Programmed Grammars and Classes of Formal Languages

DANIEL J. ROSENKRANTZ

al Electric Research and Development Center, Schenectady, New York

ABSTRACT. Programmed grammars are a generalization of phrase structure grammars where each production has a label, a core consisting of an ordinary phrase structure production, and two associated sets of production labels. If a production can be applied to an intermediate string in a derivation, it is applied as far to the left as possible, and the next production to be used is selected from the first set of labels. If the production cannot be applied to the intermediate string, the next production is selected from the other set of labels.

The properties of programmed grammars with various types of production cores are investigated. Two new classes of grammars are defined which lie between the context-free and context-sensitive grammars in their generative power. The first class consists of programmed grammars whose cores have a single symbol on the left-hand side and a nonnull string on the right. The other class consists of grammars of the above type for which the associated sets of labels act like an unconditional transfer.

KEY WORDS AND PHRASES: programmed grammar, formal language, phrase structure grammar, context-free, context-sensitive, Turing machines

CR CATEGORIES: 4.29, 5.22, 5.23

Introduction

Phrase structure grammars [1, 2] have been the most widely studied means of generating formal languages, and context-free phrase structure grammars in particular have been the most widely used in applications to programming languages. This paper deals with the properties of another device, called programmed grammars [3] (pg's), which is a generalization of phrase structure grammars. Programmed grammars have the property that after applying a production of a programmed grammar to an intermediate string in a derivation, one is restricted as to which production may be applied next. Specifically, each production has a label, a core consisting of a regular phrase structure production, and two associated "go-to" fields. If possible, the production is applied (as far to the left as possible) to the intermediate string in a derivation, and the next production to be used is selected from the first go-to field, which is called the success field. If the production cannot be applied to the string, the next production is selected from the other go-to field.

The aim in selecting the model chosen for programmed grammars was to find a way to permit specifications within the grammar as to the order in which productions can be used in generating a sentence. Several authors have considered grammars that contain some specification of the order in which the productions may be used, but the specification is of a more restricted form than that considered here.

The research described in this report was done at Columbia University under a National Science Foundation Cooperative Graduate Fellowship, at the General Electric Research and Development Center, and at Bell Telephone Laboratories. An extended abstract of this paper was presented at the Eighth Annual Symposium on Switching and Automata Theory, Austin, Texas, 1967.

Abraham [4] has defined a class of grammars that correspond in generative power to programmed grammars which have success fields only and cores which are context-free with a nonnull right-hand side. Peters [5] deals with grammars where the productions are arranged cyclicly, and each production may either be applied once or as many times as possible. Ginsburg and Spanier [6] have considered the classes of languages generated from phrase structure grammars by leftmost derivations whose production sequences lie in some language. Chomsky [7] has mentioned a model of natural languages where the grammar contains context-sensitive productions which are applied cyclicly. A group at MITRE [8] has written a program for analyzing English which utilizes productions of this form as part of its grammar.

A major advantage of using programmed grammars is that the grammar can often generate the sentences of a language in a manner which corresponds to the way in which humans would envision the generation. This is particularly true for context-sensitive languages where a phrase structure grammar might have to trace out the detailed maneuvers of a linear-bounded automaton [9, 10]. In writing and using a phrase structure grammar for a particular language, all the productions must be constantly checked to see if they are applicable at a given point. Writing a programmed grammar for a given language is similar to writing a program for the generation of its sentences.

Two new classes of languages are introduced which lie between the context-free and context-sensitive languages. The first new class of languages is generated by the set of programmed grammars, called cfpg's, whose cores have a single symbol on the left-hand side and a nonnull string on the right. Cfpg's can handle many context-sensitive features of programming languages, often acting in a more natural manner than a phrase structure grammar would. The other new class, called utcfpg's, is generated by cfpg's with identical success and failure fields. Indexed grammars [11] constitute another attempt to define a class of languages in this region.

Formal Model

A phrase structure grammar $G = (V_T, V_N, P, S)$ consists of a terminal vocabulary (V_T), a nonterminal vocabulary (V_N), a set of productions (P), and a sentence symbol (S) which is a member of V_N. The productions are of the form $\varphi \rightarrow \psi$ where φ and ψ can be strings of mixed terminal and nonterminal symbols with φ containing at least one nonterminal symbol. If the left-hand side of every production is a single symbol, the grammar is called context-free. If the right-hand side of each production has no fewer symbols than the left-hand side, the grammar is called context-sensitive.

In generating a sentence from the grammar, one starts with a string consisting of the symbol S (possibly surrounded by endmarkers) as the initial string. At any point in the derivation, any occurrence in the intermediate string of the left-hand side of some production may be replaced by the right-hand side of that production, resulting in a new intermediate string. This process is continued until a string is produced which consists entirely of terminal symbols. Such a string is a sentence generated by the grammar. Thus the language $L(G)$ generated by the grammar is defined to be the set of all strings of terminal symbols which can be derived from the symbol S by successive applications of the productions of G.

A word is in order now about the notation which will be used for strings. A string of all terminal symbols will be represented by a lowercase roman letter, all nonterminals by a capital roman letter, and a string which may contain terminals or nonterminals by a Greek letter. In addition, single symbols will be represented by letters from the beginning of the alphabet and strings by letters from the middle or end of the alphabet. An exception to this rule is S, which represents a single symbol.

A programmed grammar $G = (V_T, V_N, J, P, S)$ has, in addition to V_T, V_N, P, and S, a set of production labels J. With each r in J there is associated a unique production (r, φ, ψ, V, W). Here φ and ψ are strings of mixed terminal and nonterminal symbols with φ containing at least one nonterminal symbol. V and W are subsets of J. The production is written in the following format:

$$(r) \quad \varphi \rightarrow \psi \quad S(V)F(W)$$

The core of the production, $\varphi \rightarrow \psi$, is an ordinary phrase structure production. Note that the production format is somewhat similar to the instruction format of the SNOBOL programming language [12] and of Markov normal algorithms [13]. The interpretation of the production is also similar.

In applying the production to an intermediate string ξ, ξ is first scanned to see if it contains φ as a substring. If so, the leftmost occurrence of φ in ξ is replaced by ψ, and the next production to be applied to the ensuing string is selected from V. If ξ does not contain φ, then no change is made, and the next production is selected from W. Since the next production is selected from V if the scan is successful, V is called the success field and W is called the failure field. If V or W is absent from the explicit statement of a rule, the corresponding set of labels is the empty set. If at any point in a derivation the next production label must be selected from the empty set, the derivation comes to a halt.

The language generated by the grammar is defined to be the set of all terminal strings which can be obtained by starting with the string consisting of S, applying any applicable production, and then continuing to apply productions as directed by the success and failure fields until a terminal string is produced. Since the success and failure fields can contain several production labels, and different choices from a field can lead to different sentences, a grammar is capable of generating an infinite number of sentences.

Some additional notation will now be introduced. If ξ and ω are strings over $V_T \cup V_N$, r and p are rule labels, and rule r is

$$(r) \quad \varphi \rightarrow \psi \quad S(V)F(W),$$

we write $(\xi, r) \rightarrow (\omega, p)$ if either of the following two conditions are met:

(1) rule r succeeds on ξ, producing ω, and $p \in V$; or

(2) rule r fails on ξ, $\omega = \xi$, and $p \in W$.

If ω consists entirely of terminal symbols, we can also write $(\xi, r) \rightarrow \omega$. In addition if production r applied to ξ produces the string ω, but the set of labels from which the next production is to be selected is empty, we write $(\xi, r) \rightarrow \omega$.

Now $(\xi, r) \Rightarrow (\omega, p)$ if there is a chain such that

$$(\xi, r) = (\xi_1, r_1) \rightarrow (\xi_2, r_2) \rightarrow \cdots \rightarrow (\xi_n, r_n) = (\omega, p).$$

The language generated by the grammar is the set of terminal strings x such that

$(S, r) \Rightarrow x$ where r is a rule which has S on the left-hand side of its core. In the case of productions with cores which are not context-free, # can be used as an endmarker, and $L(G)$ is the set of terminal strings x not containing # such that $(\#S\#, r) \Rightarrow \#x\#$ for an r which is applicable to $\#S\#$.

Example 1. The following is an example of a programmed grammar where $V_T = (a, b, c)$ and $V_N = (S, A, B, C)$. The language generated is the set of sequences of the form $a^n b^n c^n$ with $n \geq 1$. This is a context-sensitive language which cannot be generated by any context-free phrase structure grammar.

(1)	$S \rightarrow ABC$	$S(2, 5)$
(2)	$A \rightarrow aA$	$S(3)$
(3)	$B \rightarrow bB$	$S(4)$
(4)	$C \rightarrow cC$	$S(2, 5)$
(5)	$A \rightarrow a$	$S(6)$
(6)	$B \rightarrow b$	$S(7)$
(7)	$C \rightarrow c$	

Example 2. Another example of a pg is the following, which generates sentences of the form nha^n where n is a nonnegative integer expressed as a binary number. A typical sentence is 101*haaaaa*. This language is a sort of pseudo Hollerith field specification since a could have been a nonterminal symbol which was later expanded into a single, arbitrary terminal symbol.

(1)	$S \rightarrow 1SB$	$S(3)$
(2)	$S \rightarrow 0S$	$S(3)$
(3)	$A \rightarrow BB$	$S(3)$ $F(4)$
(4)	$B \rightarrow A$	$S(4)$ $F(1, 2, 5)$
(5)	$S \rightarrow h$	$S(6)$
(6)	$A \rightarrow a$	$S(6)$

Note that when in a derivation by this grammar a choice must be made between several productions which can be applied next, the choice is between productions 1, 2, and 5. However each of these productions has an S on the left-hand side of its core and a distinct terminal symbol beginning the string on the right-hand side. Furthermore S is always the left-hand nonterminal symbol in an intermediate string of a derivation. Thus if we wish to parse a string according to this grammar, we can start generating a sentence, and whenever a choice must be made between rules 1, 2, and 5, we can look at the corresponding terminal symbol of the test string and, depending on whether it is 1, 0, or h, know which choice must be made if the derivation is to produce the test string. This property of grammars permits sentences to be parsed in an amount of time proportional to the length of their derivation [3].

Example 3. The following grammar generates the language $\{a^p \mid p \text{ is a prime}\}$.

(1)	$S \rightarrow SC$	$S(1, 2)$
(2)	$S \rightarrow AA$	$S(3)$
(3)	$A \rightarrow B$	$S(4)$ $F(5)$
(4)	$C \rightarrow D$	$S(3)$ $F(7)$
(5)	$C \rightarrow C$	$S(6)$
(6)	$B \rightarrow A$	$S(6)$ $F(3)$
(7)	$B \rightarrow A$	$S(7)$ $F(8)$
(8)	$D \rightarrow A$	$S(9)$ $F(10)$

(9) $D \rightarrow C$ $S(9)$ $F(3)$

(10) $A \rightarrow a$ $S(10)$

After applying rule 2 the intermediate string is of the form A^2C^{p-2}. For an intermediate string of the form A^nC^{p-n}, the grammar, beginning with rule 3, tests to see if $p - n$ is a multiple of n. If n does divide $p - n$, rule 5 will eventually fail and the grammar will stop without having produced a terminal string. If n does not divide $p - n$, rule 4 will fail, rule 8 will increment n by 1, and $n + 1$ will be tested. If rule 8 fails, then $n = p$, so that p is a prime, and rule 10 converts the A's to terminal symbols.

Effect of Cores on Generative Power

This paper is primarily concerned with placing the sets of languages generated by programmed grammars with various types of production cores within the standard hierarchy of phrase structure languages. These results are summarized below.

THEOREM 1. *The set of languages generated by programmed grammars whose rules have cores which are all right linear or terminating (or all left linear or terminating) is identical to the set of finite-state languages.*

THEOREM 2. *The set of languages generated by programmed grammars all of whose rules have cores which are linear or terminating is identical to the set of linear, context-free languages.*

THEOREM 3. *The set of languages generated by programmed grammars all of whose rules have cores which are context-sensitive is identical to the set of context-sensitive languages.*

THEOREM 4. *The set of languages generated by cfpg's (programmed grammars all of whose rules have cores with a single symbol on the left-hand side and a non-null string on the right-hand side) properly contains the set of context-free languages and is properly contained within the set of context-sensitive languages.*

THEOREM 5. *The set of languages generated by programmed grammars with arbitrary cores is identical to the set of recursively enumerable languages.*

THEOREM 6. *The set of languages generated by programmed grammars all of whose rules have cores with a single symbol on the left-hand side and an arbitrary (possibly null) string on the right-hand side is identical to the set of recursively enumerable languages.*

THEOREM 7. *The set of languages generated by utcfpg's (cfpg's with identical success and failure fields) properly contains the set of context-free languages and is properly contained within the set of cfpg languages.*

PG's for Finite-State and Context-Free Languages

A production of a phrase structure grammar is called right linear if it is of the form $A \rightarrow Bx$, terminating if of the form $A \rightarrow x$, and linear if of the form $A \rightarrow xBy$. Here x and y are terminal strings. A grammar is called linear if all its productions are linear or terminating. It is called one-sided linear if its productions are either all right linear or terminating, or all left linear or terminating. Every one-sided linear phrase structure grammar generates a finite-state language [2], i.e. one which can be recognized by a finite state machine.

This section will first show that a pg whose cores are one-sided linear (or termi-

nating) generates a finite state language, and a pg whose cores are all linear (or terminating) generates a linear language (a language generated by a linear phrase structure grammar). Thus the additional machinery of pg's does not add any power if the cores are of this form. We proceed with the proof of Theorem 1.

PROOF. First it is clear that by putting the set of all productions into the go-to fields, a one-sided linear phrase structure grammar can be converted into a pg which generates the same language. Now assume that G is a pg with one-sided linear cores. A one-sided linear phrase structure grammar G', which generates the same language as G, can be constructed.

G' has a nonterminal symbol for every production of G. Assume that G is right linear (the same procedure applies to a left linear grammar). Then a typical (ith) production of G is:

$$r_i \qquad A_i \longrightarrow x_i B_i \qquad S(V_i)F(W_i)$$

The corresponding nonterminal of G' will be $\langle r_i \rangle$.

Now define R_i as the set of rules of G which can be reached through a string of failure fields after successfully applying rule r_i and which have B_i, the nonterminal on the right-hand side of r_i, as the left-hand side of their cores. More precisely,

$$R_i = \{r_j \mid \text{there exists} \quad \nu_1, \cdots, \nu_m \quad \text{with} \quad m \geq 1, \qquad \nu_m = j,$$
$$r_{\nu_1} \in V_i, \qquad r_{\nu_{k+1}} \in W_{\nu_k} \quad \text{for} \quad k = 1, \cdots, m-1,$$
$$A_j = B_i, \quad \text{and} \quad A_{\nu_k} \neq B_i \quad \text{for} \quad k = 1, \cdots, m-1\}.$$

Thus after successfully applying production r_i, the next production which can be successfully applied to the resulting string (which has B_i as its only nonterminal) is a member of R_i.

Now corresponding to the single rule r_i of G, G' will have the set of rules

$$r_i \longrightarrow x_i r_j \quad \text{for all} \quad r_j \in R_i.$$

For a terminating rule of G,

$$r_i \qquad A_i \longrightarrow x_i \qquad S(V_i)F(W_i)$$

G' will have the single rule $r_i \longrightarrow x_i$. The sentence symbol of G' will be S'. For every production of G whose left-hand side (A_i) is equal to S (the sentence symbol of G), G' will have the production $S' \longrightarrow r_i$.

G' is thus a one-sided linear phrase structure grammar which generates the same language as G.

The proof of Theorem 2 is identical to that given above except that every appearance of $A_i \longrightarrow x_i B_i$ should be replaced by $A_i \longrightarrow x_i B_i y_i$.

It should be noted in passing that there are pg's with metalinear cores which generate languages which are not even context-free. For instance, the language $a^n b^n c^n$ can be generated by a pg with metalinear cores, as is shown in Example 1.

It is quite clear that cfpg's generate all context-free languages. A context-free grammar can be converted to a programmed grammar which generates the same languages by giving each production a label and putting the set of all labels into both go-to fields of every production.

Another approach is to group all rules with the same left-hand side together. These groups can be placed in some arbitrary linear order, but with the group of

S-rules (rules with S on the left-hand side) first. The success field of each rule in a group would contain the labels of the members of its group. The failure field would contain the labels of the next group, with the failure field of the last group containing the labels of the members of the first group.

Note that such a grammar generates sentences in depth rather than from left to right. It cycles through the nonterminals removing all occurrences (in the intermediate string) of a nonterminal before going on to the next nonterminal. With this procedure for obtaining the programmed grammar, if the original context-free grammar is unambiguous, then each sentence in the language has only one derivation in the new grammar, since for every tree produced by the context-free grammar there is only one derivation produced by the corresponding programmed grammar.

Programmed Grammars With Context-Sensitive Cores

This section deals with cspg's (programmed grammars with context-sensitive cores). It will first be shown that the use of endmarkers does not increase the power of cspg's, a result similar to a corresponding theorem [9] about context-sensitive grammars. A cspg without endmarkers is one for which none of the cores of the productions contain an endmarker. A cspg with endmarkers is one for which the cores can have endmarkers in their strings.

LEMMA 1. *The set of languages generated by cspg's with endmarkers is identical to the set of languages generated by cspg's without endmarkers.*

The proof of this lemma will only be outlined. First note that every cspg without endmarkers is trivially one with endmarkers. Now given a cspg G with endmarkers, we can obtain another cspg G' without endmarkers, which generates the same language. The new grammar will permit a symbol to be tagged with indicators as to whether or not it is the first or last symbol in an intermediate string. Productions can be written to make G' simulate G. Corresponding to productions in G which involve an endmarker, G' will have productions which involve tagged symbols. The go-to fields of G' can be arranged so that for each application of a production of G, only one of the corresponding productions of G' is applied, and it is applied as far to the left as possible. Since G' always applies a production of G as far to the left as possible, it generates the same language as G.

LEMMA 2. *The set of languages generated by cspg's is contained within the set of context-sensitive grammars.*

PROOF. Let G be an arbitrary cspg. We will assume that the cores of G do not contain endmarkers. If not, the procedure of the previous lemma can be used to rewrite G so that this condition is met. A procedure will now be described for obtaining a context-sensitive phrase structure grammar G' that generates the same language as G. G' will be a grammar with endmarkers and will correspond to a linear-bounded automata which simulates the operation of G.

At each stage in the derivation of a sentence from G', except the first and the last, the intermediate string will contain a marker. The symbols which can serve as markers will be doubly subscripted and of the form $m_{i,\alpha}$ or $m'_{i,\alpha}$. The first subscript, i, will correspond to the label of a rule of G which the grammar is currently trying to apply to the intermediate string, and the second subscript will indicate the first symbol in the intermediate string of the derivation being simulated. The

reason that the marker carries the first symbol along as a subscript is that the rules of G, being context-sensitive, must not be length-decreasing, so that when we want to liquidate the marker at the end of a derivation, we can convert it into the first symbol of the desired sentence.

The starting symbol of G' is S'. G' has a set of rules of the form

$$S' \rightarrow m_{i,s}$$

where S is the starting symbol of G and r_i is an S-rule of G. Thus G' starts off with $\#S'\#$ and replaces it with $\#m_{i,s}\#$.

Let $r_i \quad \varphi \rightarrow \psi \quad S(V_i)F(W_i)$ be a typical rule of G. A symbol of the form $m'_{i,\alpha}$ will serve as a marker which runs through an intermediate string and searches for φ. If it finds φ it replaces it by ψ and changes to an unprimed marker for a rule from V_i. If it cannot find an occurrence of φ, it changes to a marker for a rule from W_i. The $m_{i,\alpha}$ symbol moves to the beginning of a sentence and then changes to $m'_{i,\alpha}$ if φ does not occur at the beginning of the string. This is done because the scan for φ must start from the left since the leftmost occurrence of φ should be replaced by ψ. If φ occurs at the beginning of the intermediate string being simulated, then r_i is applied to the marker and the adjacent symbols which form φ. At any time the marker is at the beginning of a sentence, a production can be applied that changes it to α and ends the derivation.

Since there is only one marker in an intermediate string at a time and every production of G' (except the first) involves a marker in its left-hand side, a derivation by G' simulates a derivation by G and therefore produces a sentence of G. Similarly for every derivation of a sentence by G, there is a corresponding derivation by G'. Thus G' generates the same language as G.

The following corollary should be noted.

COROLLARY 1. *The set of languages generated by cfpg's is contained within the set of context-sensitive languages.*

THEOREM 3. *The set of languages generated by cspg's is identical to the set of context-sensitive languages.*

PROOF. Lemma 2 states that cspg languages are all context-sensitive languages. A procedure will now be described which given a context-sensitive grammar G will produce a cspg G' which generates the same language.

For a rule $\varphi \rightarrow \psi$ of G where φ contains an endmarker, G' will have the rule

$$(r_i) \qquad \varphi \rightarrow \psi \qquad S(\Gamma)F(\Gamma)$$

where Γ is the set of all rules of G'.

This simple tactic will not work if φ does not contain an endmarker, since in generating sentences from G it must be possible to replace any occurrence of φ (not just the leftmost one) by ψ. To get around this difficulty, G' will have three rules corresponding to a rule of G which does not involve endmarkers. Let $\varphi = \beta\varphi'$ and let β' be a new nonterminal symbol. Then the rules of G corresponding to $\varphi \rightarrow \psi$ will be

$$(r_{i1}) \qquad \beta \rightarrow \beta' \qquad S(r_{i1}, r_{i2})\, F(r_{i3})$$

$$(r_{i2}) \qquad \varphi \rightarrow \psi \qquad S(r_{i3})\, F(r_{i3})$$

$$(r_{i3}) \qquad \beta' \rightarrow \beta \qquad S(r_{i3})\, F(\Gamma)$$

Now G' generates the same language as G.

Pg's With Context-Free Cores and No Nullifying Rules

It has been shown that every context-free language can be generated by a cfpg, and examples have been given to show that there are cfpg's which generate languages which are not context-free, so that the set of cfpg languages (languages generated by cfpg's) properly contains the set of context-free languages. Also, from Theorem 3, every cfpg generates a language which is context-sensitive. In this section it is shown that the set of cfpg languages is properly contained within the set of context-sensitive languages. We begin with some definitions.

Let L_1 be a language over vocabulary V_T. We say that L_2 is language L_1 *with tails* if L_2 is a language over $V_T \cup \{c, d\}$, where c and d are new symbols, whose sentences are of the form xdc^m where $x \in L_1$ and m depends on x. Thus $L_1 = \{x \mid xdc^m \in L_2 \text{ for some } m\}$, and each sentence of L_2 is a sentence of L_1 followed by a tail consisting of a d and a number of c's. For $x \in L_1$ let $m(x)$ be the minimum value of j such that $xdc^j \in L_2$. Now let F be a computable, nondecreasing function defined over the nonnegative integers, and let $|x|$ denote the length of x. We will say that L_2 has *tail growth function* less than or equal to F if for all but a finite number of sentences of L_1, $m(x) \leq F(|x|)$.

Let $G = (V_T, V_N, J, P, S)$ be a cfpg with q nonterminal symbols. First place the nonterminals of G in some arbitrary linear order. The *nonterminal map*, δ_N, is a function from strings over $V_T \cup V_N$ into the set of q-tuples of natural numbers. We define $\delta_N(\varphi) = v$ where v_i (the ith component of v) is equal to the number of times the ith nonterminal occurs in φ. $\delta_N(\varphi)$ will be called the *nonterminal vector* corresponding to φ. As an example, if the nonterminals are S, A, B, C, D, then $\delta_N(aDAabCSAdbDaA) = (1, 3, 0, 1, 2)$.

LEMMA 3. *Let L_1 be a recursively enumerable language over a vocabulary which does not include c or d. Let G be a cfpg such that $L(G)$ is L_1 with tails and has a tail growth function less than or equal to F. Then there exists a context-sensitive grammar G' such that $L(G')$ is L_1 with tails and has a tail growth function less than or equal to $q(\log F + 1)$ where q is the number of nonterminals in G.*

(Throughout this section logarithms will be to the base 2.)

PROOF. First it should be observed that if some sequence of rules of G converts $\varphi_1\varphi_2$ into xdc^m where xdc^p comes from φ_1 and a string of c's from φ_2, then the same sequence of rules applied to $\varphi_1\varphi_3$ where $\delta_N(\varphi_3) = \delta_N(\varphi_2)$ will also convert φ_1 into xdc^p and φ_3 into a string of c's. Therefore the nonterminal vector corresponding to a substring which will eventually form the tail of a sentence provides all the information required to determine its behavior during a derivation.

If φ is a string of length n, then each component of its nonterminal vector is less than or equal to n and can be stored as a binary number in a space less than or equal to $\log n$. The entire vector plus q symbols which serve as separators between components of the vector can be stored in space $q \log n$.

A description is now given of a nondeterministic Turing machine, M, which never erases and which generates language L_1 with tail growth less than or equal to $q(\log F + 1)$. Since M is nonerasing, its actions can be described by a context-sensitive grammar, G', which generates the same language as M. An intermediate string in a derivation by G' always has one symbol marked to indicate the state of M and the location of its head. By shifting the marked symbol, G' can simulate the operation of M.

M simulates the operation of G. However it stores an intermediate string in a derivation as φv where φ is the substring which will eventually form the head of a sentence and v is the vector corresponding to the substring which will form the tail. M operates as follows.

1. It starts simulating the operation of G in generating a sentence, storing the intermediate string in its entirety.

2. At some point, in a nondeterministic manner, it places a marker in the cell to the right of the intermediate string. Let $n + 1$ be the number of cells (symbols) to the left of the marker.

3. M continues to simulate G, storing the first $n + 1$ symbols of an intermediate string in the first $n + 1$ cells of its tape and storing the vector corresponding to the remainder of the string on the space to the right of the marked square. In simulating a production whose core is $A \rightarrow \psi$, M behaves as follows.

(a) It first scans the string to the left of the marker for an occurrence of an A. If any are present, the leftmost A is replaced by ψ and the substring to the right of the replacement is shifted right if $|\psi| > 1$. Only the first $n + 1$ symbols of the resulting string are retained; any nonterminals in the remainder are added to the appropriate components of the vector while terminals in the remainder are deleted. The next production to be used is then nondeterministically chosen from the success field.

(b) If A does not occur in φ, but the component of the vector corresponding to A is nonzero, the production is applied to the vector. The A component is decremented by 1, and then $\delta_N(\psi)$ is added to the resulting vector. In changing the vector no symbol is ever erased, but symbols can be overprinted by a c, which will represent a blank; thus blanking with c's is used instead of erasing. When more space is needed, several components and their separators are shifted right. The next production is then taken from the success field.

(c) If A does not occur in φ and its component in the vector is zero, both φ and the vector are left changed, and the next production is selected from the failure field.

4. After applying each production of G, the machine M determines whether the string φ contains all terminal symbols and whether all the components of the vector are equal to zero. If not, M continues to simulate G'. If so, the derivation by G has produced a terminal string. M now checks whether or not it put the marker in the right place by seeing if the string to the left of the marker is of the form xd where x is a string over V_T. If not, M will halt without producing a terminal string. If so, M takes all the symbols ever used in recording the vector and converts them to c's.

The context-sensitive grammar G' which simulates M now has the desired properties. It generates the sentences of $L(G)$ but with shorter tails.

THEOREM. *There exist context-sensitive languages which cannot be generated by any cfpg.*

PROOF. This proof is based on the theory of tape complexity classes of machines [14]. Let F be a computable function defined over the nonnegative integers. A deterministic off-line Turing machine which always halts is said to operate within tape F if the amount of storage tape used for an input of length n is less than or equal to $F(n)$. A language is F-tape-recognizable if there exists a Turing machine which operates within tape F and which for any input string determines whether

or not it is in the language. It has been shown [14] that if a language is F-tape-recognizable, c is any positive constant, and $[cF]$ denotes the smallest integer greater than or equal to cF, then the language is also $[cF]$-tape-recognizable. Therefore it is only the limiting behavior of F for large values of n which is of importance. It has also been shown that if there exists a Turing machine which operates within tape F and which for each n actually uses $F(n)$ storage tape cells for some input of length n, then there is a language which is F-tape-recognizable but which cannot be recognized with essentially less tape. Specifically if $P(n)$ is a function such that $\inf_{n\to\infty}(P(n)/F(n)) = 0$, then this language cannot be recognized by any $P(n)$-tape-bounded Turing machine.

Now let M_1 be a deterministic off-line Turing machine which recognizes some language L_1 while operating within tape Q where L_1 cannot be recognized within an essentially lesser amount of tape and where $Q(n) \geq 2^{2^n}$. Such a machine exists, for instance for $Q(n) = 2^{2^n}$, since a Turing machine can be described which uses exactly that much tape.

Let us assume that every context-sensitive language can be generated by a cfpg. We can then construct the sequence of grammars shown in Figure 1.

From the description of M_1, we can obtain a context-sensitive grammar, G_2, whose language, L_2, is L_1 with tails where the tail growth function is Q. G_2 corresponds to a nonerasing Turing machine which first generates an arbitrary string, x, and then simulates M_1 given x as its input, using the squares to the right of x as the storage tape and writing c's instead of erasing. If M_1 accepts x, then G_2 prints c's on the squares to the right of x and produces a terminal string of the form xdc^m. If M_1 rejects x, then G_2 halts without producing a terminal string.

Under the assumption that every context-sensitive language is generated by some cfpg, there exists a cfpg, G_3, which generates L_2. From Lemma 3 there then exists a context-sensitive grammar, G_4, which generates the sentences of L_1 with shorter tails, specifically with tail growth function $q(\log Q + 1)$ where q is a constant. Since we are only interested in the behavior of the tails for large values of n, the tail growth function can be taken to be less than or equal to $2q\log Q$. Let L_4 be the language generated by G_4.

The tails can be further shortened by repeating the above procedure. Again under the assumption that every context-sensitive language is generated by some cfpg, there exists a cfpg, G_5, which generates L_4. Applying the Lemma to G_5,

L_1	M_1		tape Q
	⇓		
L_2	G_2	cs	tails Q
	⇓		
L_2	G_3	cfpg	tails Q
	⇓		
L_4	G_4	cs	tails $\log Q$
	⇓		
L_4	G_5	cfpg	tails $\log Q$
	⇓		
L_6	G_6	cs	tails $\log \log Q$
	⇓		
L_6	M_7		exponential tape
	⇓		
L_1	M_8		tape $[\log Q]^{p'}$

FIG. 1. Sequence of grammars used to obtain new machine

there exists a context-sensitive grammar, G_6, which generates L_4 with shorter tails. Its language is L_6, which has tail growth function less than or equal to

$$r(\log(2q \log Q) + 1) = r \log \log Q + r \log 2q + r$$

where r is the number of nonterminals in L_6. Since $r \log \log Q$ plus a constant is less than or equal to $2r \log \log Q$ for large n, we can take the tail growth function of L to be less than or equal to $2r \log \log Q$.

It will now be shown that every context-sensitive language can be recognized by an $F(n) = k^n$ tape-bounded deterministic Turing machine where k is a constant which depends on the language. The maximum length of a noncycling derivation of a sentence of length n by a context-sensitive grammar is bounded by k^n where k is the size of the vocabulary plus 1, and every sentence has a noncycling derivation. A k^n-tape-bounded Turing machine could store a string of length $\leq k^n$ formed from a set of symbols, each of which corresponds to a particular production. The machine could run through the set of all sequences of length k^n or less, and for each sequence determine whether the corresponding sequence of productions generates the input string. Therefore the machine can deterministically determine for a sample sentence whether or not it is generated by the grammar.

Thus there exists a Turing machine M_7 which recognizes L_6 within tape k^j where j is the length of the input string to M_7. From this machine we construct another machine M_8 which recognizes L_1 within tape $[\log Q(n)]^{p'}$ where p' is a constant. M_8 operates by taking an input string x of length n and successively attaching tails to produce strings of the form xdc^m for $0 \leq m \leq 2r \log \log Q(n)$. Note that M_8 is deterministic.

Since x is a member of L_1 if and only if there is some m in this range for which $xdc^m \in L_6$, M_8 will simulate M_7 with each such string as its input. Since the length of the longest such string is $n + 2r \log \log Q(n)$, the maximum space required by M_7, and therefore M_8, to do the required processing is less than or equal to $k^{n+2r \log \log Q(n)}$. Since $Q \geq 2^{2n}$, $n \leq \log \log Q(n)$ and $n + 2r \log \log Q(n) \leq (2r + 1) \log \log Q(n)$. Letting $p = k^{2r+1}$ and $p' = \log p$, we note that the amount of tape used by M_8 is less than or equal to $k^{(2r+1) \log \log Q(n)} = p^{\log \log Q(n)} = (\log Q(n))^{\log p} = (\log Q(n))^{p'}$. Therefore M_8 operates within tape $(\log Q(n))^{p'}$, and L_1 is $(\log Q(n))^{p'}$-tape-recognizable. However, L_1 was chosen as a language which could not be recognized by any machine which operates within less than $Q(n)$ tape. But

$$\inf_{n \to \infty} \frac{(\log Q(n))^{p'}}{Q(n)} = 0,$$

and M_8, if it existed, would violate this property of L_1. Therefore M_8 cannot exist and at least one of the steps in Figure 1 is invalid. Therefore either G_3 does not exist, so that L_2 is a context-sensitive language which cannot be generated by any cfpg, or if G_3 does exist, G_5 does not, and L_4 is a context-sensitive language which cannot be generated by any cfpg. In either case there exists a context-sensitive language which cannot be generated by any cfpg.

The results which have been obtained thus far about the generative power of cfpg's can be combined to give Theorem 4.

PG's For Recursively Enumerable Languages

This section is primarily occupied with a proof that all recursively enumerable languages can be generated by pg's whose cores have a single symbol on the left-hand side. But first the following result is noted.

THEOREM 5. *The set of languages generated by pg's with arbitrary cores is identical to the set of recursively enumerable languages.*

PROOF. Every recursively enumerable language is generated by an arbitrary (type 0) phrase structure grammar. Let G be an arbitrary phrase structure grammar. Let G' be the pg obtained from G by giving each production of G a label and applying the procedure of Theorem 3. Then $L(G') = L(G)$.

Now let G be a pg with arbitrary cores. It is obvious that a Turing machine can simulate the behavior of G, and so $L(G)$ is recursively enumerable.

THEOREM 6. *The set of languages generated by pg's all of whose rules have cores with a single symbol on the left-hand side and an arbitrary (possibly null) string on the right-hand side is identical to the set of recursively enumerable languages.*

PROOF. Let $G = (V_T, V_N, P, S)$ be an arbitrary phrase structure grammar. A pg, $G' = (V_T, V_N', J, P', S')$, will be constructed such that $L(G') = L(G)$.

In simulating G, the intermediate strings generated by G will be coded as numbers. Let $q = |V_T| + |V_N| + 1$. ($|X|$ is the number of elements in X if X is a set and the number of symbols in X if X is a string.) Now define an arbitrary one-one mapping

$$f: \quad V_T \cup V_N \to \{1, 2, \cdots, q-1\}.$$

An invertible mapping g from the strings over $V_T \cup V_N$ into the nonnegative integers is defined. If the string is $\varphi = \alpha_1\alpha_2 \cdots \alpha_n$, then $g(\varphi)$ is defined to be

$$g(\varphi) = f(\alpha_1) \times q^{n-1} + f(\alpha_2) \times q^{n-2} + \cdots + f(\alpha_i) \times q^{n-i} + \cdots + f(\alpha_n).$$

Thus φ is considered to be a number written in base q. The empty string is mapped into zero. Since no symbol of $V_T \cup V_N$ represents the digit 0, the mapping is unique.

Let $x \in L(G)$. Then x has a derivation of the form $S = \xi_0 \to \xi_1 \to \cdots \to \xi_r = x$. "$\xi_i \to \xi_{i+1}$" means that $\xi_i = \omega_1\varphi\omega_2$, $\xi_{i+1} = \omega_1\psi\omega_2$ where ω_1 and ω_2 can be null, and $\varphi \to \psi$ is a production of G. G' will simulate the derivations of G but will work with the codings of the strings rather than with the strings themselves.

The effect of the operation of the grammar of G' can be described by the flowchart in Figure 2. The arrows mean that the go-to field of the last production of a box contains the label of the first production of the box to which the arrow points.

The second box acts nondeterministically in that the division into ω_1, φ_1, and ω_2 can occur in any way.

If after executing the last box, ξ contains any nonterminals, then the grammar has not produced a sentence. If ξ is a terminal string, then it is a member of $L(G)$.

These blocks will now be broken down into simpler blocks which perform more primitive operations. In designating arithmetic operations, $[n/q]$ will be used for the quotient of n divided by q and $R(n/q)$ for the remainder. Also λ will be used to denote the null string.

Note that the order in which the nonterminals (other than F) of G' appear in any intermediate string does not matter since these nonterminals will be wiped out in the end. To simplify the details of the grammar, the order of the nonterminals

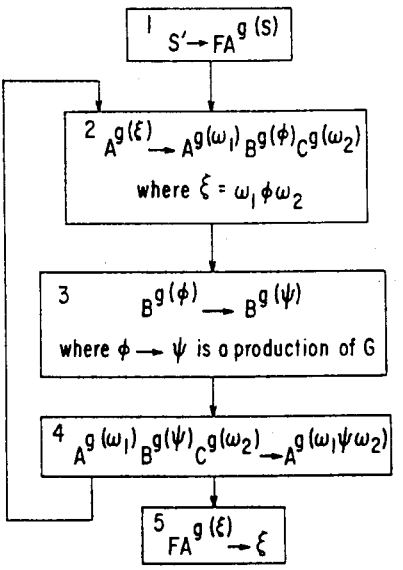

FIG. 2. Procedure for simulating an arbitrary phrase structure grammar

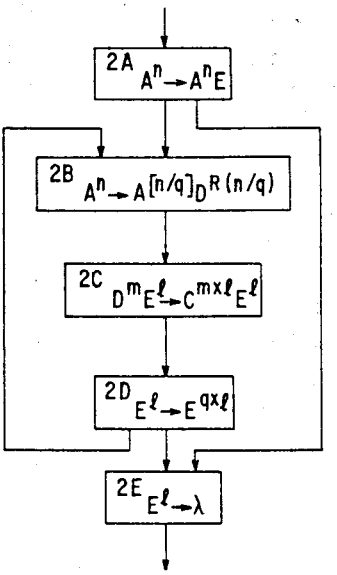

FIG. 3. Flowchart for breaking up a string

in the intermediate string will occasionally differ from that indicated by the block diagram.

Block 1 consists of a single production.

Block 2 consists of two subblocks, which perform the following operations: $A^{g(\xi)} \rightarrow A^{g(\omega_1')}C^{g(\omega_2)}$; and then $A^{g(\omega_1')} \rightarrow A^{g(\omega_1)}B^{g(\varphi)}$ where $\xi = \omega_1'\omega_2$ and $\omega_1' = \omega_1\varphi$. Since these two operations are similar, it is only necessary to indicate how to do one of them. The flowchart of Figure 3 indicates how $A^{g(\xi)} \rightarrow A^{g(\omega_1')}C^{g(\omega_2)}$ is performed.

The basis of the method used is the identity $g(\omega_1\omega_2) = g(\omega_1) \times q^{|\omega_2|} + g(\omega_2)$. In particular, for a single symbol α, $g(\omega\alpha) = g(\omega) \times q + g(\alpha)$ with $g(\alpha) < q$. The loop in the flow diagram successively picks a simple symbol off the right end of ω_1' and adds it to ω_2. At each stage the loop is entered at box 2B with the string

$$A^{g(\mu\alpha)}C^{g(\upsilon)}E^{q^{|\upsilon|}}.$$

Box 2B converts this to

$$A^{g(\mu)}C^{g(\upsilon)}D^{g(\alpha)}E^{q^{|\upsilon|}}$$

since $g(\mu) = [g(\mu\alpha)/q]$ and $g(\alpha) = R(g(\mu\alpha)/q)$. Box 2C converts this to

$$A^{g(\mu)}C^{g(\alpha\upsilon)}E^{q^{|\upsilon|}}$$

since

$$g(\alpha\upsilon) = g(\alpha) \times q^{|\upsilon|} + g(\upsilon).$$

Box 2D converts $E^{q^{|\upsilon|}}$ into $E^{q^{|\alpha\upsilon|}}$. Box 2A creates a single E since initially $|\upsilon| = 0$. The direct path from 2A to 2E is present because it should be possible for ω_2 to be null.

Block 4 will now be described since it is similar to block 2. The flowchart of

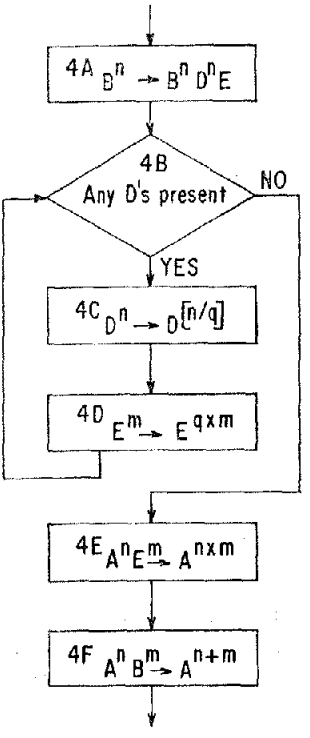

FIG. 4. Flowchart for recombining strings

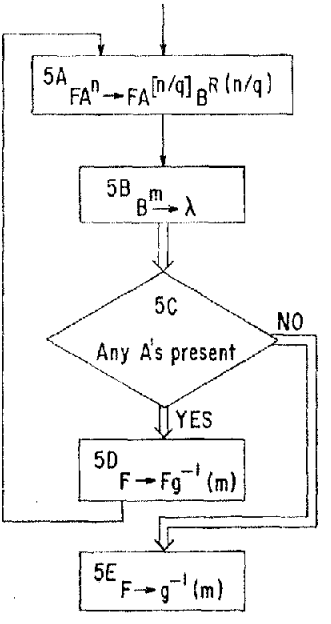

FIG. 5. Flowchart for decoding strings. Double arrows indicate multiple paths

Figure 4 will perform the operation $A^{g(\omega_1)}B^{g(\psi)} \rightarrow A^{g(\omega_1\psi)}$. It is this coding of the concatenation of strings which is the essence of block 4.

Since $g(\omega_1\psi) = g(\omega_1) \times q^{|\psi|} + g(\psi)$, the trick is to calculate $q^{|\psi|}$. This is done by the loop formed by blocks 4B, 4C, and 4D which records this value as the number of E's; i.e. on entering 4E we have $A^{g(\omega_1)}B^{g(\psi)}E^{q|\psi|}$. Box 4B can be implemented as a production whose core is $D \rightarrow D$ with 4C and 4E in the success and failure fields, respectively.

Block 5, $FA^{g(\xi)} \rightarrow \xi$, is implemented as shown in Figure 5.

Block 5A is always entered with a string of the form $F\xi''A^{g(\xi'\alpha)}$ where $\xi = \xi'\alpha\xi''$ and α is a single symbol. Block 5A converts this string to $F\xi''A^{g(\xi')}B^{g(\alpha)}$. Blocks 5B, 5C, and 5D produce $F\alpha\xi''A^{g(\xi')}$. These blocks will be further described by giving their component productions. Block 5B, whose first production is b_1, contains the following q productions.

$$(b_1) \quad B \rightarrow \lambda \quad S(b_2)$$

$$(b_i) \quad B \rightarrow \lambda \quad S(b_{i+1})F(c_{i-1}) \quad \text{for } i = 2, \cdots, q-1$$

$$(b_q) \quad B \rightarrow \lambda \quad F(c_{q-1})$$

Block 5B is entered with a string containing j B's where $1 \leq j \leq q-1$ since $j = g(\alpha)$ for some single symbol α, so that the failure field of production b_{j+1} will be taken. Production c_j will then be executed, and α will be printed out after determining if there are any A's remaining in the string.

Block 5C consists of the following $q - 1$ rules, which test for the presence of an A while retaining the value of j for $j = 1, \cdots, q - 1$.

(c_j) $\quad A \to A \quad S(d_j)F(e_j)$

Block 5D consists of the following $q - 1$ rules for $j = 1, \cdots, q - 1$, whose success fields contain the label of the first production of block 5A.

(d_j) $\quad F \to Fg^{-1}(j) \quad s(5A)$

Block 5E ends the derivation, and consists of the following $q - 1$ rules for $j = 1, \cdots, q - 1$.

(e_j) $\quad F \to g^{-1}(j)$

The same procedure which was used for 5B will be used for block 3. This block performs $B^{g(\varphi)} \to B^{g(\psi)}$ where $\varphi \to \psi$ is a production of G.

For each production $\varphi \to \psi$ of G, G' will have a production

(p) $\quad D \to B^{g(\psi)} \quad S(h)$

where h is the label of the first production of block 4 and the label p is different for each production.

Let r be the maximum integer such that $g^{-1}(r)$ is the left-hand side of a production of G. For each integer i between 1 and r let $P(i)$ be the set of labels of productions of G' corresponding to productions of G which have $g^{-1}(i)$ on their left-hand side. For some values of i, $P(i)$ may be null.

Block 3 consists of the following productions:

(s_1) $\quad B \to D \quad S(s_2)$

(s_i) $\quad B \to \lambda \quad S(s_{i+1})F(P(i-1))$ for $i = 2, \cdots, r$

(s_{r+1}) $\quad B \to \lambda \quad F(P(r))$

as well as the set of productions

(p) $\quad D \to B^{g(\psi)} \quad S(h)$

corresponding to the productions of G.

If block 2 selects a subsequence φ which is the left-hand side of a production of G, block 3 will make an appropriate replacement. If the selected subsequence does not correspond to any production of G, block 3 will halt the derivation without having produced a sentence.

To complete the proof it only remains to show how to do the arithmetic called for in some of the subblocks. Since the order of the symbols in the intermediate string does not matter, the following simple programmed grammars will suffice.

For multiplication, the following grammar converts A^mB^n into $A^mB^nC^{m\times n}$:

(1) $\quad A \to A'A'' \quad S(1)F(2)$

(2) $\quad A'' \to \lambda \quad S(3)F(5)$

(3) $\quad B \to B'C \quad S(3)F(4)$

(4) $\quad B' \to B \quad S(4)F(2)$

(5) $\quad A' \to A \quad S(5)F(\text{out})$

For division by q, the following program converts A^n into $A^{[n/q]}C^{R(n/q)}$:

(1)	$A \rightarrow A'$	$S(2)F(2q)$
(i)	$A \rightarrow \lambda$	$S(i+1)F(q+i-1), \quad i = 2, \cdots, q-1$
(q)	$A \rightarrow \lambda$	$S(1)F(2q-1)$
$(q+i)$	$A' \rightarrow C^i$	$S(2q), \quad i = 1, \cdots, q-1$
$(2q)$	$A' \rightarrow A$	$S(2q)F(\text{out})$

Properties of CFPG's

In this section some of the properties of cfpg's are given. First it is noted without proof that the set of cfpg languages is closed under some of the standard operations on languages.

THEOREM. *The set of cfpg languages is closed under union, concatenation, and Kleene closure.*

THEOREM. *Every recursively enumerable language can be generated with tails by a cfpg.*

PROOF. Let G_1 be an arbitrary phrase structure grammar which generates a recursively enumerable language L_1 over vocabulary V_T. By using the procedure of Theorem 6, we can obtain a pg, G_2, which generates L_1 and for which the left-hand side of each core consists of a single nonterminal symbol, although the right-hand side could be the null string. Let c and d be new symbols which are not in V_T. G_2 can now be modified to produce a cfpg G_3 with terminal vocabulary $V_T \cup \{c, d\}$ whose language L_3 is L_1 with tails. G_3 differs from G_2 in that block 1 of Figure 2 is changed to $S' \rightarrow FdA^{\psi(S)}$, and every core of the form $A \rightarrow \lambda$ is replaced by $A \rightarrow c$. G_3 thus operates like G_2, but it generates c's instead of erasing. Note that the only productions of G_3 which have a member of V_T on the right-hand side of their cores are the F-rules of blocks 5D and 5E of Figure 5. Since F is the leftmost symbol in an intermediate string, the occurrences of members of V_T must be to the left of the d in a sentence of L_3. Therefore $L_3 = \{xdc^m \mid x \in L_1$ and the set of allowable m depends on $x\}$, and L_3 is L_1 with tails.

A string *homomorphism* is a function h mapping strings into strings which has the property that for any two strings x and y, $h(xy) = h(x)h(y)$. Thus a homomorphism is a symbol translation of a string. It follows from the previous theorem that every recursively enumerable language is the homomorphic image of a cfpg language.

Thus far when a pg has generated a sentence, the productions have been applied as far to the left as possible; i.e. the leftmost occurrence in the intermediate string of the left-hand side of the core is replaced by the right-hand side of the core. The grammar can be said to be operating under the *leftmost interpretation*. A pg operating under the *rightmost interpretation* is one for which every production is applied as far to the right as possible. A pg operating under the *free interpretation* is one for which any occurrence in the intermediate string of the left-hand side of the core can be replaced by the right-hand side.

THEOREM. *The set of languages generated by cfpg's operating under the rightmost interpretation is identical to the set of languages generated by cfpg's operating under the leftmost interpretation.*

PROOF. Let G be a cfpg which generates language L while operating under the rightmost interpretation. Another cfpg, G', is described which generates L while operating under the leftmost interpretation. Let a production of G be

$$(r) \qquad A \to \psi \qquad S(V)\ F(W)$$

The grammar G' has the following set of productions which have the same effect as r: they replace the rightmost occurrence of A in the intermediate string by ψ.

$$\begin{aligned}
&(r\,') && A \to A' && S(r_2)\ F(W)\\
&(r_2) && A \to A' && S(r_3)\ F(r_4)\\
&(r_3) && A' \to A'' && S(r_2)\\
&(r_4) && A' \to \psi && S(r_5)\\
&(r_5) && A'' \to A && S(r_5)\ F(V)
\end{aligned}$$

If there are any A's present in the intermediate string, r_4 will be applied to the rightmost one and the next rule will be selected from V (the success field).

A similar procedure can be applied to a cfpg operating under the leftmost interpretation to obtain another cfpg which generates the same language when operating under the rightmost interpretation.

It follows from the above theorem that the set of cfpg languages is closed under reversal.

THEOREM. *The set of languages generated by cfpg's operating under the leftmost interpretation contains the set of languages generated by cfpg's operating under the free interpretation.*

PROOF. Let G be a cfpg which generates language L under the free interpretation. Another cfpg, G', is described which generates L under the leftmost interpretation.

Let a typical production of G be the following.

$$(r) \qquad A \to \psi \qquad S(V)F(W)$$

G' will have the following four rules which will correspond to this rule of G.

$$\begin{aligned}
&(r) && A \to A' && S(r, r_2)F(r_3)\\
&(r_2) && A \to \psi && S(r_4)F(r_3)\\
&(r_3) && A' \to A && S(r_2)F(W)\\
&(r_4) && A' \to A && S(r_4)F(V)
\end{aligned}$$

These four rules when operating under the leftmost interpretation have the effect of permitting any occurrence of A in the intermediate string to be replaced by ψ, thereby simulating the original rule when it operates under the free interpretation. Since the above procedure is applied to every rule of G, G' generates L under the leftmost interpretation.

Since a context-free phrase structure grammar can be effectively converted into a cfpg which generates the same language, the undecidable questions for context-free grammars are also undecidable for cfpg's. However some problems which are decidable for context-free grammars are undecidable for cfpg's.

THEOREM. *It is undecidable whether or not the language generated by a cfpg is empty.*

PROOF. Let G be an arbitrary phrase structure grammar, and let G' be the corresponding pg constructed by the method given in the proof of Theorem 6. Then $L(G')$, the language generated by G', is identical to $L(G)$, and the left-hand side of every core of G' consists of a single nonterminal symbol. Now modify G' by introducing a new terminal symbol e and replacing each production core of the form $A \rightarrow \lambda$ with the new production core $A \rightarrow e$. This new grammar, G'', is a cfpg. If G'' generates any sentences, G generates that sentence with the e's removed. Thus $L(G'')$ is empty if and only if $L(G)$ is empty. But since G is an arbitrary phrase structure grammar, it is undecidable whether or not its language is empty. Hence it is undecidable if $L(G'')$ is empty.

THEOREM. *It is undecidable if a cfpg generates a finite or an infinite number of sentences.*

PROOF. Let G be a cfpg whose sentence symbol is S. Let G' be G with the following production added.

$$(r) \qquad S \rightarrow aS \qquad S(r, \text{ or labels of } S\text{-rules of } G)$$

Now $L(G')$ is infinite if and only if $L(G)$ is nonempty, which is undecidable from the previous theorem.

THEOREM. *It is undecidable if a cfpg generates a context-free language.*

PROOF. Given a cfpg G, we can obtain another cfpg, G', whose language is the concatenation of $L(G)$ and the language $\{a^n b^n c^n \mid n \geq 1\}$ where a, b, c are new terminal symbols. If $L(G)$ is nonempty, then $L(G'')$ is also nonempty and is not context-free. If $L(G)$ is empty, then $L(G')$ is also empty and therefore context-free. Thus $L(G')$ is context-free if and only if $L(G)$ is empty, which is undecidable.

Unconditional Transfer CFPG's

An *unconditional transfer* pg is one for which the success and failure fields of each rule are identical, so that they act like an unconditional transfer. A utcfpg is an unconditional transfer grammar which is also a cfpg. An example of a utcfpg is the following grammar for generating the language nha^n.

(1)	$S \rightarrow 1SB$	$S(3)F(3)$
(2)	$S \rightarrow OS$	$S(3)F(3)$
(3)	$A \rightarrow BB$	$S(3, 4)F(3, 4)$
(4)	$A \rightarrow C$	$S(5)F(5)$
(5)	$B \rightarrow A$	$S(5, 6)F(5, 6)$
(6)	$B \rightarrow C$	$S(1, 2, 7)F(1, 2, 7)$
(7)	$S \rightarrow h$	$S(8)F(8)$
(8)	$A \rightarrow a$	$S(8)F(8)$

Note that for this grammar if C is ever written in an intermediate string, the resulting string cannot be expanded into a terminal string since no production core has C on the left-hand side. Therefore whenever rules 4 and 6 are used in the deriva-

tion of an actual sentence, they must fail because if either one were ever successful the resulting intermediate string could not be expanded into a terminal string.

It will be shown that the set of utcfpg languages is a proper subset of the cfpg languages. But first some additional machinery for dealing with nonterminal maps will be provided.

A partial ordering on the n-tuples of natural numbers will be defined by saying that $v \geq u$ if each component of v is greater than or equal to the corresponding component of u. A set of vectors (n-tuples) is *noncomparable* if for no u, v in the set is it true that $u \geq v$.

Some additional notation will be introduced. First 0 will represent the 0-vector, i.e. the vector all of whose components are 0. Also if v is a vector and r is a production whose core is $A \rightarrow \psi$, we will say that r succeeds on v if the A-component of v is nonzero. If r succeeds on v, then the application of r to v produces the vector obtained by subtracting 1 from the A-component of v and then adding $\delta_N(\psi)$ to the result (the addition being component by component).

LEMMA 4. *Every set of noncomparable vectors is finite.*

The proof of this lemma is given by Ginsburg [15] and can also be obtained from one of Konig's theorems [16].

THEOREM. *The prefix property is decidable for utcfpg's; i.e. it is decidable for any utcfpg, G, and any string, x, whether or not there exists a y such that $xy \in L(G)$.*

PROOF. A procedure, consisting of the construction of a finite directed graph based on x and G, will be given for deciding if x is a prefix of any sentence in $L(G)$. First rewrite G, if necessary, so that it contains only one S-rule whose label is r_0. Let $G = (V_T, V_N, J, P, S)$ and $n = |x|$. The nodes of the graph will be triples of the form (φ, v, r) where φ is a string over $V_T \cup V_N$ of length less than or equal to n, v is a nonterminal vector, and r is a rule label. If $|\varphi| < n$, then v will be the 0-vector.

Let (φ, v, r) and (ψ, u, q) be triples of the above type. We say that $(\varphi, v, r) \rightarrow (\psi, u, q)$ if q is in the go-to field of rule r and if a string of the form $\varphi\xi$ where $\delta_N(\xi) = v$ is converted by r into the string $\psi\xi'$ with $\delta_N(\xi') = u$. More precisely, exactly one of the following must be true.

(1) Rule r fails on φ and v, $\psi = \varphi$, and $u = v$.

(2) Rule r fails on φ but succeeds on v, $\psi = \varphi$, and u equals the vector resulting from the application of r to v.

(3) Rule r succeeds on φ, producing φ' of length less than or equal to n, $\psi = \varphi'$, and $u = v$.

(4) Rule r succeeds on φ, producing $\varphi' = \varphi_1\varphi_2$ where $|\varphi_1| = n$ and $|\varphi_2| \geq 1$, $\psi = \varphi_1$, and $u = v + \delta_N(\varphi_2)$.

Note that φ, v, and r uniquely specify ψ and u. If the go-to field of r is empty, then q is a special symbol rather than a rule label. We also say that $(\varphi, v, r) \Rightarrow (\psi, u, q)$ if there is a chain of triples such that $(\varphi, v, r) = (\xi_0, w_0, p_0) \rightarrow (\xi_1, w_1, p_1) \rightarrow \cdots \rightarrow (\xi_m, w_m, p_m) = (\psi, u, q)$, or if $(\psi, u, q) = (\varphi, v, r)$.

The graph used to decide if x is a prefix of $L(G)$ will have a tree structure. The initial node of the graph is the triple $(S, 0, r_0)$ where r_0 is the label of the S-production of G. The graph is constructed by expanding in some arbitrary order the nodes which are already present in the graph. Let (φ, v, r) be a node which is to be expanded. The triples (ψ, u, q) such that $(\varphi, v, r) \rightarrow (\psi, u, q)$ are found. For each such triple, if there is already a node in the graph of the form (ψ, u', q) with $u' \leq u$, then (ψ, u, q) is not added to the graph. Otherwise (ψ, u, q) is added and an arrow

drawn to it from (φ, v, r). This procedure is carried out for all nodes in the graph. Note that the final form of the graph will depend on the order in which the nodes are expanded.

It will now be shown that the graph is finite. For any φ, r combination let $v_1, v_2, \cdots$ be a list of vectors such that the nodes (φ, v_i, r) are successively added to the graph. Each time a node is added to the list, its vector cannot be greater than or equal to that of any of the other nodes having the same φ and r which are already on the list.

Note that for any vector v, the number of vectors u such that $u \leq v$ is finite, since no such u can have a component which is greater than the corresponding component of v. Now assume that the list of vectors is infinite. An infinite noncomparable subset of the vectors on the list can then be obtained by the following procedure. Start with v_1 and cross off the list the finite set of vectors which are less than or equal to v_1. Let v_2 be the next remaining entry on the list. Then v_1, v_2 are noncomparable since all the vectors less than or equal to v_1 have been crossed off the list and v_2 cannot be greater than or equal to v_1 as it occurs after v_1 on the list. Now cross off the list the set of vectors which are less than or equal to v_2. The resulting v_1, v_2, v_3 are noncomparable. By this procedure an infinite noncomparable set of vectors can be obtained if the original list is infinite. From the preceding lemma such an infinite set cannot exist, and the list must contain a finite number of vectors.

Since there are only a finite number of strings of length less than or equal to n and only a finite number of productions, the graph contains only a finite number of nodes.

It will now be shown that there exists a (possibly empty) terminal string y such that $(S, r_0) \Rightarrow (xy, r)$ if and only if the graph contains the node $(x, 0, r)$. Here r can be either a rule label or the special symbol which denotes the empty set of labels.

First it will be shown that if (φ, v, r) is a node in the graph, then there exists a string ξ such that $(S, r_0) \Rightarrow (\varphi\xi, r)$ and $\delta_N(\xi) = v$. Let ρ be the sequence of rule labels (third components) of the chain of nodes beginning with the initial node and leading to (φ, v, r) but omitting the label of the final node in the chain. Then by induction on the length of the chain, applying ρ to the string S will result in a string of the form $\varphi\xi$ with $\delta_N(\xi) = v$ and with r as the next applicable rule. First the result is true for the initial node of the graph, $(S, 0, r_0)$, since r_0 is an S-rule. Now assume that the result is true for all chains of length m and that $(S, 0, r_0) \Rightarrow (\varphi_m, v_m, r_m) \rightarrow (\varphi_{m+1}, v_{m+1}, r_{m+1})$. By the induction hypothesis, there is a string ξ_m such that $(S, r_0) \Rightarrow (\varphi_m\xi_m, r_m)$ and $\delta_N(\xi_m) = v_m$. However since $(\varphi_m, v_m, r_m) \rightarrow (\varphi_{m+1}, v_{m+1}, r_{m+1})$, applying r_m to $\varphi_m\xi_m$ will produce a string $\varphi_{m+1}\xi_{m+1}$ such that $\delta_N(\xi_{m+1}) = v_{m+1}$. Hence $(S, r_0) \Rightarrow (\varphi_{m+1}\xi_{m+1}, r_{m+1})$ and $\delta_N(\xi_{m+1}) = v_{m+1}$. Therefore if $(x, 0, r)$ appears in the graph, there must be a string ξ such that $(S, r_0) \Rightarrow (x\xi, r)$ and, since $\delta_N(\xi) = 0$, ξ consists solely of terminal symbols.

It will now be shown that if $(S, r_0) \Rightarrow (\varphi\xi, r)$, where $|\varphi| = n$ if ξ is nonnull, then there is a vector v such that (φ, v, r) appears in the graph and $v \leq \delta_N(\xi)$. This result can be obtained by induction on the length of the derivation of $(\varphi\xi, r)$ from (S, r_0). If the length of the derivation is zero, then $\varphi\xi = S$, and $(S, 0, r_0)$ is the initial node of the graph. Assume that the result is true for all derivations of length less than or equal to m and that $(S, r_0) \Rightarrow (\varphi_m\xi_m, r_m) \rightarrow (\varphi_{m+1}\xi_{m+1}, r_{m+1})$. Then from the induction hypothesis there is a node in the graph of the form (φ_m, v_m, r_m)

with $v_m \leq \delta_N(\xi_m)$. Consider the effect of applying rule r_m to that node. Since $(\varphi_m\xi_m, r_m) \rightarrow (\varphi_{m+1}\xi_{m+1}, r_{m+1})$, we have $(\varphi_m, v_m, r_m) \rightarrow (\varphi_{m+1}, v_{m+1}, r_{m+1})$ for some vector v_{m+1}. Furthermore since $\delta_N(\xi_m) \geq v_m$, any rule which succeeds on v_m will also succeed on ξ_m and $\delta_N(\xi_{m+1}) \geq v_{m+1}$. Because in constructing the graph, the node (δ_m, v_m, r_m) must have been expanded at some time, the graph contains a node of the form $(\varphi_{m+1}, v'_{m+1}, r_{m+1})$ where $v'_{m+1} \leq v_{m+1}$. Thus if $(S, r_0) \Rightarrow (\varphi\xi, r)$, there is a vector $v \leq \delta_N(\xi)$ such that (φ, v, r) appears in the graph. Hence if $(S, r_0) \Rightarrow (xy, r)$ where $\delta_N(y) = 0$, the graph must contain a node of the form $(x, 0, r)$.

Thus to determine if x is a prefix of $L(G)$, all we need to do is construct the graph and see if it contains a node of the form $(x, 0, r)$, since x is a prefix of $L(G)$ if and only if the graph contains such a node.

COROLLARY 2. *It is decidable whether or not the language generated by an utcfpg is empty.*

PROOF. Let $G = (V_T, J_N, J, P, S)$ be an utcfpg. For each $a \in V_T$, use the procedure of the theorem to determine if a is a prefix of $L(G)$. If for any a, a is a prefix, then there exists a (possibly null) terminal string y such that $ay \in L(G)$, which is therefore nonempty. If no $a \in V_T$ is a prefix, then $L(G)$ is empty.

COROLLARY 3. *For the class of pg's with unconditional transfer go-to fields, single symbols on the left-hand side of the cores, and arbitrary (possibly null) strings on the right-hand side, it is decidable whether or not the language generated by such a grammar is empty.*

PROOF. Let $G = (V_T, V_N, J, P, S)$ be such a grammar. Choose an $a \in V_T$, and let $G' = (V_T, V_N, J, P', S)$ be the grammar obtained by replacing each core of G of the form $A \rightarrow \lambda$ with the new core $A \rightarrow a$. Then since $L(G')$ is empty if and only if $L(G)$ is empty, and G' is an utcfpg, for which the emptiness problem is decidable, it is decidable if $L(G)$ is empty.

COROLLARY 4. *There exists a cfpg whose language cannot be generated by any utcfpg.*

PROOF. Let L_1 be a recursively enumerable language which is not recursive. As has previously been shown, a cfpg, G_2, exists which generates L_1 with tails; i.e.

$$L_1 = \{x \mid xdc^n \in L(G_2) \text{ for some } n\}.$$

Now assume that $L(G_2)$ can be generated by an utcfpg, G_3. Then given any string x, we can decide if xd is a prefix of $L(G_2)$ by using the procedure of the preceding theorem on G_3 and x. But xd is a prefix of $L(G_2)$ if and only if $x \in L_1$, which is not decidable because L_1 is not recursive. Hence no such G_3 exists, and $L(G_2)$ is a cfpg language which cannot be generated by any utcfpg.

We have now proved Theorem 7, since every utcfpg is also a cfpg, and a context-free grammar can be converted into an utcfpg which generates the same language by inserting the set of all production labels into the success and failure field of each production.

Further Properties of UTCFPG's

In this section some additional properties of utcfpg's are given. First it should be noted that the set of utcfpg languages is closed under union, concatenation, and Kleene closure. Some undecidability properties are now given.

THEOREM. *It is undecidable if a cfpg generates an utcfpg language.*

PROOF. Let L_1 be a recursively enumerable language which is not recursive, and let G_2 be a cfpg which generates L_1 with tails.

Given an arbitrary cfpg, G_3, we can effectively obtain another cfpg, G_4, whose language is the concatenation of $L(G_2)$ and $L(G_3)$.

If $L(G_3)$ is empty then $L(G_4)$ is empty and therefore is a utcfpg language. If $L(G_3)$ is nonempty, then $L(G_4)$ is nonempty, and $x \in L_1$ if and only if xd is a prefix of $L(G_4)$. If there existed an utcfpg which generates $L(G_4)$, L_1 would be recursive since the prefix property is decidable for utcfpg's. Hence if $L(G_3)$ is nonempty, then $L(G_4)$ is not an utcfpg language.

Thus $L(G_4)$ is an utcfpg language if and only if $L(G_3)$ is empty, which is not decidable. Therefore since G_4 can be constructively obtained from G_3, it is undecidable if G_4 generates an utcfpg language, and it is undecidable if a cfpg generates an utcfpg language.

THEOREM. *It is undecidable if the language generated by an utcfpg is context-free.*

PROOF. This proof is based on another proof [17] of a similar theorem.

Let G_2 be an utcfpg whose language L_2 over terminal vocabulary V_2 is not context-free. L_2 could, for instance, be the language $\{a^n b^n c^n\}$. Let G_1 be an arbitrary context-free grammar which generates language L_1 over a terminal vocabulary, V_1, which is disjoint from V_2. From G_1 and G_2 an utcfpg, G_3, can be effectively obtained such that G_3 generates the language $L_3 = L_1 V_2^* \cup V_1^* L_2$. Here V^* denotes the set of all nonnull strings over vocabulary V.

Now if $L_1 = V_1^*$, then $L_3 = V_1^* V_2^*$ and L_3 is context-free. If $L_1 \neq V_1^*$ and L_3 is context-free, then for $x \in V_1^* - L_1$, $L_3 \cap xV_2^* = xL_2$ is context-free since the intersection of a context-free language and a regular set is context-free. But then L_2 is context-free since it can be obtained from xL_2 by a homomorphism, and context-free languages are closed under homomorphisms. Thus if $L_1 \neq V_1^*$, then L_3 is not context-free. Consequently L_3 is context-free if and only if $L_1 = V_1^*$, which is undecidable [18]. Thus it is undecidable if an utcfpg generates a context-free language.

THEOREM. *The set of utcfpg languages is not closed under intersection.*

PROOF. This proof is based on a well-known technique for establishing undecidability results. Let $G_1 = (V_T, V_N, S, P_1)$ be a phrase structure grammar whose language is not recursive, and let c be a new symbol not in $V_T \cup V_N$. From G_1 we will obtain two context-free grammars, G_2 and G_3, whose languages are of the form $x_1 c x_2 c \cdots c x_{2n+1}$ where each x_i is an arbitrary (possibly null) string over $V_T \cup V_N$. $L(G_2)$ will have the property that $x_{n+1} = S$, and for each k between 1 and n, x_{2k-1} can be derived from x^r_{2k} by G_1. $L(G_3)$ will have the property that x_1 is a terminal string, and for each k between 1 and n, x_{2k} can be derived from x_{2k+1} by G_1. Here x^r denotes the reversal of x.

G_2 is of the form $(V_T \cup V_N \cup \{c\}, \{T, A\}, P_2, T)$ where P_2 contains the following productions.

$$T \rightarrow AcT$$
$$T \rightarrow S$$

$$A \rightarrow \alpha A \alpha \qquad \text{for each } \alpha \in V_T \cup V_N$$
$$A \rightarrow \psi A \varphi^r \qquad \text{for each production } \varphi \rightarrow \psi \text{ of } G_1$$
$$A \rightarrow c$$

G_3 has a similar form.

Now consider $L_4 = L(G_2) \cap L(G_3)$. If $x_1cx_2c \cdots cx_{n+1} \in L_4$, then $x_{n+1} = S$, each x_i is derivable from x_{i+1} by G_1, and x_1 is a terminal string of G_1. Similarly if $x_1 \in L(G_1)$, then corresponding to the derivation of x_1 there is a sentence in L_4 which begins with x_1c. Thus $x_1c \in L_1$ if and only if x_1c is a prefix of L_4. But if L_4 were an utcfpg language, then the prefix property would be decidable and L_1 would be recursive. Hence L_4, the intersection of two utcfpg languages, is not an utcfpg language.

COROLLARY 5. *The set of utcfpg languages is not closed under complementation.*

PROOF. Since the set of utcfpg languages is closed under union and for languages L_1 and L_2, $L_1 \cap L_2 = \overline{\bar{L}_1 \cup \bar{L}_2}$, closure under complementation would imply closure under intersection. From the previous theorem, the utcfpg languages are not closed under intersection and therefore are not closed under complementation.

Conclusions

The languages generated by pg's with various types of cores have been investigated. The additional machinery of pg's does not add any generative power if the cores are one-sided linear, linear, context-sensitive, or arbitrary. A key result is that pg's whose cores have a single symbol on the left-hand side and an arbitrary string on the right-hand side can generate all recursively enumerable languages.

The cfpg's, however, generate a class of languages which properly contains the context-free languages and is properly contained within the context-sensitive languages. Cfpg's have considerable generative power, and it is often comparatively easy to write a cfpg for a particular language. However several problems which are decidable for context-free grammars are undecidable for cfpg's.

The utcfpg languages are properly contained within the cfpg languages and properly contain the context-free languages. Utcfpg's seem to be easier to work with than the cfpg's; e.g. the prefix property is decidable for utcfpg's.

ACKNOWLEDGMENT. The author wishes to express his gratitude to Professor S. H. Unger of Columbia University for his guidance and encouragement. The author also gratefully acknowledges several interesting and helpful discussions with Dr. A. Friedes and Dr. J. D. Ullman of Bell Telephone Laboratories and with Dr. D. H. Younger of the University of Waterloo.

REFERENCES

1. CHOMSKY, N. On certain formal properties of grammars. *Inform. Contr. 2* (June 1959), 137–167.
2. ——. Formal properties of grammars. In *Handbook of Mathematical Psychology, Vol. 2*, Luce, R.D., Bush, R.R., and Galanter, E. (Eds.), Wiley, New York, 1963.
3. ROSENKRANTZ, D. J. Programmed grammars—a new device for generating formal languages. Ph.D. Thesis, Columbia U., New York, 1967.
4. ABRAHAM, S. Some questions of phrase structure grammars I. *Comput. Linguist. 4* (1965), 61-70.
5. PETERS, S. A note on ordered phrase structure grammars. Rep. No. NSF-17, Math. Linguist. and Auto. Trans., Computation Lab., Harvard U., Cambridge, Mass., Aug. 1966.
6. GINSBURG, S., AND SPANIER, E. H. Control sets on grammars. Doc. No. TM-738/036/00, System Development Corp., Santa Monica, Calif., 1967.
7. CHOMSKY, N. *Aspects of the Theory of Syntax*. M.I.T. Press, Cambridge, Mass., 1965.
8. ZWICKY, A. M., FRIEDMAN, J., HALL, B. C., AND WALKER, D. E. The MITRE syntactic

analysis procedure for transformational grammars. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Spartan Books, Washington, D.C., pp. 317–326.

9. LANDWEBER, P. S. Three theorems on phrase structure grammars of type 1. *Inform. Contr. 6* (June 1963), 131–136.
10. KURODA, S. Y. Classes of languages and linear bounded automata. *Inform. Contr. 7* (June 1964), 207–223.
11. AHO, A. V. Indexed grammars—an extension of context free grammars. IEEE Conference Record on Switching and Automata Theory, IEEE pub. no. 16-C-56, 1967.
12. FARBER, D. J., GRISWOLD, R. E., AND POLONSKY, I. P. SNOBOL, a string manipulation language. *J. ACM 11*, 1 (Jan. 1964), 21–30.
13. MARKOV, A. A. The theory of algorithms. (Russian), Tr. Math. Inst. Akad. Nauk SSSR *38* (1951), (English Trans., Amer. Math. Soc. Trans., [2], *15*, (1960), 1–14).
14. STEARNS, R. E., HARTMANIS, J., AND LEWIS, P. M. Hierarchies of memory limited computations. IEEE Conference Record on Switching Circuit Theory and Logical Design, IEEE pub. no. 16-C-13, 1965.
15. GINSBURG, S. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, New York, 1966.
16. KONIG, D. *Theorie der endlichen und unendlichen Graphen*. Chelsea, New York, 1950.
17. GINSBURG, S., GREIBACH, S. A., AND HARRISON, M. A. One-way stack automata. *J. ACM 14*, 2 (Apr. 1967), 389–414.
18. BAR-HILLEL, Y., PERLES, M., AND SHAMIR, E. On formal properties of simple phrase structure grammars. *Z. Phonetik, Sprachwissen. Kommunikationsforsch. 14* (1961), 143–172; also in Y. BAR-HILLEL, *Language and Information*, Addison-Wesley, Reading, Mass., 1965, pp. 116–150.

RECEIVED MAY, 1968; REVISED JULY, 1968