INFORMATION AND CONTROL 8, 607–639 (1965)

On the Translation of Languages from Left to Right

DONALD E. KNUTH

Mathematics Department, California Institute of Technology, Pasadena, California

There has been much recent interest in languages whose grammar is sufficiently simple that an efficient left-to-right parsing algorithm can be mechanically produced from the grammar. In this paper, we define LR(k) grammars, which are perhaps the most general ones of this type, and they provide the basis for understanding all of the special tricks which have been used in the construction of parsing algorithms for languages with simple structure, e.g. algebraic languages. We give algorithms for deciding if a given grammar satisfies the LR(k) condition, for given k, and also give methods for generating recognizers for LR(k) grammars. It is shown that the problem of whether or not a grammar is LR(k) for *some* k is undecidable, and the paper concludes by establishing various connections between LR(k) grammars and deterministic languages. In particular, the LR(k) condition is a natural analogue, for grammars, of the deterministic condition, for languages.

I. INTRODUCTION AND DEFINITIONS

The word "language" will be used here to denote a set of character strings which has been variously called a *context free language*, a (*simple*) *phrase structure language*, a *constituent-structure language*, a *definable set*, a *BNF language*, a *Chomsky type 2* (*or type 4*) *language*, a *push-down automaton language*, etc. Such languages have aroused wide interest because they serve as approximate models for natural languages and computer programming languages, among others. In this paper we single out an important class of languages which will be called *translatable from left to right*; this means if we read the characters of a string from left to right, and look a given finite number of characters ahead, we are able to parse the given string without ever backing up to consider a previous decision. Such languages are particularly important in the case of computer programming, since this condition means a parsing algorithm can be mechanically constructed which requires an execution time at worst proportional to the length of the string being parsed. Special-purpose

methods for translating computer languages (for example, the well-known precedence algorithm, see Floyd (1963)) are based on the fact that the languages being considered have a simple left-to-right structure. By considering all languages that are translatable from left to right, we are able to study all of these special techniques in their most general framework, and to find for a given language and grammar the "best possible" way to translate it from left to right. The study of such languages is also of possible interest to those who are investigating human parsing behavior, perhaps helping to explain the fact that certain English sentences are unintelligible to a listener.

Now we proceed to give precise definitions to the concepts discussed informally above. The well-known properties of *characters* and *strings* of characters will be assumed. We are given two disjoint sets of characters, the "*intermediates*" I and the "*terminals*" T; we will use upper case letters A, B, C, $\cdots$ to stand for intermediates, and lower case letters a, b, c, $\cdots$ to stand for terminals, and the letters X, Y, Z will be used to denote either intermediates or terminals. The letter S denotes the "principal intermediate character" which has special significance as explained below. Strings of characters will be denoted by lower case Greek letters α, β, γ, $\cdots$, and the *empty string* will be represented by ϵ. The notation α^n denotes n-fold concatenation of string α with itself; $\alpha^0 = \epsilon$, and $\alpha^n = \alpha\alpha^{n-1}$. A *production* is a relation $A \rightarrow \theta$ where A is in I and θ is a string on $I \cup T$; a *grammar* $\mathcal{G}$ is a set of productions. We write $\varphi \rightarrow \psi$ (with respect to $\mathcal{G}$, a grammar which is usually understood) if there exist strings α, θ, ω, A such that $\varphi = \alpha A \omega$, $\psi = \alpha\theta\omega$, and $A \rightarrow \theta$ is a production in $\mathcal{G}$. The transitive completion of this relation is of principal importance: $\alpha \Rightarrow \beta$ means there exist strings $\alpha_0, \alpha_1, \cdots, \alpha_n$ (with $n > 0$) for which $\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \cdots \rightarrow \alpha_n = \beta$. Note that by this definition it is not necessarily true that $\alpha \Rightarrow \alpha$; we will write $\alpha \Rrightarrow \beta$ to mean $\alpha = \beta$ or $\alpha \Rightarrow \beta$. A grammar is said to be *circular* if $\alpha \Rightarrow \alpha$ for some α. (Some of this notation is more complicated than we would need for the purposes of the present paper, but it is introduced in this way in order to be compatible with that used in related papers.) The *language defined by* $\mathcal{G}$ is

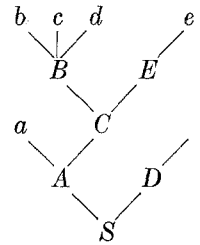
$$\{\alpha \mid S \Rightarrow \alpha \text{ and } \alpha \text{ is a string over } T\}, \tag{1}$$

namely, the set of all terminal strings derivable from S by using the productions of $\mathcal{G}$ as substitution rules. A *sentential form* is any string α for which $S \Rightarrow \alpha$.

For example, the grammar

$$S \to AD,\ A \to aC,\ B \to bcd,\ C \to BE,\ D \to \epsilon,\ E \to e \tag{2}$$

defines the language consisting of the single string "*abcde*". Any sentential form in a grammar may be given at least one representation as the leaves of a *derivation tree* or "parse diagram"; for example, there is but one derivation tree for the string *abcde* in the grammar (2), namely,



(3)

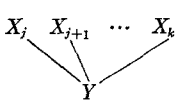
(The root of the derivation tree is S, and the branches correspond in an obvious manner to applications of productions.) A grammar is said to be *unambiguous* if every sentential form has a unique derivation tree. The grammar (2) is clearly unambiguous, even though there are several different *sequences* of derivations possible, e.g.

$$S \to AD \to aCD \to aBED \to abcdED \to abcdeD \to abcde \tag{4}$$

$$S \to AD \to A \to aC \to aBE \to aBe \to abcde \tag{5}$$

In order to avoid the unimportant difference between sequences of derivations corresponding to the same tree, we can stipulate a particular order, such as insisting that we always substitute for the leftmost intermediate (as done in (4)) or the rightmost one (as in (5)).

In practice, however, we must start with the terminal string *abcde* and try to reconstruct the derivation leading back to S, and that changes our outlook somewhat. Let us define the *handle* of a tree to be the leftmost set of adjacent leaves forming a complete branch; in (3) the handle is *bcd*. In other words, if $X_1, X_2, \cdots, X_t$ are the leaves of the tree (where each X_i is an intermediate or terminal character or ϵ), we look for the smallest k such that the tree has the form



for some j and Y. If we consider going from *abcde* backwards to reach S, we can imagine starting with tree (3), and "pruning off" its handle; then prune off the handle ("e") of the resulting tree, and so on until only the root S is left. This process of pruning the handle at each step corresponds exactly to derivation (5) in reverse. The reader may easily verify, in fact, that "handle pruning" always produces, in reverse, the derivation obtained by replacing the *rightmost* intermediate character at each step, and this may be regarded as an alternative way to define the concept of handle. During the pruning process, all leaves to the right of the handle are terminals, if we begin with all terminal leaves.

We are interested in algorithms for parsing, and thus we want to be able to recognize the handle when only the leaves of the tree are given. Number the productions of the grammar 1, 2, $\cdots$, s in some arbitrary fashion. Suppose $\alpha = X_1 \cdots X_n \cdots X_t$ is a sentential form, and suppose there is a derivation tree in which the handle is $X_{r+1} \cdots X_n$, obtained by application of the pth production. ($0 \leqq r \leqq n \leqq t$, $1 \leqq p \leqq s$.) We will say (n, p) is *a handle* of α.

A grammar is said to be *translatable from left to right with bound k* (briefly, an "LR(k) grammar") under the following circumstances. Let $k \geqq 0$, and let "$\dashv$" be a new character not in $I \cup T$. A k-sentential form is a sentential form followed by k "$\dashv$" characters. Let $\alpha = X_1X_2 \cdots X_nX_{n+1} \cdots X_{n+k}Y_1 \cdots Y_u$ and $\beta = X_1X_2 \cdots X_nX_{n+1} \cdots X_{n+k}Z_1 \cdots Z_v$ be k-sentential forms in which $u \geqq 0$, $v \geqq 0$ and in which none of $X_{n+1}, \cdots, X_{n+k}, Y_1, \cdots, Y_u, Z_1, \cdots, Z_v$ is an intermediate character. If (n, p) is a handle of α and (m, q) is a handle of β, we require that $m = n$, $p = q$. In other words, a grammar is LR(k) if and only if any handle is always uniquely determined by the string to its left and the k terminal characters to its right.

This definition is more readily understandable if we take a particular value of k, say $k = 1$. Suppose we are constructing a derivation sequence such as (5) in reverse, and the current string (followed by the delimiter $\dashv$ for convenience) has the form $X_1 \cdots X_nX_{n+1}\alpha \dashv$, where the tail end "$X_{n+1}\alpha \dashv$" represents part of the string we have not yet examined; but all possible reductions have been made at the left of the string so that the right boundary of the handle must be at position X_r for $r \geqq n$. We want to know, by looking at the next character X_{n+1}, if there is in fact a handle whose right boundary is at position X_n; if so, we want this handle to correspond to a unique production, so we can reduce the string and repeat the process; if not, we know we can move to the right

and read a new character of the string to be translated. This process will work if and only if the following condition ("LR(1)") always holds in the grammar: If $X_1X_2 \cdots X_nX_{n+1}\omega_1$ is a sentential form followed by " $\dashv$ " for which all characters of $X_{n+1}\omega_1$ are terminals or " $\dashv$ ", and if this string has a handle (n, p) ending at position n, then all 1-sentential forms $X_1X_2 \cdots X_nX_{n+1}\omega$ with $X_{n+1}\omega$ as above must have the same handle (n, p). The definition has been phrased carefully to account for the possibility that the handle is the empty string, which if inserted between X_n and X_{n+1} is regarded as having right boundary n.

This definition of an LR(k) grammar coincides with the intuitive notion of translation from left to right looking k characters ahead. Assume at some stage of translation we have made all possible reductions to the left of X_n ; by looking at the next k characters $X_{n+1} \cdots X_{n+k}$, we want to know if a reduction on $X_{r+1} \cdots X_n$ is to be made, *regardless* of what follows X_{n+k} . In an LR(k) grammar we are able to decide without hesitation whether or not such a reduction should be made. If a reduction is called for, we perform it and repeat the process; if none should be made, we move one character to the right.

An LR(k) grammar is clearly unambiguous, since the definition implies every derivation tree must have the same handle, and by induction there is only one possible tree. It is interesting to point out furthermore that nearly every grammar which is known to be unambiguous is either an LR(k) grammar, or (dually) is a right-to-left translatable grammar, or is some grammar which is translated using "both ends toward the middle." Thus, *the* LR(k) *condition may be regarded as the most powerful general test for nonambiguity that is now available.*

When k is given, we will show in Section II that it is possible to decide if a grammar is LR(k) or not. The essential reason behind this that *the possible configurations of a tree below its handle may be represented by a regular (finite automaton) language.*

Several related ideas have appeared in previous literature. Lynch (1963) considered special cases of LR(1) grammars, which he showed are unambiguous. Paul (1962) gave a general method to construct left-to-right parsers for certain very simple LR(1) languages. Floyd (1964a) and Irons (1964) independently developed the notion of *bounded context* grammars, which have the property that one knows whether or not to reduce any sentential form $\alpha\theta\omega$ using the production $A \rightarrow \theta$ by examining only a finite number of characters immediately to the left and right of θ. Eickel (1964) later developed an algorithm which would construct a

certain form of push-down parsing program from a bounded context grammar, and Earley (1964) independently developed a somewhat similar method which was applicable to a rather large number of LR(1) languages but had several important omissions. Floyd (1964a) also introduced the more general notion of a *bounded right context* grammar; in our terminology, this is an LR(k) grammar in which one knows whether or not $X_{r+1} \cdots X_n$ is the handle by examining only a given *finite* number of characters immediately to the left of X_{r+1}, as well as knowing $X_{n+1} \cdots X_{n+k}$. At that time it seemed plausible that a bounded right context grammar was the natural way to formalize the intuitive notion of a grammar by which one could translate from left to right without backing up or looking ahead by more than a given distance; but it was possible to show that Earley's construction provided a parsing method for some grammars which were *not* of bounded right context, although intuitively they should have been, and this led to the above definition of an LR(k) grammar (in which the *entire* string to the left of X_{r+1} is known).

It is natural to ask if we can in fact always parse the strings corresponding to an LR(k) grammar by going from left to right. Since there are an infinite number of strings $X_1 \cdots X_{n+k}$ which must be used to make a parsing decision, we might need infinite wisdom to be able to make this decision correctly; the definition of LR(k) merely says a correct decision *exists* for each of these infinitely many strings. But it will be shown in Section II that only a finite number of essential possibilities really exist.

Now we will present a few examples to illustrate these notions. Consider the following two grammars:

$$S \to aAc,\ A \to bAb,\ A \to b. \tag{6}$$

$$S \to aAc,\ A \to Abb,\ A \to b. \tag{7}$$

Both of these are unambiguous and they define the same language, $\{ab^{2n+1}c\}$. Grammar (6) is not LR(k) for any k, since given the partial string ab^m there is no information by which we can replace any b by A; parsing must wait until the "c" has been read. On the other hand grammar (7) is LR(0), in fact it is a bounded context language; the sentential forms are $\{aAb^{2n}c\}$ and $\{ab^{2n+1}c\}$, and to parse we must reduce a substring ab to aA, a substring Abb to A, and a substring aAc to S. This example shows that LR(k) is definitely a property of the *grammar*, not of the

language alone. The distinction between grammar and language is extremely important when semantics is being considered as well as syntax.

The grammar

$$S \to aAd,\ S \to bAB,\ A \to cA,\ A \to c,\ B \to d \tag{8}$$

has the sentential forms $\{ac^nAd\} \cup \{ac^{n+1}d\} \cup \{bc^nAB\} \cup \{bc^nAd\} \cup \{bc^{n+1}B\} \cup \{bc^{n+1}d\}$. In the string $bc^{n+1}d$, d must be replaced by B, while in the string $ac^{n+1}d$, this replacement must not be made; so the decision depends on an unbounded number of characters to the left of d, and the grammar is not of bounded context (nor is it translatable from right to left). On the other hand this grammar is clearly LR(1) and in fact it is of bounded right context since the handle is immediately known by considering the character to its right and two characters to its left; when the character d is considered the sentential form will have been reduced to either aAd or bAd.

The grammar

$$S \to aA,\ S \to bB,\ A \to cA,\ A \to d,\ B \to cB,\ B \to d \tag{9}$$

is not of bounded right context, since the handle in both ac^nd and bc^nd is "d"; yet this grammar is certainly LR(0). A more interesting example is

$$S \to aAc,\ S \to b,\ A \to aSc,\ A \to b. \tag{10}$$

Here the terminal strings are $\{a^nbc^n\}$, and the b must be reduced to S or A according as n is even or odd. This is another LR(0) grammar which fails to be of bounded right context.

In Section III we will give further examples and will discuss the relevance of these concepts to the grammar for ALGOL 60. Section IV contains a proof that the existence of k, such that a given grammar is LR(k), is recursively undecidable.

Ginsburg and Greibach (1965) have defined the notion of a *deterministic language*; we show in Section V that these are precisely the languages for which *there exists* an LR(k) grammar, and thereby we obtain a number of interesting consequences.

II. ANALYSIS OF LR(k) GRAMMARS

Given a grammar $\mathcal{G}$ and an integer $k \geqq 0$, we will now give two ways to test whether $\mathcal{G}$ is LR(k) or not. We may assume as usual that $\mathcal{G}$ does

not contain useless productions, i.e., for any A in I there are terminal strings α, β, γ such that $S \Longrightarrow \alpha A\gamma \Rightarrow \alpha\beta\gamma$.

The first method of testing is to construct another grammar $\mathfrak{F}$ which reflects all possible configurations of a handle and k characters to its right. The intermediate symbols of $\mathfrak{F}$ will be $[A\,;\,\alpha]$, where α is a k-letter string on $T \cup \{\dashv\}$; and also $[p]$, where p is the number of production in $\mathcal{G}$. The terminal symbols of $\mathfrak{F}$ will be $I \cup T \cup \{\dashv\}$.

For convenience we define $H_k(\sigma)$ to be the set of all k-letter strings β over $T \cup \{\dashv\}$ such that $\sigma \Longrightarrow \beta\gamma$ with respect to $\mathcal{G}$ for some γ; this is the set of all possible initial strings of length k derivable from σ.

Let the pth production of $\mathcal{G}$ be

$$A_p \rightarrow X_{p1} \cdots X_{pn_p}\,,\ 1 \leqq p \leqq s,\ n_p \geqq 0. \tag{11}$$

We construct all productions of the following form:

$$[A_p\,;\,\alpha] \rightarrow X_{p1} \cdots X_{p(j-1)}[X_{pj}\,;\,\beta] \tag{12}$$

where $1 \leqq j \leqq n_p$, X_{pj} is intermediate, and α, β are k-letter strings over $T \cup \{\dashv\}$ with β in $H_k(X_{p(j+1)} \cdots X_{pn_p}\alpha)$. Add also the productions

$$[A_p\,;\,\alpha] \rightarrow X_{p1} \cdots X_{pn_p}\alpha[p] \tag{13}$$

It is now easy to see that with respect to $\mathfrak{F}$,

$$[S;\ \dashv^k] \Rightarrow X_1 \cdots X_nX_{n+1} \cdots X_{n+k}[p] \tag{14}$$

if and only if there exists a k-sentential form $X_1 \cdots X_nX_{n+1} \cdots X_{n+k}Y_1 \cdots Y_u$ with handle $(n,\ p)$ and with $X_{n+1} \cdots Y_u$ not intermediates. Therefore by definition, $\mathcal{G}$ will be LR(k) if and only if $\mathfrak{F}$ satisfies the following property:

$$[S;\ \dashv^k] \Rightarrow \theta[p] \text{ and } [S;\ \dashv^k] \Rightarrow \theta\varphi[q] \text{ implies } \varphi = \epsilon \text{ and } p = q. \tag{15}$$

But $\mathfrak{F}$ is a *regular* grammar, and well-known methods exist for testing condition (15) in regular grammars. (Basically one first transforms $\mathfrak{F}$ so that all of its productions have the form $Q_i \rightarrow aQ_j$, and then if $Q_0 = [S;\ \dashv^k]$, one can systematically prepare a list of all pairs (i, j) such that there exists a string α for which $Q_0 \Rightarrow \alpha Q_i$ and $Q_0 \Rightarrow \alpha Q_j$.)

When $k = 2$, the grammar $\mathfrak{F}$ corresponding to (2) is

$$
\begin{array}{ll}
[S;\ \dashv\ \dashv] \to [A;\ \dashv\ \dashv] & [C;\ \dashv\ \dashv] \to [B; e\ \dashv] \\
[S;\ \dashv\ \dashv] \to A[D;\ \dashv\ \dashv] & [C;\ \dashv\ \dashv] \to B[E;\ \dashv\ \dashv] \\
[S;\ \dashv\ \dashv] \to AD\ \dashv\ \dashv [1] & [C;\ \dashv\ \dashv] \to BE\ \dashv\ \dashv [4] \\
[A;\ \dashv\ \dashv] \to a[C;\ \dashv\ \dashv] & [B;\ e\ \dashv] \to bcde\ \dashv [3] \\
[A;\ \dashv\ \dashv] \to aC\ \dashv\ \dashv [2] & [E;\ \dashv\ \dashv] \to e\ \dashv\ \dashv [6] \\
[D;\ \dashv\ \dashv] \to\ \dashv\ \dashv [5] &
\end{array} \tag{16}
$$

It is, of course, unnecessary to list productions which cannot be reached from $[S;\ \dashv\ \dashv]$. Condition (15) is immediate; one may see an intimate connection between (16) and the tree (3).

Our second method for testing the LR(k) condition is related to the first but it is perhaps more natural and at the same time it gives a method for parsing the grammar $\mathcal{G}$ if it is indeed LR(k). The parsing method is complicated by the appearance of ϵ in the grammar, when it becomes necessary to be very careful deciding when to insert an intermediate symbol A corresponding to the production $A \to \epsilon$. To treat this condition properly we will define $H_k'(\sigma)$ to be the same as $H_k(\sigma)$ except omitting all derivations that contain a step of the form

$$A\omega \to \omega,$$

i.e., when an intermediate as the *initial character* is replaced by ϵ. This means we are avoiding derivation trees whose handle is an empty string at the extreme left. For example, in the grammar

$$S \to BC\ \dashv\dashv\dashv,\ B \to Ce,\ B \to \epsilon,\ C \to D,\ C \to Dc,\ D \to \epsilon,\ D \to d$$

we would have

$$H_3(S) = \{\dashv\dashv\dashv,\ c\dashv\dashv,\ ce\dashv,\ cec,\ ced,\ d\dashv\dashv,\ dce,\ de\dashv,\ dec,\ ded,\ e\dashv\dashv,\ ec\dashv,ed\dashv,\ edc\}$$

$$H_3'(S) = \{dce,\ de\dashv,\ dec,\ ded\}.$$

As before we assume the productions of $\mathcal{G}$ are written in the form (11). We will also change $\mathcal{G}$ by introducing a new intermediate S_0 and adding a "zeroth" production

$$S_0 \to S\ \dashv^k \tag{16}$$

and regarding S_0 as the principal intermediate. The sentential forms are now identical to the k-sentential forms as defined above, and this is a decided convenience.

Our construction is based on the notion of a "state," which will be denoted by $[p, j; \alpha]$; here p is the number of a production, $0 \leqq j \leqq n_p$, and α is a k-letter string of terminals. Intuitively, we will be in state $[p, j; \alpha]$ if the partial parse so far has the form $\beta X_{p1} \cdots X_{pj}$, and if $\mathcal{G}$ contains a sentential form $\beta A_p \alpha \cdots$; that is, we have found j of the characters needed to complete the pth production, and α is a string which may legitimately follow the entire production if it is completed.

At any time during translation we will be in a *set* $\mathcal{S}$ of states. There are of course only a finite number of possible sets of states, although it is an enormous number. Hopefully there will not be many sets of states which can actually arise during translation. For each of these possible sets of states we will give a rule which explains what parsing step to perform and what new set of states to enter.

During the translation process we maintain a *stack*, denoted by

$$\mathcal{S}_0 X_1 \mathcal{S}_1 X_2 \mathcal{S}_2 \cdots X_n \mathcal{S}_n \mid Y_1 \cdots Y_k \omega. \tag{17}$$

The portion to the left of the vertical line consists alternately of state sets and characters; this represents the portion of a string which has already been translated (with the possible exception of the handle) and the state sets $\mathcal{S}_i$ we were in just after considering $X_1 \cdots X_i$. To the right of the vertical line appear the k terminal characters $Y_1 \cdots Y_k$ which may be used to govern the translation decision, followed by a string ω which has not yet been examined.

Initially we are in the state set $\mathcal{S}_0$ consisting of the single state $[0, 0; \dashv^k]$, the stack to the left of the vertical line in (17) contains only $\mathcal{S}_0$, and the string to be parsed (followed by $\dashv^k$) appears at the right. Inductively at a given stage of translation, assume the stack contents are given by (17) and that we are in state set $\mathcal{S} = \mathcal{S}_n$.

Step 1. Compute the "closure" $\mathcal{S}'$ of $\mathcal{S}$, which is defined recursively as the smallest set satisfying the following equation:

$$\begin{aligned} \mathcal{S}' = \mathcal{S} \cup \{[q, 0; \beta] \mid \text{there exists } [p, j; \alpha] \text{ in } \mathcal{S}', j < n_p, \\ X_{p(j+1)} = A_q, \text{ and } \beta \text{ in } H_k(X_{p(j+2)} \cdots X_{pn_p}\alpha)\}. \end{aligned} \tag{18}$$

(We thus have added to $\mathcal{S}$ all productions we might begin to work on, in addition to those we are already working on.)

Step 2. Compute the following sets of k-letter strings:

$$Z = \{\beta \mid \text{there exists } [p, j; \alpha] \text{ in } \mathcal{S}', j < n_p, \beta \text{ in } H_k'(X_{p(j+1)} \cdots X_{pn_p}\alpha)\} \tag{19}$$

$$Z_p = \{\alpha \mid [p, n_p ; \alpha] \text{ in } \mathcal{S}'\}, 0 \leqq p \leqq s. \tag{20}$$

Z represents all strings $Y_1 \cdots Y_k$ for which the handle does *not* appear on the stack, and Z_p represents all for which the pth production should be used to reduce the stack. Therefore, $Z, Z_0, \cdots, Z_s$ *must all be disjoint sets*, or the grammar is not LR(k). These formulas and remarks are meaningful even when $k = 0$. Assuming the Z's are disjoint, $Y_1 \cdots Y_k$ must lie in one of them, or else an error has occurred. If $Y_1 \cdots Y_k$ lies in Z, shift the entire stack left:

$$\mathcal{S}_0X_1\mathcal{S}_1 \cdots \mathcal{S}_nY_1 \mid Y_2 \cdots Y_k\omega$$

and rename its contents by letting $X_{n+1} = Y_1$, $Y_1 = Y_2$, $\cdots$:

$$\mathcal{S}_0X_1\mathcal{S}_1 \cdots \mathcal{S}_nX_{n+1} \mid Y_1 \cdots Y_k\omega'$$

and go on to Step 3. If $Y_1 \cdots Y_k$ lies in Z_p, let $r = n - n_p$; the stack now contains $X_{r+1} \cdots X_n$, equalling the righthand side of production p. Replace the stack contents (17) by

$$\mathcal{S}_0X_1\mathcal{S}_1 \cdots X_r\mathcal{S}_rA_p \mid Y_1 \cdots Y_k\omega \tag{21}$$

and let $n = r$, $X_{n+1} = A_p$. (Notice that obvious notational conventions have been used here to deal with empty strings; we have $0 \leqq r \leqq n$. If $n_p = 0$, i.e. if the righthand side of production p is empty, we have just *increased* the stack size by going from (17) to (21), otherwise the stack has gotten smaller.)

Step 3. The stack now has the form

$$\mathcal{S}_0X_1\mathcal{S}_1 \cdots X_n\mathcal{S}_nX_{n+1} \mid Y_1 \cdots Y_k\omega. \tag{22}$$

Compute $\mathcal{S}_n'$ by Eq. (18) and then compute the new set $\mathcal{S}_{n+1}$ as follows:

$$\mathcal{S}_{n+1} = \{[p, j + 1; \alpha] \mid [p, j; \alpha] \text{ in } \mathcal{S}_n' \text{ and } X_{n+1} = X_{p(j+1)}\}. \tag{23}$$

This is the state set into which we now advance; we insert $\mathcal{S}_{n+1}$ into the stack (22) just to the left of the vertical line and return to Step 1, with $\mathcal{S} = \mathcal{S}_{n+1}$ and with n increased by one. However, if $\mathcal{S}$ now equals $[0, 1; \dashv^k]$ and $Y_1 \cdots Y_k = \dashv^k$, the parsing is complete.

This completes the construction of a parsing method. In order to

properly take care of the most general case, this method is necessarily complicated, for all of the relevant information must be saved. The structure of this general method should shed some light on the important special cases which arise when the LR(k) grammar is of a simpler type.

We will not give a formal proof that this parsing method works, since the reader may easily verify that each step preserves the assertions we made about the state sets and the stack. The construction of all possible state sets that can arise will terminate since there are finitely many of these. The grammar will be LR(k) unless the Z sets of Eqs. (19)–(20) are not disjoint for some possible state set. The parsing method just described will terminate since any string in the language has a finite derivation, and each execution of Step 2 either finds a step in the derivation or reduces the length of string not yet examined.

III. EXAMPLES

Now let us give three examples of applications to some nontrivial languages. Consider first the grammar

$$S \to \epsilon,\ S \to aAbS,\ S \to bBaS,\quad A \to \epsilon,\ A \to aAbA,\ B \to \epsilon,\ B \to bBaB \tag{24}$$

whose terminal strings are just the *set of all strings on* $\{a, b\}$ *having exactly the same total number of* a*'s and* b*'s*. There is reason to believe (24) is the briefest possible unambiguous grammar for this language. We will prove it is unambiguous by showing it is LR(1), using the first construction in Section II. The grammar $\mathfrak{F}$ will be

$[S;\ \dashv] \to \dashv[1]$

$[S;\ \dashv] \to a[A;\ b],\quad [S;\ \dashv] \to aAb[S;\ \dashv],\quad [S;\ \dashv] \to aAbS\dashv[2]$

$[S;\ \dashv] \to b[B;\ a],\quad [S;\ \dashv] \to bBa[S;\ \dashv],\quad [S;\ \dashv] \to bBaS\dashv[3]$

$[A;\ b] \to b[4]$

$[A;\ b] \to a[A;\ b],\quad [A;\ b] \to aAb[A;\ b],\quad [A;\ b] \to aAbAb[5]$

$[B;\ a] \to a[6]$

$[B;\ a] \to b[B;\ a],\quad [B,\ a] \to bBa[B;\ a],\quad [B;\ a] \to bBaBa[7]$

The strings entering into condition (15) are therefore

$(aAb, bBa)*\dashv$[1], $(aAb, bBa)*aAbS\dashv$[2], $(aAb, bBa)*bBaS\dashv$[3]

$(aAb, bBa)*a(a, aAb)*b$[4], $(aAb, bBa)*a(a, aAb)*aAbAb$[5]

$(aAb, bBa)*b(b, bBa)*a$[6], $(aAb, bBa)*b(b, bBa)*bBaBa$[7].

Here $(\alpha, \beta)*$ denotes the set of all strings which can be formed by concatenation of α and β; clearly condition (15) is met.

Our second example is quite interesting. Consider first the set of all strings obtainable by *fully parenthesizing* algebraic expressions involving the letter a and the binary operation $+$:

$$S \to a,\ S \to (S + S) \tag{25}$$

where in this grammar "(", "+", and")" denote terminals. Given any such string we will perform the following acts of sabotage:

(i) All plus signs will be erased.

(ii) All parentheses appearing at the extreme left or extreme right will be erased.

(iii) *Both* left and right parentheses will be replaced by the letter b.

Question: After all these changes, is it still possible to recreate the original string? The answer is, surprisingly, yes; it is not hard to see that this question is equivalent to being able to parse any terminal string of the following grammar unambiguously:

Production #	Production	Production #	Production
0	$S \to B\dashv$		
1	$B \to a$	2	$B \to LR$
3	$L \to a$	4	$L \to LNb$
5	$R \to a$	6	$R \to bNR$
7	$N \to a$	8	$N \to bNNb$

(26)

Here B, L, R, N denote the sets of strings formed from (25) with alterations (i) and (iii) performed, and with parentheses removed from *both* ends, the *left* end, the *right* end, or *neither* end, respectively.

It is not immediately obvious that grammar (26) is unambiguous, nor is it immediately clear how one could design an efficient parsing algorithm for it. The second construction of Section II shows however that (26) is an LR(1) grammar, and it also gives us a parsing method. Table I shows the details, using an abbreviated notation.

In Table I, the symbol 21$\dashv$ stands for the state [2, 1; $\dashv$], and 41ab stands for two states [4, 1; a] and [4, 1; b]. "Shift" means "perform the shift left operation" mentioned in step 2; "reduce p" means "perform the transformation (21) with production p." The first lines of Table I

TABLE I
PARSING METHOD FOR GRAMMAR (26)

State set S	Additional states in S'	If Y_1 is	then	If X_{n+1} is	then go to
00$\dashv$	10$\dashv$ 20$\dashv$ 30ab 40ab	a	shift	B	01$\dashv$
				a	11$\dashv$ 31ab
				L	21$\dashv$ 41ab
01$\dashv$		$\dashv$	stop		
11$\dashv$ 31ab		$\dashv$	reduce 1		
		a, b	reduce 3		
21$\dashv$ 41ab	50$\dashv$ 60$\dashv$ 70b 80b	a, b	shift	R	22$\dashv$
				N	42ab
				a	51$\dashv$ 71b
				b	61$\dashv$ 81b
22$\dashv$		$\dashv$	reduce 2		
42ab		b	shift	b	43ab
51$\dashv$ 71ab		$\dashv$	reduce 5		
		a, b	reduce 7		
61$\dashv$ 81ab	70ab 80ab	a, b	shift	N	61$\dashv$ 82ab
				a	71ab
				b	81ab
43ab		a, b	reduce 4		
62$\dashv$ 82ab	50$\dashv$ 60$\dashv$ 70b 80b	a, b	shift	R	63$\dashv$
				N	84ab
				a	51$\dashv$ 71b
				b	61$\dashv$ 81b
63$\dashv$		$\dashv$	reduce 6		
84ab		a, b	reduce 8		

are formed as follows: Given the initial state $\mathcal{S} = \{00\dashv\}$, we must form $\mathcal{S}'$ according to Eq. (18). Since $X_{01} = B$ and $X_{02} = \dashv$ we must include $10\dashv$ and $20\dashv$ in $\mathcal{S}'$. Since $X_{21} = L$ and $X_{22} = R$ we must include $30ab$, $40ab$ in $\mathcal{S}'$ (a and b being the possible initial characters of $R\dashv$). Since $X_{41} = L$ and $X_{42} = N$ we must, similarly, include $30ab$ and $40ab$ in $\mathcal{S}'$; but these have already been included, and so $\mathcal{S}'$ is completely determined. Now $Z = \{a\}$ in this case, so the only possibility in step 2 is to have $Y_1 = a$ and shift. Step 3 is more interesting; if we ever get to Step 3 with $\mathcal{S}_n = \mathcal{S}$ (this includes later events when a reduction (21) has been performed) there are three possibilities for X_{n+1}. These are determined by the seven states in $\mathcal{S}'$, and the righthand column is merely an application of Eq. (23).

An important shortcut has been taken in Table I. Although it is possible to go into the state set "$51\dashv 71b$", we have no entry for that set; this happens because $51\dashv 71b$ is *contained in* $51\dashv 71ab$. *A procedure for a given state set must be valid for any of its subsets.* (This implies less error detection in Step 2, but we will soon justify that.) It is often possible to take the union of several state sets for which the parsing action does not conflict, thereby considerably shortening the parsing algorithm generated by the construction of Section II.

When only one possibility occurs in Step 2 there is no need to test the validity of $Y_1 \cdots Y_k$; for example in Table I line 1 there is no need to make sure $Y_1 = a$. One need do no error detection until an attempt to shift $Y_1 = \dashv$ left of the vertical line occurs. At this point the stack will contain "$\mathcal{S}_0 S \mathcal{S}_1 \mid \dashv^k$" if and only if the input string was well-formed; for we know a well-formed string will be parsed, and (by definition!) a malformed string cannot possibly be reduced to "$S\dashv^k$" by applying the productions in reverse. Thus, any or all error detection may be saved until the end. (When $k = 0$, $\dashv$ must be appended at the right in order to do this delayed error check.)

One could hardly write a paper about parsing without considering the traditional example of arithmetic expressions. The following grammar is typical:

Production #	Production	Production #	Production	
0	$S \to E\dashv$	4	$T \to P$	
1	$E \to -T$	5	$T \to T * P$	(27)
2	$E \to T$	6	$P \to a$	
3	$E \to E - T$	7	$P \to (E)$	

This grammar has the terminal alphabet $\{a, -, *, (,), \dashv\}$; for example, the string "$a - (-a*a - a)\dashv$" belongs to the language. Table II shows how our construction would produce a parsing method. In line 10, the notation "4, 5, 6" appearing in the X column means rules 4, 5, and 6 apply to this state set also. Such "factoring" of rules is another way to simplify the parsing routine produced by our construction, and the reader will undoubtedly see other ways to simplify Table II.

By means of our construction it is possible to determine exactly what information about the string being parsed is known at any given time. Because of this detailed knowledge, it will be possible to study how much of the information is not really essential (i.e., how much is redundant) and thereby determine the "best possible" parsing method for a grammar, in some sense. The two simplifications already mentioned (delayed error checking, taking unions of compatible state sets) are simplifications of this kind, and more study is needed to analyze this problem further.

In many cases it will not be necessary to store the state sets S_i in the stack, since the states S_r which are used in the latter part of Step 2 can often be determined by examining a few of the X's at the top of the stack. Indeed, this will always be true if we have a bounded right context grammar, as defined in Section I. Both grammars (26) and (27) are of bounded context.

From Table I we can see how to recover the necessary state set information without storing it in the stack. We need only consider those state sets which have at least one intermediate character in the "X_{n+1}" column for otherwise the state set is never used by the parser. Then it is immediately clear from Table I that $\{00\dashv\}$ is always at the bottom of the stack, $\{21\dashv, 41ab\}$ is always to the right of L, $\{61\dashv, 81ab\}$ is always to the right of b, and $\{62\dashv, 82ab\}$ is always to the right of N.

Grammar (27) is related to the definition of arithmetic expressions in the ALGOL 60 language, and it is natural to ask whether ALGOL 60 is an LR(k) language. The answer is a little difficult because the definition of this language (see Naur (1963)) is not done completely in terms of productions; there are "comment conventions" and occasional informal explanations. The grammar cannot be LR(k) because it has a number of syntactic ambiguities; for example, we have the production

$$\langle \text{open string} \rangle \rightarrow \langle \text{open string} \rangle \langle \text{open string} \rangle$$

which is always ambiguous. Another type of ambiguity arises in the parsing of ⟨identifier⟩ as ⟨actual parameter⟩. There are eight ways to do

TABLE II

PARSING METHOD FOR GRAMMAR (27)

State set S	Additional states in S	Y_1	Step 2 action	Rule #	X_{n+1}	Go to
00┤ 71┤)−∗	10┤)− 20┤)− 30┤)− 40┤)−∗ 50┤)−∗ 60┤)−∗ 70┤)−∗	−(*a*	shift	1 2 3 4 5 6	*E* − *T* *P* *a* (	01┤ 72┤)−∗ 31┤)− 11┤)− 21┤)− 51┤−∗ 41┤)−∗ 61┤)−∗ 71┤)−∗
01┤ 72┤)−∗ 31┤)−		┤) −	stop shift	7 8	) −	73┤)−∗ 32┤)−
11┤)−	40┤)−∗ 50┤)−∗ 60┤)−∗ 70┤)−∗			9	*T* 4, 5, 6	12┤)− 51┤)−∗
21┤)− 51┤)−∗		∗ ┤) −	shift reduce 2	10	∗	52┤)−∗
32┤)−	40┤)−∗ 50┤)−∗ 60┤)−∗ 70┤)−∗			11	*T* 4, 5, 6	33┤)− 51┤)−∗
12┤)− 51┤)−∗		∗ ┤) −	shift reduce 1	12	∗	52┤)−∗
52┤)−∗	60┤)−∗ 70┤)−∗	(*a*	shift	13	*P* 5, 6	53┤)−∗
33┤)− 51┤)−∗		∗ ┤) −	shift reduce 3	14	∗	52┤)−∗
pn_pX		X	reduce p			

this:

⟨actual parameter⟩ → ⟨array identifier⟩ → ⟨identifier⟩

⟨actual parameter⟩ → ⟨switch identifier⟩ → ⟨identifier⟩

⟨actual parameter⟩ → ⟨procedure identifier⟩ → ⟨identifier⟩

⟨actual parameter⟩ → ⟨expression⟩ → ⟨designational expression⟩
⇒ ⟨identifier⟩

⟨actual parameter⟩ → ⟨expression⟩ → ⟨Boolean expression⟩
⇒ ⟨variable⟩ ⇒ ⟨identifier⟩

⟨actual parameter⟩ → ⟨expression⟩ → ⟨Boolean expression⟩
⇒ ⟨function designator⟩ ⇒ ⟨identifier⟩

⟨actual parameter⟩ → ⟨expression⟩ → ⟨arithmetic expression⟩
⇒ ⟨variable⟩ ⇒ ⟨identifier⟩

⟨actual parameter⟩ → ⟨expression⟩ → ⟨arithmetic expression⟩
⇒ ⟨function designator⟩ ⇒ ⟨identifier⟩

These syntactic ambiguities reflect bona fide *semantic* ambiguities, if the identifier in question is a formal parameter to a procedure, for it is then impossible to determine what sort of identifier will be the actual argument in the absence of specifications. At the time the ALGOL 60 report was written, of course, the whole question of syntactic ambiguity was just emerging, and the authors of that document naturally made little attempt to avoid such ambiguities. In fact, the differentiation between array identifiers, switch identifiers, etc. in this example was done intentionally, to provide explanation along with the syntax (referring to identifiers which have been declared in a certain way). In view of this, a *ninth* alternative

⟨actual parameter⟩ → ⟨string⟩ → ⟨formal parameter⟩ → ⟨identifier⟩

might also have been included in the ALGOL 60 syntax (since section 4.7.5.1 specifically allows formal parameters whose actual parameter is a string to be used as actual parameters, and this event is not reflected in any of the eight possibilities above). The omission of this ninth alternative is significant, since it indicates the philosophy of the ALGOL 60 re-

port towards formal parameters: they are to be conceptually replaced by the actual parameters *before* rules of syntax are employed.

At any rate when parsing is considered it is desirable to have an unambiguous syntax, and it seems clear that with little trouble one could redefine the syntax of ALGOL 60 so that we would have an LR(1) grammar for the same language.

By the "ALGOL 60 language" we mean the set of strings meeting the *syntax* for ALGOL 60, not necessarily satisfying any semantical restrictions. For example,

begin array $x[100000: 0]$; $y := z/0$ **end**

would be regarded as a string in the ALGOL 60 language.

It is interesting to observe that it might be impossible to define ALGOL 60 using an RL(k) grammar (where by RL(k) we mean "translatable from right to left," defined dually to LR(k)). Several features of that language make it most suited to a left-to-right reading; for example, going from right to left, note that the basic symbol **comment** radically affects the parsing of the characters to its right. A similar language, for which some LR(k) grammars but no RL(k) grammars exist, is considered in Section V of this paper; but we also will give an example there which makes it appear possible that ALGOL 60 *could* be RL(k).

IV. AN UNSOLVABLE PROBLEM

Post (1947) introduced his famous *correspondence problem* which has been used to prove quite a number of linguistic questions undecidable. We will define here a similar unsolvable problem, and apply it to the study of LR(k) grammars.

THE PARTIAL CORRESPONDENCE PROBLEM. *Let* (α_1, β_1), (α_2, β_2), $\cdots$, (α_n, β_n) *be ordered pairs of nonempty strings. Do there exist, for all* $p > 0$, *ordered* p*-tuples of integers* $(i_1, i_2, \cdots, i_p)$ *such that the first* p *characters of the string* $\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_p}$ *are respectively equal to the first* p *characters of* $\beta_{i_1}\beta_{i_2}\cdots\beta_{i_p}$?

The ordinary correspondence problem asks for the existence of a $p > 0$ for which the entire strings $\alpha_{i_1}\cdots\alpha_{i_p}$ and $\beta_{i_1}\cdots\beta_{i_p}$ are equal. A solution to the ordinary correspondence problem implies an affirmative answer to the partial correspondence problem, but the general solvability of either problem is not directly related to the solvability of the other. There are relations between the partial correspondence problem and

the Tag problem (see Cocke and Minsky (1964)) but no apparent simple connection. We can, however, prove that the partial correspondence problem is recursively unsolvable, using methods analogous to those devised by Floyd (1964b) for dealing with the ordinary correspondence problem and using the determinacy of Turing machines.

For this purpose, let us use the definition and notation for Turing machines as given in Post (1947); we will construct a partial correspondence problem for any Turing machine and any initial configuration. The characters used in our partial correspondence problem will be

$$\vdash q_i\bar{q}_i S_j \bar{S}_j h \bar{h},\ 1 \leqq i \leqq R,\ 0 \leqq j \leqq m.$$

If the initial configuration is

$$S_{j_1} S_{j_2} \cdots S_{j_{k-1}} q_{i_1} S_{j_k} \cdots S_{j_\kappa}$$

the pair of strings

$$(\vdash,\ \vdash h S_{j_1} \cdots S_{j_{k-1}} q_{i_1} S_{j_k} \cdots S_{j_\kappa} h) \tag{28}$$

will enter into our partial correspondence problem. We also add the pairs

$$(\bar{h}, h), (h, \bar{h}), (S_j, \bar{S}_j), (\bar{S}_j, S_j), (\bar{q}_i, q_i), 1 \leqq i \leqq R, 0 \leqq j \leqq m. \tag{29}$$

Finally, we give pairs determined by the quadruples of the Turing machine:

Form of quadruple	Corresponding pairs, $0 \leqq t \leqq m$:	
$q_i S_j L q_l$	$(hq_i S_j, \bar{h}\bar{q}_l \bar{S}_0 \bar{S}_j), (S_t q_i S_j, \bar{q}_l \bar{S}_t \bar{S}_j)$	
$q_i S_j R q_l$	$(q_i S_j h, \bar{S}_j \bar{q}_l \bar{S}_0 \bar{h}), (q_i S_j S_t, \bar{S}_j \bar{q}_l \bar{S}_t)$	(30)
$q_i S_j S_k q_l$	$(q_i S_j, \bar{q}_l \bar{S}_k)$	

Now it is easy to see that these corresponding pairs will simulate the behavior of the Turing machine. Since the pair (28) is the only pair having the same initial character, and since the pairs in (30) are the only ones involving any q_i in the lefthand string, the only possible strings which can be initial substrings of both $\alpha_{i_1}\alpha_{i_2} \cdots$ and $\beta_{i_1}\beta_{i_2} \cdots$ are initial substrings of

$$\vdash \alpha_0 \bar{\alpha}_1 \alpha_1 \bar{\alpha}_2 \alpha_2 \bar{\alpha}_3 \alpha_3 \cdots, \tag{31}$$

where α_0, α_1, α_2, etc. represent the successive stages of the Turing machine's tape (with h's placed at either end, and where $\bar{\alpha}$ is an obvious

notation signifying the "barring" of each letter of α). For these pairs, therefore, *the partial correspondence problem has an affirmative answer if and only if the Turing machine never halts.* And the problem of telling if a Turing machine will ever halt is, of course, well known to be recursively unsolvable.

We will apply this result to LR(k) grammars as follows:

THEOREM. *The problem of deciding, for a given grammar* $\mathcal{G}$, *whether or not there exists a* $k \geqq 0$ *such that* $\mathcal{G}$ *is* LR(k), *is recursively unsolvable.*

This theorem is in contrast to the results of Section II, where we showed the problem to be solvable when k is also given. To prove this theorem we will reduce the partial correspondence problem to the LR(k) problem for a particular class of grammars.

Let $(\alpha_1, \beta_1), \cdots, (\alpha_n, \beta_n)$ be pairs of strings entering into the partial correspondence problem, and let

$$X_1 X_2 \cdots X_n +$$

be $n + 1$ characters distinct from those appearing among the α's and β's. Let $\mathcal{G}$ be the following grammar:

$$\begin{gathered} S \to A,\ S \to B,\ A \to X_i + \alpha_i,\ B \to X_i + \beta_i \\ A \to X_i A \alpha_i,\ B \to X_i B \beta_i,\ 1 \leqq i \leqq n. \end{gathered} \tag{32}$$

The sentential forms are

$$\{X_{i_m} \cdots X_{i_1} A \alpha_{i_1} \cdots \alpha_{i_m}\} \cup \{X_{i_m} \cdots X_{i_1} B \beta_{i_1} \cdots \beta_{i_m}\}$$
$$\cup \{X_{i_m} \cdots X_{i_1} + \alpha_{i_1} \cdots \alpha_{i_m}\} \cup \{X_{i_m} \cdots X_{i_1} + \beta_{i_1} \cdots \beta_{i_m}\}.$$

We will show $\mathcal{G}$ is LR(k) for some k if and only if the partial correspondence problem has a negative answer. If the answer is affirmative, for every p we have sentential forms $X_{i_p} \cdots X_{i_1} + \alpha_{i_1} \cdots \alpha_{i_p}$, $X_{i_p} \cdots X_{i_1} + \beta_{i_1} \cdots \beta_{i_p}$ in which the first p characters following "+" agree. The handle must include the "+" sign, but the $p - q$ characters following the handle do not tell us whether the production $A \to X_{i_1} + \alpha_{i_1}$ or $B \to X_{i_1} + \beta_{i_1}$ is to be applied, if q is the maximum length of the strings α_i, β_i. Hence the grammar is not LR(q). On the other hand, if the answer to the partial correspondence problem is negative, there is a p for which, knowing $(i_1, \cdots, i_{min(p,t)})$ and the first p characters of $\alpha_{i_1}\alpha_{i_2} \cdots \alpha_{i_t} \dashv^p$ or $\beta_{i_1}\beta_{i_2} \cdots \beta_{i_t} \dashv^p$, we can distinguish whether it is a string of α's or a string of β's, and therefore $\mathcal{G}$ is in fact a bounded context grammar.

We have proved slightly more, answering a question posed by Floyd (1964a, p. 66):

THEOREM. *The problems of deciding whether a given grammar* (i) *has bounded context, or* (ii) *has bounded right context, are recursively unsolvable.*

These theorems could be sharpened in the usual ways to show that we can assume the grammar $\mathcal{G}$ is unambiguous, linear, has at most two terminals, and has either a bounded number of productions or a bounded length of string in a production, and can still prove the problem to be unsolvable.

V. CONNECTIONS WITH DETERMINISTIC LANGUAGES

Ginsburg and Greibach (1965) define a *deterministic language* as one which is accepted by a so-called *deterministic push-down automaton* (DPDA). The latter is a device which has a finite number of states q_0, q_1, q_2, $\cdots$ q_r and which manipulates strings of characters in two alphabets T and I, according to the production rules of the following two types:

$$Aq_i \rightarrow \theta q_j \tag{33}$$

$$Aq_i a \rightarrow \theta q_j \tag{34}$$

Here A and a are single characters in I and T, respectively, and θ is any string over I. When A is the special character $\vdash$ we require θ to be a nonempty string whose initial character is $\vdash$. For each pair Aq_i, where A is in I and $0 \leqq i \leqq r$, we stipulate there is either a unique rule of type (33) and none of type (34), or there are no rules of type (33) and at most one of type (34) for each a in T. Some of the states are designated as "final states", and the terminal string α is *accepted* by the DPDA if and only if $\vdash q_0\alpha \Longrightarrow \vdash \omega q_i$ for some final state q_i and some string ω. Here the relation "$\Longrightarrow$" is generated from "$\rightarrow$" as in Section I.

THEOREM. *If $\mathcal{G}$ is an* LR(k) *grammar, and if $\mathcal{G}$ defines the language L, there is a* DPDA *which accepts the language $L\dashv^k$.*

The second construction of Section II is in fact closely related to a DPDA. The grammar $\mathcal{G}$ *augmented* by production (16) defines the language $L\dashv^k$. To construct such a DPDA we will take as our states, q_i, terminal k-letter strings $[Y_1 \cdots Y_k]$, and there will also be various auxiliary states. The terminal alphabet for the DPDA will be $T \cup \{\dashv\}$ and the intermediate alphabet will be $\{\$\} \cup I \cup T \cup \{\vdash\}$. We want our

DPDA to arrive at the configuration

$$\vdash \mathcal{S}_0 X_1 \mathcal{S}_1 \cdots X_n \mathcal{S}_n [Y_1 \cdots Y_k]\omega \tag{35}$$

if and only if the stack in the parsing algorithm of Section II is "$\mathcal{S}_0 X_1 \mathcal{S}_1 \cdots X_n \mathcal{S}_n \mid Y_1 \cdots Y_k\omega$" at a corresponding stage of the calculation.

Clearly we can construct productions of form (34) which read the first k characters of our input string $Y_1 \cdots Y_k\omega$ and get us to the initial configuration $\vdash\{[0, 0; \dashv^k]\}[Y_1 \cdots Y_k]\omega$. Now assume the DPDA has arrived at the configuration (35); as in steps 1 and 2 of the parsing algorithm we can compute the sets Z and Z_p. If $Y_1 \cdots Y_k$ is in Z, we create instructions of the form (34)

$$\mathcal{S}_n [Y_1 \cdots Y_k] a \to \mathcal{S}_n Y_1 \mathcal{S}_{n+1} [Y_2 \cdots Y_k a] \tag{36}$$

where $\mathcal{S}_{n+1}$ is determined by $X_{n+1} = Y_1$ (or a if $k = 0$) in (23). If $Y_1 \cdots Y_k$ is in Z_p, we let $q^{(0)}, q^{(1)}, \cdots, q^{(2n_p)}$ be new auxiliary states and write

$$\begin{aligned}
&\mathcal{S}_n [Y_1 \cdots Y_k] \to \mathcal{S}_n q^{(0)} \\
&\mathcal{S} q^{(2t)} \to q^{(2t+1)},\ X_{p(n_p - t)} q^{(2t+1)} \to q^{(2t+2)},\ 0 \leqq t < n_p, \text{ all } \mathcal{S}. \\
&\mathcal{S} q^{(2n_p)} \to \mathcal{S} A_p \mathcal{S}_{n+1} [Y_1 \cdots Y_k], \text{ all } \mathcal{S}.
\end{aligned} \tag{37}$$

where $\mathcal{S}_{n+1}$ is determined from $\mathcal{S}$ by using (23) with $\mathcal{S}_n = \mathcal{S}$, $X_{n+1} = A_p$. We make one exception to this rule, namely, if $Y_1 \cdots Y_k = \dashv^k$ and $\mathcal{S} = \{[0, 0; \dashv^k]\}$, we change the last instruction to

$$\mathcal{S} q^{(2n_p)} \to q_f$$

where q_f is the unique final state of our DPDA.

The rules (36) and (37) for all possible combinations of $\mathcal{S}_n$ and $[Y_1 \cdots Y_k]$, plus the few initial and final ones, give us a DPDA which exactly follows the procedure of the parsing algorithm in Section II.

COROLLARY. *If* $\mathcal{G}$ *is an* LR(k) *grammar and if* $\mathcal{G}$ *defines the language* L, *there is a* DPDA which accepts the language L.

For Ginsburg and Greibach (1965) have proved, among several other interesting theorems, that if L_0 is deterministic and R is regular, then $\{\alpha \mid \alpha\beta \text{ in } L_0 \text{ for some } \beta \text{ in } R\}$ is deterministic. We take $L_0 = L \dashv^k$ and $R = \{\dashv^k\}$.

We now prove a converse result.

THEOREM. *If L is deterministic, there is an* LR(1) *grammar* $\mathcal{G}$ *which defines L.*

To prove this theorem, we want to take an arbitrary DPDA with its instructions of the forms (33) and (34), and construct a corresponding grammar. First it will be necessary to simplify the problem a little, and so we will require that all of the instructions of our DPDA are of three types:

$$\begin{aligned} &\text{type (i)}: Aq_ia \to Aq_j \\ &\text{type (ii)}: Aq_i \to q_j \\ &\text{type (iii)}: Aq_i \to ABq_j \end{aligned} \tag{38}$$

where A, B are intermediates, a is terminal. This involves no loss of generality, since a rule (34) can be replaced by $Aq_ia \to Aq$, $Aq \to \theta q_j$ for some new state q, and we are left with type (i) and rules of the form (33). The rule $Aq_i \to \theta q_j$ is of type (ii) if θ is empty, otherwise assume $\theta = A_1A_2 \cdots A_t$ with $t \geqq 1$. If $A_1 \neq A$ we have $A \neq \vdash$ so we can replace (33) by

$$Aq_i \to q,\ Bq \to BA_1q' \text{ for all intermediates } B,\ A_1q' \to \theta q_j$$

where q, q' are new states. Thus we may assume $A = A_1$, and hence the rule (33) may be replaced by a sequence of $t - 1$ rules of type (iii), introducing $t - 2$ new states, provided $t > 1$. Finally if $t = 1$, the rule $Aq_i \to Aq_j$ may be replaced by

$$Aq_i \to AAq,\ Aq \to q_j$$

where q is a new state, thereby reducing all rules to the forms (38).

For any pair Aq_i we still have the deterministic property that if more than one rule appears with Aq_i on the left, all such rules are of type (i), and there is at most one such rule for any particular terminal character a.

A further assumption is needed about *final states.* If q_f, q_f' are final states (possibly identical), we want to avoid the situation

$$\alpha q_f \Rightarrow \beta q_f' \tag{39}$$

since this would imply an input string would be "accepted twice" by the DPDA. To exclude this possibility, we double the number of states in the DPDA, using two states q_i, $\bar{q}_i$ for each original state q_i. The instructions (38) are then replaced by

type (i) $Aq_ia \to Aq_j$, $A\bar{q}_ia \to Aq_j$.

type (ii) $Aq_i \to q_j$ if q_i is *not* final, $Aq_i \to \bar{q}_j$ if q_i *is* final, $A\bar{q}_i \to \bar{q}_j$.
type (iii) $Aq_i \to ABq_j$ if q_i is *not* final, $Aq_i \to AB\bar{q}_j$ if q_i *is* final, $A\bar{q}_i \to AB\bar{q}_j$.

One easily verifies that (39) cannot occur, and the same set of strings is accepted; basically we get into a state $\bar{q}_j$ if the current string has been accepted, and then we do not accept the string again, but return to an unbarred state when the next rule of type (i) is used.

Once the DPDA has been modified to meet these assumptions, let it have the states $q_0, \cdots, q_r$; we are ready to construct a grammar for the language it accepts. We begin by defining the languages L_{iAt} for $0 \leqq i, t \leqq r$ and for all intermediates A of the DPDA:

$$L_{iAt} = \{\alpha \mid Aq_i\alpha \overset{\prime}{\Longrightarrow} Aq \to q_t \text{ for some } q\} \tag{40}$$

where *no step in the derivation represented by "$\overset{\prime}{\Longrightarrow}$" affects the A appearing at the left.*

Construct the following productions for all rules (38) of the DPDA:

Rule	Productions for $\mathcal{G}$	
type (i) $Aq_ia \to Aq_j$	$L_{iAt} \to aL_{jAt}$, $0 \leqq t \leqq r$.	
type (ii) $Aq_i \to q_j$	$L_{iAj} \to \epsilon$	(41)
type (iii) $Aq_i \to ABq_j$	$L_{iAt} \to L_{jBs}L_{sAt}$, $0 \leqq s, t \leqq r$.	

An easy induction based on the length of the derivation "$\overset{\prime}{\Longrightarrow}$" or the derivation in $\mathcal{G}$ establishes the equality of the sets of strings defined in (40) and the sets of strings derivable from L_{iAt} using the productions (41).

Another set of languages is also important:

$$L_{iA} = \{\alpha \mid Aq_i\alpha \overset{\prime}{\Longrightarrow} A\omega q_f, \text{ some string } \omega, \text{ some final state } q_f\}. \tag{42}$$

We construct the following further productions:

Rule	Productions for $\mathcal{G}$	
type (i) $Aq_ia \to Aq_j$	$L_{iA} \to aL_{jA}$	
type (ii) $Aq_i \to q_j$	(none)	(43)
type (iii) $Aq_i \to ABq_j$	$L_{iA} \to L_{jB}$, $L_{iA} \to L_{jBs}L_{sA}$, $0 \leqq s \leqq r$.	
q_i is final	$L_{iA} \to \epsilon$, all A.	

Again, induction establishes the equivalence of (42) and (43). *The language derivable from $L_0\vdash$ using $\mathcal{G}$ is precisely the language L of the theorem,* by the definition of a DPDA.

Now remove all useless productions from $\mathcal{G}$, i.e., those which can never appear in a derivation of a terminal string starting from $L_0\vdash$. We claim the resulting grammar $\mathcal{G}$ is LR(1). This result could be proved using either of the constructions in Section II, where the state sets have a rather simple form, but for purposes of exposition we will give here a more intuitive explanation which shows the connection between the operation of the DPDA and the parsing process.

Consider any string $\alpha\dashv$ where α is accepted by the DPDA, and consider the step-by-step behavior of the DPDA as it processes α. At the same time we will be building a partial derivation tree which reflects all of the information known at a given stage of the parse. The nodes of this partial tree will contain symbols $[i, A, *]$ which means that in the only possible parsing of the string the intermediate L_{iAt}, for *some* $t = 0, 1, \cdots, r$ or t "blank", must fill that position. We will be "at" some node $[i, A, *]$ of the tree, meaning this particular node below the handle is of interest, and at the same time the DPDA will contain the configuration $\cdots Aq_i \cdots$.

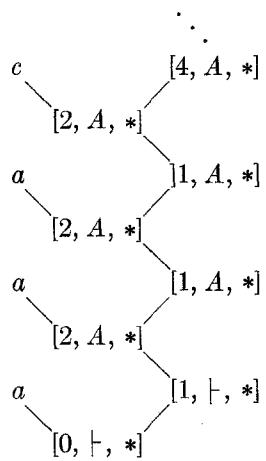
All of this can be clarified by considering an example, so we will consider the following "random" DPDA:

Rules of DPDA	Productions of $\mathcal{G}$ (useless ones deleted)
$\vdash q_0 a \to \vdash q_1$	$L_0\vdash \to aL_1\vdash$
$\vdash q_1 \to \vdash Aq_2$	$L_1\vdash \to L_{2A}$, $L_1\vdash \to L_{2A5}L_5\vdash$
$Aq_2a \to Aq_1$	$L_{2At} \to aL_{1At} (t = 2, 5, 6)$, $L_{2A} \to aL_{1A}$
$Aq_1 \to AAq_2$	$L_{1A} \to L_{2A}$, $L_{1A} \to L_{2A2}L_{2A}$,
	$L_{1A2} \to L_{2A6}L_{6A2}$, $L_{1At} \to L_{2A2}L_{2At}$
$Aq_2b \to Aq_3$	$L_{2A5} \to bL_{3A5}$, $L_{2A} \to bL_{3A}$
$Aq_2c \to Aq_4$	$L_{2A6} \to cL_{4A6}$ (44)
$Aq_3 \to q_5$	$L_{3A5} \to \epsilon$
$Aq_4 \to q_6$	$L_{4A6} \to \epsilon$
$Aq_6 \to q_2$	$L_{6A2} \to \epsilon$
$\vdash q_5 c \to \vdash q_1$	$L_5\vdash \to cL_1\vdash$
q_1 final	$L_{1A} \to \epsilon$, $L_1\vdash \to \epsilon$
q_3 final	$L_{3A} \to \epsilon$

Consider the action of the DPDA when given the string $aaacb\dashv$. We have

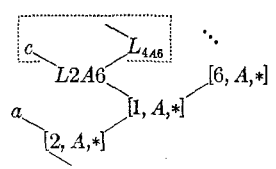
$$\vdash q_0aaacb\dashv \to \vdash q_1aacb\dashv \to \vdash Aq_2aacb\dashv \to \vdash Aq_1acb\dashv \to \vdash AAq_2acb\dashv$$
$$\to \vdash AAq_1cb\dashv \to \vdash AAAq_2cb\dashv \to \vdash AAAq_4b\dashv$$

Corresponding to these seven transitions we will build the following partial tree, one node at a time:



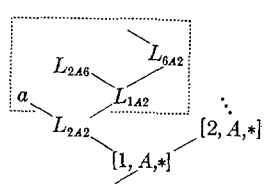
(45)

We are now "at" node [4, A, *], signified by the three dots above it. At this point the DPDA uses the rule $Aq_4 \rightarrow q_6$ and we transform the top of tree (45) to



(46)

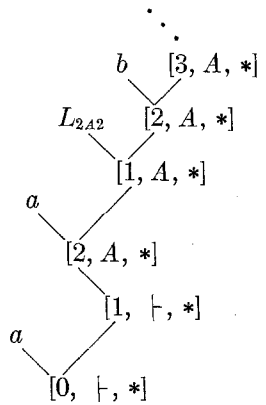
(Thus, two handles are recognized and then removed from the tree.) Then the DPDA uses the rule $Aq_6 \rightarrow q_2$ and (46) becomes



(47)

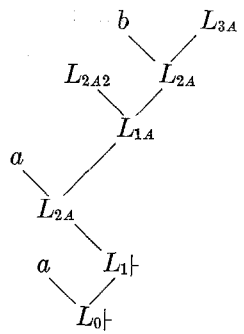
by reducing three more handles. When the rule $Aq_2b \rightarrow Aq_3$ is next ap-

plied, the tree becomes



(48)

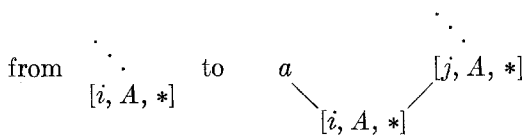
Now q_3 is a final state and the next character is " $\dashv$ ", so we complete the parsing; (48) becomes



(49)

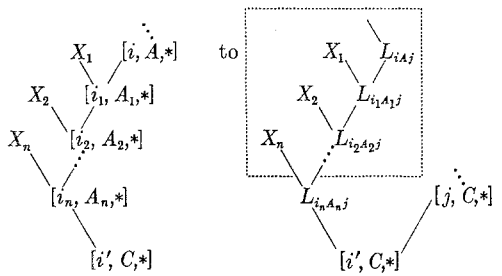
Having worked the example, we can consider the general case. Suppose the DPDA is in the configuration $\cdots CAq_ia \cdots$, and suppose we are at node $[i, A, *]$ of the tree. If q_i is a final state and $a =$ " $\dashv$ ", by condition (39) we must now complete the parsing, so we proceed to replace each $[i, A, *]$ in the tree by L_{iA} until the root is reached (as in going from (48) to (49)). If q_i is not final or $a \neq$ " $\dashv$ ", there are three cases depending on the pair Aq_i :

Case (*i*). The DPDA contains a rule of the form $Aq_ia \rightarrow Aq_j$. Then the only possible parse must occur by changing



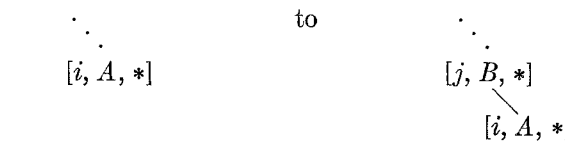
as we did in changing (47) to (48).

Case (ii). The DPDA contains a rule of the form $Aq_i \rightarrow q_j$. Then our tree must be changed from



as we did in changing from (45) to (46) and (46) to (47). Here $n \geqq 0$.

Case (iii). The DPDA contains a rule of the form $Aq_i \rightarrow ABq_j$. Then the only possible parse must occur by changing



as we did while building tree (45).

Cases (i), (ii), (iii) are mutually exclusive by the definition of DPDA, and the arguments are justified by the fact that our tree represents all possible productions of the grammar that could conceivably work. Notice that in the parsing *we actually have almost an* LR(0) *grammar* since it was necessary to look at the character following the handle only when q_i was a final state, to see if the next character is " $\dashv$ " or not.

As a consequence of our two theorems, we find a language can be generated by an LR(k) grammar if and only if it is deterministic, if and only if it can be generated by an LR(1) grammar.

The theorem cannot be improved to "LR(0) grammar", since ob-

viously even the simple language $\{\epsilon, a\}$ cannot be given an LR(0) grammar. However, it is possible to show that *the language* $L\dashv$ *can always be given an* LR(0) *grammar*; simply take the LR(1) grammar of the second theorem, and reapply the first theorem to get another DPDA for $L\dashv$. This DPDA has only one final state q_f, which leads to no further states, so the construction of the second theorem applied to this new grammar will be LR(0). A deterministic language in which no accepted string is a proper initial substring of any other will likewise have an LR(0) grammar.

Our last theorem shows that "deterministic" is essentially an asymmetric property, for there are languages which are translatable from right to left but which are not deterministic.

THEOREM. *The following productions constitute an* RL(0) *grammar for which the corresponding language is not deterministic*:

$$S \to Ac,\ S \to B,\ A \to aAbb,\ A \to abb,\ B \to aBb,\ B \to ab. \quad (50)$$

Proof: The terminal strings of this language are either $a^nb^{2n}c$ or a^nb^n, where $n > 0$. The grammar is clearly RL(0). On the other hand, suppose we could find an LR(k) grammar for the same language. (The problem is, of course, the appearance of "c" at the extreme right.) If we consider the derivations of the infinitely many strings a^nb^n we must find one in which a *recursive* intermediate appears; thus, there will be an intermediate C and strings α, φ, μ, ψ, ω such that $S \Rightarrow \alpha C\omega \Rightarrow \alpha\varphi C\psi\omega \Rightarrow \alpha\varphi\mu\psi\omega = a^nb^n$ for some n. Now $\alpha\varphi^t\mu\psi^t\omega$ must be in the language for all $t \geqq 0$, and $\varphi\psi$ is not empty since the grammar is unambiguous. We see therefore that $\varphi = a^p, \psi = b^p$ for some $p > 0$. This implies that C cannot appear in the derivation of any of the strings $a^nb^{2n}c$. For arbitrarily large t, the language contains strings $\alpha\varphi^{t+1}\mu\psi^{t+1}\omega = a^{n+pt}b^{n+pt}$ in which, by nonambiguity, the handle must be at least $p(t+1)$ characters from the right and must lead to a sentential form $\alpha\varphi^{t+1}C\psi^{t+1}\omega$ with $p(t+1)$ characters to the right of the handle; yet the language also contains the strings $a^{n+pt}b^{2(n+pt)}c$ which must *not* have the same handle, so the grammar cannot be LR(k). By the preceding theorem the language is not deterministic in the left-to-right sense.

When this paper was being prepared, an attempt was made to show that the language $\{a^nb^n\}d \cup (a, b)*c$ cannot be given an LR(k) grammar. Although this seemed plausible at first, the following grammar actually does work:

$$
\begin{aligned}
&S \to A,\ S \to bC,\ S \to Bd,\ S \to BcC,\ S \to c\\
&A \to Bc,\ A \to BaC,\ A \to aA,\\
&B \to ab,\ B \to aBb,\\
&C \to c,\ C \to aC,\ C \to bC.
\end{aligned}
\tag{51}
$$

This is an LR(0) grammar.

Indeed, we can note that a DPDA is able to recognize the complement of the strings it accepts, so that if L is a deterministic language not involving the character "c," the language $L \cup \{\alpha c \mid \alpha$ a string on the terminal symbols of $L\}$ would actually be deterministic, contrary to expectations. This weakens the argument that "**comment**" in Algol 60 might make it a non-RL language.

VI. REMARKS AND OPEN QUESTIONS

The concept of LR(k) grammars sheds much light on the translation problem for phrase structure languages, and it suggests several interesting areas for further investigation.

Of principal interest would be the study of grammatical transformations which preserve the LR(k) condition. Many such transformations are well known (for example, the removal of "empty" from a grammar; elimination of left-recursion; reducing to a "normal form" in which all productions are of type $A \to BC$ or $A \to a$; the operation of transduction which converts a grammar to another grammar for its translation; and many special cases of the latter). Which of these grammatical modifications take LR(k) grammars into LR(k) grammars? Similar questions apply to bounded context and bounded right context grammars.

Another important area of research is to develop algorithms that accept LR(k) grammars, or special classes of them, and to mechanically produce efficient parsing programs. In Section III we indicated three ways to simplify the general parsing schemes produced by our construction and many more techniques certainly exist. A table such as Table II shows essentially all of the information available during the parsing, and much of it can be recognized as repetitive or redundant.

There are also implications for automata theory. We have shown that a deterministic push-down automaton accepts precisely those languages that can be given an LR(k) grammar. This result can be strengthened to show that in fact such languages can always be given a *bounded right*

context grammar: We simply modify the construction (41), (43) by changing

$$L_{iAt} \to \alpha \text{ to } L_{iAt} \to M_{iA}\alpha$$

$$L_{iA} \to \alpha \text{ to } L_{iA} \to M_{iA}\alpha$$

and adding the productions $M_{iA} \to \epsilon$ for all i, A. This has the effect of keeping the necessary information in the sentential form that has been parsed.

The question is, however, what type of automaton is capable of accepting precisely those languages for which a *bounded context* grammar can be given. The bounded context condition is symmetric with respect to left and right, and we have shown that the deterministic property is not; for example, the mirror reflection of language (50) is a deterministic language which cannot be defined by a bounded context grammar.

The speed of parsing is another area of interest. Although LR(k) grammars can be efficiently parsed with an execution time essentially proportional to the length of string, there are more general grammars which can be parsed at a linear rate of speed. This may involve, for example, backing up a bounded number of times, or scanning back and forth from left to right and right to left in combination, etc. For every general parsing method known, there are grammars which cause it to take an *exponential* amount of time; yet it has never been proved that the parsing problem is necessarily inefficient in general. Are there particular grammars for which no conceivable parsing method will be able to find one parse of each string in the language with running time at worst linearly proportional to the length of string? Are there general parsing methods for which a linear parsing time can be guaranteed for all grammars? (In these questions, a parsing method means a process of constructing a derivation sequence from a terminal string by scanning a bounded number of characters at a time.)

Finally, we might mention another generalization of LR(k) to be explored. The "second handle" of a tree may be regarded as the left-most complete branch of terminals lying to right of the handle, and similarly we can consider the r-th handle. A parsing process which always reduces one of the first t handles leads to what might be called an LR(k, t) grammar. (In our case, $t = 1$.) The grammar

$$S \to ACc,\ S \to BCd,\ A \to a,\ B \to a,\ C \to Cb,\ C \to b \qquad (52)$$

is not LR(k, 1) for any k, since "a" is the handle in both ab^nc and ab^nd; but it is LR(0, 2). The following reduction rules serve to parse (52):

$$ab \to aC,\ Cb \to C,\ aCc \to ACc,\ aCd \to BCd,\ ACc \to S,\ BCc \to S.$$

One might choose to call this left-to-right translation, although we had to back up a finite amount.

RECEIVED: June 23, 1965

REFERENCES

COCKE, J., AND MINSKY, M. (1964), Universality of Tag systems with $P = 2$. *J. Assoc. Comput. Mach.* **11,** 15–20.

EARLEY, J. (1964), "Generating Productions from BNF" (preliminary report). Carnegie Institute of Technology.

EICKEL, J. (1964), Generation of parsing algorithms for Chomsky type 2 languages. *Tech. Hoch. München, Ber.* #6401.

FLOYD, R. W. (1963), Syntactic analysis and operator precedence. *J. Assoc. Comput. Mach.* **10,** 316–333.

FLOYD, R. W. (1964a), Bounded context syntactic analysis. *Commun. Assoc. Comput. Mach.* **7,** 62–66.

FLOYD, R. W. (1964b), "New Proofs of Old Theorems in Logic and Formal Linguistics." Computer Associates, Inc., Wakefield, Massachusetts.

GINSBURG, S., AND GREIBACH, S. (1965), "Deterministic Context-Free Languages" (preliminary report). *Am. Math. Soc. Not.* **12,** 246, 367.

IRONS, E. T. (1964), "Structural connections" in formal languages. *Commun. Assoc. Comput. Mach.* **7,** 67–71.

LYNCH, W. C. (1963), "Ambiguities in BNF Languages." Thesis, Univ. of Wisconsin.

NAUR, P., ed. (1963), Revised Algol 60 report. *Commun. Assoc. Comput. Mach.* **6,** 1–17.

PAUL, M. (1962), A general processor for certain formal languages. *Proc. Symp. Symbolic Languages in Data Processing, Rome, 1962.* Gordon and Breach, New York.

POST, E. L. (1947), Recursive unsolvability of a problem of Thue. *J. Symbolic Logic* **12,** 1–11.