Practical Arbitrary Lookahead LR Parsing

MANUEL E. BERMUDEZ

University of Florida, Gainesville, Florida 32611

AND

KARL M. SCHIMPF*

University of California, Santa Cruz, California 95064

Received January 2, 1988; revised November 12, 1988

We present a practical technique for computing lookahead for an LR(0) parser, that progressively attempts single-symbol, multi-symbol, and arbitrary lookahead. The technique determines the amount of lookahead required, and the user is spared the task of guessing it. The class of context-free grammars defined by our technique is a subset of the LR-regular grammars; we show that unlike LR-regular, the problem of determining whether an arbitrary grammar is in the class, is decidable. When restricted to k-symbol lookahead, the technique has the power of LALR(k) parsers. It has been successfully used to resolve multi-symbol lookahead conflicts in grammars for FORTRAN, Ada, C, COBOL, and PL/I, and its performance compares favorably with that of two well-known, commercially available parser generators. © 1990 Academic Press, Inc.

1. INTRODUCTION

Many researchers have examined LR-based parsing techniques, i.e., parsers constructed automatically from context-free specifications, based on LR(k) methods. Knuth defined LR(k) grammars [12], but the technique was not considered practical until Korenjak [13] produced reasonable-sized parsers, and until DeRemer [6, 7] devised the SLR and LALR methods. Since then, LR techniques, and particularly the LALR(1) method, have become commonplace. The usual strategy for constructing an LR-based parser is as follows:

(1) Construct the LR(0) parser. In most cases the LR(0) parser is *nondeterministic*, i.e. the LR(0) push-down automaton has one or more *inconsistent* states. Inconsistent states have one or more *conflicts*, each of which can be a *shift/reduce* conflict or a *reduce/reduce* conflict.

(2) Provide determinism to the LR(0) parser by adding "lookahead" sets of strings to each inconsistent state. This is done by partitioning the input strings

* Present address: Reasoning Systems, Inc., Palo Alto, CA 94304.

according to those that can follow each action (either shift or reduce) and by verifying that the sets of strings are disjoint. If such is the case, the LR(0) parser can be modified (and made deterministic) by forcing it to "look ahead" a certain number of symbols whenever it "lands" in an inconsistent state.

Both the SLR and LALR techniques utilize this strategy; the difference between them resides in the calculation of the lookahead sets. Algorithms for computing LALR(k) lookahead have been presented by LaLonde [15], Anderson, Eve, and Horning [1], Pager [16], Kristensen and Madsen [14], DeRemer and Pennello [8], and Park, Choe, and Chang [17]. Regular languages as lookahead sets have also been considered [2, 5], but until now regular lookahead has not been deemed practical.

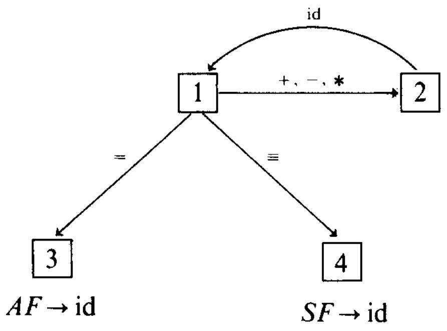
Most techniques used today are based on single-symbol lookahead, since they are considered to be sufficient for grammars that describe the phrase structure of programming languages. However, there are a number of situations that require multi-symbol or even arbitrary lookahead for LR parsing decisions. These situations arise when some form of grammar pre-processor is used to translate a high-level phrase-structure specification to a pure context-free grammar, which is then processed by an LR analyzer. For example, "syntax macros" may have to be pre-processed, as well as regular right-part grammars. The resulting context-free grammar usually bears little resemblance to the original specification. If the LR analyzer is incapable of producing a deterministic parser, it is usually very difficult for the user to identify and correct the problem. The reasons are twofold. First, the system-generated diagnostics (if any!) of the unresolved conflicts are written in terms of dozens of nonterminals that were created by the system to produce a pure context-free grammar and hence are gibberish to the user. Second, when the user finally traces the problem back to the original specification, he finds that the required modifications wreak havoc with semantic processing and render his syntactic specification unnecessarily cluttered and unnatural. For example, a published Ada grammar [9] is LALR(1). The "subprogram-specification is" section of this grammar is regarded as "inelegant but inevitable, ...unless we go up to LALR(2)." The natural (in fact, the original) version of this grammar is LALR(2). It was not without effort that the original grammar was transformed into the final one.

Typically, grammars that describe the phrase-structure of programming languages are "mostly" LR(0), i.e., the majority (around 70%) of the states in its LR(0) push-down automaton have no conflicts. The vast majority (more than 95%) of the remaining states have conflicts that can be resolved with single-symbol lookahead. Thus multi-symbol lookahead is necessary only for a small number of states, usually only one or two. It is annoying to see a grammar rejected by an LR-based system that is so "close" to obtaining a deterministic parser, particularly if the remaining conflicts are so few and so difficult to correct by re-writing the specification. Our technique addresses this problem, by progressively attempting single-symbol, multi-symbol, and even arbitrary lookahead, only when required, thereby increasing the chances of obtaining a deterministic parser automatically, without painful user intervention. We present the essence of our technique with the following context-free grammar:

$$\begin{array}{lll}
S \rightarrow S\perp & AE \rightarrow AE + AT & SE \rightarrow SE + ST \\
S \rightarrow B & AE \rightarrow AE - AT & SE \rightarrow SE - ST \\
B \rightarrow AE = AE & AE \rightarrow AT & SE \rightarrow ST \\
B \rightarrow SE \equiv SE & AT \rightarrow AT * AF & ST \rightarrow ST * SF \\
 & AT \rightarrow AF & ST \rightarrow SF \\
 & AF \rightarrow \text{id} & SF \rightarrow \text{id}
\end{array}$$

Here the language consists of a single boolean expression, followed by an arbitrary number of end-of-file markers $\perp$.[1] The boolean expression is a comparison of two arithmetic expressions, or of two set expressions. Both types of expressions have the same syntactic structure. The LR(0) automaton (not shown) has 29 states, only nine of which are inconsistent. Of these, eight require only single symbol lookahead and can be dealt with using the well-known techniques mentioned above. The one remaining inconsistent state (call it q) has a reduce–reduce conflict: after shifting on the first "id" in the expression, the parser cannot decide whether the "id" is to be a set factor (*SF*), or an arithmetic factor (*AF*). The parser must look ahead an arbitrary number of symbols, to the end of the expression, and make its decision upon encountering either $=$ or $\equiv$.

To solve this problem, we propose to automatically construct an FSA such as the one shown below, and "attach" it to q.



Whenever the main parser "lands" in q, it is temorarily suspended. The above FSA is invoked to scan the lookahead symbols and eventually, barring erroneous input, to encounter the deciding symbol ($=$ or $\equiv$). On that symbol, the FSA moves to one of the two final states, which is annotated with the correct production upon which to reduce. The main parser then resumes at q, and reduces accordingly.

[1] We will assume that every sentence is "padded" with such a sequence of $\perp$'s.

Without question, the FSA solves a problem that would have been difficult to correct by re-writing the grammar.

To handle problems of this nature, we present an automatic parser generation technique in which we (1) construct the LR(0) automaton, (2) construct one lookahead FSA per inconsistent LR(0) state, and (3) test each lookahead FSA for certain (rather simple) properties, to determine whether single-symbol, multi-symbol, or arbitrary lookahead is required, or whether lookahead can be used to resolve the conflict at all. Thus it is the system, not the user, that determines the amount of lookahead required. We call the corresponding class of context-free grammars "LAR(m)." The user-supplied parameter "m" is *not* the amount of lookahead requested, but instead the limit on the size of certain paths that are utilized in constructing the lookahead FSAs. The class of LAR(m) grammars is a subset of the LR-regular grammars [5], but unlike the LR-regular class, it is decidable whether an arbitrary context-free grammar is LAR(m), for a given m. Furthermore, every LALR(k) grammar is LAR(m), for some m. In practice, the required value of m is reasonable: our implementation has used values of $m \leqslant 6$ to resolve multi-symbol lookahead problems in grammars for FORTRAN, C, Ada, COBOL, and PL/I. In doing so, lookahead FSAs of reasonable size have been built, and the technique's running time compares favorably to that of YACC [11] and the Metaware® TWS.[2]

We now give some background and terminology, describe the construction of lookahead automata, define LAR(m) grammars and their properties, discuss the relationship between LAR(m) and other grammar classes, present statistics for various programming language grammars, and conclude.

2. Background and Terminology

Here we briefly review context-free grammars, finite-state automata, LR(0) and LALR(k) parsers. A detailed presentation is given in [10]. A *context-free grammar* (CFG) is a quadruple $G = (N, T, S, P)$, where N and T are finite disjoint sets of nonterminals and terminals respectively, $S \in N$ is the *start symbol*, and P is a finite set of productions of the form $A \rightarrow \omega$, where $A \in N$ and $\omega \in (N \cup T)^*$. We adhere to the following (usual) notation:

$A, B, C, \ldots$	nonterminal symbols	$\alpha, \beta, \gamma, \ldots, \omega$	strings of grammar symbols
$t, a, b, c, \ldots$	terminal symbols	ε	the empty string
$\ldots, x, y, z$	terminal strings	$\Rightarrow$	right-most derivation
$\ldots, X, Y, Z$	grammar symbols	p, q, r, s	states in an automaton

The *language* generated by G, denoted $L(G)$, is the set $L(G) = \{z \in T^* \mid S \Rightarrow^* z\}$. A *sentence* is an element of $L(G)$. We assume all CFGs to be *reduced*, i.e., every

[2] Metaware is a registered trademark of Metaware, Inc.

production is used in the derivation of some sentence. We also require every CFG to contain two productions of the form $S \rightarrow S\bot$ and $S \rightarrow S'$, where S and $\bot$ appear in no other production. Sentences are thus "padded" with an arbitrary number of end-of-file markers "$\bot$."

A *finite-state automaton* (FSA) M is a quintuple (K, T, Δ, Start, F), where K is a finite set of *states*, T is a finite set of symbols, Δ is a finite set of *transitions* of the form $p \rightarrow^{t} q$, Start $\in K$ is the *start state*, and $F \subseteq K$ is the set of *final states*. A *path* in FSA M is a sequence of states $q_0 q_1 \cdots q_n$ such that for some $t_1 t_2 \cdots t_n \in T^*$ $(n \geqslant 0)$, $q_0 \rightarrow^{t_1} q_1 \rightarrow^{t_2} \cdots \rightarrow^{t_n} q_n$, and is denoted $[q_0 : t_1 t_2 \cdots t_n]_M$. This notation uniquely defines the path only if the FSA is deterministic. In this paper, the notation will be used only whenever this is the case. The subscript M will be omitted whenever the intent is clear. Top$[q_0 : t_1 t_2 \cdots t_n]$ denotes q_n, and the *length* of path $[q_0 : t_1 t_2 \cdots t_n]$ is defined as $n+1$. The first state in a path is omitted if it is the start state of M; thus $[t_1 t_2 \cdots t_n]$ denotes $[\text{Start}: t_1 t_2 \cdots t_n]$, and $[\ ]$ denotes the start state alone. Paths can be truncated to a given maximum length m:

$$[q:tz]:m = \begin{cases} [r:z]:m, & \text{if } |tz| \geqslant m \text{ and } r = \text{Top}[q:t] \\ [q:tz], & \text{otherwise.} \end{cases}$$

Note that when a path is truncated to length m, the *last* m states in the path are kept. The *language* recognized by FSA M, denoted $L(M)$, is $\{z \in T^* \mid \text{Top}[z] \in F\}$. A FSA M is *reduced* if for all $q \in K$, there exists a path from Start to q, and a path from q to some final state.

An LR(0) parser for a CFG G is a quintuple $LR_0 = (G, K, \text{Start}, \text{Next}, \text{Reduce})$, where K is a finite set of *parse states*, Start $\in K$ is the *start state*, Next is the transition function for the characteristic FSA $CA = (K, N \cup T, \text{Next}, \text{Start}, K)$, and Reduce: $K \rightarrow$ *Powerset*(P) is the *reduce function*. By construction (see [6, 10]), CA is deterministic. However, LR_0 might be nondeterministic, as we show next.

A *configuration* of parser LR_0 is a pair denoted $[\alpha]z$, where $[\alpha]$ is the parse-time state stack (a path through CA, beginning at Start) and $z \in T^*$ is the remaining input to be parsed. The moves made by LR_0 are defined by relation "$\mapsto$" (pronounced "moves-to"), as follows:

- *shift-move*: $[\alpha]\, tz \mapsto [\alpha t]z$ iff Next(Top$[\alpha]$, t) is defined;
- *reduce-move*: $[\alpha\omega]z \mapsto [\alpha A]z$ iff $A \rightarrow \omega \in$ Reduce(Top$[\alpha\omega]$).

A shift move (on t) is accomplished by removing t from the input, moving (on t) to a new state in the automaton and pushing that state on the stack. A reduce move (on a given production) consists of popping from the stack as many states as there are symbols on the right-hand side of the production and then moving forward on the left-hand symbol of the production. Note that it is the top state of the stack that determines which move(s) apply. An LR(0) state is *inconsistent* if either (or both) of the following occur: (1) both moves apply (shift/reduce conflict); (2) more than one reduce-move applies (reduce/reduce conflict). LR_0 is nondeterministic if it has one or more inconsistent states. Note that LR_0 is "built upon" CA, by adding the

Reduce function and the parse-time stack. The nondeterminism is introduced by the Reduce function, since every conflict involves at least one reduction. The *language* recognized by LR_0, denoted $L(LR_0)$, is the set $L(LR_0)=\{z\in T^* \mid [\varepsilon]\, z\bot \mapsto^+ [S]\bot\}$. Both Knuth and De Remer [6, 12] have proven that $L(LR_0)=L(G)$, i.e., LR_0 (nondeterministically) accepts only correct strings.

LALR(k) lookahead sets are defined for each state q and each conflicting action (shift or reduce), as the set of strings of length k that the LR(0) parser could accept by (1) performing the conflicting action in question at state q and (2) performing some sequence of applicable moves,

$$LA_k(q, A\to\omega)=\{t_1t_2\cdots t_k \mid [\alpha\omega]\, t_1t_2\cdots t_k\, y \mapsto [\alpha A]\, t_1t_2\cdots t_k\, y \mapsto^* [S']\bot, \mathrm{Top}[\alpha\omega]=q, \text{for some } y\}.$$

$$LA_k(q, t)=\{t_1t_2\cdots t_k \mid [\alpha]\, t_1t_2\cdots t_k\, y \mapsto [\alpha t_1]\, t_2\cdots t_k\, y \mapsto^* [S']\bot, \mathrm{Top}[\alpha]=q, t=t_1, \text{for some } y\}.$$

A CFG G is LALR(k) if and only if for every inconsistent state q in its LR(0) automaton and for each $\Omega_1, \Omega_2\in(T\cup P)$ such that $\Omega_1\neq\Omega_2$, $LA_k(q,\Omega_1)$ and $LA_k(q,\Omega_2)$ are disjoint.

3. Lookahead Automata

Our intent is to construct one lookahead FSA per inconsistent state in the LR(0) automaton. We begin by defining the "continuation" language of any state q.

Definition 3.1. Given LR_0 and $q\in K$, Continuation(q) $=\{x\in T^* \mid [\alpha]x\mapsto^* [S]\bot, \mathrm{Top}[\alpha]=q\}$.

State q can be the start state of LR_0. If so, the continuations comprise the entire language. Thus, the continuations in general form a context-free language. We intend to approximate it with a *regular* language, by simulating the effect of shift and reduce moves on a *suffix* of the stack. The stack suffix's depth is restricted to a maximum of "m," the parameter in "LAR(m)". We consider only $m\geqslant 1$, since a maximum stack depth of zero is useless. The simulation begins at q, with the stack suffix containing only state q. This is essentially Čulik and Cohen's notion of "...modifying the LR(0) push-down automaton so as to "forget" all but a bounded amount of information on its push-down stack..." [5], which they propose to do essentially by hand and which we do automatically as part of our construction algorithm. The following relations accomplish this.

Definition 3.2. Relation Reduces is the union of all relations Reduces$_{A\to\omega}$, and Reads is the union of all relations Reads$_t$, where for all $A\to\omega\in P$, for all $t\in T$, and for all $m\geqslant 1$:

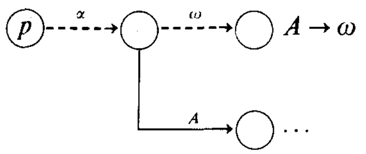
(i) $[p{:}\alpha\omega]$ $\text{Reduces}_{A\to\omega}[p{:}\alpha A]{:}m$ iff $A\to\omega\in\text{Reduce}(\text{Top}[p{:}\alpha\omega])$;

(ii) $[p{:}\gamma]$ $\text{Reduces}_{A\to\omega}[r{:}A]{:}m$ iff $A\to\omega\in\text{Reduce}(\text{Top}[p{:}\gamma])$, $\omega=\alpha\gamma$, and $\text{Top}[r{:}\alpha]=p$;

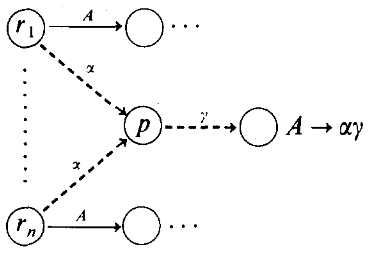
(iii) $[p{:}\alpha]$ $\text{Reads}_t[p{:}\alpha t]{:}m$ iff $\text{Top}[p{:}\alpha t]$ is defined.

Pictorially,

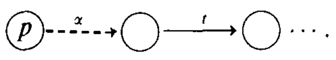
Case (i)



Case (ii)



Case (iii)



In case (i), the stack suffix $[p{:}\alpha\omega]$ is long enough to accommodate the entire phrase being reduced (ω). We backtrack on ω and then move forward on A; the resulting (unique) stack suffix is $[p{:}\alpha A]$, truncated to maximum length m.[3] In case (ii), the phrase being reduced ($\alpha\gamma$) is at least as long as the stack suffix ($[p{:}\gamma]$); i.e., we must backtrack on γ (to state p), *and further* on α, before moving forward on A. We must backtrack on α in all possible ways, i.e., as many as there are states r such that $\text{Top}[r{:}\alpha]=p$. The resulting stack suffixes are of the form $[r{:}A]$, truncated to maximum length m.[4] When $\alpha=\varepsilon$, cases (i) and (ii) are the same. In case (iii), a shift on t is simulated, and the stack is truncated to length m.

3.1. *Non-Deterministic Lookahead Automata*

Here we define the non-deterministic version of the lookahead automaton. Intuitively, it has (1) one state per stack suffix and conflicting action, (2) one ε-transition per Reduces step, (3) one t-transition per Reads_t step, and (4) appropriately defined transitions leading out from q (the start state).

[3] Truncation occurs only if $\omega=\varepsilon$.

[4] Truncation occurs only if $m=1$.

DEFINITION 3.3. A *lookahead item* is a pair of the form $([p{:}\alpha], \Omega)$, where $\Omega \in T \cup P$ is called an *action*. A *lookahead state* is a collection of lookahead items. A lookahead state is *final* iff every item in it has the same action.

In the non-deterministic lookahead FSA each state has only one item; thus every state is final. In the deterministic FSA, as we shall see shortly, a lookahead state can have more than one item.

DEFINITION 3.4. Given an inconsistent LR(0) state q, its *non-deterministic lookahead automaton* is defined as $\mathrm{NLAA}_q = (\mathrm{LAS}_q, T, \mathrm{Lookahead}_q, q, \mathrm{LAS}_q)$, where

(i) $\mathrm{Lookahead}_q(([p{:}\alpha], \Omega), \varepsilon) = \{([p'{:}\alpha'], \Omega) \mid [p{:}\alpha]\ \mathrm{Reduces}[p'{:}\alpha']\}$;

(ii) $\mathrm{Lookahead}_q(([p{:}\alpha], \Omega), t) = \{([p'{:}\alpha'], \Omega) \mid [p{:}\alpha]\ \mathrm{Reads}_t[p'{:}\alpha']\}$;

(iii) $\mathrm{Lookahead}_q(q, \varepsilon) = \{([r{:}A]{:}m, A \to \omega) \mid [q{:}\varepsilon]\ \mathrm{Reduces}_{A \to \omega}[r{:}A]{:}m\}$

(iv) $\mathrm{Lookahead}_q(q, t) = \{([q{:}t]{:}m, t) \mid [q{:}\varepsilon]\ \mathrm{Reads}_t[q{:}t]{:}m\}$.

LAS_q contains *lookahead states*, every one of which (for now) is final. The term "lookahead state" also applies to q, i.e., $q \in \mathrm{LAS}_q$. $\mathrm{Lookahead}_q$ is NLAA_qs transition function. The automaton accepts strings of terminal symbols, and its start state is q itself. Actions originate at q; in fact, they are the actions that cause the conflict. Thereafter, actions are simply copied from item to item. Pictorially:

Case (i) $\boxed{[p{:}\alpha], \Omega} \xrightarrow{\varepsilon} \boxed{[p'{:}\alpha'], \Omega}$,

if $[p{:}\alpha]\ \mathrm{Reduces}[p'{:}\alpha']$

Case (ii) $\boxed{[p{:}\alpha], \Omega} \xrightarrow{t} \boxed{[p'{:}\alpha'], \Omega}$,

if $[p{:}\alpha]\ \mathrm{Reads}_t[p'{:}\alpha']$

Case (iii) $\textcircled{q} \xrightarrow{\varepsilon} \boxed{[r{:}A]{:}m, A \to \omega}$,

if $[q{:}\varepsilon]\ \mathrm{Reduces}_{A \to \omega}[r{:}A]{:}m$

Case (iv) $\textcircled{q} \xrightarrow{t} \boxed{[q{:}t]{:}m, t}$,

if $[q{:}\varepsilon]\ \mathrm{Reads}_t[q{:}t]{:}m$

An example, including the grammar, the LR(0) automaton and the lookahead automaton for the (one) inconsistent LR(0) state, is shown in Fig. 3.1. The value of m used is 2. The left and right halves of NLAA_{12} recognize languages $c^* + c^*a\bot^*$ and $c^* + c^*b\bot^*$ as those that the parser could accept if it reduced via $A \to C$ and $B \to C$ in state 12, respectively. To resolve the conflict at parse time, one must determine which half of the lookahead automaton accepts the actual lookahead, which cannot be done efficiently with NLAA_{12} in its present form, since it is

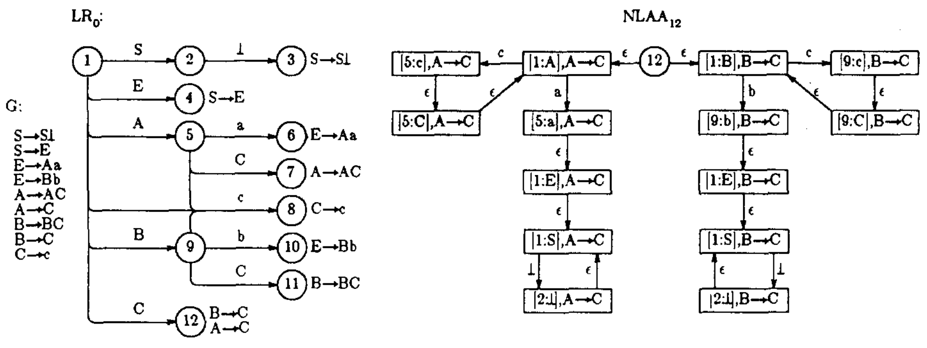


FIG. 3.1. Sample non-deterministic lookahead FSA, with $m = 2$.

non-deterministic. In its present form, however, it is easy to see that for a given state q, every continuation string is accepted by the lookahead automaton.

LEMMA 3.1. Continuation$(q) \subseteq L(\text{NLAA}_q)$.

Proof. Given $y \in \text{Continuation}(q)$, a sequence of moves accepts it. Each reduce or shift-t move in this sequence is matched by a Reads or Reduces step, and hence by an ε- or a t-transition in NLAA_q, respectively. Any such sequence of transitions ends in a final lookahead state, and thus $y \in L(\text{NLAA}_q)$. ∎

3.2. *Deterministic Lookahead Automata*

The deterministic version of LAA_q (called DLAA_q) is obtained by (1) transforming NLAA_q to a DFA in the traditional manner, (2) redefining the set of final states in the resulting machine, and (3) "pruning" certain transitions. For our example, the machine resulting from the first of these three steps is shown in Fig. 3.2(i). In this DFA, every state is final, since every state in NLAA_q was final.

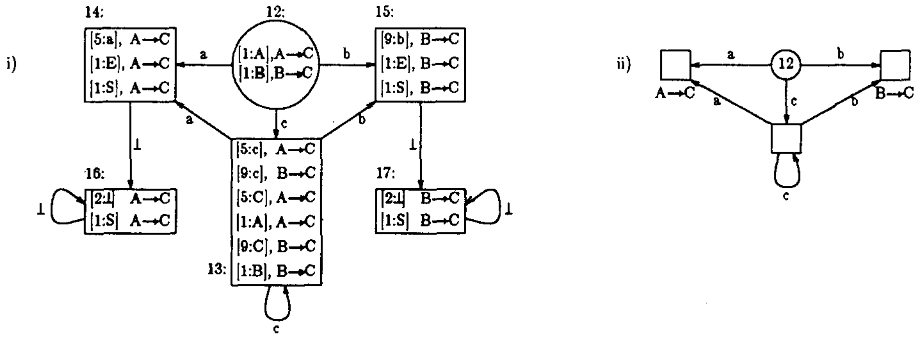


FIG. 3.2. Sample deterministic lookahead automaton, before and after "pruning" and discarding items.

The transformation of NLAA_{12} to a DFA has rendered both $\perp$-transitions useless, since in each of lookahead states 14, 15, 16, and 17 every item has the same action. To resolve the conflict, the "lookahead scanner" should recognize either c^*a or c^*b, arriving at either state 14 or 15, *where it need go no further*. Rather than every state being final, lookahead states 14, 15, 16 and 17 (and no others) should be final. Hence the second step alluded to above: we re-define the set of final states as in Definition 3.3. Transitions leading out from these final states are now useless. Thus we have step 3: "prune" all transitions leading out from final states. This may render some states unreachable, e.g., states 16 and 17 in Fig. 3.2. These states must be removed as well. The final states are annotated with their unique actions. Finally, once the lookahead FSA is built, all items can be discarded. The resulting automaton is shown in Figure 3.2(ii).

DEFINITION 3.5. DLAA_q is defined as the result of (1) transforming NLAA_q to a DFA, (2) re-defining final states as those in which all items have the same action, and (3) pruning all transitions leading out from final states and eliminating unreachable states.

The start state of DLAA_q contains all items of the form $([q:\varepsilon], \Omega)$, where Ω is a conflicting shift action at q. Due to the NFA-to-DFA transformation, the start state of DLAA_q also contains the "Reduces" closure of those items of the form $([q:\varepsilon], \Omega)$, where Ω is conflicting reduce action. It should be evident that NLAA_q and DLAA_q do *not* accept the same language. Specifically, NLAA_{12} accepts prefixes of strings of the form $c^*(a+b)\perp^*$, whereas DLAA_{12} accepts $c^*(a+b)$. More on this shortly.

3.3. *Direct Construction of Deterministic Lookahead Automata*

Here we outline the algorithm for constructing DLAA_q directly, i.e., without the intermediate step of building NLAA_q. Hereafter, we refer to DLAA_q as simply LAA_q. The algorithm is presented in detail in [4], and it is similar to that of LR(0) parsers: each state is an item-set, and Closure and Successor operations are applied to each item-set. We begin by assuming that each of LAA_qs lookahead states (including q, its start state) is an empty set of lookahead items. Then, for each conflicting action $A \to \omega$, we add (to q) the item produced by $\mathrm{Reduces}_{A \to \omega}$. Also, for each conflicting action t, the item produced by Reads_t is added to the t-successor of q. We then compute the Closure and the Successors of each non-final lookahead state, until no more can be computed, as follows.

$$\mathrm{Closure}(r) = r \cup \{([p:\alpha], \Omega) \mid ([p':\alpha'], \Omega) \in \mathrm{Closure}(r),\ [p':\alpha']\ \mathrm{Reduces}[p:\alpha]\};$$

$$\mathrm{Successors}(r) = \{\mathrm{Nucleus}(r, t) \mid t \in T,\ r \notin \mathrm{FLAS}_q\};$$

$$\mathrm{Nucleus}(r, t) = \{([p:\alpha], \Omega) \mid ([p':\alpha'], \Omega) \in r,\ [p':\alpha']\ \mathrm{Reads}_t[p:\alpha]\};$$

$$\mathrm{FLAS}_q = \{r \in \mathrm{LAS}_q \mid r \text{ is final}\}.$$

Final lookahead states, i.e., those whose items all have the same action, are "annotated" via the Decision function, which returns the unique action from each final lookahead state.

DEFINITION 3.6. $\text{Decision}_q : \text{FLAS}_q \rightarrow (T \cup P)$ is defined such that $\text{Decision}_q(r) = \Omega$ iff $([p{:}\alpha], \Omega)$ is an item in r.

The direct construction method is illustrated in Fig. 3.3, in which we have the same grammar as in Fig. 3.1, but have chosen a value of $m = 1$. With this DFA, the conflict cannot be resolved. Comparing Fig. 3.3 with Fig. 3.2, the reasons for which $m = 2$ "works" and $m = 1$ does not are clear. With $m = 1$, the lookahead machine "forgets" how state 8 is entered (from either 5 or 9, depending on the action chosen). The Closure operation then "backs out" of state 8 in all possible ways, leading eventually to a "trap", i.e., a loop on $\perp$ with no way out. With $m = 2$, the two paths to state 8 are kept separate. Clearly, any value $m \geqslant 2$ would do. Three facts should be evident at this point:

(1) Given m, the above construction terminates, since there are finitely many items and states are item-sets.

(2) By construction, every lookahead state is reachable from q. However, final states are not reachable from every lookahead state, e.g., when there is a loop on $\perp$, as in Fig. 3.3. In that case, LAA_q is not reduced.

(3) After constructing the lookahead FSA, it is useful to examine it. If it is not reduced, as in Fig. 3.3, then lookahead will not resolve the conflict. If it is

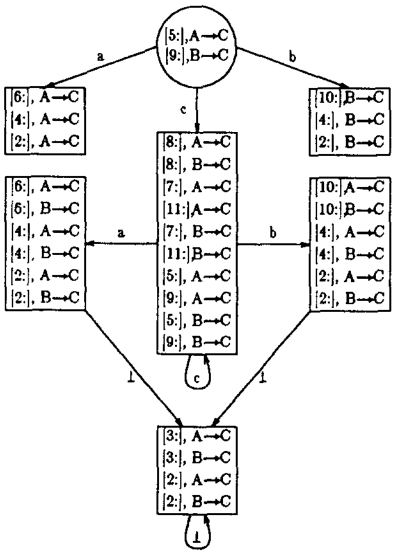


FIG. 3.3. Sample deterministic lookahead FSA, with $m = 1$.

reduced but contains cycles (as in Fig. 3.2), then arbitrary lookahead is required to resolve the conflict. If it is reduced and acyclic, then the length of the longest path is the amount of lookahead required. Since Successors are not computed for final states, the technique determines the amount of lookahead required *progressively*; i.e., multi-symbol lookahead is attempted only when single-symbol lookahead fails, and arbitrary lookahead only when multi-symbol lookahead fails.

We can now characterize the language accepted by LAA_q.

LEMMA 3.2. *If* LAA_q *is reduced, it accepts some prefix of every string in* Continuation(q).

Proof. Let $z \in \text{Continuation}(q)$. Then $[\alpha]\, z\bot \mapsto^* [S]\,\bot$ and $\text{Top}[\alpha] = q$. Since a final state can always be reached in LAA_q, $[q\!:](\text{Reduces}^* \circ \text{Reads})^n\, [p\!:\!\beta]\!:\!m$, for some n, where the Reads steps are on the first n symbol of z (say, $t_1 \cdots t_n$), and $\text{Top}[q\!:\!t_1 \cdots t_n]_{\text{LAA}_q} = f$, a final lookahead state. Hence LAA_q accepts prefix $t_1 \cdots t_n$ of z. ▮

If LAA_q is not reduced, it may scan all of z, but might not accept it if it enters a loop on $\bot$ and never reaches a final state. Such is the case for any string of the form $cc^*(a+b)\bot^*$ in Fig. 3.3.

It is important to note that LAA_q might accept prefixes of some "incorrect" strings (i.e., not in Continuation(q)) and might even produce parsing decisions for them. We disregard those because the underlying LR(0) parser will eventually reject them anyway. In the worst case error detection may be postponed.

4. LAR(m) PARSERS

The LAR(m) parser is "built upon" the LR(0) parser by adding a collection of finite-state automata, one such *lookahead automaton* per inconsistent LR(0) state. At parse time, whenever LR_0 "lands" in an inconsistent state q, LR_0 is temporarily suspended, and q's lookahead automaton is invoked to scan the remaining input in an attempt to resolve the conflict. Assuming that all lookahead automata are reduced, a final state in the lookahead FSA is reached before the stream of end-of-file markers is encountered. The correct action (either the symbol on which to shift or the production on which to reduce) is found at the final state. The LR(0) parser takes that action and resumes normal parsing. In this section we formally define this mode of operation and prove the correctness of the LAR(m) parser.

DEFINITION 4.1. Given LR_0, its collection of lookahead automata is $\text{LAA} = \{(\text{LAA}_q, \text{Decision}_q) \mid q \in K \text{ is inconsistent}\}$. For any $m \geqslant 1$, an LAR(m) parser for a CFG G is a pair $\text{LAR}_m = (LR_0, \text{LAA})$.

A *configuration* of an LAR(m) parser is denoted $[\alpha]\, y \mid z$, where $[\alpha]\, yz$ is the configuration for the corresponding LR(0) parser, and "|" is the *current lookahead* *marker*, which identifies the portion of the remaining input that has already been scanned by LAA_q, while attempting to resolve the conflict of q. LAR_m can make five types of moves. Intuitively, these are (1) shift as LR_0 would, (2) reduce as LR_0 would, (3) look ahead one more symbol, (4) resolve a conflict in favor of a shift move, and (5) resolve a conflict in favor of a reduce move. These five are described by relation "$\mapsto^{\mathrm{lar}}$."

DEFINITION 4.2. $\mapsto^{\mathrm{lar}}$ is the union of the following five moves:

(1) *shift move.* $[\alpha] \mid tz \mapsto^{\mathrm{lar}} [\alpha t] \mid z$ iff $\mathrm{Top}[\alpha]$ is consistent and $[\alpha]\, tz \mapsto [\alpha t] z$;

(2) *reduce move.* $[\alpha\omega] \mid z \mapsto^{\mathrm{lar}} [\alpha A] \mid z$ iff $\mathrm{Top}[\alpha\omega]$ is consistent and $[\alpha\omega] z \mapsto [\alpha A] z$;

(3) *LA-scan move.* $[\alpha] w \mid tz \mapsto^{\mathrm{lar}} [\alpha]\, wt \mid z$ iff $q = \mathrm{Top}[\alpha]$ is inconsistent and $\mathrm{Lookahead}_q(\mathrm{Top}[q{:}w], t)$ is defined;

(4) *LA-shift move.* $[\alpha]\, ty \mid z \mapsto^{\mathrm{lar}} [\alpha t] \mid yz$ iff $q = \mathrm{Top}[\alpha]$ is inconsistent, $[\alpha]\, tyz \mapsto [\alpha t]\, yz$, and $\mathrm{Decision}_q(\mathrm{Top}[q{:}ty]_{\mathrm{LAA}_q}) = t$;

(5) *LA-reduce move.* $[\alpha\omega] w \mid z \mapsto^{\mathrm{lar}} [\alpha A] \mid wz$ iff $q = \mathrm{Top}[\alpha\omega]$ is inconsistent, $[\alpha\omega]\, wz \mapsto [\alpha A]\, wz$, and $\mathrm{Decision}_q(\mathrm{Top}[q{:}w]_{\mathrm{LAA}_q}) = A \to \omega$.

The first two moves are identical to their LR_0 counterparts, and in them the lookahead marker remains at the beginning of the input. In the LA-scan move, LAA_q has already scanned string w; more input is needed before a parse decision can be made. LAA_q advances on t; the lookahead marker is moved past t. In the LA-shift move, LAA_q has reached a final state, and the correct action, provided by the Decision function, is to shift on t. The parser shifts on t, and the lookahead marker is reset to the beginning of the input. Finally, the LA-reduce move operates analogously to the lookahead-shift move: the correct action is to reduce using $A \to \omega$, the parser performs the reduction and resets the lookahead marker accordingly. Clearly at most one of the five moves applies to a given configuration. Hence the LAR(m) parser is deterministic.

DEFINITION 4.3. For any $m \geqslant 1$, the language recognized by the LAR(m) parser, denoted $L(\mathrm{LAR}_m)$, is the set

$$L(\mathrm{LAR}_m) = \{z \in T^* \mid [\varepsilon] \mid z\bot \xmapsto{\mathrm{lar}}{}^{+} [S] \mid \bot\}.$$

The LAR(m) parser's moves are ultimately LR_0's moves, interspersed with LA-scan moves that perform the lookahead and determine the correct action. Thus LR_0 accepts any string accepted by the LAR(m) parser.

LEMMA 4.1. *For any $m \geqslant 1$, $L(\mathrm{LAR}_m) \subseteq L(LR_0)$.*

Proof. Given a sequence of LAR moves that accepts some string z, consider (1) removing all lookahead-scan moves and (2) replacing all lookahead-shift and lookahead-reduce moves with shift and reduce moves, respectively. Clearly, the resulting sequence of moves can be used by LR_0 to accept z. ▮

The converse, i.e., $L(LR_0) \subseteq L(\mathrm{LAR})$ holds *only* if all lookahead automata are reduced, i.e., if there exist no "traps" in them. In fact, the existence (or lack thereof) of such traps is our criterion for pronouncing a CFG an element of the class of LAR(m) grammars.

DEFINITION 4.4. A CFG G is LAR(m) iff for all inconsistent states q in G's LR(0) parser, LAA_q is reduced.

According to this definition, the grammar in Fig. 3.1 is LAR(m), for $m \geqslant 2$, but it is not LAR(1). It should be evident that if a CFG is LAR(m), then it is also LAR(m'), for $m' \geqslant m$.

THEOREM 4.1. *For any* $m \geqslant 1$, *if a CFG* G *is* LAR(m), *then* LAR_m *is correct, i.e.,* $L(\mathrm{LAR}_m) = L(LR_0)$.

Proof. We need only prove that $L(LR_0) \subseteq L(\mathrm{LAR}_m)$. Let $z \in L(LR_0)$. Then $[\varepsilon]\, z\bot \mapsto^* [S]\,\bot$. We will build a sequence of LAR_m moves that parses z. First, replace every move of the form $[\alpha]x \mapsto [\beta]\,y$ such that $\mathrm{Top}[\alpha]_{CA}$ is consistent, with the corresponding LAR_m move $[\alpha]\,|x \mapsto^{\mathrm{lar}} [\beta]|\; y$. The remaining LR_0 moves in the sequence are of one of the following two forms:

(1) $[\alpha]\; ty \mapsto [\alpha t]\; y$, $\mathrm{Top}[\alpha]_{CA}$ is inconsistent.

(2) $[\alpha\omega]\; y \mapsto [\alpha A]\; y$, $\mathrm{Top}[\alpha\omega]_{CA}$ is inconsistent.

Consider a move of type (1) above. By Lemma 3.2, $\mathrm{LAA}_{\mathrm{Top}[\alpha]}$ accepts some prefix x of ty, so assuming that $xz = ty$, we have $[\alpha] \mid xz \mapsto^{\mathrm{lar}*} [\alpha]x \mid z \mapsto^{\mathrm{lar}} [\alpha t] \mid y$. Replace the single LR_0 move with this sequence of LAR_m moves. An analogous argument applies to moves of type (2) above. In short, replace each LR_0 move in the sequence that parses z with one or with many LAR_m moves, depending on whether the LR_0 state involved is consistent or not. The result is a sequence of LAR_m moves that (deterministically) parses z. Hence $z \in L(\mathrm{LAR}_m)$. ▮

5. RELATIONSHIP WITH OTHER GRAMMAR CLASSES

Here we examine the relationship between the class of LAR(m) grammars and other well-known classes of context-free grammars. First, it should be evident that for each $m \geqslant 1$, the LAR(m) grammars form a proper subset of the LR-regular grammars, since FSAs are used to resolve conflicts. The class of LR-regular grammars is "too large" to be of practical use, because (1) it is undecidable whether a CFG is LR-regular, (2) no suitable method of constructing an LR-regular machine has been given, and (3) it requires *right-to-left* scanning of the input (see [5]). In contrast, it is decidable whether a CFG is LAR(m), for a given m, since algorithms exist for both constructing LAA_q and testing it for the LAR(m) property. Furthermore, the LAR(m) parser accepts the input left-to-right. LAR(m) parsing time is not linear in the size of the input; in fact the parser's running time is, in the worst case, $O(n^2)$, since every input symbol may require a parsing decision and every decision may require looking ahead to the end of the input. However, that is the worst case, and as we shall see in the next section, it rarely (if ever) occurs in practical situations.

LAR(m) grammars have an interesting relationship with the class of LALR(k) grammars.

THEOREM 5.1. *If a CFG G is* LALR(k), *then G is* LAR(m), *for some* $m \geq 1$.

Proof. We describe the procedure for finding a suitable m. Let G be a LALR(k) grammar. Let q be an inconsistent state in G's $LR(0)$ automaton, with two conflicting actions Ω_1 and Ω_2. Let $x_1 \cdots x_k = x \in \mathrm{LA}_k(q, \Omega_1)$, and $y_1 \cdots y_k = y \in \mathrm{LA}_k(q, \Omega_2)$. Then $[\alpha]\, xx' \mapsto^* [\beta]\, x'$, and $[\alpha']\, yy' \mapsto^* [\beta']\, y'$, for some α, α', β, and β', where $\mathrm{Top}[\alpha] = \mathrm{Top}[\alpha'] = q$. Since G is LALR(k), $x \neq y$. Let j be the smallest value such that $x_j \neq y_j$. We will choose m as the stack suffix length required to "discriminate" between these two sequences (i.e., these specific $\alpha, \alpha', \beta, \beta'$, and no others) with Reads and Reduces. Let m_1 be the length of the longest path in the above sequence, between $[\alpha]$ and $[\beta]$, or between $[\alpha']$ and $[\beta']$. Let m_2 be the length of the longest common prefix among all these paths. The value desired is $m = m_1 - m_2$: if LAA_q were built using this value, $\mathrm{Top}[q{:}x_1 \cdots x_j]$ and $\mathrm{Top}[q{:}y_1 \cdots y_j]$ would be different. Furthermore, Ω_1 would appear in $\mathrm{Top}[q{:}x_1 \cdots x_j]$ but not in $\mathrm{Top}[q{:}y_1 \cdots y_j]$; Ω_2 would appear in the latter but not in the former. Now maximize m over *all* such strings x, y, and over all pairs of actions Ω_1, Ω_2 at q. The corresponding LAA_q accepts both $x_1 \cdots x_j$ and $y_1 \cdots y_j$, $\mathrm{Decision}_q(\mathrm{Top}[q{:}x_1 \cdots x_j]) = \Omega_1$, and $\mathrm{Decision}_q(\mathrm{Top}[q{:}y_1 \cdots y_j]) = \Omega_2$. In considering all such pairs of actions, we are clearly considering at least one prefix for every string in Continuation(q), and since all such prefixes are different, each one leads (in LAA_q) to a different final lookahead state. Thus LAA_q is reduced. Finally, maximize m (again) over all inconsistent LR(0) states q. Thus all LAA_q are reduced, and G is LAR(m). ∎

It is important to note that no value of m "covers" all LALR(k) grammars. An example of this is shown in Fig. 5.1, in which an infinite family of LALR(1) grammars is presented. Although each individual grammar is LAR(m) for some m, there exists no m' such that every grammar in the family is LAR(m').

For an arbitrary CFG, it is futile to repeatedly increment m and test for the LAR(m) condition, unless the grammar is known to be LALR(k) for some k. This is not as serious a limitation as one might at first assume, since a similar limitation holds for the LALR class: it is undecidable whether a CFG is LALR(k) for some k (see [12, 18]), and thus it is futile to repeatedly increment k and test for the

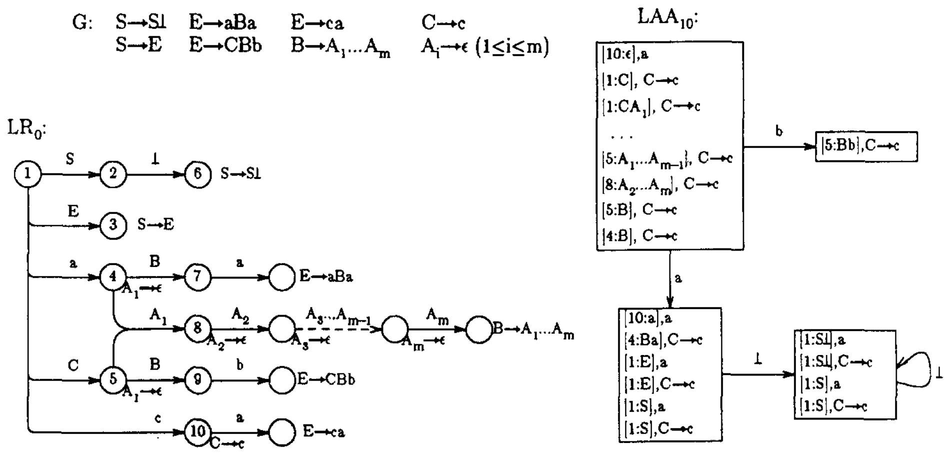


FIG. 5.1. A family of LALR(1) grammars that are not "covered" by any fixed m.

LALR(k) condition. Summarizing, the LAR(m) technique has the power of LALR(k) parsers.

Another extension of fixed look-ahead to arbitrary look-ahead is due to Baker in [2]. His technique, called "extended LR" (XLR), consists of adding "reduce-arcs" to the LR(0) automaton, to indicate that from a given LR(0) state q, one may reduce, say, ω to A, land in state p, perform a sequence of reductions, and then shift on t, landing finally in state r. The corresponding reduce-arc is labeled $(t, A \to \omega)$, and it connects states q and r. Multi-symbol and arbitrary lookahead are achieved by chaining the reduce-arcs, along with terminal transitions in the LR(0) parser. Baker proved that his XLR technique is equivalent to LALR(1), but weaker than LALR(k), for $k > 1$. Hence the following result.

THEOREM 5.2. *If a CFG G is* XLR(k), *then G is* LAR(m), *for some* $m \geqslant 1$.

Proof. If G is XLR(k), then it is LALR(k), as proven by Baker in [2]. By Theorem 5.1, G must also be LAR(m), for some $m \geqslant 1$. ∎

LAR(m) is a stronger technique than XLR. A grammar that shows the additional strength of LAR is shown in Fig. 5.2. This grammar appeared in [2], as an example of a grammar that is LALR(2), and even SLR(2), but not XLR(∞). The XLR reduce-arcs are shown as dashed arrows. For action $C \to d$ at state 13, we may chain the reduce arcs through states 13, 12, 16, and 10: the corresponding string is $ca\perp$. For action $D \to d$ at state 13, we may follow the arcs (and LR(0) transitions) through states 13, 14, 17, and 10: the corresponding string is also $ca\perp$. Clearly, the XLR technique cannot solve the conflict. The reason is that although each individual reduce-arc entails "optimal" (i.e., full LALR(1)) lookahead information, the chaining of such reduce-arcs does not preserve that information because it is

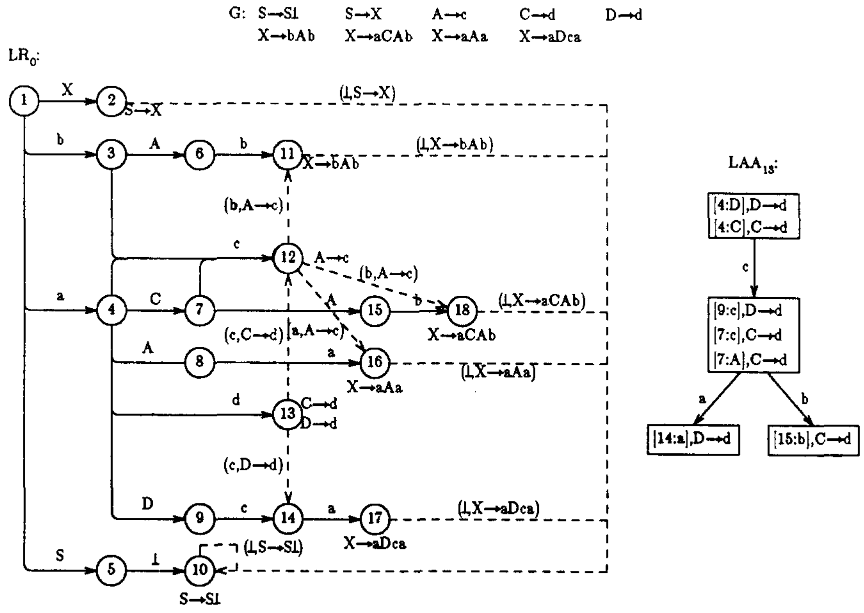


FIG. 5.2. A grammar that is LAR(2), but not XLR(∞).

performed on a state-by-state basis. This explains why XLR is equivalent to LALR(1), but weaker than LALR(k), for $k > 1$. In Fig. 5.2, the chaining of reduce-arcs $13 \rightarrow 12$ and $12 \rightarrow 16$ does not preserve full LALR(2) information, since the reduce-arc from 13 to 12 entails entering state 12 from state 7, not from state 4, and the reduce-arc from 12 to 16 does not take this into consideration when "backing out" of state 12. More explicitly, the parser, after reducing d to C in state 13 (landing in 7), and shifting (on c) to 12, will reduce c to A and land in state 15, not in state 8. It will then shift (on b) to 18, not (on a) to 16.

The grammar is LAR(2), as shown in Fig. 5.2. The value of $m = 2$ is sufficient, because only two states are required to "remember" that state 12 is entered from state 7, not state 4, after reducing d to C in state 13. It should be clear that the grammar is not LAR(1).

Both the XLR and the LAR techniques utilize the LR(0) characteristic FSA as the starting point for lookahead computation, but any characteristic machine that recognizes viable prefixes would serve just as well. Thus either technique can be applied to LR parsers obtained from regular right-part grammars, or from "state-splitting" techniques. Clearly, because of arbitrary lookahead, both techniques can handle (some) grammars that are not LALR(k), or even LR(k), for any k.

Summarizing, the class of LAR(m) grammars is a subset of the LR-regular class. LAR(m) parsers are as powerful as LALR(k) parsers, and more powerful than XLR parsers.

6. Practicality of the LAR(m) Technique

The LAR(m) technique has been successfully applied to a variety of programming language grammars. Here we compare its performance to that of YACC [11] and to that of the Metaware TWS. The results of the comparison with TWS are shown in Fig. 6.1, which have appeared previously in [3].

These tests were run under the UNIX® operating system,[5] on an HP 9000 Series 840 computer. This is a powerful mini-computer, conservatively rated at 4.5 MIPS. The languages are HP Pascal 3000, HP-FORTRAN/77, and HP-COBOL. The syntax of HP-COBOL is defined by three separate grammars, which describe the data division, the environment division, and the procedure division. The grammars for the first two were easily handled. The procedure division, however, is considerably more difficult, and neither TWS nor the LAR(m) implementation could handle it. It is currently being handled by generating incomplete LALR(1) tables and altering the tables by hand. The last three columns show the performance of the two systems. Listed in the table is the *user* time, as measured by /bin/time on UNIX. Process times in UNIX consist of *user* time and *system* time. *User* time is the amount of time spent executing instructions in the *user's* address space, and *system* time is the amount of time spent in the kernel address space on behalf of the process. System time was disregarded for several reasons. First, it was small in relation to the user time. Second, UNIX systems in general cannot account for system time accurately and considerable fluctuation occurs. Third, there was not much difference in the system time between the LAR and TWS tests. The LAR(m) system (written in C) was compiled both with and without the C compiler optimizer. The only available implementation of the TWS was unoptimized, and it was run with the *do try-lalrk* option. The number of lookahead states is fairly large, but we will address that shortly.

More exhaustive comparisons were made with YACC, as shown in Fig. 6.2. These tests were also run under UNIX, on a Gould PowerNode 9080, a dual-processor mainframe computer with each processor rated at 5 MIPS. Throughout Fig. 6.2, X is the LALR(1) version of X'. There are four examples of this: Ada, C, Fortran, and PL/I. In all four cases, X' was not obtained from X, but instead the other way around: the "natural" grammar is not LALR(1), and hand transformation was necessary to obtain the LALR(1) grammar. Six of the 12 grammars could not be handled by YACC. Three of these were because of the grammar's size, three others because of single-symbol lookahead being insufficient, and one (Ada') for both reasons. Column five lists the required amount of lookahead k. Column four lists the number of LR(0) states whose lookahead automata require multi-symbol lookahead, i.e., $k > 1$. Column seven lists the number of lookahead states in those lookahead automata requiring $k > 1$, and column six lists the total number of lookahead states, regardless of the lookahead required. The figures support our remarks in the Introduction of this paper, and our contention that LAR(m) is a practical technique. Spefically,

[5] UNIX is a trademark of Bell Laboratories.

Grammar	LR(0) States	Inconsistent States	Lookahead States	Lookahead Required	Value of "m"	User Time (sec.)		
						LAR(m)	LAR(m) (Opt.)	TWS
Pascal	369	65	778	2	6	5.2	2.7	6.3
FORTRAN	745	115	836	1	4	21.3	10.0	22.9
COBOL Data Div.	316	80	1040	1	5	1.2	0.8	2.8
COBOL Env. Div.	407	100	872	2	5	1.4	0.9	3.7

FIG. 6.1. Comparison of LAR(m) and the Metaware TWS.

- Approximately 20% of the LR(0) states are inconsistent (columns 2 and 3).
- The vast majority of these LR(0) inconsistent states are resolved with single-symbol lookahead. (columns 3 and 4).
- Additional ($k > 1$) lookahead is frequently needed (column 4).
- The additional amount of lookahead needed is reasonable (column 5).
- The total number of lookahead states required is fairly large, but not unreasonable (column 6).
- The number of lookahead states required, for those LR(0) states that need multi-symbol lookahead, is very reasonable (column 7).
- The value of m required is reasonable (column 8).
- LAR(m) produced a deterministic parser for all 12 grammars. YACC failed to do so for half of them (column 11).
- The user time required (columns 9 and 10) is less than YACC's in almost half the cases. LAR(m) did perform poorly for both C and Fortran, in relation to YACC. However, note that YACC could not produce a parser for two of these four grammars, and more importantly, it takes *several hours* for the typical user to transform these LALR(2) grammars into LALR(1).

Grammar	LR(0) States	Inconsistent States	L A R (m)							Y A C C	
			States with k>1	Lookahead required (k)	Lookahead States	LA States with k>1	Value of m	User Time		User Time	Unresolved Conflicts
Ada	924	151	0	1	1581	0	4	25.62		*	*
Ada'	892	146	1	2	1566	19	5	25.90		**	**
Basic	939	122	0	1	1281	0	4	11.95		*	*
C	379	75	0	1	1231	0	4	28.82		10.4	0
C'	373	59	17	2	1567	874	4	29.62		7.6	31
CLU	256	41	0	1	365	0	4	1.37		1.6	0
Fortran	587	93	0	1	656	0	4	15.30		10.1	0
Fortran'	596	93	2	2	742	12	4	16.58		11.6	2
Modula	401	123	0	1	1394	0	5	2.10		4.8	0
Pascal	365	69	0	1	968	0	3	7.52		7.3	0
PL/I	318	128	0	1	1480	0	5	3.85		4.7	0
PL/I'	558	89	11	3	2165	643	5	6.92		7.3	13

* Grammar too large to run on YACC.
** Unresolved conflicts would occur even if the grammar were not too large.

FIG. 6.2. Comparison of LAR(m) and YACC.

For large grammars that are LALR(1), YACC seems to perform better. Thus, in practice, it seems best to adopt a hybrid technique: use an efficient LALR(1) algorithm to solve the vast majority of the conflicts, and then use LAR(m) to resolve the small number of conflicts that remain. In that case, it would only be necessary to build the number of lookahead states listed in column seven, and not *all* of those listed in column six. Using the hybrid technique, the space requirements are clearly reasonable. There remains the problem of determining a suitable value for m, since repeatedly increasing it may lead nowhere. One simple heuristic that we have found useful is as follows: let m be the length of the longest acyclic path in the LR(0) automaton. If this value of m is not suitable, it is unlikely that a larger value will.

Finally, arbitrary lookahead is the last resort, at the expense of a slower (quadratic) parser. The need for it did not arise in any of the grammars presented here, but as mentioned earlier, such situations do arise. For each of the various grammars shown here, parsing time is linear in the size of the input. Furthermore, parsing time should not vary much from one grammar to another, since multi-symbol lookahead occurs infrequently, and does not extend beyond a small number of symbols.

7. Conclusions

LAR(m) is a practical, correct technique for computing lookahead for an LR(0) parser, that progressively attempts single-symbol, multi-symbol, and arbitrary lookahead. The technique determines the amount of lookahead required, and the user is spared the task of guessing it. The class of context-free grammars defined by our technique is a subset of the LR-regular grammars, but unlike LR-regular, the problem of determining whether an arbitrary grammar is in the class is decidable. The technique has the power of LALR(k) parsers, and is more powerful than XLR. Its performance is quite good, especially when used in conjunction with an efficient LALR(1) algorithm. We have shown it to be useful for programming language grammars that are not LALR(1), namely grammars for FORTRAN, Ada, C, COBOL, and PL/I. The technique requires a modest amount of additional grammar analysis time and storage, but it reduces the amount of painful and time-consuming grammar debugging performed by the user.

Acknowledgment

The authors wish to thank Michael Baumann for his implementation efforts.

References

1. T. Anderson, J. Eve, and J. J. Horning, Efficient LR(1) parsers, *Acta Inform.* **2**, No. 1 (1973), 12–39.
2. T. P. Baker, Extending lookahead for LR parsers, *J. Comput. System Sci.* **22**, No. 2 (1981), 243–259.

3. M. BAUMANN, "The LAL(m, NS, k) Parser Generator," Senior thesis project report, Board of Studies in Computer and Information Sciences, University of California, Santa Cruz, June 1987.
4. M. E. BERMUDEZ AND K. SCHIMPF, A practical arbitrary lookahead LR parsing technique, *SIGPLAN Notices* **21**, No. 7 (1986), 136–144.
5. C. ČULIK AND R. COHEN, LR-regular grammars—An extension of LR(k) grammars, *J. Comput. System Sci.* **7** (1973), 66–96.
6. F. L. DEREMER, "Practical Translators for LR(k) Languages," Ph.D. thesis, Department of Electrical Engineering, MIT, Cambridge, MA, 1969.
7. F. L. DEREMER, Simple LR(k) grammars, *Comm. ACM* **14**, No. 7 (1971), 453–460.
8. F. L. DEREMER AND T. J. PENNELLO, Efficient computation of LALR(1) lookahead sets, *ACM TOPLAS* **4**, No. 4 (1982), 615–649.
9. F. L. DEREMER, T. J. PENNELLO, AND W. M. MCKEEMAN, Ada syntax chart, *SIGPLAN Notices* **16**, No. 9 (1981), 48–59.
10. M. HARRISON, "An Introduction to Formal Language Theory," Addison–Wesley, Reading, MA, 1978.
11. S. C. JOHNSON, "YACC—Yet Another Compiler Compiler," Tech. Report CSTR 32, Bell Labs, Murray Hill, NJ, 1974.
12. D. E. KNUTH, On the translation of languages from left to right, *Inform. and Control* **8** (1965), 607–639.
13. A. J. KORENJAK, A practical method for constructing LR(k) processors, *Comm. ACM* **12**, No. 11 (1969), 613–623.
14. B. B. KRISTENSEN AND O. L. MADSEN, Methods for computing LALR(k) lookahead, *ACM TOPLAS* **3**, No. 1 (1981), 6–82.
15. W. R. LALONDE, "An Efficient LALR Parser Generator," Tech. Rep. 2, Computer Systems Research Group, University of Toronto, 1971.
16. D. PAGER, A practical general method for constructing LR(k) parsers, *Acta Inform.* **7** (1977), 249–268.
17. J. C. H. PARK, K. M. CHOW, AND C. H. CHANG, A new analysis of LALR formalisms, *ACM TOPLAS* **7**, No. 1 (1985), 159–175.
18. S. SIPPU, E. SOISALON-SOININEN, AND E. UKKONEN, The complexity of LALR(k) testing, *J. Assoc. Comput. Mech.* **30**, No. 2 (1983), 259–270.