Information Processing Letters 31 (1989) 233–238
North-Holland

12 June 1989

SIMPLE COMPUTATION OF LALR(1) LOOKAHEAD SETS

Manuel E. BERMUDEZ

Computer & Info Sciences, University of Florida, Gainesville, FL 32611, U.S A.

George LOGOTHETIS

AT&T Bell Laboratories, Murray Hill, NJ 07974, U.S.A.

Communicated by E.C.R. Hehner
Received 2 December 1988
Revised 1 February 1988

Recent LALR(1) algorithms involve the computation (via graph algorithms) of certain Follow sets, for nonterminal transitions in the LR(0) automaton. In this paper we show that these Follow sets are merely the ordinary, well-known Follow sets, in a certain context-free grammar called G', which is related to the user's grammar G. By using G' instead of G, the well-known, simple algorithms ordinarily used for computing Follow sets (and SLR(1) parsers), can be used to obtain LALR(1) lookaheads. G' captures the essential structure of the problem of computing LALR(1) lookahead sets, and leads to the most straightforward and concise LALR(1) algorithm yet discovered.

Keywords: Context-free grammar, SLR(1), LALR(1), Follow set

1. Introduction

SLR(1) and LALR(1) parsers, invented by DeRemer [4,5], are common choices of compiler-writers for obtaining parsers for programming languages. These types of parsers are typically obtained by

(1) constructing the LR(0) automaton from the user's grammar, and

(2) computing sets of lookahead symbols for the reductions at each "conflict" state.

Many LALR(1) algorithms have been published. At first, DeRemer [4] proposed building the LALR(1) parser by first building a full LR(1) parser, and then merging states according to certain criteria. DeRemer also defined SLR(k) parsers [5] as those built using ordinary Follow sets as the lookaheads. Kristensen and Madsen [11] compute LALR(k) lookahead sets by resolving recursive equations. Other algorithms have been presented by LaLonde [12], Anderson, Eve and Horning [2], Korenjak [10], and Pager [14]. Our approach is similar to the more recent one of DeRemer and Pennello [6]. They compute LALR(1) lookahead sets by performing two graph traversals, the graphs being induced by relations on nonterminal transitions in the LR(0) state diagram. The graph traversals yield what they call Follow(p, A) sets, i.e. the set of symbols that can follow A, in the context of p. These Follow sets are then unioned appropriately to form the LALR(1) lookahead sets. Improved versions of their technique have appeared in [9,15].

SLR(1) Follow sets are computed from the user's grammar G using well-known algorithms [1,7,13]. The principal result presented in this paper is that Follow(p, A) can be similarly computed, *using the same algorithms, but on a different grammar*. This grammar, which we call G', is obtained from the LR(0) automaton, and embodies the notion of a symbol following A "in the context of" a certain state p. Thus LALR(1) lookahead set

computation can be carried out in SLR(1) style, i.e. by computing ordinary Follow sets, except that G' is used instead of G. This result eluded DeRemer and Pennello in [6], who did not notice that the essence of their two-graph algorithm is, after all, the ordinary computation of Follow sets, but carried out on a different grammar. Using G', the LALR(1) algorithm is as follows.

(1) Construct G'.

(2) Compute a Follow set for each nonterminal symbol in G'.

(3) Union the Follow sets appropriately to form the LALR(1) lookahead sets.

This is the simplest LALR(1) algorithm yet discovered. In fact, it is "simple" in two ways:

(1) literally, since the three steps shown above are quite straightforward, and

(2) figuratively, since our LALR(1) algorithm consists of computing Follow sets in the same way as for SLR(1) [1].

We now proceed to give some background and terminology, define G', argue its correctness, present an example, and conclude.

2. Background and terminology

We assume the reader is familiar with context-free grammars, LR parsing, and deterministic finite-state automata, in particular LR(0) automata. For a detailed presentation, the reader is referred to [1,7,8]. We adhere to the following (usual) notation.

$A, B, C, \ldots$	nonterminal symbols;
$t, a, b, c, \ldots$	terminal symbols;
$\ldots, X, Y, Z$	grammar symbols;
$\ldots, x, y, z$	terminal strings;
$\alpha, \beta, \gamma, \ldots$	strings of grammar symbols;
ε	the empty string;
$\perp$	the end-of-file symbol;
$A \to \omega$	a production in a CFG;
$\Rightarrow$	right-most derivation;
First(α)	$\{t \mid \alpha \Rightarrow^* tx$, for some $x\}$;
p, q, r, s	states in the LR(0) automaton.

[1] Recall that SLR stands for "Simple LR".

In the LR(0) automaton, $[p: X]$ denotes a transition from state p on symbol X, and Go$[p: X]$ is the target state of that transition. In general, $[p: \alpha]$ denotes the path that begins at state p, and "spells" α; Go$[p: \alpha]$ is the target state of that path. $[\alpha]$ denotes the path that begins at the start state of the LR(0) automaton, i.e. $[\alpha] = [\text{Start}: \alpha]$. A reduction by prodcution $A \to \omega$ in state q is denoted $(q, A \to \omega)$.

SLR(1) and LALR(1) parsers are obtained by first constructing the LR(0) parser, and then computing lookahead sets for "conflict" states, i.e. those with shift/reduce or reduce/reduce conflicts. One lookahead set, denoted LA$(q, A \to \omega)$, is computed for each reduction. To distinguish the type of parser, we use the notation SLRLA$(q, A \to \omega)$ and LALRLA$(q, A \to \omega)$. At parse time, the reduction will take place only if the next input symbol, t, is a valid lookahead symbol for that reduction, i.e. if $t \in \text{LA}(q, A \to \omega)$. SLR(1) and LALR(1) differ only in the lookahead sets. Both SLR(1) and LALR(1) lookahead symbols are those that can "follow", in a suitable sense, the reduction that causes the conflict. For a reduction $(q, A \to \omega)$, SLR(1) lookahead symbols are those that can follow nonterminal A in some sentential form, i.e. the symbols in the set

$$\text{SLRLA}(q, A \to \omega) = \text{Follow}(A) = \{t \mid S \Rightarrow^* \alpha A t x\}.$$

LALR(1) lookaheads also "follow" nonterminal A, but they do so in the context of state q. LALRLA$(q, A \to \omega)$ contains symbols t that can follow A in a sentential form $\alpha A t x$, provided that Go$[\alpha\omega] = q$, i.e. that $\alpha\omega$ spell a path from the LR(0) automaton's start state to q. Thus LALRLA$(q, A \to \omega)$ is the union of all sets of the form

$$\text{Follow}[p: A] = \{t \mid S \Rightarrow^* \alpha A t x, \text{ and Go}[\alpha] = p\}$$

such that Go$[p: \omega] = q$. Note that there are two definitions of "Follow"; we will let the argument of Follow (either a symbol or a transition) indicate which one is intended. Summarizing, LALR(1) lookaheads follow A in the context implied by state q, whereas SLR(1) lookaheads follow A independently of any such context. This "contextual

dependency" of LALR(1) is captured by G', which we define next.

3. Construction of G'

We begin by pointing out a well-known property of LR(0) parsers: for every nonterminal transition $[p_1 : A]$ in the LR(0) automaton, and for every production $A \to \omega$ in G, there exists a path of the form $[p_1 : \omega]$ such that (Go$[p_1 : \omega]$, $A \to \omega$) is a reduction. The situation is depicted in Fig. 1, where $\omega = X_1 \ldots X_n$.

We wish to construct G' so that its vocabulary of terminal and nonterminal symbols consists of the terminal and nonterminal *transitions* in the LR(0) automaton. G' will contain one production for each path of the type described in Fig. 1. The left-part of the production (in G') is nonterminal transition $[p_1 : A]$; the right part of the production (in G') consists of the transitions on the symbols that comprise the corresponding right-part in G. Thus, from Fig. 1 we obtain a production for G' of the form

$$[p_1 : A] \to [p_1 : X_1][p_2 : X_2] \ldots [p_n : X_n].$$

In particular, if the right-part is the empty string, we obtain a null production, $[p_1 : A] \to \varepsilon$. We stress again that the vocabulary of G' consists of the *transitions* in the LR(0) parser. Thus, grammar G' is similar to G, except for the fact that a certain amount of "symbol splitting" has taken place during construction of the LR(0) parser. Each symbol in G is split into many transitions in G'. Distinguishing among these various "split versions" of a nonterminal symbol is the essence of the LALR(1) technique. This is in contrast with the SLR(1) technique, in which no such distinction takes place. We now formalize the definition of G'.

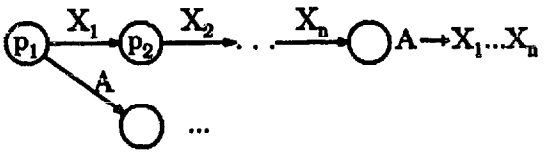

Fig. 1. A path for nonterminal transition $[p_1 : A]$.

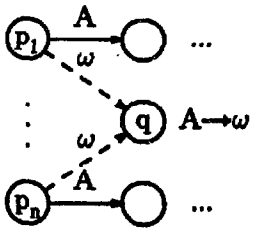

Fig. 2. Relationship between Follow$[p : A]$ and LALRLA$(q, A \to \omega)$.

Definition. For a CFG $G = (N, T, P, S)$, we define $G' = (N', T', P', S')$, where

$N' = \{[p : A] \mid \text{Go}[p : A] \text{ is defined}\}$,

$T' = \{[p : t] \mid \text{Go}[p : t] \text{ is defined}\}$,

$S' = [\text{Start} : S]$, and

$P' = \{[p_1 : A] \to [p_1 : X_1][p_2 : X_2] \ldots [p_n : X_n] \mid [p_1 : A] \in N', [p_i : X_i] \in N' \cup T'$ for $1 \leqslant i \leqslant n$, and $A \to X_1 X_2 \ldots X_n \in P\}$.

Our focus is the computation of Follow sets in G', i.e. Follow sets of nonterminal transitions. G' embodies the "state specific" information required for LALR(1), as we show next.

4. Correctness of G'

We now show that G' is correct, i.e. that it faithfully provides the Follow sets required to produce a correct LALR(1) parser. We begin by establishing the criteria for unioning Follow sets to form the lookahead sets. One lookahead set is computed for each reduction at each conflict state.

$$\text{LALRLA}(q, A \to \omega) = \{t \mid [r : t] \in \text{Follow}[p : A], \text{Go}[p : \omega] = q\}. \quad (1)$$

The lookahead set LALRLA$(q, A \to \omega)$ consists of those terminal symbols that appear in terminal transitions of the form $[r : t]$ which in turn appear in the Follow sets of all nonterminal transitions $[p : A]$, such that Go$[p : \omega] = q$. The situation is depicted in Fig. 2. The parser, when reducing ω to A in state q, can back out on ω to any state p_i,

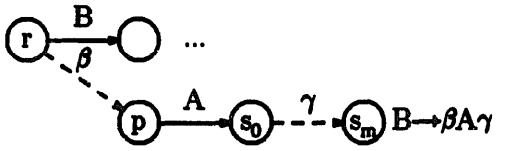

Fig. 3. Relationship between Follow[r: B] and Follow[p: A].

before advancing on A. Thus all Follow[p_i: A] sets contribute symbols to LALRLA(q, $A \to \omega$). These Follow sets contain terminal transitions, from which we extract the terminal symbols.

Next, we focus on the computation of Follow sets. It is well known that Follow sets are obtained by unioning other Follow sets, along with certain First sets. Specifically, recall that the ordinary computation of Follow sets for a context-free grammar G [1,7,13] involves the following two definitions:

$$\text{Follow}(B) \subseteq \text{Follow}(A)$$
$$\text{iff } B \to \beta A\gamma \text{ and } \gamma \Rightarrow^* \varepsilon; \quad (2)$$
$$\text{First}(X) \subseteq \text{Follow}(A)$$
$$\text{iff } B \to \alpha A\gamma X\delta \text{ and } \gamma \Rightarrow^* \varepsilon. \quad (3)$$

The reformulation of these two definitions for G' is shown, and explained, below.

$$\text{Follow}[r: B] \subseteq \text{Follow}[p: A]$$
$$\text{iff } B \to \beta A\gamma, \text{Go}[r: \beta] = p \text{ and } \gamma \Rightarrow^* \varepsilon; \quad (4)$$
$$\text{First}(X) \subseteq \text{Follow}[p: A]$$
$$\text{iff } B \to \alpha.A\gamma X\delta \in p \text{ and } \gamma \Rightarrow^* \varepsilon. \quad (5)$$

The situation for definition (4) is depicted in Fig. 3. Follow[r: B] $\subseteq$ Follow[p: A] if the parser, having reduced some phrase to A (landing in state s_0), can perform a sequence of reductions of the empty string to γ, and arrive at a state s_m, where it can reduce using production $B \to \beta A\gamma$. The reduction must back out through $A\gamma$ to state p, *and further through β to state r*, before advancing from r on B. The "backing out" must go through transition [p: A] to preserve the "context dependence" of LALR, versus the independence of such context of SLR. Hence the requirement that states r and p be connected via β. Assuming that $\beta = \beta_1 \ldots \beta_n$ and $\gamma = \gamma_1 \ldots \gamma_m$, G' will contain a production of the form

$$[r: B] \to [r: \beta_1][p_1: \beta_2] \ldots [p_{n-1}: \beta_n] [p: A][s_0: \gamma_1] \ldots [s_{m-1}: \gamma_m].$$

In the right part of this production, every symbol beyond [p: A] is nullable. This suffices to establish that Follow[r: B] $\subseteq$ Follow[p: A]. Thus definition (4) is the correct LALR criterion for unioning Follow sets of nonterminal transitions.

The situation for definition (5) is shown in Fig. 4. To ensure that First(X) $\subseteq$ Follow[p: A], not only must prodcution $B \to \alpha A\gamma X\delta$ exist in G, but LR(0) item $B \to \alpha.A\gamma X\delta$ must appear [2] in state p. The presence of this item ensures that from state p there exists a path on $A\gamma X$ such that the parser, after reducing some phrase ω to A (landing in r_0), can perform a sequence of reductions of the empty string, land in r_k, and then shift on any symbol in First(X). Assuming $\alpha = \alpha_1 \ldots \alpha_n$, $\gamma = \gamma_1 \ldots \gamma_k$, and $\delta = \delta_1 \ldots \delta_m$, G' contains a prodcution of the form

$$[p_0: B] \to [p_0: \alpha_1] \ldots [p_{n-1}: \alpha_n] [p: A][r_0: \gamma_1] \ldots [r_{k-1}: \gamma_k] [r_k: X][s_0: \delta_1] \ldots [s_{m-1}: \delta_m].$$

In the right part of this production, every transition separating [p: A] and [r_k: X] is nullable. Thus First(X) $\subseteq$ Follow[p: A], i.e. definition (5) is the correct LALR criterion for initializing Follow sets with the First sets.

[2] Recall that during the construction of the LR(0) automaton, a state is a collection of LR(0) items; an item is a production with a "dot" marker somewhere in its right part.

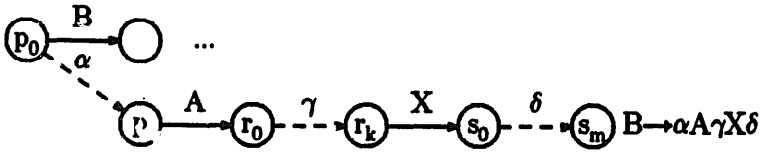

Fig. 4. Relationship between First(X) and Follow[p: A].

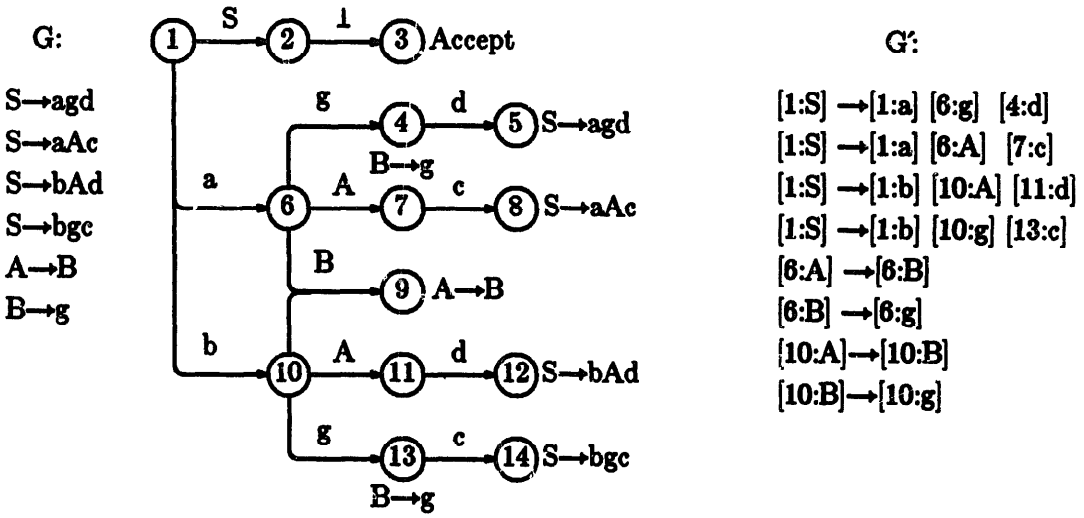

Fig. 5. An LALR(1) grammar that is non-SLR(1) and non-NQLALR(1).

Summarizing, Follow sets in G', when unioned appropriately, provide the correct LALR(1) lookahead sets. Hence our LALR(1) algorithm.

(1) Construct G'.

(2) Compute a Follow set for each nonterminal symbol in G'.

(3) Union the Follow sets appropriately to form the LALR(1) lookahead sets.

In practice, constructing G' requires no significant effort, since the structure of the LR(0) machine reflects the productions of G'. The computation of Follow sets has been studied extensively (see [1,7,13]). Finally, the Follow sets should be unioned "on demand", i.e. only for those LR(0) states that have conflicts.

5. An example

Here we present an example that illustrates how G' provides the "state dependent" lookahead information required by LALR, versus the "state independent" lookaheads of SLR. The example was used by DeRemer and Pennello [6] to show the inadequacy of an "oversimplification" technique that they called "Not-Quite" LALR(1). SLR(1) and NQLALR(1) are unrelated classes of CFG's (see [3]), although both are subsets of LALR(1). The grammar shown in Fig. 5 is neither SLR(1) nor NQLALR(1), but it is LALR(1).

We begin with the SLR(1) analysis. For both conflict states 4 and 13, Follow(B) is required. Using definitions (2) and (3),

$$\text{Follow}(B) \supseteq \text{Follow}(A) = \{c, d\}.$$

In neither case do these resolve the conflict. The grammar is not SLR(1). For the sake of brevity we do not elaborate on the (rather subtle) details of the NQLALR(1) technique. The NQLALR(1) lookahead sets are

$$\text{NQLA}(4, B \to g) = \{c, d\} \quad \text{and}$$
$$\text{NQLA}(13, B \to g) = \{c, d\}.$$

Clearly, neither of these resolves the conflicts at states 4 and 13. Roughly, the reason is that the NQLALR(1) technique "forgets" how state 9 is entered. State 9 can be entered from state 6 or from state 10, depending on whether the conflict in state 4 or state 13 is currently under consideration. The technique then "backs out" of state 9 *both ways* (i.e. landing in states 7 and 11) for each conflict. Thus both "c" and "d" appear in each lookahead set. The grammar is not NQLALR(1).

In contrast, G' has separate productions for nonterminal transitions [6 : A] and [10 : A], as well as for transitions [6 : B] and [10 : B] that lead into state 9. Thus, with G', Follow[6 : A] = {[7 : c]}, and Follow[10 : A] = {[11 : d]}, yielding

$$\text{LALRLA}(4, B \to g) = \{c\} \quad \text{and}$$
$$\text{LALRLA}(13, B \to g) = \{d\}.$$

Clearly, these lookahead sets solve the conflicts in states 4 and 13, respectively. The grammar is LALR(1).

6. Conclusions

We have presented an algorithm for computing LALR(1) lookahead sets that is the simplest to date. The algorithm is based on the context-free grammar G', which is structurally similar to the user's grammar G, but embodies the "symbol splitting" that is inherent in the construction of the LR(0) automaton. G' contains exactly the lookahead information required for a correct LALR(1) construction. We have formally defined G', proven its correctness, and illustrated it with an example.

The importance of G' is that it permits the use of efficient bit-matrix techniques for computing Follow sets, that are available in the literature. Thus, LALR(1) is "simple", both in the sense that it is straightforward, and in the sense that it is as easy as S(imple)LR, if one deals with the appropriate grammar.

We conjecture that our result extends to LALR(k), with $k > 1$. Specifically, we conjecture that Follow$_k$ sets, from G', yield correct LALR(k) lookahead sets.

7. References

[1] A. Aho, R. Sethi and J. Ullman, *Compilers. Principles, Techniques and Tools* (Addison-Wesley, Reading, MA, 1986).

[2] T. Anderson, J. Eve and J.J. Horning, Efficient LR(1) parsers, *Acta Inform.* 2 (1973) 12–39.

[3] M. Bermudez and K. Schimpf, On the (non)relationship between SLR(1) and NQLALR(1) grammars, *ACM TOPLAS* 10 (1988) 338–342.

[4] F. DeRemer, *Practical Translators for LR(k) Languages*, Ph.D. Thesis, Department of Electrical Engineering, MIT, Cambridge, MA, 1969.

[5] F. DeRemer, Simple LR(k) grammars, *Comm. ACM* 14 (1971) 453–460.

[6] F. DeRemer and T. Pennello, Efficient computation of LALR(1) look-ahead sets, *ACM TOPLAS* 4 (1982) 615–649.

[7] C. Fischer and R.J. LeBlanc, *Crafting a Compiler* (Benjamin/Cummings, Menlo Park, 1988).

[8] M. Harrison, *An Introduction to Formal Language Theory* (Addison-Wesley, Reading, MA, 1978).

[9] F. Ives, Unifying view of recent LALR(1) algorithms, In: *Proc. SIGPLAN '86 Symp. on Compiler Construction*, Palo Alto (June 1986).

[10] A.J. Korenjak, A practical method for constructing LR(k) processors, *Comm. ACM* 12 (1969) 613–623.

[11] B.B. Kristensen and O.L. Madsen, Methods for computing LALR(k) lookahead, *ACM TOPLAS* 3 (1981) 60–82.

[12] W.R. LaLonde, *An Efficient LALR Parser Generator*, Tech. Rept. 2, Computer Systems Research Group, Univ. of Toronto, 1971.

[13] P.M. Lewis, D.J. Rosenkrantz and R.E. Stearns, *Compiler Design Theory* (Addison-Wesley, Reading, MA, 1976).

[14] D. Pager, A practical general method for constructing LR(k) parsers, *Acta Inform.* 7 (1977) 249–268.

[15] J.C.H. Park, K.M. Choe and C.H. Chang, A new analysis of LALR formalisms, *ACM TOPLAS* 7 (1985) 159–175.