Programming Languages

J. C. REYNOLDS, Editor

A Practical Method for Constructing LR(k) Processors

A. J. KORENJAK*
RCA Laboratories, Princeton, New Jersey

A practical method for constructing LR(k) processors is developed. These processors are capable of recognizing and parsing an input during a single no-backup scan in a number of steps equal to the length of the input plus the number of steps in its derivation.

The technique presented here is based on the original method described by Knuth, but decreases both the effort required to construct the processor and the size of the processor produced. This procedure involves partitioning the given grammar into a number of smaller parts. If an LR(k) processor can be constructed for each part (using Knuth's algorithm) and if certain conditions relating these individual processors are satisfied, then an LR(k) processor for the entire grammar can be constructed for them. Using this procedure, an LR(1) parser for ALGOL has been obtained.

KEY WORDS AND PHRASES: LR(k) grammar, syntactic analysis, parser, deterministic language, syntax-directed compiler, language processor, context-free language, ALGOL
CR CATEGORIES: 4.12, 5.22, 5.23

1. Introduction

The use of formal syntax as the basis of automatically constructed compilers for programming languages has gained wide acceptance in recent years. These applications have, to a great extent, used only the class of context-free languages as a model. For, although features of programming languages such as the requirement that identifiers be declared before use require a more powerful descriptive mechanism, these restrictions can be more efficiently handled by other means (e.g. symbol tables) than by using a more powerful language model. In fact, rather than generalizing the context-free model, practical applications have almost exclusively dealt with more restricted language classes [2]. In particular, the majority of the automatically constructed syntax recognizers are based on a subclass of context-free grammars whose strings can be processed in a single left-to-right no-backup scan, using a pushdown stack. This general approach is formally studied in [3], where the deterministic pushdown automata and the related class of deterministic (context-free) languages are defined. Knuth [4] defines the LR(k) grammars,[1] which have the property that they generate all and only the deterministic languages. (In fact, he shows that the LR(1) grammars also have this "completeness" property with respect to the deterministic languages.) Thus the LR(k) grammars (and, in particular, the LR(1) grammars) are an extremely useful tool for the study of the single-scan context-free languages.

Knuth has described an algorithm for testing an arbitrary context-free grammar for the LR(k) condition (for any particular value of $k \geq 0$). If it succeeds, it generates a deterministic pushdown automaton which can recognize and parse sentences of the given grammar. However, the effort required to make this check and the size of the processor produced grow very rapidly with the complexity of the grammar [1, p. 128]. For a very large grammar, such as one describing a syntax for a programming language, the procedure is not practical. In the first part of this paper a practical method, the basis of which is the Knuth algorithm but which will allow us to work with very large grammars, is investigated. This method involves dividing the given grammar into a number of smaller grammars, proving each part is LR(1) using the Knuth algorithm, and concluding that the original grammar is LR(1) if certain conditions relating the parts are satisfied. The second part of the paper applies this method to a grammar for ALGOL and discusses the implementation of the resulting LR(1) processor.

2. Definitions and Notation

A *vocabulary* V is the union of two disjoint sets of symbols, V_N (the *nonterminal symbols*) and V_T (the *terminal symbols*). For any set of symbols V, the notation V^* is used for the set of all finite-length strings over V, including the empty string, ϵ; $V^+ \triangleq V^* - \{\epsilon\}$. For $k \geq 0$, V^k de-

A short version of this paper was presented at the ACM Symposium on the Theory of Computing, Marina del Rey, California, May 5–7, 1969.

* Present Address: Applied Logic Research Institute, Princeton, N. J.

[1] k is a nonnegative integer.

notes all strings in V^* of length k. Unless otherwise specified, uppercase Latin letters (A, B, C, $\cdots$) are used for nonterminal symbols, lowercase Latin letters at the beginning of the alphabet (a, b, c, $\cdots$) for terminal symbols, lowercase Latin letters at the end of the alphabet (t, u, v, $\cdots$) for strings in V_T^*, and lowercase Greek letters (α, β, γ, $\cdots$) for strings in V^*.

A *context-free grammar* G is a 4-tuple:

$$G = (V_N, V_T, \mathcal{P}, Z_0).$$

V_N and V_T are respectively the nonterminal and terminal vocabularies associated with G. $\mathcal{P}$ is a finite set of *productions* (or *rules*) of the form $A \to \omega$; A is called the *subject* of this rule. The set V_N must include all subjects; V_T must include all symbols appearing on the right side of any production that are not members of V_N. The symbol Z_0 is a particular member of V_N, called the *starting symbol.*

If a grammar G includes the rule $A \to \omega$, then for any strings α and β we write $\alpha A \beta \to \alpha\omega\beta$. If $\alpha_1 \to \alpha_2 \to \cdots \to \alpha_n$, $n \geq 1$, we write $\alpha_1 \Rightarrow \alpha_n$, and say that α_n is *derivable from* α_1 or that α_n can be *reduced to* α_1. The sequence $\alpha_1, \alpha_2, \cdots, \alpha_n$ is a *derivation* of α_n from α_1 in G; the sequence $\alpha_n, \alpha_{n-1}, \cdots, \alpha_1$ is a *reduction* of α_n to α_1. If each step of a derivation is of the form $\alpha A t \to \alpha\omega t$, then the derivation is a *rightmost derivation*, since $t \in V_T^*$ and hence the rightmost nonterminal symbol is rewritten at each step; any description of the corresponding reduction is called a *leftmost parse.* If $Z_0 \Rightarrow \omega$ by a rightmost derivation, then ω is a *rightmost sentential form* of G. If $Z_0 \Rightarrow t$ and $t \in V_T^*$, t is a *sentence* of G. The set of all sentences of G is the *language* generated by G, denoted $L(G)$.

3. LR(k) Grammars and the Knuth Algorithm

Although the construction and operation of LR(k) processors has been described in detail by Knuth [4], to clarify the remainder of this paper it is necessary to repeat the formal description. The operation of the processor is described first, followed by the algorithm to construct the *parsing table* that drives it.

For any grammar $G = (V_N, V_T, \mathcal{P}, Z_0)$, number the productions in $\mathcal{P}$ from 1 to π; the pth production is represented by $A_p \to X_{p1} \cdots X_{pn_p} (n_p \geq 0)$. For fixed $k \geq 0$ and Z_0', $\# \notin V$, let $G' = (V_N', V_T', \mathcal{P}', Z_0')$, where $V_N' = V_N \cup \{Z_0'\}$, $V_T' = V_T \cup \{\#\}$ and

$$\mathcal{P}' = \mathcal{P} \cup \{Z_0' \to Z_0 \#^k\}.^2$$

Number the productions of G' as in G, assigning $Z_0 \to Z\#^k$ the number 0. The symbol $\#$ is called the endmarker. G' is called the *k-augmented grammar* associated with G.

If G is LR(k), a *parsing table* $\mathfrak{I}(G)$ which describes the LR(k) processor for G can be constructed. $\mathfrak{I}(G)$ is composed of a finite number of rows, each of which is associated with a *state* of the processor. A typical row of $\mathfrak{I}(G)$, corresponding to a state $\mathcal{S}$, has the following form:[3]

State Name	Lookahead Strings	Actions	Stack Symbols	Goto States
	y_1	*Shift*	Y_1	$\mathcal{S}_1$
			Y_2	$\mathcal{S}_2$
$\mathcal{S}$		⋮		
			⋮	
	y_l	*Reduce p*	Y_m	$\mathcal{S}_m$

$y_1, \cdots, y_l \in (V_T')^k$ are distinct *lookahead strings* ($l \geq 1$); $Y_1, \cdots, Y_m \in V$ are distinct *stack symbols* ($m \geq 0$); $\mathcal{S}_1, \cdots, \mathcal{S}_m$ are *goto states.* The set of lookahead strings is denoted as $L(\mathcal{S})$ and the action associated with each $y \in L(\mathcal{S})$ is denoted as $A_{\mathcal{S}}(y)$. Each action is one of the following: *Shift*, *Reduce p* ($1 \leq p \leq \pi$), or *Accept.* The set of stack symbols is denoted $S(\mathcal{S})$ and the goto state associated with each $Y \in S(\mathcal{S})$ is denoted $G_{\mathcal{S}}(Y)$.

Driven by a table $\mathfrak{I}(G)$ composed of rows such as these, the LR(k) processor examines an input string from left to right and attempts to reduce it to Z_0—in fact, it does so by attempting to generate its leftmost parse. A two-track pushdown stack is maintained for control purposes. We denote the position of the input tape and the contents of the stack by the notation:

$$\begin{array}{l|l} \mathcal{S}_0\ \mathcal{S}_1 \cdots \mathcal{S}_n & \\ \bullet\ X_1 \cdots X_n & a_1 \cdots a_k t \end{array}$$

Let the original input string be $xa_1 \cdots a_k t \in V_T^* \#^k$. Then, when the above configuration has been reached, x has been reduced to $X_1 \cdots X_n$, the lookahead string is $a_1 \cdots a_k$, and t is that portion of the input not yet examined. The top track of the pushdown stack contains the state names $\mathcal{S}_0, \cdots, \mathcal{S}_n$; roughly, $\mathcal{S}_n$ represents the possible parses at this point.

The processor operates as follows. A special start state $\mathcal{S}_0$ is initially placed on the stack and the first k symbols of the input string are the initial lookahead string. (The symbol $\bullet$ is just a marker.) Assume the processor has reached the configuration shown above.

First, the lookahead string $a_1 \cdots a_k$ is compared with each of the strings in $L(\mathcal{S}_n)$, causing *exactly one* of four possible actions:

(i) If $A_{\mathcal{S}_n}(a_1 \cdots a_k)$ = *Shift*, a_1 is pushed onto the bottom track of the stack, giving

$$\begin{array}{l|l} \mathcal{S}_0\ \mathcal{S}_1 \cdots \mathcal{S}_n & \\ \bullet\ X_1 \cdots X_n\ a_1 & a_2 \cdots a_k t \end{array}$$

(ii) If $A_{\mathcal{S}_n}(a_1 \cdots a_k)$ = *Reduce p*, the rightmost n_p

[2] For any symbol X, X^k denotes the k-fold concatenation of X with itself.

[3] A complete table is shown in Table I.

items on both tracks of the stack are popped off and A_p is pushed onto the bottom track of the stack, resulting in:

$$\begin{array}{l} \mathcal{S}_0\mathcal{S}_1 \cdots \mathcal{S}_{n-n_p} \\ \bullet\ X_1 \cdots X_{n-n_p}A_p \end{array} \Bigg| \; a_1 \cdots a_k t$$

(iii) If $A_{\mathcal{S}_n}(a_1 \cdots a_k) = \textit{Accept}$, processing stops and the input is accepted (see below);

(iv) If $a_1 \cdots a_k \notin L(\mathcal{S}_n)$, the input is rejected.

Secondly, if the action was *Shift* or *Reduce p*, a goto state is determined from the parsing table and placed in the empty position at the right end of the top stack track, in the following manner:

(i) If the action was *Shift*, a_1 will be in $S(\mathcal{S}_n)$ and the goto state is $G_{\mathcal{S}_n}(a_1)$;

(ii) If the action was *Reduce p*, $\mathcal{S}_{n-n_p}$ has been exposed; by consulting the $\mathcal{S}_{n-n_p}$ row in the parsing table, the goto state is determined to be $G_{\mathcal{S}_{n-n_p}}(A_p)$.

In either case (after relabeling) the stack again has the form

$$\begin{array}{l} \mathcal{S}_0\ \ \mathcal{S}_1 \cdots \mathcal{S}_n \\ \bullet\ X_1 \cdots X_n \end{array} \Bigg| \; a_1 \cdots a_k t$$

where $\mathcal{S}_n$ is the goto state just computed. The process is iterated; eventually either the input will be rejected or the special *accepting configuration* will be reached:

$$\begin{array}{l} \mathcal{S}_0\ \ \mathcal{S}_F \\ \bullet\ Z_0 \end{array} \Bigg| \; \#^k$$

S_F is the unique state containing the entry $A_{\mathcal{S}_F}(\#^k) = \textit{Accept}$. No infinite loops are possible, since each move either decreases the length of the input not yet examined or produces another line of the rightmost derivation.

Before proceeding further, it is necessary to explain in some detail the meaning of a state. A *state* is a set of *partial states* of the form $[p, j, w]$, where $0 \leq p \leq \pi$, $0 \leq j \leq n_p$ and $w \in (V_T')^k$. The processor is said to be in the partial state $[p, j, w]$ iff, for some string β, the bottom track of the stack is $\bullet\beta X_{p1} \cdots X_{pj}$ and there exists a terminal string t such that $\beta A_p w t$ is a rightmost sentential form of G'. Thus, in scanning from left to right, the processor has made all reductions necessary for a leftmost parse and has reached a point where the first j elements of the pth production are at the top of the stack; w is a k-character terminal string which may appear if the pth production is completed. At any given time, the processor will be in the state composed of all the partial states which fulfill the conditions above; this is the state at the top of the stack.

Before describing the construction of the parsing table, two definitions are needed. The first defines the set of all terminal strings of length k that are prefixes of strings derivable from α. The second is similar, except the derivations involved may not include the application of a rule $A \rightarrow \epsilon$ to a string which begins with A.

Definition. For a grammar $G = (V_N, V_T, \mathcal{P}, S)$, $\alpha \in V^+$ and $k \geq 0$,

(a) $H(\alpha) = \{t \in {V_T}^k \mid \exists \beta \in V^* \text{ such that } \alpha \Rightarrow t\beta \text{ in } G\}$;

(b) $H'(\alpha) = \{t \in {V_T}^k \mid \exists \beta \in V^* \text{ such that } \alpha \Rightarrow t\beta \text{ in } G$, but $\alpha \Rightarrow t\beta$ does *not* contain a step of the form $A\gamma \rightarrow \gamma$ (due to $A \rightarrow \epsilon$ in $\mathcal{P}$)$\}$.

Both sets are easily computed from the grammar.

During the construction of the parsing table, a list of states is maintained whose associated rows are yet to be computed. Initially, this list contains the start state,[4] $\mathcal{S}_0 = \{[0, 0, \#^k]\}$; as each row is computed, additional states may be added to the list. Since there is a finite number of possible states, the process will terminate. The entries for a typical row of the table, corresponding to a state $\mathcal{S}$, are determined as follows:

First, we compute a set $\mathcal{S}'$ which is needed to find the table entries for row $\mathcal{S}$. Roughly, $\mathcal{S}'$ adds partial states to $\mathcal{S}$ which represent the introduction of new productions. Formally, $\mathcal{S}'$ is iteratively computed as the smallest set satisfying the equation

$$\mathcal{S}' = \mathcal{S} \cup \{[q, 0, x] \mid \exists [p, j, w] \in \mathcal{S}' \text{ such that } j < n_p,$$

$$X_{p(j+1)} = A_q \text{ and } x \in H(X_{p(j+2)} \cdots X_{pn_p} w)\}.^5$$

The partial states added to $\mathcal{S}$ in computing $\mathcal{S}'$ indicate the following situation: the processor is in partial state $[p, j, w]$ and the $(j+1)$-th element of the pth production is the subject of the qth production; thus, the next input symbols may have been derived from A_q according to the qth production.

Second, determine the members of $L(\mathcal{S})$ and their corresponding actions:

(i) If $\exists [p, j, w] \in \mathcal{S}'$ such that $j < n_p$ and

$$y \in H'(X_{p(j+1)} \cdots X_{pn_p} w),$$

then $y \in L(\mathcal{S})$ and $A_{\mathcal{S}}(y) = \textit{Shift}$, since y allows the possibility of continuing the pth production;

(ii) If $[p, n_p, y] \in \mathcal{S}'$, then $y \in L(\mathcal{S})$ and $A_{\mathcal{S}}(y) = \textit{Reduce } p$, since the entire pth production is on the stack and y can follow the completion of this production.

If the action for any symbol in $L(\mathcal{S})$ is not unique, the algorithm terminates; then G is not LR(k), by the definition below.

[4] This state does not quite conform to the informal description of a state given above, since $\#^k$ cannot *follow* the rule $Z_0' \rightarrow Z_0\#^k$. However, it does correctly indicate that processing starts at the 0th position of rule 0.

[5] For example, to compute $\mathcal{S}_0'$ from $\mathcal{S}_0 = \{[0, 0, \#^k]\}$: at the first iteration, add to $\mathcal{S}_0$ all partial states of the form $[q, 0, \#^k]$, where Z_0 is the subject of the qth rule; successive iterations add all partial states corresponding to the 0th position of rules whose subjects can begin a string derived from Z_0. Thus, $\mathcal{S}_0'$ represents all grammar positions in which the processor might be before the first input symbol is read.

Third, compute $S(\mathcal{S})$ and the corresponding goto states:
(i) $S(\mathcal{S}) = \{Y \mid \exists[p, j, w] \in \mathcal{S}'$ such that $Y = X_{p(j+1)}\}$;
(ii) For each $Y \in S(\mathcal{S})$,

$$G_{\mathcal{S}}(Y) = \{[p, j+1, w] \mid \exists[p, j, w] \in \mathcal{S}' \text{ such that } Y = X_{p(j+1)}\}.$$

That is, when the processor is in a state containing the partial state $[p, j, w]$ and the $(j+1)$-th element of the pth production is found, it advances to a state containing the partial state $[p, j+1, w]$.

Finally, the list of states whose rows must yet be computed is updated by:

(i) Removing the present state, $\mathcal{S}$;

(ii) Adding all those states which appear in the goto column of the row for $\mathcal{S}$ that are not on the list and whose rows have not been computed.

These steps are then repeated for each entry on the list, until it is empty. In order to accept an input string as described above, in the unique state $\mathcal{S}_F$ which contains $[0, 1, \#^k]$, set $A_{\mathcal{S}_F}(\#^k)$ to *Accept*.

This algorithm provides a *definition of* LR(k): a grammar G is said to be LR(k) iff, for each state $\mathcal{S}$ in $\mathfrak{I}(G)$, each symbol in $L(\mathcal{S})$ has a unique action. Knuth has proved that if G is an LR(k) grammar (for some $k \geq 0$), then $L(G)$ is a deterministic language—that is, $L(G)$ can be recognized by a deterministic pushdown automaton. Since the processor defined by the parsing table is of the pushdown type, it is merely necessary to incorporate the lookahead mechanism into the finite control unit of a canonical pushdown automaton. He has also shown that every deterministic language has an LR(1) grammar.

4. The Partitioning Scheme—A Simple Case

The computation required for Knuth's algorithm increases rapidly with the size and complexity of the grammar and with the value of k. For a grammar G, let $P = \sum_{p=0}^{\pi} n_p$ and $F = T^k$, where T is the size of V_T'; then a bound on the number of states in $\mathfrak{I}(G)$ is 2^{PF}. Although the actual number of states is typically smaller, Earley exhibits a class of grammars for which the number of states grows exponentially with grammar size [1]. The partitioning scheme described here attempts to reduce parsing table size by reducing growth reflected in the factor F. In particular, it reduces the number of states whose partial states differ only in their third components. This state reduction decreases the computation required to generate the parsing table in two ways. First, fewer table rows need be computed. Second, certain necessary bookkeeping chores are shortened—e.g. determining whether a goto state is identical to a previously generated state, which is necessary to terminate the Knuth algorithm. Most important, the reduction in table size occurs *during* table generation; it is not achieved by generating the entire Knuth table and eliminating or combining states a posteriori.

As the first step, since k is a very strong factor, only LR(1) grammars are considered.[6] From a theoretical point of view, there is no loss of generality, since LR(1) grammars have the same generative capability as LR(k) grammars. More important, this restriction does not seem to be a severe practical limitation. Experience indicates that grammars for most programming languages are LR(1)—or almost so. If a few non-LR(1) situations do arise, the practical solution is not to try to construct an LR(2) processor. In the programming language grammars we have worked with, the few problems encountered have been easily eliminated in one of two ways.

(i) By the preprocessor (lexical analyzer): for example, one problem frequently cited in ALGOL involves the statement prefix "*XYZ* :=". If ":=" is treated as two symbols, there may be some problem in correctly parsing "*XYZ*" with an LR(1) processor. However, there is no good reason for not letting the preprocessor pass the single symbol ":=" to the parser. This avoids the problem and probably should be done in any case. Certainly strings such as **begin** and **end** should be treated as single symbols.

(ii) By making small local changes in the grammar, so that it becomes LR(1): a few potential problems in ALGOL were eliminated this way, as mentioned in Section 7.

If some trouble spots cannot be avoided (perhaps for semantic reasons), it may be necessary for the parser to process these sections in a non-LR(1) manner. (The partitioning scheme seems to be a well-suited framework for such an approach; it is an area for further study.) But it is certainly not generally practical to process significant portions of the input with $k > 1$.

The proposed method of processor construction involves generating the processors for certain LR(1) subgrammars and then combining them to form an LR(1) processor for the entire grammar. To begin with, a very simple case is considered which demonstrates the technique, but avoids the complexities, of the general method. An example is given.

Let $G = (V_N, V_T, \mathcal{P}, Z_0)$ be a large grammar for which we wish to construct an LR(1) processor. Choose a nonterminal Z_1 other than Z_0 and let $\bar{z}_1$ be a new terminal symbol. Let $G_0 = (V_{N_0}, V_{T_0}, \mathcal{P}_0, Z_0)$ be the reduced form[7] of $(V_N, V_T, \overline{\mathcal{P}}, Z_0)$, where $\overline{\mathcal{P}} = \{Y \to \bar{\alpha} \mid Y \to \alpha \in \mathcal{P}$, and $\bar{\alpha}$ is obtained from α by replacing each occurrence of Z_1 by $\bar{z}_1\}$. Let $G_1 = (V_{N_1}, V_{T_1}, \mathcal{P}_1, Z_1)$ be the reduced form of $(V_N, V_T, \mathcal{P}, Z_1)$. Thus G_0 is the reduced form of the grammar obtained from G by replacing every occurrence of the nonterminal symbol Z_1 by the new terminal symbol $\bar{z}_1$, and G_1 is the reduced subgrammar of G which has Z_1 as its starting symbol. Let $G_0' = (V_{N_0}', V_{T_0}', \mathcal{P}_0', Z_0')$ and $G_1' = (V_{N_1}', V_{T_1}', \mathcal{P}_1', Z_1')$ be the 1-augmented grammars associated with G_0 and G_1, with endmarkers $\#_0$ and $\#_1$, respectively. Let the productions of $\mathcal{P}_0$ and $\mathcal{P}_1$ retain the

[6] However, the partitioning scheme can be generalized to $k > 1$ straightforwardly.

[7] A grammar is in *reduced form* if all "useless" nonterminals are removed: for each $A \in V_N$, $\exists \alpha, \beta, t$ such that $S \Rightarrow \alpha A \beta \Rightarrow \alpha t \beta$.

numbering they had in $\mathcal{P}$. The two rules involving endmarkers may both be numbered zero.

Consider a sentence xz_1y that has a derivation $Z_0 \Rightarrow xZ_1y \Rightarrow xz_1y$ in G such that $Z_0 \Rightarrow x\bar{z}_1y$ in G_0 and $Z_1 \Rightarrow z_1$ in G_1. Assume both G_0 and G_1 are LR(1). Consider the recognition of xz_1y by an LR(1)-type processor, knowing that both $x\bar{z}_1y$ and z_1 can be recognized in an LR(1) fashion. The LR(1) recognizer for G might operate in two modes—one for G_0 and the other for G_1. The only difficulty is in ensuring that the master LR(1) processor could determine when to switch between the two modes as it scans xz_1y from left to right. This requires that the master processor be able to determine that a mode change is to be made on the basis of the state it is in after processing x and the first symbol of z_1.

Similarly, the processor must be able to detect the end of the substring z_1. This can be accomplished if whenever $Z_0' \Rightarrow x\bar{z}_1at$ in G_0' and $Z_1' \Rightarrow z_1bu$ in G_1', then $a \neq b$. Otherwise, having seen xz_1a, the processor would not know whether to return to the G_0-mode or to continue to find a longer instance of Z_1 in the G_1-mode.

Rather than formalize these two conditions in terms of the grammar, we describe a method of constructing, if possible, an LR(1) parsing table $\mathfrak{I}$ for G from $\mathfrak{I}_0 = \mathfrak{I}(G_0)$ and $\mathfrak{I}_1 = \mathfrak{I}(G_1)$. At various points during this construction certain conditions must be met in order to continue. These are sufficient conditions for obtaining the LR(1) parser $\mathfrak{I}$. The construction algorithm proceeds by making certain modifications to $\mathfrak{I}_0$ and $\mathfrak{I}_1$, and then forming $\mathfrak{I}$ as the union of the resulting tables.

Consider the left-to-right parsing of the sentence $xz_1y\#$. The LR(1) tables $\mathfrak{I}_0$ and $\mathfrak{I}_1$ are available to recognize the strings $x\bar{z}_1y\#_0$ and $z_1\#_1$. The composite table $\mathfrak{I}$ should start in state $\mathcal{S}_0^0$, the start state of $\mathfrak{I}_0$, and proceed to parse x (which may be null) according to $\mathfrak{I}_0$ until the initial character of z_1 first becomes the lookahead symbol. The state $\mathcal{S}$ it has reached and all preceding states are in $\mathfrak{I}_0$. Since $x\bar{z}_1$ is the prefix of a sentence of $L(G_0')$, $\bar{z}_1 \in L(\mathcal{S})$. However, since the input is $xz_1y\#$, not $x\bar{z}_1y\#_0$, $\mathfrak{I}$ will see the initial symbol of z_1, not the symbol $\bar{z}_1$. The possible initial symbols of z_1 can be found in S_0^1, the start state of $\mathfrak{I}_1$; they are exactly the members of $L(\mathcal{S}_0^1)$. Thus, $L(\mathcal{S})$ should be updated by replacing $\bar{z}_1$ by $L(\mathcal{S}_0^1)$. The action associated with each of these new symbols depends on $A_{\mathcal{S}}(\bar{z}_1)$. Let b be the member of $L(\mathcal{S}_0^1)$ which is the first symbol of z_1:

(i) If $A_{\mathcal{S}}(\bar{z}_1) =$ *Reduce* p, the action does not affect z_1. Correct parsing of x still requires *Reduce* p, since b now signals the end of rule p for $\mathfrak{I}$ just as $\bar{z}_1$ did for $\mathfrak{I}_0$. Thus $A_{\mathcal{S}}(b)$ should be set to *Reduce* p.

(ii) If $A_{\mathcal{S}}(\bar{z}_1) =$ *Shift*, processing of z_1 should begin. This would be done in $\mathfrak{I}_1$ by S_0^1. To effect this switch to the $\mathfrak{I}_1$-mode, $A_{\mathcal{S}}(b)$ should be set to $A_{\mathcal{S}_0^1}(b)$.

These changes should be made to every state $\mathcal{S}$ in $\mathfrak{I}_0$ for which $\bar{z}_1 \in L(\mathcal{S})$. In order for the algorithm to proceed, however, the following condition must be met for each such state: if $b \in L(\mathcal{S}) \cap L(\mathcal{S}_0^1)$, then the action assigned to b during the updating of $\mathcal{S}$ must be the same as the action it originally had in $\mathcal{S}$—i.e. no conflicts may be generated.

Before proceeding to modifications in $\mathfrak{I}_1$, changes needed in the stack and goto entries of $\mathfrak{I}_0$ must be considered. Of the states examined so far, the only ones for which $\bar{z}_1 \in S(\mathcal{S})$ are those for which $A_{\mathcal{S}}(\bar{z}_1) =$ *Shift*. To obtain the correct stack and goto entries for these states, the existing entries must be merged with those for $\mathcal{S}_0^1$, in order to effect the mode change. $S(\mathcal{S})$ must be replaced by $[S(\mathcal{S}) - \{\bar{z}_1\}] \cup S(\mathcal{S}_0^1)$. The goto states corresponding to the new members of $S(\mathcal{S})$ are obtained as follows:

(i) for all $Y \neq Z_1$, $G_{\mathcal{S}}(Y)$ is set to $G_{\mathcal{S}}(Y) \cup G_{\mathcal{S}_0^1}(Y)$;

(ii) $G_{\mathcal{S}}(Z_1)$ is set to $G_{\mathcal{S}}(\bar{z}_1) \cup \tilde{G}_{\mathcal{S}_0^1}(Z_1)$, where $\tilde{G}_{\mathcal{S}_0^1}(Z_1)$ is obtained from the table entry for $G_{\mathcal{S}_0^1}(Z_1)$ by deleting $\#_1$ and its action.

The union operation used on states indicates the formation of a (possibly) *new* state, whose table entries are the "union" of the entires of its members.[8] In (i) above, if $Y \notin S(\mathcal{S})$ before updating, then $G_{\mathcal{S}}(Y)$ is set to $G_{\mathcal{S}_0^1}(Y)$ to allow $\mathcal{S}$ to act like $\mathcal{S}_0^1$ while $\mathfrak{I}$ is processing z_1 in the $\mathfrak{I}_1$-mode. However, if Y was already in $S(\mathcal{S})$, indicating that Y might also be a continuation of the $\mathfrak{I}_0$-mode, then a union state must be formed to allow the processor to follow both paths until the indecision is resolved. In (ii), $G_{\mathcal{S}}(\bar{z}_1)$ indicates that when a Z_1 has been found, processing should return to the $\mathfrak{I}_0$-mode, just as if $\bar{z}_1$ had been in the input string. However, if $Z_1 \Rightarrow Z_1\alpha$ in G_1, processing should also continue in the $\mathfrak{I}_1$-mode to find a longer instance of Z_1. In this case, $\tilde{G}_{\mathcal{S}_0^1}(Z_1)$ must be added. $\tilde{G}_{\mathcal{S}_0^1}(Z_1)$ is used because $G_{\mathcal{S}_0^1}(Z_1)$ is $\mathcal{S}_F^1$, the accepting state of $\mathfrak{I}_1$, and has the entry $A(\#_1) =$ *Accept*. This entry is not needed in the new state, since it indicates that all of z_1 has been reduced to Z_1, in which case the actual lookahead symbol will be a member of $L(G_{\mathcal{S}}(\bar{z}_1))$. If Z_1 cannot generate a string $Z_1\alpha$ with $\alpha \neq \epsilon$, then $\tilde{G}_{\mathcal{S}_0^1}(Z_1)$ is null and a union state is not actually formed by (ii).

At this point, all updating of $\mathfrak{I}_0$ has been completed and the correct transfer of control from the $\mathfrak{I}_0$-mode to the $\mathfrak{I}_1$-mode has been effected. For the input $xz_1y\#$, parsing of z_1 as Z_1 now continues in $\mathfrak{I}_1$, since the goto states generated in (i) above (neglecting union states) are in $\mathfrak{I}_1$. When the initial character of y is first encountered as the lookahead symbol, $\mathfrak{I}$ is in some state $\mathcal{S}$ of $\mathfrak{I}_1$ such that $\#_1 \in L(\mathcal{S})$; but the actual lookahead symbol is some member of the set $\text{Follow}_{G_0'}(\bar{z}_1)$.[9] Correct parsing requires that $L(\mathcal{S})$ be updated by replacing $\#_1$ by $\text{Follow}_{G_0'}(\bar{z}_1)$. The action for each new member of $L(\mathcal{S})$ should be set to $A_{\mathcal{S}}(\#_1)$, since any one of these symbols can now signal the end of Z_1, just as $\#_1$ did for $\mathfrak{I}_1$. This updating should be done for all states $\mathcal{S}$ in $\mathfrak{I}_1$ for which $\#_1 \in L(\mathcal{S})$, except $\mathcal{S}_F^1$. ($\mathcal{S}_F^1$ appears in $\mathfrak{I}$ only, if at all, as part of the union state formed in (ii) above; it may be deleted.) As was the case in updating states of $\mathfrak{I}_0$, conflicts must not be generated by the symbols

[8] A detailed description of union state formation appears later.

[9] For any grammar G with starting symbol S and any terminal symbol b, $\text{Follow}_G(b) = \{a \mid \exists \alpha,\beta \in V^*$ such that $S \Rightarrow \alpha ba\beta$ in $G\}$.

which replace $\#_1$. That is, if any symbol b was in $L(\mathcal{S})$ before updating and is also in $\text{Follow}_{G_0'}(\bar{z}_1)$, then $A_{\mathcal{S}}(b)$ is required to have been the same as $A_{\mathcal{S}}(\#_1)$.

Note that replacing $\#_1$ by the entire set $\text{Follow}_{G_0'}(\bar{z}_1)$ may delay the discovery of certain errors in the input string, since, for any particular prefix tz_1 of a sentence, only *some* elements of $\text{Follow}_{G_0'}(\bar{z}_1)$ might follow *this instance* of z_1. However, strings not in $L(G)$ will eventually be rejected by $\mathfrak{Z}$, since an LR(1) processor reduces a string ω to Y only if $Y \Rightarrow \omega$ and accepts an input only if it is reduced to the starting symbol.

The updating of $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$ is now complete, since the switch back to the $\mathfrak{Z}_0$-mode after z_1 has been reduced to Z_1 has been taken care of: once the reduction to Z_1 has been done, the goto state is computed from the state $\mathcal{S}$ which appears below Z_1 on the stack; but $\mathcal{S}$ is in $\mathfrak{Z}_0$ and $\bar{z}_1 \in S(\mathcal{S})$, and $G_{\mathcal{S}}(Z_1)$ was computed above. The table $\mathfrak{Z}$ is the union of the updated $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$; its start state is $\mathcal{S}_0^0$ and its accepting state is $\mathcal{S}_F^0$. Occurrences of $\#_0$ should be replaced by $\#$.

5. An Example

Table I shows a grammar G and its parsing table $\mathfrak{K}$, generated by the Knuth algorithm. Its format is as described in Section 2, except for two additional columns: the partial states of each state are listed under the heading $\mathcal{S}$; the partial states added to $\mathcal{S}$ to form $\mathcal{S}'$ are listed under $\mathcal{S}'-\mathcal{S}$. A compact notation for partial states has been used: "*pjabc*" denotes $\{[p, j, a], [p, j, b], [p, j, c]\}$, etc. Tables II and III show the partition grammars G_0 and G_1 and their respective parsing tables $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$, also generated by the Knuth algorithm. Finally, Table IV shows the parsing table $\mathfrak{Z}$ formed from $\mathfrak{Z}_0$ and $\mathfrak{Z}_1$.

The parsing table $\mathfrak{Z}$ has fewer states than $\mathfrak{K}$. Very roughly, this savings occurs because $\mathfrak{Z}$ uses a single closed subroutine to parse all substrings generated by Z_1, whereas $\mathfrak{K}$ may use several pieces of in-line code. In this example, once an "a" has been found, indicating that a Z_1 is being sought, $\mathfrak{Z}$ transfers to the "subroutine" consisting of the states which were originally in $\mathfrak{Z}_1$—$\mathcal{S}_2^1$, $\mathcal{S}_3^1$, $\mathcal{S}_4^1$, and $\mathcal{S}_5^1$. $\mathfrak{Z}$ ignores the information that the first occurrence of Z_1 must be followed by b or c and that the second occurrence must be followed by c or d; while in the subroutine, $\mathfrak{Z}$ allows any of b, c, or d to follow. Only when it exits from the subroutine and returns to the $\mathfrak{Z}_0$-mode does $\mathfrak{Z}$ check the validity of this symbol, thereby causing delayed error checking in some cases. On the other hand, $\mathfrak{K}$ has two sets of states which search for an occurrence of Z_1 once the initial a has been found—$\mathcal{S}_3$, $\mathcal{S}_7$, $\mathcal{S}_8$, $\mathcal{S}_{11}$ and $\mathcal{S}_{10}$, $\mathcal{S}_{15}$, $\mathcal{S}_{16}$, $\mathcal{S}_{17}$.[10] Each of these states stores the information as to which symbols can follow a *particular occurrence* of Z_1. Thus $\mathfrak{K}$ checks the validity of the symbol following Z_1 when it first becomes the lookahead symbol and so error checking is immediate.

[10] In general, each of these sets which is "repeated" in $\mathfrak{K}$ is not as large as the "subroutine" in $\mathfrak{Z}$, which consists of all states in $\mathfrak{Z}_1$ except $\mathcal{S}_0^1$ and $\mathcal{S}_F^1$.

TABLE I. Grammar G and Parsing Table $\mathfrak{K}$

$G = (\{Z_0, Z_1, A, B, C\}, \{a, b, c, d, e\}, \mathcal{P}, Z_0)$

1 $Z_0 \to Z_1AZ_1B$	5 $B \to c$
2 $Z_1 \to aC$	6 $B \to d$
3 $A \to b$	7 $C \to Ce$
4 $A \to c$	8 $C \to e$

State Name	$\mathcal{S}$	$\mathcal{S}'-\mathcal{S}$	Lookahead Symbol	Action	Stack Symbol	Goto State
$\mathcal{S}_0$	00#	10# 20*bc*	*a*	Shift	Z_0 Z_1 *a*	$\mathcal{S}_1$ $\mathcal{S}_2$ $\mathcal{S}_3$
$\mathcal{S}_1$	01#		#	Accept		
$\mathcal{S}_2$	11#	30*a* 40*a*	*b* *c*	Shift Shift	*A* *b* *c*	$\mathcal{S}_4$ $\mathcal{S}_5$ $\mathcal{S}_6$
$\mathcal{S}_3$	21*bc*	70*bce* 80*bce*	*e*	Shift	*C* *e*	$\mathcal{S}_7$ $\mathcal{S}_8$
$\mathcal{S}_4$	12#	20*cd*	*a*	Shift	Z_1 *a*	$\mathcal{S}_9$ $\mathcal{S}_{10}$
$\mathcal{S}_5$	31*a*		*a*	Reduce 3		
$\mathcal{S}_6$	41*a*		*a*	Reduce 4		
$\mathcal{S}_7$	22*bc* 71*bce*		*b* *c* *e*	Reduce 2 Reduce 2 Shift	*e*	$\mathcal{S}_{11}$
$\mathcal{S}_8$	81*bce*		*b* *c* *e*	Reduce 8 Reduce 8 Reduce 8		
$\mathcal{S}_9$	13#	50# 60#	*c* *d*	Shift Shift	*B* *c* *d*	$\mathcal{S}_{12}$ $\mathcal{S}_{13}$ $\mathcal{S}_{14}$
$\mathcal{S}_{10}$	21*cd*	70*cde* 80*cde*	*e*	Shift	*C* *e*	$\mathcal{S}_{15}$ $\mathcal{S}_{16}$
$\mathcal{S}_{11}$	72*bce*		*b* *c* *e*	Reduce 7 Reduce 7 Reduce 7		
$\mathcal{S}_{12}$	14#		#	Reduce 1		
$\mathcal{S}_{13}$	51#		#	Reduce 5		
$\mathcal{S}_{14}$	61#		#	Reduce 6		
$\mathcal{S}_{15}$	22*cd* 71*cde*		*c* *d* *e*	Reduce 2 Reduce 2 Shift	*e*	$\mathcal{S}_{17}$
$\mathcal{S}_{16}$	81*cde*		*c* *d* *e*	Reduce 8 Reduce 8 Reduce 8		
$\mathcal{S}_{17}$	72*cde*		*c* *d* *e*	Reduce 7 Reduce 7 Reduce 7		

TABLE II. GRAMMAR G_0 AND PARSING TABLE $\mathfrak{I}_0$

$G_0 = (\{Z_0, A, B\}, \{b, c, d\}, \mathcal{P}_0, Z_0)$

$1\ Z_0 \rightarrow \bar{z}_1 A \bar{z}_1 B \qquad 5\ B \rightarrow c$
$3\ A \rightarrow b \qquad 6\ B \rightarrow d$
$4 \rightarrow c$

State Name	$\mathcal{S}$	$\mathcal{S}' - \mathcal{S}$	Lookahead Symbol	Action	Stack Symbol	Goto State
$\mathcal{S}_0^0$	$00\#_0$	$10\#_0$	$\bar{z}_1$	Shift	Z_0 $\bar{z}_1$	$\mathcal{S}_1^0$ $\mathcal{S}_2^0$
$\mathcal{S}_1^0$	$01\#_0$		$\#_0$	Accept		
$\mathcal{S}_2^0$	$11\#_0$	$30\bar{z}_1$ $40\bar{z}_1$	b c	Shift Shift	A b c	$\mathcal{S}_3^0$ $\mathcal{S}_4^0$ $\mathcal{S}_5^0$
$\mathcal{S}_3^0$	$12\#_0$		$\bar{z}_1$	Shift	$\bar{z}_1$	$\mathcal{S}_6^0$
$\mathcal{S}_4^0$	$31\bar{z}_1$		$\bar{z}_1$	Reduce 3		
$\mathcal{S}_5^0$	$41\bar{z}_1$		$\bar{z}_1$	Reduce 4		
$\mathcal{S}_6^0$	$13\#_0$	$50\#_0$ $60\#_0$	c d	Shift Shift	B c d	$\mathcal{S}_7^0$ $\mathcal{S}_8^0$ $\mathcal{S}_9^0$
$\mathcal{S}_7^0$	$14\#_0$		$\#_0$	Reduce 1		
$\mathcal{S}_8^0$	$51\#_0$		$\#_0$	Reduce 5		
$\mathcal{S}_9^0$	$61\#_0$		$\#_0$	Reduce 6		

TABLE III. GRAMMAR G_1 AND PARSING TABLE $\mathfrak{I}_1$

$G_1 = (\{Z_1, C\}, \{a, e\}, \mathcal{P}_1, Z_1)$

$2\ Z_1 \rightarrow aC$
$7\ C \rightarrow Ce$
$8\ C \rightarrow e$

State Name	$\mathcal{S}$	$\mathcal{S}' - \mathcal{S}$	Lookahead Symbol	Action	Stack Symbol	Goto State
$\mathcal{S}_0^1$	$00\#_1$	$20\#_1$	a	Shift	Z_1 a	$\mathcal{S}_1^1$ $\mathcal{S}_2^1$
$\mathcal{S}_1^1$	$01\#_1$		$\#_1$	Accept		
$\mathcal{S}_2^1$	$21\#_1$	$70\#_1 e$ $80\#_1 e$	e	Shift	C e	$\mathcal{S}_3^1$ $\mathcal{S}_4^1$
$\mathcal{S}_3^1$	$22\#_1$ $71\#_1 e$		$\#_1$ e	Reduce 2 Shift	e	$\mathcal{S}_5^1$
$\mathcal{S}_4^1$	$81\#_1 e$		$\#_1$ e	Reduce 8 Reduce 8		
$\mathcal{S}_5^1$	$72\#_1 e$		$\#_1$ e	Reduce 7 Reduce 7		

TABLE IV. PARSING TABLE $\mathfrak{I}$ CONSTRUCTED FROM $\mathfrak{I}_0$ AND $\mathfrak{I}_1$

$\text{Follow}_{G_0'}(\bar{z}_1) = \{b, c, d\}$

State Name	Lookahead Symbol	Action	Stack Symbol	Goto State
$\mathcal{S}_0^0$	a	Shift	Z_0 Z_1 a	$\mathcal{S}_1^0$ $\mathcal{S}_2^0 \cup \tilde{\mathcal{S}}_1^1 = \mathcal{S}_2^0$ $\mathcal{S}_2^1$
$\mathcal{S}_1^0$	$\#$	Accept		
$\mathcal{S}_2^0$	b c	Shift Shift	A b c	$\mathcal{S}_3^0$ $\mathcal{S}_4^0$ $\mathcal{S}_5^0$
$\mathcal{S}_3^0$	a	Shift	Z_1 a	$\mathcal{S}_6^0 \cup \tilde{\mathcal{S}}_1^1 = \mathcal{S}_6^0$ $\mathcal{S}_2^1$
$\mathcal{S}_4^0$	a	Reduce 3		
$\mathcal{S}_5^0$	a	Reduce 4		
$\mathcal{S}_6^0$	c d	Shift Shift	B c d	$\mathcal{S}_7^0$ $\mathcal{S}_8^0$ $\mathcal{S}_9^0$
$\mathcal{S}_7^0$	$\#$	Reduce 1		
$\mathcal{S}_8^0$	$\#$	Reduce 5		
$\mathcal{S}_9^0$	$\#$	Reduce 6		
$\mathcal{S}_2^1$	e	Shift	C e	$\mathcal{S}_3^1$ $\mathcal{S}_4^1$
$\mathcal{S}_3^1$	b c d e	Reduce 2 Reduce 2 Reduce 2 Shift	e	$\mathcal{S}_5^1$
$\mathcal{S}_4^1$	b c d e	Reduce 8 Reduce 8 Reduce 8 Reduce 8		
$\mathcal{S}_5^1$	b c d e	Reduce 7 Reduce 7 Reduce 7 Reduce 7		

6. General Algorithm

The algorithm described in this section generalizes the simple scheme by allowing the grammar to be divided into more than two parts. For $G = (V_N, V_T, \mathcal{P}', Z_0)$, let $Z_1, \cdots, Z_n$ be distinct nonterminals other than Z_0, and let $\bar{z}_0, \cdots, \bar{z}_n$ be new terminal symbols not in V. Number the members of $\mathcal{P}$ from 1 to π. Let $G' = (V_N', V_T', \mathcal{P}, Z_0')$ be the 1-augmented grammar associated with G, and assign the production $Z_0' \rightarrow Z_0\#$ the number 0. For $0 \leq i \leq n$,

let $G_i = (V_{N_i}, V_{T_i}, \mathcal{P}_i, Z_i)$ be the reduced form of $(V_N, \bar{V}_T, \mathcal{P}_i, Z_i)$, where $\bar{V}_T = V_T \cup \{\bar{z}_j \mid 0 \leq j \leq n\}$ and $\mathcal{P}_i = \{A \to \bar{\alpha} \mid A \to \alpha \in \mathcal{P}$ and $\bar{\alpha}$ is obtained from α by replacing every occurrence of Z_j $(j \neq i)$ by $\bar{z}_j\}$. Assign the rule $A \to \bar{\alpha}$ in $\mathcal{P}_i$ the number that $A \to \alpha$ has in $\mathcal{P}$. For each i, let $G_i' = (V_{N_i}', V_{T_i}', \mathcal{P}_i', Z_i')$ be the 1-augmented grammar associated with G_i, with endmarker $\#_i$.

Thus G_0 is analogous to the G_0 of the simple case, and each G_i is similar to the G_1 of the simple case. The composite parsing table $\mathfrak{Z}$ will operate as follows. It starts in the $\mathfrak{Z}_0$-mode. When a substring generated by Z_i is encountered, it switches to the $\mathfrak{Z}_i$-mode; if this substring itself contains a substring generated by Z_j, it will have to switch to the $\mathfrak{Z}_j$-mode before returning to the $\mathfrak{Z}_0$-mode. Thus for each pair (i, j) such that $\bar{z}_j \in V_{T_i}$, we must make sure that the master processor is capable of finding the beginning and end of strings generated by Z_j when they are included in strings generated by Z_i. The algorithm for constructing a parsing table $\mathfrak{Z}$ for G from the tables for the G_i is based on the same reasoning used for the simple case of Section 4.

Construction Algorithm. A parsing table $\mathfrak{Z}_i$ is generated for each G_i using the Knuth algorithm. The following algorithm, if it does not FAIL, will produce an updated version of each $\mathfrak{Z}_i$. The composite parsing table $\mathfrak{Z}$ for G is then formed as the union of the resulting tables (and possibly some union states), with start state $\mathcal{S}_0^0$ and final state $\mathcal{S}_F^0$. The algorithm is specified by the following two flow charts: MAIN ALGORITHM (Figure 1) and subroutine UPDATE($\mathcal{S}$) (Figure 2). No doubt the reader will see a number of ways in which the efficiency of the algorithm could be improved by the addition of various programming switches, some reorganization, etc.; however, the purpose here is to describe the algorithm in the most convenient way.

Box 1: All $\#_0$ entries in $\mathfrak{Z}_0$ are duplicated with $\#$. This allows $\#_0$ to be treated like the other $\#_j$'s when $\bar{z}_0$ appears in other tables; $\#$ is the actual endmarker appended to input strings.

Box 2: This initializes $\mathfrak{Z}_{n+1}$, the "table" composed of any union states that will be generated.

Boxes 3–9: This loop computes the set FOL(i) for each subgrammar. FOL(i) includes those symbols that could follow instances of Z_i in G', except those that could do so only in the ith subgrammar. It is analogous to $\text{Follow}_{G0}'(\bar{z}_1)$ in the simple case. The set HIST is used to eliminate looping.

Boxes 11–14: This loop updates the start states by calls to the subroutine UPDATE, which replaces occurrences of any $\bar{z}_j$ or $\#_j$ in these states. Although updating of the start states need not be done in advance, this saves considerable duplication of computation later. HIST is used to guarantee termination of the UPDATE subroutine.

Boxes 16–19: These update the remaining states in the same way.

Boxes 20–26: Some union states may have been created by the calls of UPDATE in boxes 12 or 17. Their table

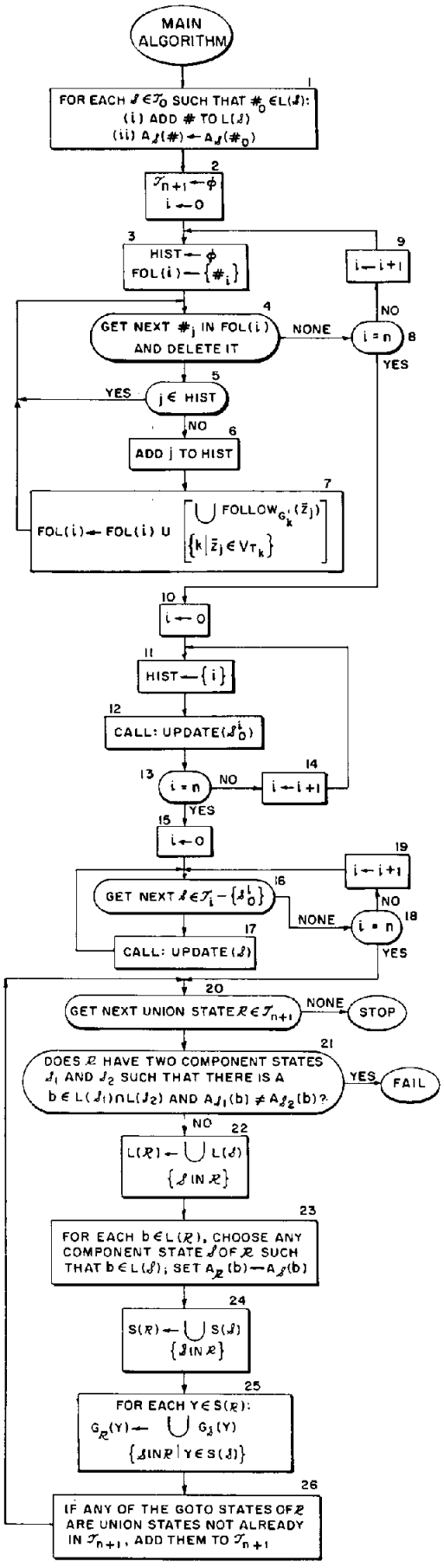

FIG. 1

entries are computed in boxes 21–26 by unioning the table entries of their component states. Since the component states have already been updated, no $\bar{z}_j$'s or $\#_j$'s are involved. However box 21 must check to make sure no lookahead symbol would be assigned conflicting actions by this process. Additional union states may be generated by box 25; these are added to $\mathfrak{I}_{n+1}$ if they are not already

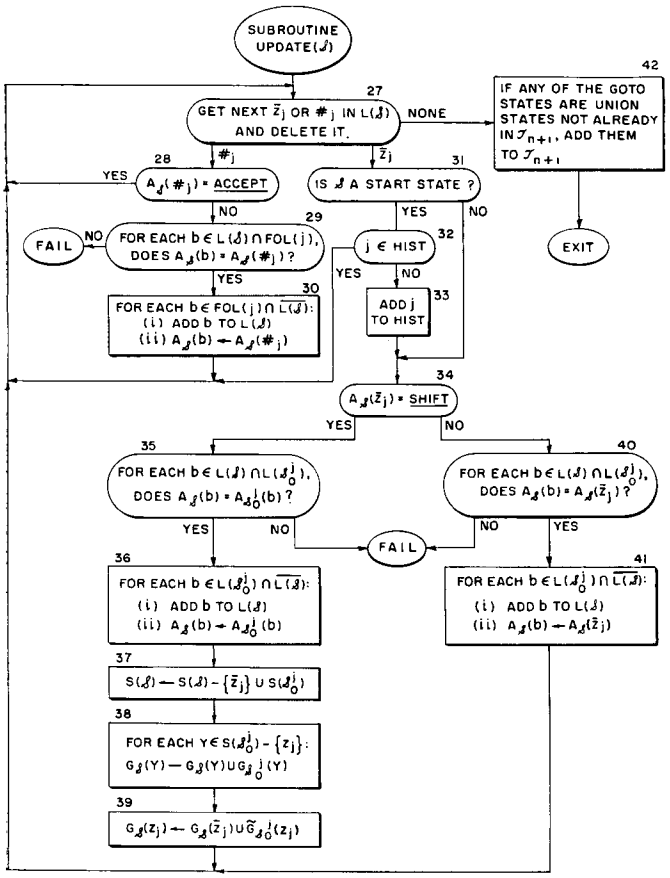

FIG. 2

there. The algorithm terminates when the table entries for each union state have been computed. The notation {S in $\mathfrak{R}$} is used to denote the set of component states of $\mathfrak{R}$.

The subroutine UPDATE, defined in the flowchart of Figure 2, is the heart of the algorithm. It is completely analogous to the updating method used in the simple case: it replaces every occurrence of $\bar{z}_j$ or $\#_j$ by the symbols in V_T' that might actually occur in the same position in a sentence of G'. If no conflicts arise, these new symbols are assigned appropriate actions, and the stack and goto entries are updated accordingly.

Boxes 28–30: If an $\#_j$ has been found in box 27, it should be replaced by the symbols in FOL(j). However, if S is a final state, the replacement need not be made, since this entry will not appear in table $\mathfrak{I}$. Box 29 makes sure conflicting actions will not be assigned to the symbols being added to $L(\mathfrak{S})$. Finally, box 30 makes the replacement and assigns $A_{\mathfrak{S}}(\#_j)$ to each new symbol, as in the simple case.

Boxes 31–33: If S is a start state, there is a possibility that even though $\bar{z}_j$ was once deleted from $L(\mathfrak{S})$ in box 27, it has since been added to $L(\mathfrak{S})$ by boxes 36 or 41 (see below). In this case, to avoid looping, $\bar{z}_j$ is not processed again.

Boxes 35–39: These update S due to the presence of $\bar{z}_j$ in $L(\mathfrak{S})$ with $A_{\mathfrak{S}}(\bar{z}_j) = \mathit{Shift}$. In this case, $\bar{z}_j$ is replaced by the symbols in $L(\mathfrak{S}_0^j)$, each of which retains the action it had in $\mathfrak{S}_0^j$. The updating process FAILs if any conflicts would be generated. The stack symbols and their goto states are computed in exactly the same way as in the simple case. Note that once the start states have been updated (boxes 11–14), $L(\mathfrak{S}_0^j)$ contains no $\bar{z}_k$'s or $\#_k$'s; thus, all symbols added to $L(\mathfrak{S})$ in box 36 are in V_T'. On the other hand, during the updating of start states, box 36 (as well as box 41) may introduce additional $\bar{z}_k$'s and $\#_k$'s which must be removed by later execution of box 27. This is why box 32 is needed.

Boxes 40–41: If the $\bar{z}_j$ in $L(\mathfrak{S})$ has the action *Reduce p*, it is replaced by the symbols in $L(\mathfrak{S}_0^j)$, which are assigned the action *Reduce p*, if no conflicts arise.

Box 42: When all $\bar{z}_j$'s and $\#_j$'s have been removed from $L(\mathfrak{S})$, the updating is complete. Any new union states which have been created during the updating of S are added to $\mathfrak{I}_{n+1}$. Their table entries are computed later in boxes 20–26.

It should be clear that the additional complexity of the general algorithm is not due to any inherent difference between it and the method used for the simple case. It is only a matter of keeping track of some additional details.

It is shown in [5] that any grammar for which this algorithm succeeds is LR(1), since the Knuth algorithm would also have succeeded in producing an LR(1) parsing table for G. The converse, however, is not true: there exist LR(1) grammars for which certain partitions will cause the general algorithm to fail.

7. An LR(1) Grammar for ALGOL

Obtaining a parsing table using the partitioning scheme is somewhat of a trial-and-error process, since a partition must be chosen and the corresponding tables $\mathfrak{I}_i$ generated before the updating and merging process can begin. If a particular partition fails to produce an LR(1) parsing table, the entire process may have to be repeated. However, in practical situations the partitioning scheme requires very few iterations. The difficulties that do arise are more often due to non-LR(1) subgrammars than to poor partition choices. Using this scheme, parsers for ALGOL, BASIC and CDL1[11] have been generated. In this section, the process of producing an LR(1) parser for ALGOL will be briefly described.

[11] The preliminary syntax of a computer description language; there are 350 grammar rules [7].

ALGOL is defined by a set of Backus Normal Form syntax rules and some additional "semantic" rules [6]. These semantic rules include certain restrictions which imply that the set of all well-formed ALGOL programs is not even context-free, much less deterministic. However, our interest here is in efficient syntax-directed compilation and a language specification that will be useful for this purpose. To this end, we define the structure of ALGOL by a grammar G, given in [5]. G is not identical to the official ALGOL syntax—in fact, it does not define the same set of strings as ALGOL's BNF rules. But, combined with an appropriate semantic interpretation, it does define the same language as the ALGOL BNF rules together with the ALGOL semantics. It is a matter of shifting the dividing line between syntax and semantics. The main differences are:

1. For convenience and efficiency in applying Knuth's algorithm to various G_i, certain finite sets of basic ALGOL symbols are replaced by a single terminal symbol in G, e.g. *type*, *ad.op*, *digit*, *string.symbol*, etc.

2. In a very few instances changes were made in the ALGOL grammar in order to obtain an LR(1) structure. These changes do not affect the strings generated and do not cause any meaningful change in syntactic structure. For example, the syntax of ⟨string⟩ was changed to eliminate an ambiguity.

3. The symbols *let.string* and *label* are terminal symbols in G. The addition of grammars for these structures would cause no difficulties in proving G to be LR(1), and is omitted for simplicity. Integer labels are not allowed, since an integer appearing as an actual parameter would be syntactically ambiguous: Is it to be parsed as a ⟨designational expression⟩ or as an ⟨arithmetic expression⟩? This restriction is common in ALGOL implementations.

4. The following *terminal* symbols appear in G: *identifier*, *arithmetic identifier*, *Boolean identifier*, *arithmetic array identifier*, *Boolean array identifier*, *arithmetic function identifier*, *Boolean function identifier*, *switch identifier*, *procedure identifier*. These symbols (and their interpretation) constitute the only major difference between G and the standard ALGOL syntax. They syntactically reflect certain semantic restrictions on an ALGOL program. Their interpretation (with respect to an ALGOL compiler which is based on G) is as follows. Any string parsable as an ⟨identifier⟩ is to be accepted by the compiler in place of any one of the nine symbols listed above. In the ⟨declaration⟩ subgrammar (the only place where *identifier* appears), successful recognition of such a string will cause a symbol table entry to be made, consisting of the string itself and an attribute indicating how it was declared. The parsing then proceeds as if the symbol *identifier* had been encountered. In all other subgrammars (where only the other eight types of identifiers appear), successful recognition of an ⟨identifier⟩ string in place of one of these symbols will cause the compiler to check the symbol table entry for that string to make sure that its attributes correspond to one of the identifiers allowable in this particular instance. If this check succeeds, parsing continues as if the corresponding terminal symbol had been encountered.

The validity of this interpretation of G depends on the assumption that the compiler framework in which it is embedded as a block-structured symbol table with the necessary identifier entries and their attributes available at the time of use. This is a reasonable assumption, since the ALGOL declaration structure lends itself to this general parsing philosophy. The point is that the relation of G to the official ALGOL syntax is such that an ALGOL compiler could be constructed around the LR(1) parsing table $\mathfrak{Z}$ for G which will permit particularly efficient processing. The discussion above is meant to justify this point. It is not intended to describe the detailed construction of a compiler.

The initial problem in applying the algorithm is to choose a partition for the given grammar. There are no steadfast rules on how this is to be done. The number of nonterminals chosen, n, is affected by a trade-off in the effort required to complete the two parts of the procedure. A large value of n will result in small subgrammars, to which Knuth's algorithm may be easily applied, but the construction of $\mathfrak{Z}$ from the $\mathfrak{Z}_i$ will be more complex. On the other hand, the algorithm is more likely to succeed if n is small. The following guidelines may be useful in determining how to choose the partitioning nonterminals.

1. Each G_i should be small enough so that Knuth's algorithm can be easily applied.

2. Choice of a Z_i whose initial symbols appear infrequently in other parts of the grammar decreases the work required to construct $\mathfrak{Z}$ and increases the chances of success.

3. Choice of a Z_i which appears in many different parts of the grammar will help to reduce the size of $\mathfrak{Z}$.

4. Choice of a pair of Z_i, each of which appears in the subgrammar of the other, will markedly decrease the effort needed to apply Knuth's algorithm and will help reduce the size of $\mathfrak{Z}$.

In the case of a grammar defining the syntax of a programming language, some choices may be rather natural. In the case of grammar G for ALGOL, the choice of ⟨declaration⟩ and ⟨statement⟩ was made first. The initial symbols of ⟨declaration⟩ do not appear in the grammar for ⟨statement⟩. Since ⟨arithmetic expression⟩ and ⟨Boolean expression⟩ are widely used throughout G, they were chosen next. This still left a rather large grammar for ⟨statement⟩; so ⟨actual parameter part⟩ and ⟨designational expression⟩ were chosen. Finally, ⟨simple arithmetic expression⟩ and ⟨simple Boolean⟩ were chosen because of their complex intertwining with ⟨arithmetic expression⟩, and to avoid duplication of the large number of rules defining ⟨simple arithmetic expression⟩ in both ⟨arithmetic expression⟩ and in ⟨Boolean expression⟩.

Once the partition was chosen, the grammar and partitioning nonterminals were used as input to a program

TABLE V

z_i	*Computation Time*	*Number of States in* $\mathfrak{z}_i$
⟨program⟩	1	6
⟨unlabeled program⟩	1	15
⟨declaration⟩	4	77
⟨statement⟩	12	110
⟨actual parameter part⟩	2	32
⟨arithmetic expression⟩	1	10
⟨simple arithmetic expression⟩	6	45
⟨Boolean expression⟩	1	15
⟨simple Boolean⟩	5	38
⟨designational expression⟩	2	42
union states	1	53

which does the following:

1. Forms the subgrammars and applies the Knuth algorithm to each.
2. Updates each $\mathfrak{z}_i$ and forms the union states.
3. Encodes the resulting table $\mathfrak{z}$ into assembly language format.

The table is then appended to a simple interpreter to form a parser for the given language, in this case ALGOL. Table V indicates the results for ALGOL.

The following is an example of the savings effected by the partitioning scheme: Instead of treating ⟨simple arithmetic expression⟩ as a $\bar{z}_j$ in the grammar for ⟨arithmetic expression⟩, as was done above, the Knuth algorithm was applied to the ⟨arithmetic expression⟩ grammar directly. (The other $\bar{z}_j$'s were retained.) The result was a parsing table with three times as many states as the old ⟨arithmetic expression⟩ and ⟨simple arithmetic expression⟩ tables combined, and which took nine times as long to generate.

8. Conclusion

LR(k) grammars are useful syntactic descriptors for programming languages because they combine two important features: processing efficiency and wide applicability. The original algorithm given by Knuth for testing a grammar for the LR( ) condition and automatically constructing a parser for its sentences has a serious drawback: it is not practical for large grammars. The partitioning scheme described in this paper significantly reduces the computation required to generate an LR(1) parsing table for a large grammar. In addition, the table it produces is usually smaller than the table that would have been produced if the Knuth algorithm were applied directly.

Knuth has also suggested a way of decreasing the computation time for an LR(k) table. Namely, if a state is generated that is a subset of another state, the latter state can be substituted for the former and the table entries for the former state need not be computed. This shortcut can, of course, be used in conjunction with the partitioning scheme when generating the individual $\mathfrak{z}_i$. We have found, however, that the savings is negligible. Knuth also points out that any two states whose parsing action does not conflict can be merged into a single state to cut down on the size of the table. Again, this technique can be applied along with the partitioning method to both the individual $\mathfrak{z}_i$ and the final table $\mathfrak{z}$.

Additional savings in both time and storage can be obtained by various implementation techniques. For instance, the bottom track of the stack (the track containing vocabulary symbols) is needed only if one of the two kinds of shortcuts suggested by Knuth are used. In that case, when the action *Reduce* p is called for, it is necessary to check whether $X_{p1} \cdots X_{pn_p}$ is at the top of the stack; otherwise, this check is not needed. It is also useful to distinguish between states which have only one action and those with more than one action. In particular, no lookahead symbols need be stored for a state whose only action is *Reduce* p. Whenever the processor is in such a state, it can perform the reduce action without examining the lookahead symbol; delayed error-checking will result, but no incorrect strings will be accepted.

A number of problems remain to be studied. Considering the trade-off between the time it takes to form the individual $\mathfrak{z}_i$, the time required to update and combine them and the chances of successfully doing so, how many ways should a grammar be partitioned? What are some reasonable heuristics to guide the choice of the partitioning nonterminals? Although an LR(1) processor (even with delayed error-checking) can point to the leftmost unacceptable symbol in a nonsentence, what additional automatic error analysis can be incorporated into the scheme? Perhaps more important, how can error recovery be included? Since the stack contains very detailed information about the input already processed, the possibility of good error processing seems better here than in less formalized processors.

RECEIVED SEPTEMBER, 1967; REVISED MARCH, 1969

REFERENCES

1. EARLEY, J. An efficient context-free parsing algorithm. Thesis, Carnegie-Mellon U., Pittsburgh, Pa., Aug., 1968.
2. FELDMAN, J., AND GRIES, D. Translator writing systems. *Comm. ACM 11*, 2 (Feb. 1968), 77–113.
3. GINSBURG, S., AND GREIBACH, S. A. Deterministic context-free languages. *Inform. Contr. 9* (Dec. 1966), 620–648.
4. KNUTH, D. E. On the translation of languages from left to right. *Inform. Contr. 8* (Dec. 1965), 607–639.
5. KORENJAK, A. J. Deterministic language processing. Thesis, Princeton, U., Princeton, N. J., Sept. 1967.
6. NAUR, P. (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6*, 1 (Jan. 1963), 1–17.
7. SRINIVASAN, C. V. An introduction to CDL1, a computer description language. Scientific Rep. No. 1, Contract No. AF19(628)4789, RCA Laboratories, Princeton, N. J., Sept. 1967.