-Weber simple precedence relations [19], such that $<\cdot$ and $\cdot>$ indicate the left and right ends, respectively, of a phrase. By requiring that at most one precedence relation hold between any pair of symbols, at least one phrase of every sentential form is uniquely distinguished by the relations at its left and right ends. Colmerauer showed that the total precedence languages are incommensurate with both the deterministic languages and their reflections.

Noncanonical Extension of Bounded Context (BC)

A grammar is said to be *m, n bounded context* (BC(m, n)) if *every* phrase of any sentential form is uniquely distinguished by the m symbols to its left and the n symbols to its right [8]. Williams [18] derived a noncanonical extension of the BC method and defined a class of grammars, called the *m, n bounded context parsable* (BCP(m, n)) grammars, in which *at least one* phrase of any sentential form is uniquely distinguished by the m symbols to its left and the n symbols to its right. Currently, the BCP method is the most powerful of all parsing methods that have both decidability[2] and linear time parsability. However, this method is

[1] In [2] a handle is a leftmost phrase of a right-sentential form.

[2] Decidability means that the question of whether or not an arbitrary grammar is parsable by using a particular parsing method is decidable.

not practical because it requires that a large table be scanned to distinguish a phrase.

Noncanonical Extensions of LR(k)

Szymanski and Williams [14] attempted to construct a general framework for bottom-up parsers and, within that framework, to examine noncanonical extensions of some existing bottom-up parsing methods. One noncanonical extension of LR(k), which was suggested by Knuth [9] and called LR(k, t), requires that in any sentential form one of the leftmost t phrases be uniquely distinguished by its left context and the first k symbols of its right context. Szymanski and Williams showed that the LR(k, t) grammars are a superset of the LR(k) grammars, but the LR(k, t) languages are exactly the LR(k) languages (i.e., the deterministic languages).

Two other noncanonical extensions of LR(k) were studied by Szymanski and Williams. One of them, FSPA(k), requires that a finite *s*tate *p*arsing *a*utomaton be able to find a phrase in any sentential form by using a left-to-right scan with k symbols of lookahead. The other, which includes FSPA(k) as a proper subclass and is called *LR*(k, ∞), requires that at least one phrase of every sentential form can be found solely by examining its left context and the first k symbols of its right context. They showed that the question of whether an arbitrary CF grammar is FSPA(k) or LR(k, ∞) for any fixed value of k is undecidable and suggested that further restrictions to the FSPA method to achieve decidability should be considered.

Szymanski [13] proposed a noncanonical extension of SLR(k) called *RNP*(k),[3] which is a proper subclass of FSPA(k). With no ϵ-productions, the RNP(1) grammars (after a minor error in Algorithm 3.4 of [13] is corrected) are exactly the NSLR(1) grammars to be defined in this paper.[4]

Culik and Cohen [4] defined another extension of LR(k) grammars, called the *LR-regular* (LRR) grammars, in which the "lookahead information" for determining handles is represented by a finite number of regular sets. Using two scans, LRR grammars can be parsed deterministically in linear time. During the right-to-left "prescan" of the input string, the lookahead information is computed. This information is used in the succeeding left-to-right "main scan" to parse the input string.

Noncanonical Extensions for Concurrent Compilation

Fischer [7] investigated noncanonical extensions of several bottom-up parsing methods in a parallel environment, including simple precedence, simple mixed strategy precedence, and LR(k). He suggested that such methods could be used to produce practical parsers for parallel computers such as Illiac IV and CDC Star-100.

---

[3] The RNP(k) grammars are REDNEXT parsable using k symbols of lookahead, where REDNEXT$_k(A)$, for a nonterminal A, is defined as $\{y \text{ in } V^k \mid S \underset{lm}{\Rightarrow}^* wAyz \text{ for some } w \text{ and } z \text{ in } V^*\}$.

[4] RNP and NSLR methods are based on the same idea, but were developed independently in different approaches. Implicit in Section 3.3 of [13] is a parser construction algorithm that yields parsers isomorphic to the ones produced by the NSLR(1) parser construction algorithm of this paper.

4. SUMMARY OF THE SLR(1) METHOD

In this section the SLR(1) metod [5] is presented to provide a framework for understanding how SLR(1) parsers can be modified for noncanonical parsing.

An *SLR*(1) *parser* consists of a set Q of states and two functions, the *parsing action function* f and the *goto function* g. State s_0 is designated as the initial state. Assume that the symbol \$, not in V_T, is appended to the right end of each input string. The parsing action function f takes an arbitrary state s and an input symbol b as arguments, where b is a terminal or the symbol \$. The value of $f(s, b)$ is either **shift**, **reduce** i, **error**, or **accept**. The goto function g takes a state s and a symbol Y in V as arguments, and the value of $g(s, Y)$ is either a state or **error**.

To construct a parser for a grammar $G = (V_N, V_T, P, S)$, G is augmented with a new production $S' \rightarrow S$ and start symbol S', not in V_N. $S' \rightarrow S$ is assumed to be the zeroth production and productions in P are numbered 1, 2, . . . , and t, where $t = |P|$. Let P_i denote the ith production in P, $0 \leq i \leq t$, and let n_i denote the length of the right part of P_i. Define T_FOLLOW(A), for a nonterminal A, to be the set of terminals that can follow A in some sentential form, and if A can be the rightmost symbol of a sentential form, then \$ is included in T_FOLLOW(A). That is, T_FOLLOW(A) = $\{b$ in $V_T \cup \{\$\} \mid S'\$ \Rightarrow^* uAbv$ for some u and v in $V^* \{\$\}^*\}$.

Each state in an SLR(1) parser is represented by a set of *items* (or *configurations*) $[i, j]$ where i is the number of a production and j is the position of a special marker (here, a period) to be inserted in the right part of P_i. Let $P_i = A \rightarrow x$. An item $[i, j]$ is equivalent to the notation $[A \rightarrow u.v]$ where $u = x_1x_2 \cdots x_j$ and $v = x_{j+1}x_{j+2} \cdots x_{n_i}$ (both notations are used in this paper). If a state contains an item $[i, j]$ with $j < n_i$, then it has a *read transition* under the symbol x_{j+1} (called the *read symbol* for that item). If a state contains an item $[i, n_i]$, then it has a *reduce transition* for P_i.

For any set I of items, let CLOSURE(I) be defined as the smallest set satisfying the following properties: (1) every item in I is in CLOSURE(I), and (2) if $[A \rightarrow u.Bv]$ is in CLOSURE(I) and $B \rightarrow y$ is a production, then the item $[B \rightarrow .y]$ is in CLOSURE(I).

Construction Algorithm for SLR(1) Parsers

(1) Initially, let s_0 = CLOSURE({[0, 0]}) and $Q = \{s_0\}$ with s_0 "unmarked."
(2) For each unmarked state s in Q, mark it by performing the following steps:
  (2.1) For each i, $0 \leq i \leq t$, let

$$L_i = \begin{cases} \text{T\_FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \rightarrow x \\ \phi & \text{otherwise} \end{cases}$$

  be the *simple* 1-*lookahead set* associated with the reduce transition for P_i.
  (2.2) Let $L = \{Y \mid s$ contains $[i, j], j < n_i, P_i = A \rightarrow x$, and $x_{j+1} = Y\}$ be the *simple* 1-*lookahead set* associated with all read transitions from s.
  (2.3) If $L, L_0, L_1, \ldots,$ and L_t are not pairwise disjoint, then G is not SLR(1).
  (2.4) For each terminal b in L, set $f(s, b)$ = **shift**.
  If $L_0 = \{\$\}$, then set $f(s, \$)$ = **accept**.
  For each b in L_i, $1 \leq i \leq t$, set $f(s, b)$ = **reduce** i.

For each b in $V_T \cup \{\$\}$ but not in $L \cup L_0 \cup L_1 \cup \cdots \cup L_t$, set $f(s, b) =$ **error**.

(2.5) For each Y in L, compute GOTO(s, Y) as follows:
GOTO(s, Y) = CLOSURE$(\{[i, j+1] \mid s$ contains $[i, j], j < n_i, P_i = A \rightarrow x$, and $x_{j+1} = Y\})$.
If GOTO(s, Y) is not already in Q, then add it to Q as an unmarked state. Set $g(s, Y) =$ GOTO(s, Y).

(2.6) For each Y in V but not in L, set $g(s, Y) =$ **error**. □

Any state with read transitions only is called a *read* state. Any state with exactly one reduce transition and no read transitions is called a *reduce* state. States which are neither read states nor reduce states are called *inadequate* states. A CF grammar G is said to be *simple LR*(1) or *SLR*(1) if and only if every inadequate state in the SLR(1) parser for G has pairwise disjoint simple 1-lookahead sets associated with its read and reduce transitions.

SLR(1) Parsing Algorithm

Initially, the pushdown stack contains the initial state s_0.

(1) Determine the current input symbol b.
(2) Let s be the state at the top of the stack.
  (2.1) If $f(s, b) =$ **shift**, then remove b from the input, push the state $g(s, b)$ onto the stack, and go to (1).
  (2.2) If $f(s, b) =$ **reduce** i, **where** $P_i = A \rightarrow x$, then pop n_i states from the stack. A new state s' is then exposed as the top of the stack. Push the state $g(s', A)$ onto the stack and go to (1).
  (2.3) If $f(s, b) =$ **error**, then halt and declare error.
  (2.4) If $f(s, b) =$ **accept**, then halt and declare acceptance.

5. LEFTMOST SLR(1) GRAMMARS

Let inadequate states which do not have pairwise disjoint simple 1-lookahead sets be called *SLR*(1)-*inadequate* states. In each SLR(1)-adequate state, there exist parsing conflicts which cannot be resolved by using simple 1-lookahead sets. One way of resolving such parsing conflicts is to postpone decisions on "possible" phrases until phrases to the right have been fully reduced, because right contexts beginning with the same terminal may possibly be reduced to strings beginning with different nonterminals. By doing so, lookahead sets associated with reduce transitions from SLR(1)-inadequate states may contain "fully reduced" symbols instead of "unreduced" terminals.

Let LM_FOLLOW(A), for a nonterminal A, be the set of symbols that can follow A in some left-sentential form, and if A can be the rightmost symbol of a left-sentential form, then \$ is included in LM_FOLLOW(A). That is, LM_FOLLOW$(A) = \{Y$ in $V \cup \{\$\} \mid S'\$ \underset{\text{lm}}{\Rightarrow}^* uAYv$ for some u and v in $V^* \{\$\}^*\}$. Then LM_FOLLOW$(A)$ is the set of "fully reduced" symbols following A and can be used as the lookahead set associated with a reduce transition for $A \rightarrow x$.

Consider the grammar G_1 with the following productions:

$$S \rightarrow Aa \mid Bb$$
$$A \rightarrow \bar{A}A \mid \bar{A}$$
$$B \rightarrow \bar{B}B \mid \bar{B}$$

$$\bar{A} \rightarrow c$$
$$\bar{B} \rightarrow c$$

G_1 is not LR(k) for any k because the nonterminal $\bar{A}$ or $\bar{B}$ to which c should be reduced depends upon whether the last input symbol is a or b, respectively. Figure 1(a) shows the SLR(1) parser for G_1 in which s_1 is the only SLR(1)-inadequate state. Note that LM_FOLLOW($\bar{A}$) = {a, A} and LM_FOLLOW($\bar{B}$) = {b, B}. Since LM_FOLLOW($\bar{A}$) and LM_FOLLOW($\bar{B}$) are disjoint, parsing conflicts in s_1 can be resolved by using "fully reduced" lookahead symbols.

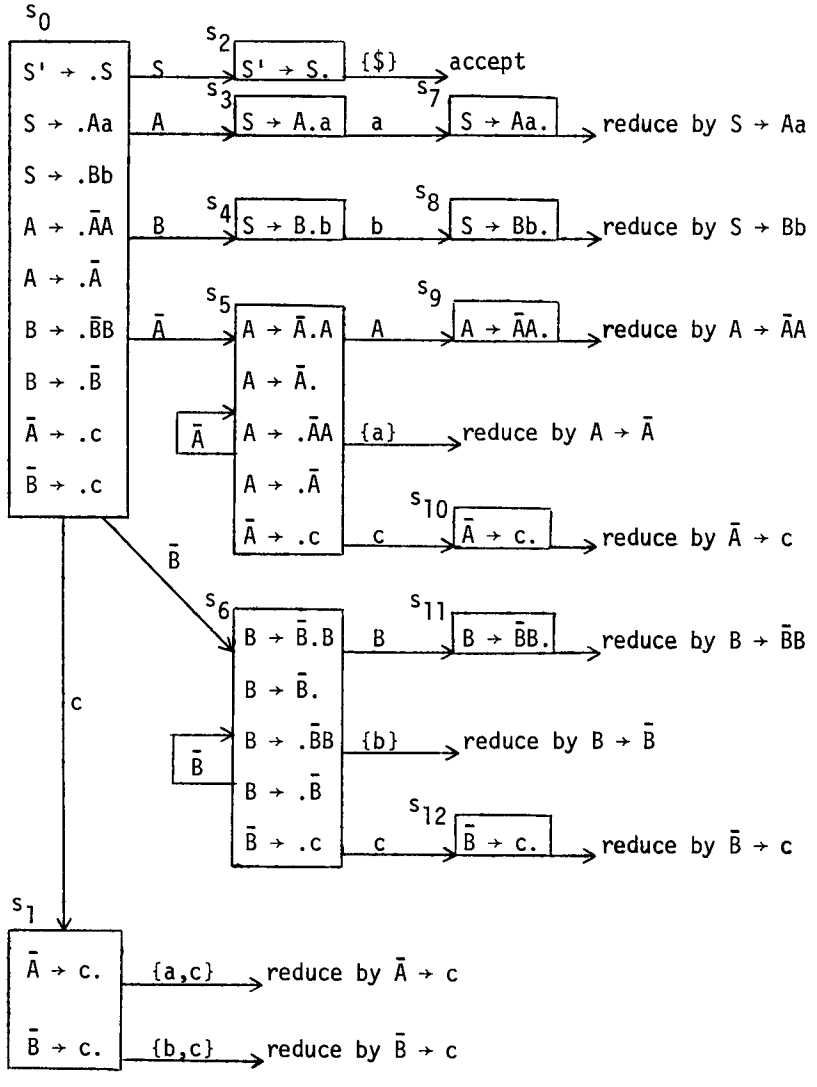

Fig. 1(a). SLR(1) parser for G_1
(Lookahead sets associated with reduce transitions are surrounded by braces.)

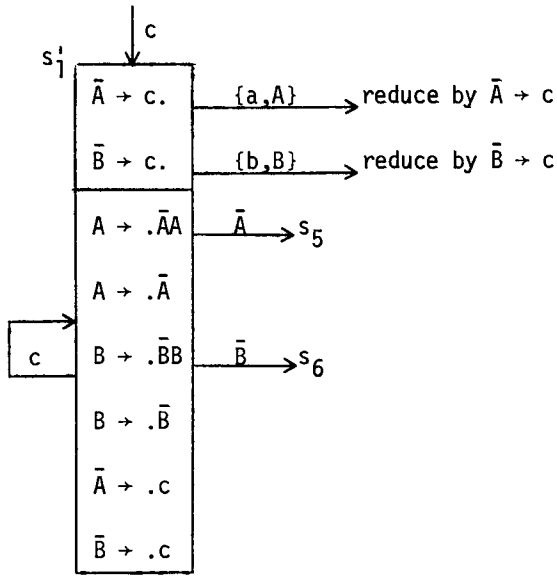

Fig. 1(b). Expanded s_1 with LSLR(1)-lookahead sets

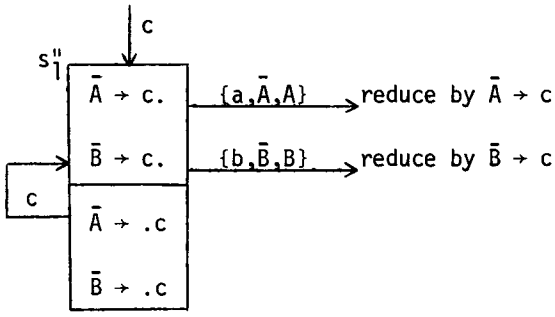

Fig. 1(c). Expanded s_1 with NSLR(1)-lookahead sets

Noncanonical SLR(1) Parsing

In order to obtain nonterminals for lookahead, noncanonical parsing must be applied to reduce nonleftmost phrases of sentential forms. Detailed modifications to SLR(1) parsers to allow noncanonical parsing is discussed later. After entering an SLR(1)-inadequate state, a noncanonical SLR(1) parser will postpone the decision on "possible" phrases, continue parsing the right context, and later obtain a nonterminal for lookahead. The parsing action function f of a noncanonical SLR(1) parser is different from that of a canonical SLR(1) parser. It takes a state s and a symbol Y in $V \cup \{\$\}$ (as opposed to $V_T \cup \{\$\}$) as arguments, and the value of $f(s, Y)$ is either **shift**, **reduce** i, **error**, or **accept**.

A two-stack pushdown automaton (2PDA) is needed for noncanonical parsing [2]. One of these two stacks, called SYMBOL_STK, acts as the input tape and contains the input string before the parsing starts. The other stack, called

STATE_STK, contains a sequence of states in a noncanonical SLR(1) parser during parsing. The parsing action function of a noncanonical SLR(1) parser takes the top symbols of both stacks as arguments. In a **shift** action, the parser pops one symbol from SYMBOL_STK and pushes one state (determined by the goto function) onto STATE_STK. In a **reduce** action, the parser pops zero or more states from STATE_STK and pushes one symbol (the resulting nonterminal) onto SYMBOL_STK. Thus, after a reduction is made, the resulting nonterminal acts as a lookahead symbol. If the next parsing action is still to make a reduction, another nonterminal will be pushed onto SYMBOL_STK. Therefore, the contents of SYMBOL_STK (from top to bottom) during parsing is a string in $V_N^* V_T^* \$$.

State Expansion

In order to allow noncanonical parsing, each SLR(1)-inadequate state is expanded by adding new items derived from nonterminals in the lookahead sets associated with reduce transitions. When successors of expanded SLR(1)-inadequate states are computed, new read transitions and possibly new states must be created.

In Figure 1(b) state s_1 is expanded to s_1' by adding all items derived from nonterminals in lookahead sets LM_FOLLOW($\bar{A}$) and LM_FOLLOW($\bar{B}$) associated with reduce transitions for $\bar{A} \rightarrow c$ and $\bar{B} \rightarrow c$, respectively. State s_1' has three new read transitions. By replacing s_1 with s_1', the resulting noncanonical parser (called the LSLR(1) parser for G_1) can deterministically parse sentences in $L(G_1)$. Figure 2 shows how the input string *cca* is parsed by the LSLR(1) parser for G_1. Initially, the *c*'s are shifted until the *a* is encountered. Then these *c*'s are reduced in a right-to-left manner. The first *c* is reduced to $\bar{A}$ after the second *c* has been reduced to *A*.

SLR(1)-inadequate states may be expanded for noncanonical parsing in different ways. The simplest method of state expansion is to add all items derived from nonterminals in the lookahead sets associated with reduce transitions. Adding new items to an SLR(1)-inadequate state may enlarge the set of read symbols and thus may create parsing conflicts, even though that state has pairwise disjoint lookahead sets associated with its reduce transitions. For example, let *s* be an SLR(1)-inadequate state and let L_i and L_j be the lookahead sets associated with reduce transitions for items $[i, n_i]$ and $[j, n_j]$ in *s*, respectively. Assume that L_i contains a symbol *Y* and L_j contains a nonterminal *A* which derives an item $[B \rightarrow .Yu]$, where $A \Rightarrow^* B$ and $A \neq Y$. If $[B \rightarrow .Yu]$ is added to *s*, then a parsing conflict arises because there are both read and reduce transitions under *Y*.

Nevertheless, some of the parsing conflicts due to state expansion can be eliminated by adding fewer items for noncanonical parsing. Consider the case that $i = j$ in the above example. Then *A* and *Y* are in L_i and $[B \rightarrow .Yu]$ is an item derived from *A*. The purpose of adding $[B \rightarrow .Yu]$ to *s* is to obtain *A* for lookahead to reduce by P_i. However, since the reduction by P_i can be distinguished by *Y*, it is unnecessary to add $[B \rightarrow .Yu]$ to *s* to postpone the reduction. Thus, items which are derived from nonterminals in L_i and have their read symbols in L_i are not necessarily needed for noncanonical parsing. Excluding this type of items from each SLR(1)-inadequate state may eliminate some parsing conflicts and therefore increase the chance of having pairwise disjoint lookahead sets.

STATE_STK	SYMBOL_STK	Action
s_0	cca$	shift c
s_0s_1'	ca$	shift c
$s_0s_1's_1'$	a$	reduce by $\bar{A} \to c$
s_0s_1'	$\bar{A}$a$	shift $\bar{A}$
$s_0s_1's_5$	a$	reduce by $A \to \bar{A}$
s_0s_1'	Aa$	reduce by $\bar{A} \to c$
s_0	$\bar{A}$Aa$	shift $\bar{A}$
s_0s_5	Aa$	shift A
$s_0s_5s_9$	a$	reduce by $A \to \bar{A}A$
s_0	Aa$	shift A
s_0s_3	a$	shift a
$s_0s_3s_7$	$	reduce by S → Aa
s_0	S$	shift S
s_0s_2	$	accept

Fig. 2. Parsing of the string *cca* using the LSLR(1) parser for G_1

Note that during state expansion of s, if L_i contains a nonterminal C such that $C \to \epsilon$ is in P, then $[C \to \epsilon.]$ is added to s (assume s does not contain $[C \to \epsilon.]$ before state expansion). Adding $[C \to \epsilon.]$ to s creates a new reduce transition whose associated lookahead set may further cause new items to be added to s. Thus the state expansion of s can be divided into two steps: (1) add items of the form $[C \to \epsilon.]$ to create new reduce transitions and (2) add items of the form $[B \to .y]$, where $y \neq \epsilon$, derived from nonterminals in the lookahead sets associated with reduce transitions.

Let FOLLOW(A), for a nonterminal A, be the set of symbols that can follow A in some sentential form, and if A can be the rightmost symbol of a sentential form, then \$ is included in FOLLOW(A). That is, FOLLOW(A) = $\{Y$ in $V \cup \{\$\} \mid S'\$ \Rightarrow^* uAYv$ for some u and v in $V^*\{\$\}^*\}$. Both T_FOLLOW($A$) and LM_FOLLOW($A$) are subsets of FOLLOW($A$). For any set I of items, let ϵ_CLOSURE(I) be defined as the smallest set satisfying the following properties: (1) every item in I is in ϵ_CLOSURE(I), and (2) if $[A \to x.]$ is in I and FOLLOW(A) contains B such that $B \to \epsilon$ is in P, then $[B \to \epsilon]$ is in ϵ_CLOSURE(I).

Construction Algorithm for Leftmost SLR(1) Parsers

(1) Initially, let s_0 = CLOSURE({[0, 0]}) and $Q = \{s_0\}$ with s_0 "unmarked."
(2) For each unmarked state s in Q, mark it by performing the following steps:
  (2.1) Perform steps (2.1) and (2.2) of the SLR(1) parser construction algorithm described in Section 4. If s has pairwise disjoint simple 1-lookahead sets, then go to (2.7).
  (2.2) Let $s = \epsilon$_CLOSURE(s) to add items for new reduce transitions. (This step is unnecessary if G is ϵ-free.)
  (2.3) For each i, $0 \le i \le t$,

$$LM_i = \begin{cases} \text{LM\_FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \to x \\ \phi & \text{otherwise} \end{cases}$$

$$F_i = \begin{cases} \text{FOLLOW}(A) & \text{if } s \text{ contains } [i, n_i] \text{ and } P_i = A \to x \\ \phi & \text{otherwise.} \end{cases}$$

  (2.4) For each i, $0 \le i \le t$,
    (2.4.1) Let $L_i = LM_i$ be the *LSLR(1)-lookahead set* associated with the reduce transition for P_i.
    (2.4.2) Add the set I_i of new items to s, where

$$I_i = \{[q, 0] \mid P_q = B \to y, B \text{ is in } F_i, y \neq \epsilon \text{ and } y_1 \text{ is not in } L_i\}.$$

  (2.5) Let $L = \{Y \mid s \text{ contains } [i, j], j < n_i, P_i = A \to x, \text{ and } x_{j+1} = Y\}$ be the *LSLR(1)-lookahead set* associated with all read transitions from s.
  (2.6) If L, L_0, L_1, ..., and L_t are not pairwise disjoint, then abort (thus G is not leftmost SLR(1)).
  (2.7) For each Y in L, set $f(s, Y)$ = **shift**. If $L_0 = \{\$\}$, then set $f(s, \$)$ = **accept**. For each Y in L_i, $1 \le i \le t$, set $f(s, Y)$ = **reduce** i. For each Y in $V \cup \{\$\}$ but not in $L \cup L_0 \cup L_1 \cup \cdots \cup L_t$, set $f(s, Y)$ = **error**.
  (2.8) For each Y in L, compute GOTO(s, Y) as follows:

$$\text{GOTO}(s, Y) = \text{CLOSURE}(\{[i, j+1] \mid s \text{ contains } [i, j], j < n_i, P_i = A \to x, \text{ and } x_{j+1} = Y\}).$$

  If GOTO(s, Y) is not already in Q, then add it to Q as an unmarked state. Set $g(s, Y)$ = GOTO(s, Y).
  (2.9) For each Y in V but not in L, set $g(s, Y)$ = **error**. □

A CF grammar G is said to be *leftmost SLR*(1) or *LSLR*(1) if and only if every SLR(1)-inadequate state in the LSLR(1) parser for G has pairwise disjoint LSLR(1)-lookahead sets associated with its read and reduce transitions. It is shown later that the class of LSLR(1) grammars includes the SLR(1), reflected SMSP, and total precedence grammars as proper subclasses. The LSLR(1) parsing algorithm is the same as the NSLR(1) parsing algorithm presented in Section 6.

6. NONCANONICAL SLR(1) GRAMMARS

The class of LSLR(1) grammars is defined by using lookahead symbols which appear in left-sentential forms. Thus, during LSLR(1) parsing, reductions in each SLR(1)-inadequate state are postponed until the right context has been fully reduced (in most cases).[5] However, it is possible to distinguish a phrase before its right context has been fully reduced. Furthermore, it is unnecessary to postpone

[5] For each SLR(1)-inadequate state s, the LSLR(1) parser construction algorithm does not always add to s all the items derived from nonterminals in the lookahead sets associated with reduce transitions. Therefore, some reductions can be made before the right context is fully reduced.

STATE_STK	SYMBOL_STK	Action
s_0	cca$	shift c
$s_0 s_1''$	ca$	shift c
$s_0 s_1'' s_1''$	a$	reduce by $\bar{A} \rightarrow c$
$s_0 s_1''$	$\bar{A}$a$	reduce by $\bar{A} \rightarrow c$
s_0	$\bar{A}\bar{A}$a$	shift $\bar{A}$
$s_0 s_5$	$\bar{A}$a$	shift $\bar{A}$
$s_0 s_5 s_5$	a$	reduce by $A \rightarrow \bar{A}$
$s_0 s_5$	Aa$	shift A
$s_0 s_5 s_9$	a$	reduce by $A \rightarrow \bar{A}A$
s_0	Aa$	shift A
$s_0 s_3$	a$	shift a
$s_0 s_3 s_7$	$	reduce by S → Aa
s_0	S$	shift S
$s_0 s_2$	$	accept

Fig. 3. Parsing of the string *cca* using the NSLR(1) parser for G_1

all reductions in an SLR(1)-inadequate state just because a few of them are in conflict. Therefore, the lookahead sets associated with reduce transitions should be enlarged so that phrases which have been scanned can be reduced as early as possible. Enlarging lookahead sets associated with reduce transitions will cause fewer items to be added for noncanonical parsing and fewer transitions and states to be created during state expansion.

Assume that s is an SLR(1)-inadequate state having a reduce transition for $A \rightarrow x$. Any symbol in FOLLOW(A) could be a lookahead symbol for $A \rightarrow x$. The LSLR(1)-lookahead set for $A \rightarrow x$ is LM_FOLLOW(A), a subset of FOLLOW(A). In order to make the reduction by $A \rightarrow x$ in s as early as possible, the lookahead set for $A \rightarrow x$ should be the largest subset R_A of FOLLOW(A) such that every symbol in R_A can distinguish the reduction by $A \rightarrow x$ in s. (How to construct R_A is shown later.) Note that if s has pairwise disjoint LSLR(1)-lookahead sets, then R_A should contain LM_FOLLOW(A).

Consider state s_1 in Figure 1(a), which has reduce transitions for $\bar{A} \rightarrow c$ and $\bar{B} \rightarrow c$. Since FOLLOW($\bar{A}$) = {a, A, $\bar{A}$, c} and FOLLOW($\bar{B}$) = {b, B, $\bar{B}$, c}, a

reduction by $\bar{A} \rightarrow c$ ($\bar{B} \rightarrow c$) can be distinguished by any symbol in $\{a, A, \bar{A}\}$ ($\{b, B, \bar{B}\}$). Using $\{a, A, \bar{A}\}$ and $\{b, B, \bar{B}\}$ as lookahead sets for $\bar{A} \rightarrow c$ and $\bar{B} \rightarrow c$, respectively, s_1 is expanded to s_1'' by adding items $[\bar{A} \rightarrow .c]$ and $[\bar{B} \rightarrow .c]$ (see Figure 1(c)). In state s_1'' reductions by $\bar{A} \rightarrow c$ and $\bar{B} \rightarrow c$ are postponed only if the current input symbol is c. Compared with s_1', s_1'' has larger lookahead sets associated with reduce transitions and a smaller lookahead set associated with all read transitions. By replacing s_1 with s_1'', the resulting noncanonical parser (called the NSLR(1) parser for G_1) can deterministically parse sentences in $L(G_1)$. Figure 3 shows how the input string *cca* is parsed by the NSLR(1) parser for G_1. The parsing of *cca* in Figure 3 is different from that in Figure 2 in steps (4) through (7). After the second c is reduced to $\bar{A}$ in step (3), in Figure 3 the first c is reduced to $\bar{A}$, while in Figure 2 the second c is further reduced to A.

Construction Algorithm for Noncanonical SLR(1) Parsers

The algorithm is the same as the LSLR(1) parser construction algorithm shown in Section 5, except that steps (2.4) through (2.6) of the latter are modified as follows:

(2.4)′ Let $R = \{Y \mid s \text{ contains } [i, j], j < n_i, P_i = A \rightarrow x \text{ and } x_{j+1} = Y\}$ be the set of read symbols of s. For each i, $0 \leq i \leq t$,

(2.4.1)′ Compute the largest subset R_i of F_i that can distinguish the reduction by P_i:

$$R_i = \{Y \text{ in } F_i \mid Y \text{ is neither in } R \text{ nor in } F_j, \text{ where } 0 \leq j \leq t \text{ and } j \neq i\}.$$

(2.4.2)′ Let $L_i = LM_i \cup R_i$ be the *NSLR(1)-lookahead set* associated with the reduce transition for P_i.

(2.4.3)′ Add the set I_i of new items to s, where $I_i = \{[q, 0] \mid P_q = B \rightarrow y, B \text{ is in } F_i, y \neq \epsilon \text{ and } y_1 \text{ is not in } L_i\}$.

(2.5)′ Let $L = \{Y \mid s \text{ contains } [i, j], j < n_i, P_i = A \rightarrow x, \text{ and } x_{j+1} = Y\}$ be the *NSLR(1)-lookahead set* associated with all read transitions from s.

(2.6)′ If L, L_0, L_1, ..., and L_t are not pairwise disjoint, then abort (thus G is not noncanonical SLR(1)).

A CF grammar G is said to be *noncanonical SLR(1)* or *NSLR(1)* if and only if every SLR(1)-inadequate state in the NSLR(1) parser for G has pairwise disjoint NSLR(1)-lookahead sets associated with its read and reduce transitions.

NSLR(1) Parsing Algorithm

An NSLR(1) parser needs two pushdown stacks SYMBOL_STK and STATE_STK. Initially, SYMBOL_STK contains the input string and STATE_STK contains the initial state s_0.

(1) Determine the symbol Y at the top of SYMBOL_STK.
(2) Let s be the state at the top of STATE_STK.
  (2.1) If $f(s, Y)$ = **shift**, then pop Y from SYMBOL_STK, push the state $g(s, Y)$ onto STATE_STK, and go to (1).
  (2.2) If $f(s, Y)$ = **reduce** i, where $P_i = A \rightarrow x$, then pop n_i states from STATE_STK, push A onto SYMBOL_STK, and go to (1).
  (2.3) If $f(s, Y)$ = **error**, then halt and declare error.
  (2.4) If $f(s, Y)$ = **accept**, then halt and declare acceptance.

THEOREM 1. *Let s be an SLR(1)-inadequate state. s has pairwise disjoint NSLR(1)-lookahead sets if and only if s has pairwise disjoint LSLR(1)-lookahead sets.*

PROOF. Let L, L_0, L_1, ..., and L_t be the NSLR(1)-lookahead sets of s. Then $L_i = LM_i \cup R_i$ for $0 \le i \le t$. By the definition of I_i, the set of read symbols for items in I_i is $\{y_1 \mid B$ is in F_i and $B \to y$ is in $P\} - L_i$. Since L_i contains LM_i and $\{y_1 \mid B$ is in F_i and $B \to y$ is in $P\} \cup LM_i = F_i$, it follows that the set of read symbols for items in I_i is $F_i - L_i$. Therefore, $L = R \cup (F_0 - L_0) \cup (F_1 - L_1) \cup \cdots \cup (F_t - L_t)$.

Let L', L_0', L_1', ..., and L_t' be the LSLR(1)-lookahead set of s. Then $L_i' = LM_i$ for $0 \le i \le t$ and $L' = R \cup (F_0 - L_0') \cup (F_1 - L_1') \cup \cdots \cup (F_t - L_t')$.

If. If L', L_0', L_1', ..., and L_t' are pairwise disjoint, then each L_i', $0 \le i \le t$, is disjoint from L' and L_j', $0 \le j \le t$ and $j \ne i$. Since $L_i' = LM_i$, each LM_i is disjoint from R, $F_j - L_j'$, and L_j', where $0 \le j \le t$ and $j \ne i$. Therefore, LM_i is a subset of R_i and $L_i = R_i$ for $0 \le i \le t$. By the definition of R_i, sets L, R_0, R_1, ..., and R_t are pairwise disjoint. It follows that L, L_0, L_1, ..., and L_t are pairwise disjoint.

Only If. If L', L_0', L_1', ..., and L_t' are not pairwise disjoint, then one of the following two cases must hold:

(1) For some i and j, $i \ne j$, L_i' and L_j' are not disjoint. Since L_i' and L_j' are subsets of L_i and L_j, respectively, it follows that L_i and L_j are not disjoint.

(2) For some i, $0 \le i \le t$, L_i' is not disjoint from L'. Then one of the following two cases must hold:

(2.1) L_i' is not disjoint from R. Since L_i' and R are subsets of L_i and L, respectively, L_i and L are not disjoint.

(2.2) L_i' is not disjoint from $F_j - L_j'$ for some $j \ne i$. Then there exists a symbol Y in both L_i' and $F_j - L_j'$. Since L_i' is a subset of L_i, Y is in L_i. By the definition of R_j, Y is not in R_j. Since $L_j = L_j' \cup R_j$, it follows that Y is in $F_j - L_j$. Thus L_i is not disjoint from $F_j - L_j$. $F_j - L_j$ is a subset of L and therefore L_i is not disjoint from L.

By (1) and (2), if L', L_0', L_1', ..., and L_t' are not pairwise disjoint, then L, L_0, L_1, ..., and L_t are not pairwise disjoint. Q.E.D.

Obviously, every LSLR(1) grammar is NSLR(1). Theorem 1, however, does not imply that every NSLR(1) grammar is LSLR(1). The reason is that the NSLR(1) parser for a grammar G may have fewer states than the LSLR(1) parser for G. It is possible that some of the extra states that the LSLR(1) parser has do not have pairwise disjoint LSLR(1)-lookahead sets. Consider the grammar G_2 with the following productions:

$$S \to cACe \mid dADe \mid A\bar{A} \mid B\bar{B}$$
$$A \to a$$
$$B \to a$$
$$\bar{A} \to fg$$
$$\bar{B} \to fh$$
$$C \to d$$
$$D \to d$$

Figure 4(a) shows part of the LSLR(1) parser for G_2. G_2 is not LSLR(1) because state s_2 does not have pairwise disjoint LSLR(1)-lookahead sets. However, there is no such state in the NSLR(1) parser for G_2. Figure 4(b) shows the corresponding part of the NSLR(1) parser for G_2. Since state s_1' is the only SLR(1)-inadequate

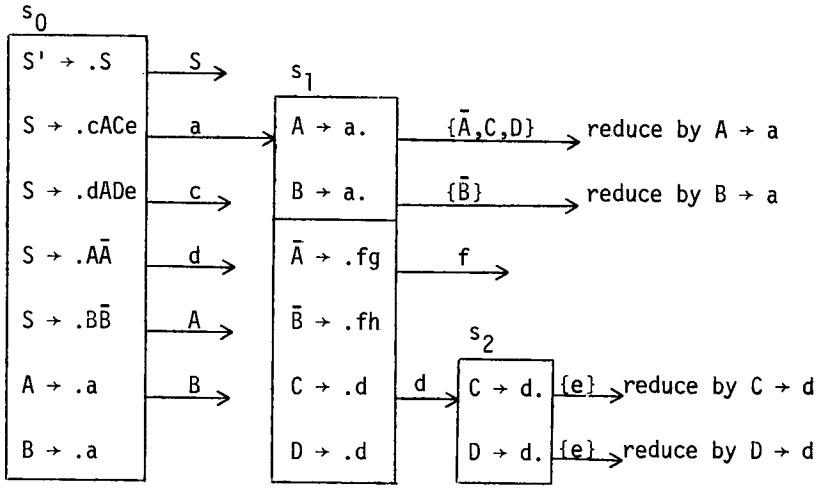

Fig. 4(a). Part of the LSLR(1) parser for G_2

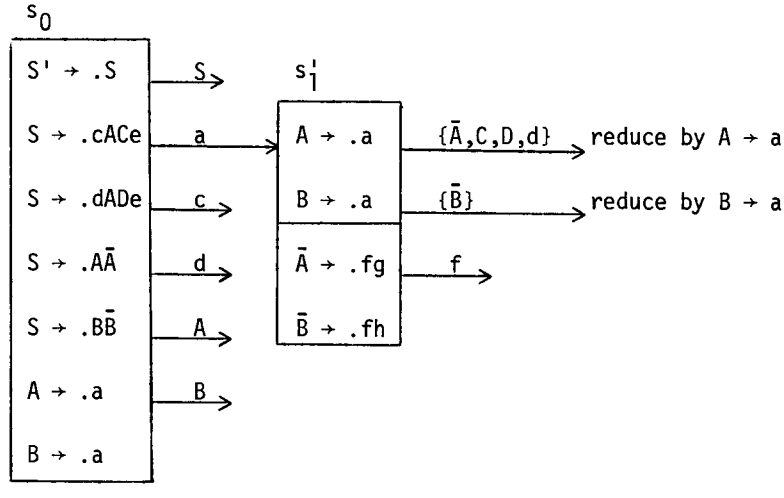

Fig. 4(b). Part of the NSLR(1) parser for G_2

state in the NSLR(1) parser for G_2 and has pairwise disjoint NSLR(1)-lookahead sets, G_2 is NSLR(1). Compared with s_1 in Figure 4(a), s_1' has a larger lookahead set for $A \rightarrow x$ and does not contain items $[C \rightarrow .d]$ and $[D \rightarrow .d]$, which induce the creation of state s_2 in Figure 4(a).

THEOREM 2. *The class of LSLR(1) grammars is a proper subclass of the NSLR(1) grammars.*

PROOF. The theorem follows from Theorem 1 and the above discussion. Q.E.D.

Because of the extension for noncanonical parsing, LSLR(1) and NSLR(1) parsers have some properties different from those of SLR(1) parsers. Most significantly, LSLR(1) and NSLR(1) parsers do not have the *correct prefix*

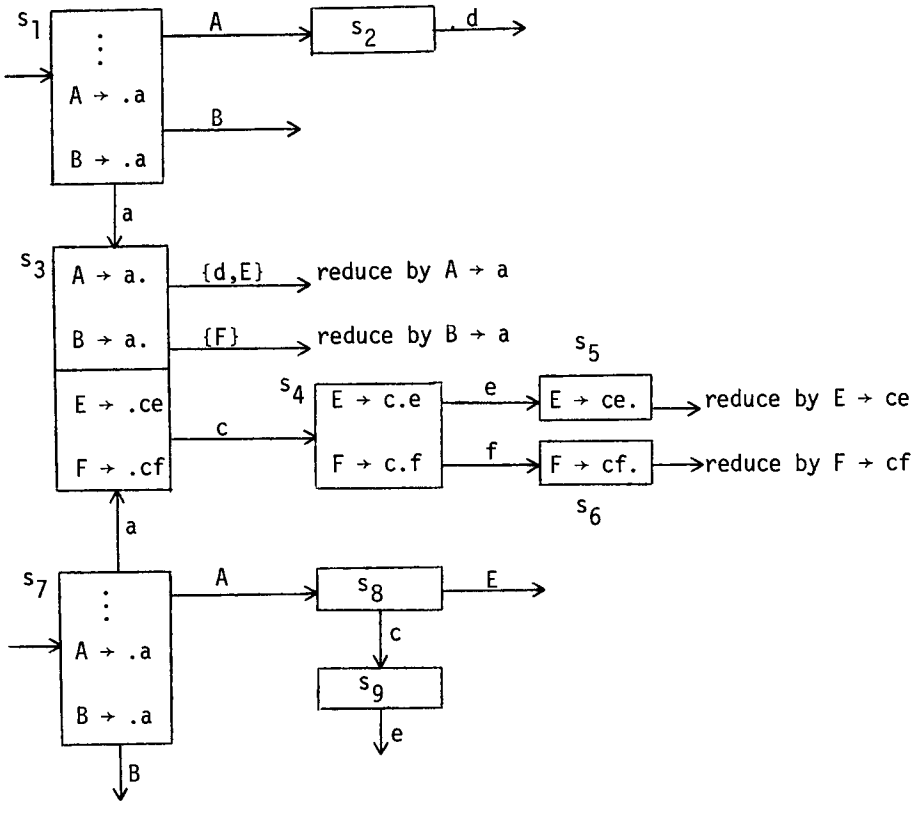

Fig. 5.

property that LR(k) parsers and their variants have. A parsing method has the correct prefix property if a syntax error is detected as soon as the parsed portion of the input is no longer a prefix of a valid sentence in the language being parsed. However, loss of the correct prefix property does not imply that LSLR(1) and NSLR(1) parsers will accept invalid input strings. LSLR(1) and NSLR(1) parsers merely detect syntax errors later than SLR(1) parsers.

The parser shown in Figure 5 illustrates the above point. Suppose that during the parsing of an input string, the top of STATE_STK begins with states s_3s_1 and that the remaining input string begins with ce. However, ce cannot legally follow the parsed portion of the input unless s_3 is entered through s_7. The parser cannot detect this error immediately. Instead, the parser accesses states s_4 and s_5 and makes a reduction by $E \rightarrow ce$. After this reduction, the tops of STATE_STK and SYMBOL_STK are s_3 and E, respectively. The parser then makes a reduction by $A \rightarrow a$ and later accesses state s_2. Now the top of SYMBOL_STK is E, and the parser detects the error because s_2 has no transition under E.

One possible advantage of the postponement of the detection of syntax errors is that more contextual information is available for error recovery or correction. The LR error recovery and correction techniques (e.g. [6, 11, 12, 16, 17]) can be extended to LSLR(1) and NSLR(1) parsers.

7. PROPERTIES OF LSLR(1) AND NSLR(1) GRAMMARS

It has been shown that the class of LSLR(1) grammars is a proper subclass of the NSLR(1) grammars. This section presents properties of LSLR(1) and NSLR(1) grammars and their languages.

The reflection of a grammar $G = (V_N, V_T, P, S)$ is $G^R = (V_N, V_T, P^R, S)$, where P^R is P with all right parts reversed. The reflection of a language L is defined as $L^R = \{a_n \cdots a_1 \mid a_1 \cdots a_n \text{ is in } L\}$. Then $L(G^R)$ is the reflection of $L(G)$ and vice versa. A set A is said to be *incommensurate* with another set B if A and B are not disjoint and A is not a subset of B or vice versa.

THEOREM 3. *The class of LSLR(1) (NSLR(1)) grammars includes each of the following classes of grammars as a subclass*: (1) *SLR(1) grammars*, (2) *reflections of SMSP (simple mixed strategy precedence) grammars, and* (3) *TP (total precedence) grammars.*

The proof of this theorem is given in the Appendix.

THEOREM 4. *The following three classes of languages are pairwise incommensurate*: (1) *deterministic languages*, (2) *reflections of deterministic languages, and* (3) *TP languages, and the class of LSLR(1) (NSLR(1)) languages includes each of these three classes of languages as a proper subclass.*

PROOF. Colmerauer [3] showed that the TP languages are incommensurate with both the deterministic languages and their reflections. Also, the deterministic languages are incommensurate with their reflections. Thus, these three classes of languages are pairwise incommensurate. By Theorem 3, the class of LSLR(1) languages includes the following three classes as subclasses: (1) SLR(1) languages, (2) SMSP^{R} languages, and (3) TP languages. The SLR(1) languages are exactly the deterministic languages [10]. The SMSP^{R} languages are the reflections of the SMSP languages. Since the SMSP languages are exactly the deterministic languages [1], it follows that the SMSP^{R} languages are the reflections of the deterministic languages. Therefore the class of LSLR(1) languages includes the deterministic languages, their reflections, and TP languages as subclasses. Since these subclasses are pairwise incommensurate, the class of LSLR(1) languages includes each subclass as a proper subclass. Q.E.D.

THEOREM 5. *The following three classes of grammars are pairwise incommensurate*: (1) *SLR(1) grammars*, (2) *$SMSP^R$ grammars, and* (3) *TP grammars, and the class of LSLR(1) (NSLR(1)) grammars includes each of these three classes of grammars as a proper subclass.*

PROOF. Obviously, the classes of SLR(1) grammars, SMSP^{R} grammars, and TP grammars are not pairwise disjoint. By Theorem 4, the classes of languages defined by these classes of grammars are pairwise incommensurate. Therefore these three classes of grammars are pairwise incommensurate. By Theorem 3, the class of LSLR(1) grammars includes these three classes of grammars as subclasses. Since these subclasses are pairwise incommensurate, the class of LSLR(1) grammars includes each subclass as a proper subclass. Q.E.D.

Figure 6 shows how different classes of grammars and languages are interrelated. As shown in Figure 6(a), there exist grammars G_2 which are NSLR(1) but not LSLR(1), and G_4 which are LSLR(1) but not SLR(1), SMSP^{R}, or TP. (G_4 is given in Section 8.) Whether the LSLR(1) languages are a proper subset of the NSLR(1) languages and whether the union of the deterministic languages, their

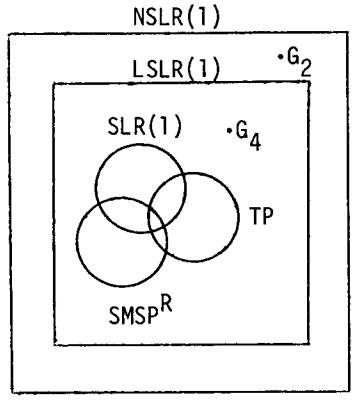

Fig. 6(a). Inclusion diagram for grammars

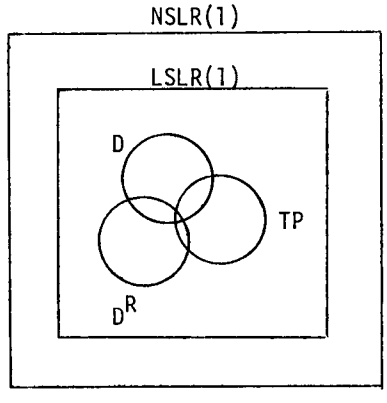

Fig. 6(b). Inclusion diagram for languages
(D ≡ {deterministic languages} ≡ {SLR(1) languages}; D^R ≡ {deterministic languages}^{R} ≡ {$SMSP^R$ languages}.)

reflections, and total precedence languages is a proper subset of the LSLR(1) languages are not known.

THEOREM 6. *Every NSLR(1) (LSLR(1)) grammar is unambiguous and is parsable in linear time on a two-stack pushdown automaton.*

PROOF. A CF grammar is said to be FSPA(k) if it is parsable by using a finite state parsing automaton with k symbols of lookahead. Obviously, every NSLR(1) grammar is FSPA(1). The theorem follows directly from Theorem 4.5 of Szymanski and Williams [14], which shows that every FSPA(k) grammar is unambiguous and is parsable in linear time on a two-stack pushdown automaton. Q.E.D.

8. EXPERIMENTAL RESULTS

An NSLR(1) parser generator has been implemented. Based on the NSLR(1) parser construction algorithm described in Section 6, an SLR(1) parser generator used at Cornell University was modified. The original SLR(1) parser generator has 599 PL/I statements. The resulting NSLR(1) parser generator has 662 PL/I statements, only 10 percent larger.

During the construction of an NSLR(1) parser, for each new state s, the parser generator first determines whether s is an inadequate state. If s is inadequate, then it determines whether s has pairwise disjoint simple 1-lookahead sets. If s is SLR(1)-inadequate, then it determines whether s has pairwise disjoint NSLR(1)-lookahead sets. Only when s is SLR(1)-inadequate are the NSLR(1)-lookahead sets computed and the state expansion technique applied. The computation of NSLR(1)-lookahead sets is not difficult. Since the parser generator uses bit strings to represent sets of symbols, only logical operations are needed for computing lookahead sets. The process of state expansion is similar to that of computing the closure of a state. Because the number of SLR(1)-inadequate states is usually small, the speed of NSLR(1) parser construction is only a little slower than that of SLR(1) parser construction.

The parser generator has been run with two sets of NSLR(1) grammars. The first set contains grammars which are not SLR(1) but still generate deterministic languages. It is known that every deterministic language can be generated by an SLR(1) grammar. However, some SLR(1) grammars for a deterministic language are considered "unnatural" or impractical because these grammars do not reflect the "phrase structure" imposed on the language for other phases of compilation, e.g., semantic processing and code generation. With the NSLR(1) method, compiler writers can have more chance of using "natural" grammars to define programming languages.

Consider the deterministic language $L_3 = \{da^n b^n \mid n > 0\} \cup \{da^{n+p} b^n \mid n, p > 0\}$ for which the semantic or code generation action on d depends upon whether the numbers of a's and b's in the input string are equal. Let G_3 be the grammar with the following productions:

$$S \rightarrow DE \mid \bar{D}F$$
$$D \rightarrow d$$
$$\bar{D} \rightarrow d$$
$$E \rightarrow aEb \mid ab$$
$$F \rightarrow aF \mid aE$$

Then $L(G_3) = L_3$ and d will be reduced to either D or $\bar{D}$ depending upon whether or not the numbers of a's and b's in the input string are equal. G_3 is not LR(k) for any k, but it is NSLR(1).

Consider another deterministic language $L_4 = \{b^{2n} \mid n > 0\} \cup \{b^{2n+1} \mid n > 0\}$ for which the semantic or code generation action on each symbol b depends upon whether the number of b's in the input string is even or odd. This process can be easily accomplished by using the grammar G_4 with the following productions:

$$S \rightarrow E \mid FB$$
$$E \rightarrow bbE \mid bb$$
$$F \rightarrow bbF \mid bb$$
$$B \rightarrow b$$

$L(G_4) = L_4$ and productions $S \rightarrow E$, $E \rightarrow bbE$ and $E \rightarrow bb$ generate strings in $\{b^{2n} \mid n > 0\}$, while other productions generate strings in $\{b^{2n+1} \mid n > 0\}$. G_4 is not

LR(k) for any k and not LR(k, t) for any k and t. Also, G_4 is neither SMSP^R nor TP.[6] But G_4 is LSLR(1) and thus NSLR(1).

The second set of NSLR(1) grammars contains grammars which generate nondeterministic languages. There has been no practical method for parsing all nondeterministic languages. The NSLR(1) method, however, can produce efficient parsers for some nondeterministic languages, including the reflections of deterministic languages. Thus compiler writers can have more freedom to select appropriate languages for their own purposes.

Consider the nondeterministic language $L_5 = \{a^n b^n c \mid n > 0\} \cup \{a^n b^{2n} d \mid n > 0\}$. L_5 can be generated by the NSLR(1) grammar G_5 with the following productions:

$$S \rightarrow Ec \mid Fd$$
$$E \rightarrow aB \mid aEB$$
$$B \rightarrow b$$
$$F \rightarrow a\bar{B}\bar{B} \mid aF\bar{B}\bar{B}$$
$$\bar{B} \rightarrow b$$

Consider a more complicated nondeterministic language $L_6 = \{a^m b^m c^n d^{n+p} \mid m, n, p > 0\} \cup \{a^m b^{2m} c^n d^n \mid m, n > 0\}$. L_6 can be generated by the NSLR(1) grammar G_6 with the following productions:

$$S \rightarrow AD \mid \bar{A}\bar{D}$$
$$A \rightarrow aAB \mid aB$$
$$B \rightarrow b$$
$$\bar{A} \rightarrow a\bar{A}\bar{B}\bar{B} \mid a\bar{B}\bar{B}$$
$$\bar{B} \rightarrow b$$
$$D \rightarrow Dd \mid \dot{E}d$$
$$\bar{D} \rightarrow E$$
$$\dot{E} \rightarrow cEd \mid cd$$

Details about the NSLR(1) parsers for G_1 through G_6 are given in Table I. Although G_1 through G_6 are too small to define "real" programming languages, certain portions of existing "real" grammars can be easily extended by using the NSLR(1) method.

Two problems concerning the optimization of NSLR(1) parsers need to be mentioned. One is the detection of "useless" lookahead symbols. Some lookahead symbols for inadequate states may be useless because of the use of approximations to exact lookahead information. (This is also true for SLR(1) parsers.) Moreover, a lookahead nonterminal for an SLR(1)-inadequate state s is useless if none of the items directly derivable from the nonterminal is in s after state expansion. Suppose that C is a nonterminal in some lookahead set L_i of s, $0 \leq i \leq t$. Define DFIRST(C) to be $\{Y \mid C \rightarrow x$ is a production and $x_1 = Y\}$. According to the NSLR(1) parser construction algorithm, if DFIRST(C) is a subset of L_i, then no items of the form $[C \rightarrow .x]$ are added to s during state expansion. In Figure 1(c)

[6] G_4 is not SMSP^R because $b\ \rho^*\alpha\ b$ and $b\ \rho^*\alpha\lambda^+\ b$, and G_4 is not TP because $b\ \alpha\ b$ and $b\ \rho^*\alpha\lambda^+\ b$.

Table I. About the NSLR(1) parsers for G_1 through G_6

Grammar	G_1	G_2	G_3	G_4	G_5	G_6
No. of productions[1]	9	11	9	8	9	14
No. of states[2]	13	23	16	11	18	27
No. of inadequate states	3	1	2	2	1	3
No. of SLR(1)-inadequate states	1	1	1	1	1	1
No. of states added for noncanonical parsing	0	1	0	1	0	1
No. of redundant states	2	2	4	1	2	4

[1] One extra production $S' \rightarrow S$ is added by the NSLR(1) parser generator.
[2] Reduce states are included.

lookahead nonterminals A and B for s_1'' are useless because s_1'' does not contain items $[A \rightarrow .\bar{A}A]$, $[A \rightarrow .\bar{A}]$, $[B \rightarrow .\bar{B}B]$, and $[B \rightarrow .\bar{B}]$.

Another problem is the detection of redundant transitions and states in NSLR(1) parsers. Some transitions and states will never be accessed during parsing because of the transitions and states added for noncanonical parsing. For example, consider state s_{10} in Figure 1(a). If s_1 is replaced with s_1'' in Figure 1(c), then the transition from s_5 to s_{10} is redundant because every c in the input string has been reduced to $\bar{A}$ before s_5 is entered. After the transition from s_5 to s_{10} is deleted, there is no transition to s_{10} and thus s_{10} becomes redundant. The same conclusions can be obtained for state s_{12} in Figure 1(a). Table I shows the number of redundant states in the NSLR(1) parsers for G_1 through G_6. Note that for each grammar, the number of states added to its original SLR(1) parser for noncanonical parsing is less than or equal to the number of states which are redundant in the resulting NSLR(1) parser. Therefore, extending an SLR(1) parser for noncanonical parsing may reduce the number of states in the parser. Since most redundant states are reduce states, the amount of storage wasted due to redundant states need not be significant if some optimization techniques are applied to the parser (see [2]).

9. CONCLUSION

Two noncanonical extensions of the SLR(1) method have been presented. Parsers for these two extensions are constructed by applying the state expansion technique to modify SLR(1) parsers for noncanonical parsing. The NSLR(1) method is superior to the SLR(1) method because it enlarges both the classes of acceptable grammars and languages. More importantly, the NSLR(1) method appears to be the most practical method yet described for deterministically parsing some

nondeterministic languages. Extensions of the NSLR(1) method to cover larger classes of context-free grammars are being investigated [15].

Most of the existing programming languages are deterministic languages. With the NSLR(1) method, these languages may be extended to nondeterministic languages without loss of parsing efficiency. The NSLR(1) method may also improve the parsing and translation of programming languages.

Assume that a grammar for a programming language is not SLR(1) just because some inadequate states need lookahead strings of length greater than 1. If lookahead sets contain only strings of length k, $k > 1$, then the parser needs an excessive amount of memory space. If lookahead sets contain strings of variable length, then the parser has to determine the number of lookahead symbols during parsing. However, this grammar can be easily transformed to an NSLR(1) grammar without serious loss of the "phrase structure" imposed by the original grammar. The NSLR(1) grammar G_2 shown in Section 6 can be considered as a transformed grammar of the SLR(2) grammar G_7 with the following productions:

$$S \rightarrow cACe \mid dADe \mid Afg \mid Bfh$$
$$A \rightarrow a$$
$$B \rightarrow a$$
$$C \rightarrow d$$
$$D \rightarrow d$$

As shown in Section 8, NSLR(1) grammars can be used to improve the translation of languages. One practical example in programming languages is the determination of the type of identifiers in a declaration statement. Consider the grammar G_8 with productions

⟨declaration⟩ → **var**⟨id-list⟩ : ⟨type⟩;

⟨id-list⟩ → identifier, ⟨id-list⟩ | identifier

⟨type⟩ → **integer** | **real**

G_8 defines the syntax of declaration statements in Pascal (with **integer** and **real** types only). In G_8, when each identifier is reduced, its type is unknown. With the NSLR(1) method, the reduction of each identifier can be postponed until the type of the identifier is determined. Let G_9 be the grammar with productions

⟨declaration⟩ → **var**⟨integer-id-list⟩⟨integer-type⟩; | **var**⟨real-id-list⟩⟨real-type⟩;

⟨integer-id-list⟩ → identifier, ⟨integer-id-list⟩ | identifier

⟨integer-type⟩ → : **integer**

⟨real-id-list⟩ → identifier, ⟨real-id-list⟩ | identifier

⟨real-type⟩ → : **real**

$L(G_9) = L(G_8)$ and G_9 is NSLR(1). In G_9, ⟨integer-id-list⟩ derives a list of identifiers with type **integer**, while ⟨real-id-list⟩ derives a list of identifiers with type **real**. Therefore the type of each identifier is determined by the production applied for reducing that identifier.

APPENDIX

To prove Theorem 3, several definitions and lemmas are given.

Given a CF grammar $G = (V_N, V_T, P, S)$, for all X, Y in V the relations α, λ, ρ are defined by

$X \alpha Y \equiv$ [there exists A in V_N and u, v in V^* such that $A \rightarrow uXYv$ is in P],
$X \lambda Y \equiv$ [there exists v in V^* such that $X \rightarrow Yv$ is in P],
$X \rho Y \equiv$ [there exists u in V^* such that $Y \rightarrow uX$ is in P].

For a nonterminal A, FOLLOW(A) = $\{Y \mid A\ \rho^*\alpha\lambda^*\ Y\}$ and LM_FOLLOW(A) = $\{Y \mid A\ \rho^*\alpha\ Y\}$. Define READ($A$) to be $\{Y \mid A\ \rho^*\alpha\lambda^+\ Y$ but not $A\ \rho^*\alpha\ Y\}$. Then LM_FOLLOW(A) and READ(A) are disjoint and FOLLOW(A) = LM_FOLLOW(A) $\cup$ READ(A).

LEMMA 1. *Let s be an SLR(1)-inadequate state in the LSLR(1) parser for G, where G is ϵ-free. If s does not have pairwise disjoint LSLR(1)-lookahead sets, then at least one of the following conditions is true:*

(1) *s contains two distinct items $[A \rightarrow u.]$ and $[B \rightarrow v.]$, and* LM_FOLLOW(A) *and* LM_FOLLOW(B) *are not disjoint.*

(2) *s contains two items $[A \rightarrow u.]$ and $[B \rightarrow v.]$, where $A \neq B$, and* LM_FOLLOW(A) *and* READ(B) *are not disjoint.*

(3) *s contains two items $[A \rightarrow u.]$ and $[B \rightarrow v.\bar{Y}w]$, where $v \neq \epsilon$, and* LM_FOLLOW(A) *contains a symbol Y such that $\bar{Y}\ \lambda^*\ Y$.*

PROOF. Let L, L_0, L_1, . . . , and L_t be the LSLR(1)-lookahead sets of s. If s does not have pairwise disjoint LSLR(1)-lookahead sets, then at least one of the following two cases must hold:

(a) For some i and j, $0 \leq i \neq j \leq t$, L_i and L_j are not disjoint. Since both L_i and L_j are nonempty, s contains items $[A \rightarrow u.]$ and $[B \rightarrow v.]$, where $P_i = A \rightarrow u$ and $P_j = B \rightarrow v$. Then L_i = LM_FOLLOW(A) and L_j = LM_FOLLOW(B). Therefore condition (1) of Lemma 1 is true.

(b) L is not disjoint from L_i for some i, $0 \leq i \leq t$. Assume $P_i = A \rightarrow u$. Then s contains the item $[A \rightarrow u.]$, L_i = LM_FOLLOW(A), and the set of read symbols for items derived from nonterminals in L_i is FOLLOW(A) − LM_FOLLOW(A). Note that READ(A) = FOLLOW(A) − LM_FOLLOW(A). By the definition of the LSLR(1)-lookahead set associated with all read transitions from s,

$$L = \{Y \mid s \text{ contains } [B \rightarrow v.] \text{ and } Y \text{ is in READ}(B)\} \cup \{Y \mid s \text{ contains } [B \rightarrow v.\bar{Y}w],\ v \neq \epsilon, \text{ and } \bar{Y}\ \lambda^*\ Y\}.$$

Since LM_FOLLOW(A) and L are not disjoint, it follows that one of the following two cases must hold:

(b.1) s contains an item $[B \rightarrow v.]$ and there exists a symbol Y in both LM_FOLLOW(A) and READ(B). The fact that LM_FOLLOW(A) and READ(A) are disjoint implies that $A \neq B$. Therefore condition (2) of Lemma 1 is true.

(b.2) s contains an item $[B \rightarrow v.\bar{Y}w]$, $v \neq \epsilon$, and there exists a symbol Y in LM_FOLLOW(A) such that $\bar{Y}\ \lambda^*\ Y$. Therefore condition (3) is true. Q.E.D.

LEMMA 2. *Let s be one of the states in the LSLR(1) parser for G, where G is ϵ-free. If s contains items $[A \rightarrow uX.Yv]$ and $[B \rightarrow .Yz]$, then $X\ \alpha\ Y$ and $X\ \rho^*\alpha\lambda^+\ Y$.*

PROOF. Since $A \rightarrow uXYv$ is a production, $X\ \alpha\ Y$. Note that s is entered after reading X. Since s contains $[B \rightarrow .Yz]$, one of the following conditions is true:

(1) s contains an item $[C \rightarrow wX.\bar{B}y]$ and $\bar{B}\ \lambda^*\ B$. Since $X\ \alpha\ \bar{B}$, $\bar{B}\ \lambda^*\ B$, and $B\ \lambda\ Y$, it follows that $X\ \alpha\lambda^+\ Y$.

(2) s contains an item $[C \rightarrow wX.]$, LM_FOLLOW(C) contains a symbol $\bar{B}$, and $\bar{B}\ \lambda^*\ B$. Since $X\ \rho\ C$, $C\ \rho^*\alpha\ \bar{B}$, $\bar{B}\ \lambda^*\ B$, and $B\ \lambda\ Y$, it follows that $X\ \rho^+\alpha\lambda^+\ Y$.

By (1) and (2), $X\ \rho^*\alpha\lambda^+\ Y$. Q.E.D.

LEMMA 3. *Let s be one of the states in the LSLR(1) parser for G, where G is ϵ-free. If s contains items $[A \rightarrow u.w]$ and $[B \rightarrow v.z]$, where u and v are in V^+, $u \neq v$, and w and z are in V^*, then there exist symbols X and Y such that $X\ \alpha\ Y$ and $X\ \rho^*\alpha\lambda^+\ Y$.*

PROOF. Since $[A \rightarrow u.w]$ and $[B \rightarrow v.z]$ are in the same state and $u \neq v$, v is a proper suffix of u or vice versa. Assume by symmetry that v is a proper suffix of u. Then $v = Yv'$ and $u = u'XYv'$, where X and Y are in V and u' and v' are in V^*. By construction of LSLR(1) parsers, there exists a state which contains items $[A \rightarrow u'X.Yv'w]$ and $[B \rightarrow .Yv'z]$. By Lemma 2, $X\ \alpha\ Y$ and $X\ \rho^*\alpha\lambda^+\ Y$. Q.E.D.

THEOREM 3. *The class of LSLR(1) grammars includes each of the following classes of grammars as a sublass*: (1) *SLR(1) grammars*, (2) *SMSP^R grammars*, *and* (3) *TP grammars.*

PROOF. (1) It is obvious that every SLR(1) grammar is LSLR(1).

(2) A grammar G is an SMSP^R grammar[7] if and only if
   (a) G is a proper CF grammar with no ϵ-productions.
   (b) $\rho^*\alpha \cap \rho^*\alpha\lambda^+ = \phi$.
   (c) If G has productions $A \rightarrow u$ and $B \rightarrow uYv$, then $A\ \rho^*\alpha\ Y$ does not hold.
   (d) If G has productions $A \rightarrow u$ and $B \rightarrow u$, where $A \neq B$, then there is no X such that $A\ \rho^*\alpha\ X$ and $B\ \rho^*\alpha\ X$.

Let G be a proper CF grammar with no ϵ-productions. Suppose G is not LSLR(1). Then there exists an SLR(1)-inadequate state s which does not have pairwise disjoint LSLR(1)-lookahead sets. By Lemma 1, one of the following conditions is true:

(2.1) s contains two distinct items $[A \rightarrow u.]$ and $[B \rightarrow v.]$, and LM_FOLLOW(A) and LM_FOLLOW(B) are not disjoint.

(2.1.1) $u = v$. Then $A \neq B$. Since LM_FOLLOW(A) and LM_FOLLOW(B) are not disjoint, there exists a symbol X such that $A\ \rho^*\alpha\ X$ and $B\ \rho^*\alpha\ X$. Therefore G is not SMSP^R.

(2.1.2) $u \neq v$. By Lemma 3, there exist symbols X and Y such that $X\ \alpha\ Y$ and $X\ \rho^*\alpha\lambda^+\ Y$. Therefore G is not SMSP^R.

(2.2) s contains two items $[A \rightarrow u.]$ and $[B \rightarrow v.]$, and LM_FOLLOW(A) and

[7] The definition of SMSP^R grammars is derived from the definition of SMSP grammars and from the fact that α, λ, and ρ are the reflections of α, ρ, and λ, respectively. G is an SMSP grammar [1] if and only if (a) G is a proper CF grammar with no ϵ-productions; (b) $\alpha\lambda^* \cap \rho^+\alpha\lambda^* = \phi$; (c) G has productions $A \rightarrow v$ and $B \rightarrow uYv$, then $Y\ \alpha\lambda^*\ A$ does not hold; (d) G has productions $A \rightarrow u$ and $B \rightarrow u$, where $A \neq B$, then there is no X such that $X\ \alpha\lambda^*\ A$ and $X\ \alpha\lambda^*\ B$.

READ(B) are not disjoint. Let X be the last symbol of both u and v, and let Y be a symbol in both LM_FOLLOW(A) and READ(B). Then $X\,\rho^+\alpha\ Y$ and $X\,\rho^+\alpha\lambda^+\ Y$. Therefore G is not SMSP^R.

(2.3) s contains two items $[A \rightarrow u.]$ and $[B \rightarrow v.\bar{Y}w]$, $v \neq \epsilon$, and LM_FOLLOW(A) contains a symbol Y such that $\bar{Y}\,\lambda^*\ Y$.

(2.3.1) $u \neq v$. By Lemma 3, there exist symbols X and Y such that $X\,\alpha\ Y$ and $X\,\rho^*\alpha\lambda^+\ Y$. Therefore G is not SMSP^R.

(2.3.2) $u = v$ and $\bar{Y} \neq Y$, i.e., $\bar{Y}\,\lambda^+\ Y$. Let X be the last symbol of both u and v. Since $X\,\rho\ A$ and $A\,\rho^*\alpha\ Y$, $X\,\rho^+\alpha\ Y$. Moreover, $X\,\alpha\ \bar{Y}$ and $\bar{Y}\,\lambda^+\ Y$ imply that $X\,\alpha\lambda^+\ Y$. Therefore G is not SMSP^R.

(2.3.3) $u = v$ and $\bar{Y} = Y$. Then G has productions $A \rightarrow u$ and $B \rightarrow uYw$. Since $A\,\rho^*\alpha\ Y$, G is not SMSP^R.

By (2.1), (2.2), and (2.3), every SMSP^R grammar is LSLR(1).

(3) A grammar G is a total precedence (TP) grammar [3] if and only if

(a) G is a proper CF grammar with no ϵ-productions.

(b) G is uniquely invertible, that is, all its productions have distinct right parts.

(c) $\alpha \cap \rho^*\alpha\lambda^+ = \phi$ and $\rho^+\alpha \cap \alpha\lambda^* = \phi$.

Let G be a uniquely invertible, proper CF grammar with no ϵ-productions. Suppose G is not LSLR(1). By Lemma 1, there exists an SLR(1)-inadequate state s for which one of the following conditions is true:

(3.1) s contains two distinct items $[A \rightarrow u.]$ and $[B \rightarrow v.]$, and LM_FOLLOW(A) and LM_FOLLOW(B) are not disjoint. Since G is uniquely invertible, $u \neq v$. By Lemma 3, there exist symbols X and Y such that $X\,\alpha\ Y$ and $X\,\rho^*\alpha\lambda^+\ Y$. Therefore G is not TP.

(3.2) s contains two items $[A \rightarrow u.]$ and $[B \rightarrow v.]$, and LM_FOLLOW(A) and READ(B) are not disjoint. By the same argument as in (3.1), G is not TP.

(3.3) s contains two items $[A \rightarrow u.]$ and $[B \rightarrow v.\bar{Y}w]$, $v \neq \epsilon$, and LM_FOLLOW(A) contains a symbol Y such that $\bar{Y}\,\lambda^*\ Y$. Let X be the last symbol of both u and v. Since $X\,\rho\ A$ and $A\,\rho^*\alpha\ Y$, it follows that $X\,\rho^+\alpha\ Y$. Moreover, $X\,\alpha\ \bar{Y}$ and $\bar{Y}\,\lambda^*\ Y$ imply that $X\,\alpha\lambda^*\ Y$. Therefore G is not TP.

By (3.1), (3.2), and (3.3), every TP grammar is LSLR (1).

By (1), (2), and (3), the class of LSLR(1) grammars includes the classes of SLR(1) grammars, SMSP^R grammars, and TP grammars as subclasses. Q.E.D.

ACKNOWLEDGMENTS

The author is very grateful to T.G. Szymanski for his valuable comments, to D.M. Tolbert for his helpful readings of this paper, and to S. Huang for his efforts on the implementation of the NSLR(1) parser generator.

REFERENCES

1. AHO, A.V., DENNING, P.J., AND ULLMAN, J.D. Weak and mixed strategy precedence parsing. *J. ACM 19*, 2 (April 1972), 225–243.
2. AHO, A.V., AND ULLMAN, J.D. *The Theory of Parsing, Translation and Compiling. Vols. I and II.* Prentice-Hall, Englewood Cliffs, N.J., 1972 and 1973.
3. COLMERAUER, A. Total precedence relations. *J. ACM 17*, 1 (Jan. 1970), 14–30.

4. CULIK, K., AND COHEN, R. LR-regular grammars—an extension of LR(k) grammars. *J. Comptr. Syst. Sci. 7* (1973), 66-96.
5. DEREMER, F.L. Simple *LR(k)* grammars. *Comm. ACM 14*, 7 (July 1971), 453-460.
6. DRUSEIKIS, F.C., AND RIPLEY, G.D. Error recovery for simple LR(k) parsers. Proc. ACM Annual Conf., Houston, Tex., 1976, pp. 396-400.
7. FISCHER, C.N. On parsing context free languages in parallel environments. Ph.D. Thesis, Tech. Rep. 75-237, Dep. Comptr. Sci., Cornell U., 1975.
8. FLOYD, R.W. Bounded context syntactic analysis. *Comm. ACM 7*, 2 (Feb. 1964), 62-67.
9. KNUTH, D.E. On the translation of languages from left to right. *Inform. Contr. 8*, 6 (Oct. 1965), 607-639.
10. MICKUNAS, M.D., LANCASTER, R.L., AND SCHNEIDER, V.B. Transforming LR(k) grammars to LR(1), SLR(1), and (1, 1) bounded right-context grammars. *J. ACM 23*, 3 (July 1976), 511-533.
11. MICKUNAS, M.D., AND MODRY, J.A. Automatic error recovery for LR parsers. *Comm. ACM 21*, 6 (June 1978), 459-465.
12. PENNELLO, T.J., AND DEREMER, F. A forward move algorithm for LR error recovery. Conf. Rec. 5th ACM Symp. Principles of Programming Languages, 1978, pp. 241-254.
13. SZYMANSKI, T.G. Generalized bottom-up parsing. Ph.D. Thesis, Tech. Rep. 73-168, Dep. Comptr. Sci., Cornell U., 1973.
14. SZYMANSKI, T.G., AND WILLIAMS, J.H. Noncanonical extensions of bottom-up parsing techniques. *SIAM J. Comptg. 5*, 2 (June 1976), 231-250.
15. TAI, K.C. Noncanonical LR parsing. In preparation.
16. TAI, K.C. Syntactic error correction in programming languages. *IEEE Trans. Software Eng. SE-4*, 5 (Sept. 1978), 414-425.
17. TAI, K.C. The recovery of parsing configurations for LR parsers. Tech. Rep. 77-06, Dep. Comptr. Sci., North Carolina State U., 1977.
18. WILLIAMS, J.H. Bounded context parsable grammars. *Inform. Contr. 28*, 4 (Aug. 1975), 314-334.
19. WIRTH, N., AND WEBER, H. EULER—a generalization of ALGOL and its definition, parts I and II. *Comm. ACM 9*, 1-2 (Jan.-Feb. 1966), 13-25, 89-99.

Received January 1979; revised May 1979