Detection of Generative Ambiguities in Context-Free Mechanical Languages*

SAUL GORN

University of Pennsylvania, Philadelphia, Pennsylvania

1. *Introduction*

By a *context-free* mechanical language we mean one specified recursively by a finite set of replacement rules (a production system) of the form $\langle \alpha_i \rangle ::= \psi_i$ where each ψ_i is a finite concatenation of characters from a *base* alphabet (often called terminal characters, or vocabulary) and the alphabet of the $\langle \alpha_i \rangle$ (often called nonterminal characters, or syntactic types) such that no ψ_i is the null-string.

Such a system generates strings of characters from the base alphabet called *words* of the language $\langle \alpha_1 \rangle$; the set of generated words, we call the *extent* of the language $\langle \alpha_1 \rangle$ (called the *support* of the language by Schützenberger and Chomsky [4]). The bracketing characters "$\langle$" and "$\rangle$" indicate the extent.

Each finite program of sequenced applications of these replacement rules beginning with $\langle \alpha_1 \rangle$ and leading to a word of the language is called a *derivation* of that word. A production system in which some $\langle \alpha_i \rangle$ never appear in the derivation of a word we will call *generatively inadmissible*. With the requirement that the recursive generator (i.e. the production system) be generatively admissible, our definition of context-free language agrees in extent with that of Schützenberger and Chomsky.

We are concerned here with the detection of a type of syntactic ambiguity of words of a context-free language. We call a word *generatively ambiguous* (also called *structurally ambiguous*) with respect to the production system if it has essentially more than one derivation.

All context-free languages are *decidable*. By this we mean that for each one we can construct a processor called a *recognizer* which, for any string from the terminal alphabet, can decide in a finite time whether that string does or does not belong to the language. It will be easy to see below that any generatively admissible context-free language possesses an *ordinal generator* (one which produces all words of the language in sequence) which is *approximately monotonic*; by this we mean that there is a recursively increasing sequence of natural num-

* This paper was presented to the Congress of the International Federation for Information Processing, Aug. 27–Sept. 1, 1962, Munich, Germany, at Session 39 (Symposium on Languages for Processor Construction). The abstract appears in the proceedings of the Congress.

The background work on mechanical languages in this paper was originally supported by the U.S. Army Signal Corps from 1958 to 1960 under Contract DA-36-039-SC-74047 to the University of Pennsylvania's Institute for Cooperative Research and The Moore School of Electrical Engineering. The developments in this paper were made possible by joint support from the National Science Foundation (NSF-G-14096) and the Air Force Office of Scientific Research (AF-49(638)-951).

bers n_m such that if $n_m < k \leqq n_{m+1}$ then the kth word produced must have length bigger than m. Any mechanical language with such an approximately monotonic ordinal generator is decidable.

Nevertheless, the problem of designing a processor which will accept the specifications of any context-free language and will decide in a finite time whether that language has words which are generatively ambiguous is unsolvable (see Schützenberger and Chomsky [4]; Bar-Hillel, Perles and Shamir [1]; Kantor [6]; Floyd [7]).

This paper will therefore restrict itself to the design of a *limited ambiguity detector.* We will design a generalized prefix language (see Gorn [2]) for the specification of derivations, and construct a *derivation generator.* We will also be able to define levels of derivation and therefore be able to construct an ambiguity detector working up to any level.

2. *The Schützenberger-Chomsky Formal Power Series*

The cited paper by Schützenberger and Chomsky considers words in the extent of a context-free language as the result of noncommutative products (concatenations) of characters from the (terminal) alphabet and uses an addition operation corresponding to unions of sets. Thus the rules of the production system become algebraic equations in a noncommutative and associative system.

For example, the language over the terminal alphabet $\{a, b\}$ which is specified in Backus form by the production

$$\langle \alpha_1 \rangle ::= a \mid b \mid a\langle \alpha_1 \rangle \mid \langle \alpha_1 \rangle b$$

is written as the implicit polynomial equation

$$\alpha_1 = a + b + a\alpha_1 + \alpha_1 b.$$

Treating this as an implicit function in the iterative functional equation form, we can begin to develop the formal power series for α_1 :

$$\begin{aligned} \alpha_1 &= a + b + a\alpha_1 + \alpha_1 b \\ &= a + b + a(a + b + a\alpha_1 + \alpha_1 b) + (a + b + a\alpha_1 + \alpha_1 b)b \\ &= a + b + aa + 2ab + bb + aa\alpha_1 + 2a\alpha_1 b + \alpha_1 bb \\ &= \text{etc.} \end{aligned}$$

The coefficients bigger than one in such developments indicate the number of distinct ways in which their word can be derived from the production rules.

The extent of this language (called the *support* by the cited authors) is the class of words represented by $\{a^m b^n\}$ where not both m and n are zero. It is the same as that of the language specified by:[1]

$$\begin{aligned} \alpha_1 &= \alpha_2 + \alpha_3 + \alpha_2\alpha_3 , \\ \alpha_2 &= a + a\alpha_2 , \\ \alpha_3 &= b + b\alpha_3 . \end{aligned}$$

[1] In the first production system the property, "there is no b to the left of an a," is true for words produced by the first two productions and is left unchanged by the third and

On the one hand, the latter form has, for example, $\langle\alpha_1\rangle ::= \langle\alpha_2\rangle$ as a rule in the system—a type of rule eliminated from consideration by the cited authors. On the other hand, formal expansion yields a power series for α_1 with the same words as before whose coefficients are not zero but in which all words are unambiguously derived and therefore have coefficient one. Thus the word ab has the two derivations, $\alpha_1 \rightarrow a\alpha_1 \rightarrow ab$ and $\alpha_1 \rightarrow \alpha_1 b \rightarrow ab$, in the first system, but only $\alpha_1 \rightarrow \alpha_2\alpha_3 \rightarrow a\alpha_3 \rightarrow ab$ in the second, since $\alpha_1 \rightarrow \alpha_2\alpha_3 \rightarrow \alpha_2 b \rightarrow ab$ is not considered a *distinct* derivation.[2] The method of analysis which will be described in this paper can be reflected back into the formal power series approach by permitting *derivation operator* coefficients instead of natural numbers to designate *amount of ambiguity*.

3. *The $\vee C$-Flow-Graph of a Context-Free Language*

We can use a two-dimensional representation of the recursive generator of a context-free language which is a modification of what we have elsewhere called *flow graphs* (see Gorn [3]). This language uses in its basic alphabet:

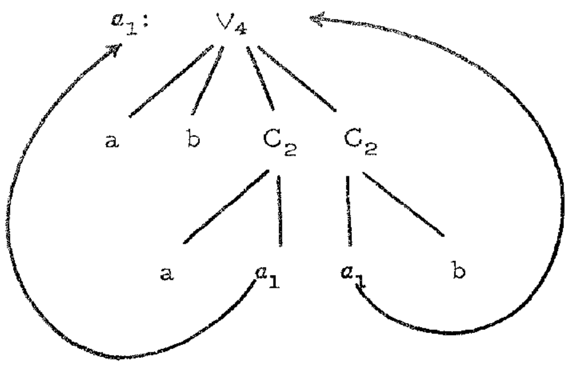
(a) directed lines (arrows)
(b) the characters C for *concatenation* and $\vee$ for *logical* OR
(c) the control alphabet of natural number subscripts
(d) the alphabet of *syntactic type* names, corresponding to Chomsky's non-terminal characters
(e) selected words from the alphabet of *terminal characters* over which the *object language* is to be defined
(f) the character ":"

The syntactic types will be *names* or *labels* of certain nodes of the graph, the first node of which will have the language name α_1 as label. No two distinct nodes will have the same label, and every node with more than one *entrance* will be labelled. The *control character* ":" will delimit the name of a node from its *content*. The content of a node which is not an *endpoint* of the graph will be a C or an $\vee$ with a subscript from the control alphabet of natural numbers; C_n will indicate a concatenation of n elements (we will not use C_0 or C_1) and a node with such content will have n *exiting* arrows; similarly $\vee_n$ will indicate n alternatives (we will not use $\vee_0$ or $\vee_1$) and such a node will also have n exiting arrows. Endpoints will have strings from the object (i.e. terminal) alphabet of type (e) above as their contents. Thus Figure 1 shows the $\vee C$-flow-graphs of the two recursive generators described above for the languages whose common extent (support) is the set $\{a^m b^n$, not both m and $n = 0\}$.

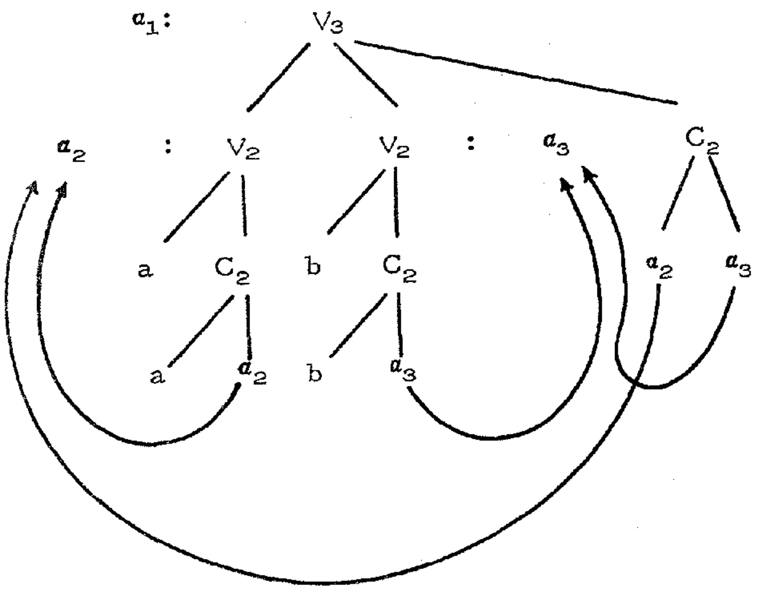
The structure of the $\vee C$-flow-graph of the context-free language is to be that

fourth. This is a recursive proof that all words of $\langle\alpha_1\rangle$ are in the stated form. It is easy to see that any such word can be produced. In the second system it is just as easy to see that $\langle\alpha_2\rangle = \{a^m;\ m > 0\}$ and $\langle\alpha_3\rangle = \{b^n;\ n > 0\}$; whence, the stated equivalence.

[2] Notice that the symbol "$\rightarrow$" used here does not have the same meaning as the substitution command used in flow charts. It is a "permissive" replacement rule which may be interpreted as "objective" replacement commands for the "production machine."



$\langle a_1 \rangle ::= a \mid b \mid a\langle a_1 \rangle \mid \langle a_1 \rangle b$



$\langle a_1 \rangle ::= \langle a_2 \rangle \mid \langle a_3 \rangle \mid \langle a_2 \rangle \langle a_3 \rangle$

$\langle a_2 \rangle ::= a \mid a\langle a_2 \rangle$

$\langle a_3 \rangle ::= b \mid b\langle a_3 \rangle$

FIG. 1. $\bigvee C$-flow-graphs of two production systems with extent $\{a^m b^n; m + n > 0\}$

of a flow graph, i.e. a finite connected graph possessing the following properties:

1. Every branch has an arrow in one and only one direction.
2. One and only one node is designated as the *root* or entrance (this will place the same restriction on the production system as the requirement of a unique α_1 does).
3. There is at least one node from which there is no exiting branch. Each such node is called an exit or endpoint (this condition will be made more rigid to permit only generatively admissible production systems).
4. Every node except the root possesses at least one entering branch (see remark on restriction 2).
5. No endpoint has more than one entering branch.

As remarked in Gorn [3], connected flow charts with a single entrance and at least one exit possess the structure of flow graphs; if C-nodes are considered to be AND-nodes where simultaneous action begins and $\bigvee$-nodes are considered to be OR-nodes where selection takes place and if the contents of nodes are to be programs of commands and their labels are to be *symbolic addresses*, then flow charts—whether purely sequential or calling for simultaneous action—are also $\bigvee C$-flow-graphs.

An $\bigvee C$-flow-graph is, then, a flow graph in which every node has either an $\bigvee$ or a C in it subscripted by the number of exiting branches, and every node with more than one entering branch, as well as the root, has a label.

We also require that an $\bigvee C$-flow-graph for a production system have the property we have needed in the cited reference for *output state diagrams*, namely, that at most one branch connects any two nodes.

Beginning with a context-free production system, we construct an $\bigvee C$-flow-graph for it by a standard translation procedure, phase 2; but first (phase 1) we subject it to a recognizer of its generative admissibility, and then of its connectivity.

4. *The Recognizer of Generative Admissibility (Phase 1) and the Construction of the Flow Graph (Phase 2)*

List the replacement rules in the following order:

a. Group all those with the same $\langle\alpha_i\rangle$ on the left into one group.
b. Within each group, list first the *ad hoc replacements*, i.e. those with only terminal symbols on the right.
c. Place the group for α_1 first.
d. Place next the groups for the α_i appearing at the right of the α_1 rules in the order of first appearance in the α_1 group.
e. Beginning with the group following α_1, place those groups not yet placed in order of first appearance as in d, and iterate this process.
f. Because the number of replacement rules is finite, this procedure must end; if any α_i have not been ordered by this procedure, the language is not *connected*; i.e. some α_i are not needed in derivations of words of α_1.

If the system is conected, we test it for generative admissibility as follows.

a. In a first pass through the ordered system of replacement producers, tag all the ad hoc rules (which will yield branches leading to endpoints in the flow graph) as belonging to *height one*.
b. In a second pass through the set of rules, tag all those as *height two* in which only symbols from the alphabet of terminal characters and syntactic type symbols appearing at the left of height one productions appear on the right.
c. Iterating this process, when the producers (replacement rules) are grouped by height up through height n, label as height $(n+1)$-productions those which have not yet been labelled and in which none but left-side symbols of already tagged productions up through height n appear on the right.
d. This process will also come to a conclusion, because of the finite number of rules, when we can find no rules for the new height. When this happens, if all rules have been tagged with a height the system is generatively admissible; otherwise, it is not.

The first ordering has tested the α_1-system for connectivity, and the second has tested for generative admissibility; when both tests have been passed, the first ordering yields a specification of the $\vee C$-flow-graph as follows.

a. Label the entering node α_1.
b. If there is more than one rule for α_1, place an $\vee$ at that node, subscripted by the number of such rules; except for the priority on ad hoc rules, the ordering of branches—one for each rule—is considered unimportant (different orderings will yield "equivalent" flow graphs). Place one branch for each rule in left-to-right order and fill the nodes reached by terminal strings for the height-one rules or label them for the others, as the case may be.
c. If thre is only one rule for α_1, place a C at the node subscripted by the number of concatenands in the rule (each maximal substring of terminal symbols is considered to be one concatenand). Fill the nodes at the branch ends or label them as in step b. Here the left-to-right order *is* important.
d. Iterate steps b and c for all labelled nodes reached; the only exception to rules b and c is that a label which has already appeared will be followed by a single branch arrow back to its previous appearance.

5. *The Infinite Periodic Tree of an $\vee C$-Flow-Graph and the Derivation Subtrees*

As in [3], the $\vee C$-flow-graph may be specified in a prefix language (generalized—see Gorn [2]) as follows. For any labelled node α_i, the $\vee C$-flow-graph has a maximal subtree whose root is labelled α_i and whose nodes which are not tree endpoints are unlabelled; the endpoints of this maximal subtree will therefore be either endpoints or labelled nodes of the graph. Each of these trees may be specified in some tree-naming language to yield a set of equations, one for each labelled node, which recursively specifies the graph. For example, the first graph in Figure 1, yields, in a generalized prefix language, the single equation

$$\alpha_1 = \vee_4(a)(b)C_2(a)\alpha_1C_2\alpha_1(b).$$

The second yields three equations:

$$\alpha_1 = \vee_3\alpha_2\alpha_3C_2\alpha_2\alpha_3\,, \qquad \alpha_2 = \vee_2(a)C_2(a)\alpha_2\,, \qquad \alpha_3 = \vee_2(b)C_2(b)\alpha_3\,.$$

These correspond, in prefix form, exactly to the polynomial equations of Schützenberger and Chomsky. They have the advantage, however, of permitting us, by use of standard prefix language processors, to generate all derivation trees. This is done by deriving successive *level-n approximating finite trees* of the desired infinite periodic tree as follows:

a. $\alpha_1^{(0)}$ is simply the given designation of α_1.

b. $\alpha_1^{(n+1)}$ is obtained from $\alpha_1^{(n)}$ by making a single pass, say from left to right, through $\alpha_1^{(n)}$ and replacing each occurrence of an α_i by the right-hand side of the equation specifying α_i.

Thus, for the first production system of Figure 1 we have

$$\alpha_1^{(0)} = \vee_4(a)(b)C_2(a)\alpha_1C_2\alpha_1(b),$$

$$\alpha_1^{(1)} = \vee_4(a)(b)C_2(a)\vee_4(a)(b)C_2(a)\alpha_1C_2\alpha_1(b)C_2\vee_4(a)(b)C_2(a)\alpha_1C_2\alpha_1(b)(b),$$

etc.,

while in the second system the approximating finite trees are:

$$\alpha_1^{(0)} = \vee_3\alpha_2\alpha_3C_2\alpha_2\alpha_3,$$

$$\alpha_1^{(1)} = \vee_3\vee_2(a)C_2(a)\alpha_2\vee_2(b)C_2(b)\alpha_3C_2\vee_2(a)C_2(a)\alpha_2\vee_2(b)C_2(b)\alpha_3,$$

etc.

$\alpha_1^{(1)}$ in each case is shown in two-dimensional representation in Figures 2 and 3.

Figure. 4 illustrates a generatively inadmissible system.

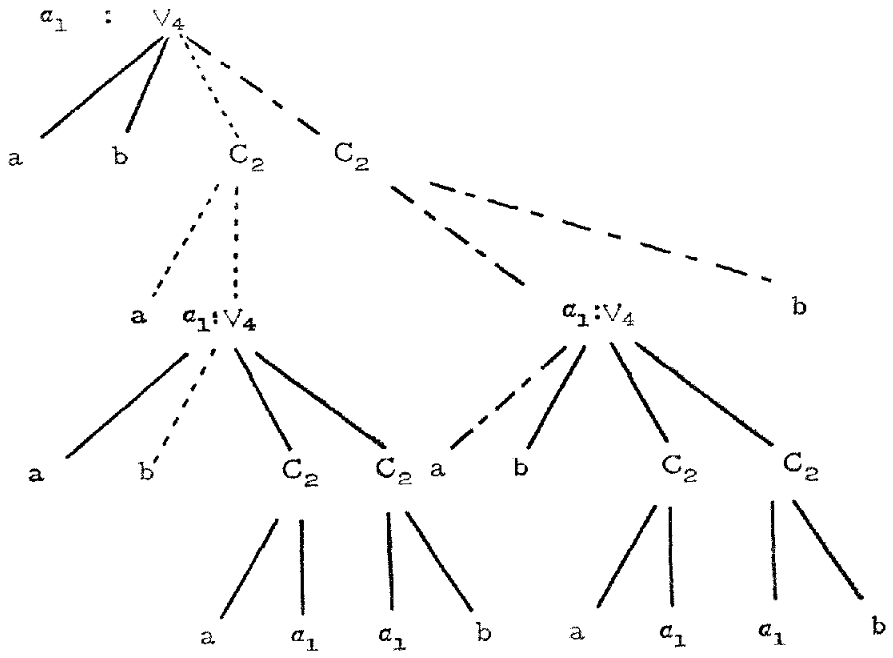


FIG. 2. $\alpha_1^{(1)}$ for system 1, and two derivation subtrees for *ab*

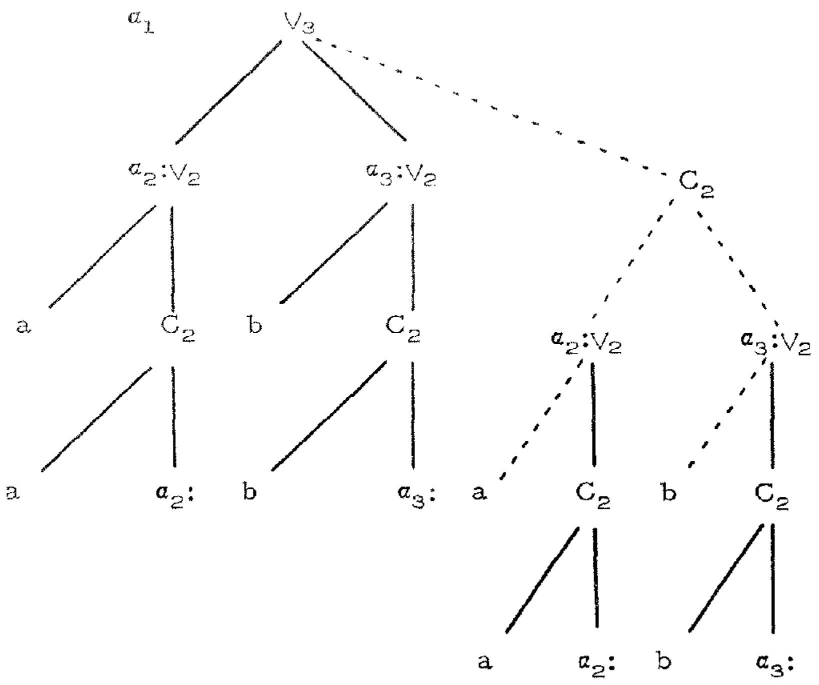
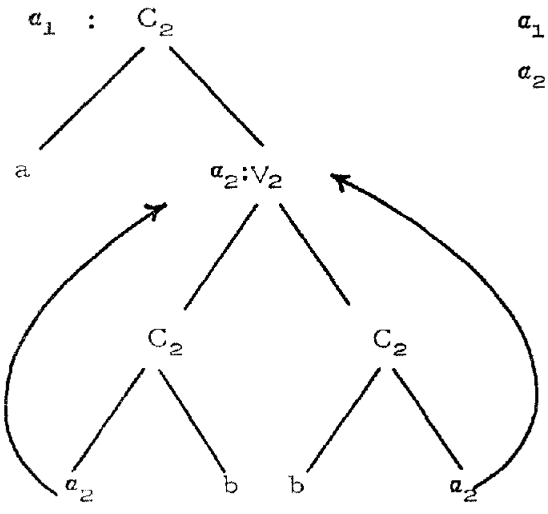


FIG. 3. $\alpha_1^{(1)}$ for system 2, and the derivation subtree for *ab*



$$\alpha_1 = C_2(a)\alpha_2$$
$$\alpha_2 = V_2C_2\alpha_2(b)C_2(b)\alpha_2$$

FIG. 4. A generatively inadmissible system

Many flow graphs will expand into the same infinite periodic tree; for example, completing each $\alpha_1^{(i)}$ by arrows back from the labelled endpoints to previous nodes with the same label would give us such equivalent graphs. The fact that we might have several different infinite periodic trees for what we might choose to call equivalent languages is, however, a consequence of the unsolvability of the *equivalence problem*.

By a (finite) derivation subtree of level n of an infinite periodic tree we mean the following.

a. It is a subtree of $\alpha_1^{(n)}$ but not of $\alpha_1^{(n-1)}$.

b. Its root is the root of $\alpha_1^{(n)}$.

c. At each $\vee$-node one and only one exiting branch of $\alpha_1^{(n)}$ appears.

d. At each C-node every exiting branch appears.

e. Every endpoint is an unlabelled endpoint of $\alpha_1^{(n)}$; i.e. all endpoints contain strings from the terminal alphabet only.

An $\vee C$-flow-graph is generatively admissible if and only if every *incomplete* derivation tree at level n is part of a complete derivation tree at some higher level. If in reading the derivation subtrees from an $\alpha_1^{(n)}$ we place the number of the selected branch of an OR-node (reading from left to right) as a superscript of $\vee$, then in prefix form we can name all derivation trees. For example, in Figure 2 we have at level 0: $\vee_4^1 a$ and $\vee_4^2 b$ are the derivation trees for the words a and b, respectively, and at level 1: $\vee_4^3 C_2(a)\vee_4^1(a)$, $\vee_4^3 C_2(a)\vee_4^2(b)$, $\vee_4^4 C_2\vee_4^1(a)(b)$, and $\vee_4^4 C_2\vee_4^2(b)(b)$ are derivation trees of the words aa, ab, ab, and bb, respectively; this indicates clearly the ambiguous derivation of ab.

Similarly, in Figure 3, there are no derivation trees at level 0, and the following at level 1: $\vee_3^1\vee_2^1(a)$, $\vee_3^2\vee_2^1(b)$, $\vee_3^3 C_2\vee_2^1(a)\vee_2^1(b)$ for the words a, b, and ab, respectively.

6. *Derivation Operators and the "Free Language" of an $\vee C$-Flow-Graph*

If we consider all the derivation trees of an $\vee C$-flow-graph and mark all the end points 0 instead of by the appropriate strings of terminal characters, we can interpret the results as *operators* which will automatically produce words from the production system as ordered by the connectivity test of Section 4. Thus the level-0 operators in Figure 2 are $\vee_4^1 0$ and $\vee_4^2 0$; the level-1 operators in Figure 2 are $\vee_4^3 C_2 0 \vee_4^1 0$, $\vee_4^3 C_2 0 \vee_4^2 0$, $\vee_4^4 C_2 \vee_4^1 00$ and $\vee_4^4 C_2 \vee_4^2 00$; the level-1 operators in Figure 3 are $\vee_3^1\vee_2^1 0$, $\vee_3^2\vee_2^1 0$ and $\vee_3^3 C_2\vee_2^1 0\vee_2^1 0$; etc. In each case a unique word is determined because of the priority rules accepted at the OR-nodes.

The derivation operators themselves of a particular ordered production system form a context-free language closely related to the one they are applied to.

$\delta\alpha_1 \rightarrow \vee_4^1 0$
$\delta\alpha_1 \rightarrow \vee_4^2 0$
$\delta\alpha_1 \rightarrow \vee_4^3 C_2 0 \delta\alpha_1$
$\delta\alpha_1 \rightarrow \vee_4^4 C_2 \delta\alpha_1 0$

(a) The derivation language for Figure 2

$\delta\alpha_1 \rightarrow \vee_3^1 \delta\alpha_2$
$\delta\alpha_1 \rightarrow \vee_3^2 \delta\alpha_3$
$\delta\alpha_1 \rightarrow \vee_3^3 C_2 \delta\alpha_2 \delta\alpha_3$
$\delta\alpha_2 \rightarrow \vee_2^1 0$
$\delta\alpha_2 \rightarrow \vee_2^2 C_2 0 \delta\alpha_2$
$\delta\alpha_3 \rightarrow \vee_2^1 0$
$\delta\alpha_3 \rightarrow \vee_2^2 C_2 0 \delta\alpha_3$

(b) The derivation language for Figure 3

FIG. 5

For example, all derivation operators for the language of Figures 2 and 3 may be generated respectively by the two production systems in Figure 5.

It is clear that the derivation language can be specified from the ordered productions of a context-free language by a mechanically produced production system, with a one-to-one correspondence between the productions of the two. Equivalent $\bigvee C$-flow-graphs will yield equivalent derivation languages because their flow graphs have identical structure. One has only to replace in the $\bigvee C$-flow-graph of the language:

(a) each label α_i by $\delta\alpha_i$,

(b) the jth node emanating from an OR-node by $\bigvee_n{}^j 0$ if it is a terminal point, and

(c) each terminal point branching from a C-node by 0.

Thus any ordinal generator of the derivation language simultaneously provides an ordinal generator of the original language—permitting repeats for the ambiguous words—by allowing each operator, as it is produced, to make the reverse substitutions to a, b and c. In the case of c, the ordering of the 0's determines the ordering of the terminal strings in the corresponding concatenation rule.

The derivation language of an ordered production system is unambiguous. If we changed the base (terminal) alphabet of the given system by marking the endpoints of the infinite tree $x_1, x_2, x_3, \cdots$ without repeats, we would obtain the infinite tree of an unambiguous context-free language isomorphic with the derivation language. This would be a *free language* with an infinite alphabet for the $\bigvee C$-flow-graph. The derivation language is already practically such a free language with only a finite alphabet which is a generalized prefix language, and as such the derivation language is generatively unambiguous and uniquely deconcatenable. Its processors are context-dependent. The complete operator language is therefore universal for all CF-languages, a generalized prefix language, but not context-free.

Sometimes a context-free free language exists, for a given language, which also uses only a finite alphabet.

Thus a free language for Figure 2 is given by the production system:

$$\bar{\alpha}_1 \rightarrow x_1, \quad \bar{\alpha}_1 \rightarrow x_2, \quad \bar{\alpha}_1 \rightarrow x_3\bar{\alpha}_1, \quad \text{and} \quad \bar{\alpha}_1 \rightarrow \bar{\alpha}_1 x_4 .$$

Thus, any generator, including an ordinal generator, of the original language (with possible repeats for ambiguities) is obtained from a corresponding generator of the free language by interrupting the generator of the free language—whenever a word is produced—and substituting for the x_i (an example in which this cannot be done is $\langle\alpha_1\rangle ::= a \mid \langle\alpha_1\rangle\langle\alpha_1\rangle$).

For example, in the language of Figure 2 the substitutions would be:[3]

$$a \rightarrow x_1, \quad b \rightarrow x_2, \quad a \rightarrow x_3, \quad \text{and } b \rightarrow x_4 .$$

[3] Note that the arrows here are not production symbols; they are obligatory "objective" substitution commands; i.e. it is the "objects" at the beginning of the arrow which are substituted, not the contents of storage cells which they designate; similarly, at the right of the arrows, objects are designated, not storage locations.

It is now clear that we can use the derivation operators as coefficients in the Schützenberger-Chomsky formal power series, instead of natural numbers. For example, in our first example we could write:

$$\begin{aligned}
\alpha_1 &= (\vee_4^1 0)a + (\vee_4^2 0)b + (\vee_4^3 C_2 00)a\alpha_1 + (\vee_4^4 C_2 00)\alpha_1 b \\
&= (\vee_4^1 0)a + (\vee_4^2 0)b \\
&\quad + (\vee_4^3 C_2 00)a[(\vee_4^1 0)a + (\vee_4^2 0)b + (\vee_4^3 C_2 00)a\alpha_1 + (\vee_4^4 C_2 00)\alpha_1 b] \\
&\quad + (\vee_4^4 C_2 00)[(\vee_4^1 0)a + (\vee_4^2 0)b + (\vee_4^3 C_2 00)a\alpha_1 + (\vee_4^4 C_2 00)\alpha_1 b]b \\
&= (\vee_4^1 0)a + (\vee_4^2 0)b + (\vee_4^3 C_2 0 \vee_4^1 0)aa \\
&\quad + [(\vee_4^3 C_2 0 \vee_4^2 0) + (\vee_4^4 C_2 \vee_4^1 00)]ab + (\vee_4^4 C_2 \vee_4^2 00)bb \\
&\quad + (\vee_4^3 C_2 0 \vee_4^3 C_2 00)aa\alpha_1 + [(\vee_4^3 C_2 0 \vee_4^4 C_2 00) + (\vee_4^4 C_2 \vee_4^3 C_2 000)]a\alpha_1 b \\
&\quad + (\vee_4^4 C_2 \vee_4^4 C_2 000)\alpha_1 bb = \text{etc.}
\end{aligned}$$

7. *The Derivation Generator and the Limited Ambiguity Detector*

Let us now describe a four-tape device which uses only the syntactic information in the production system and which generates derivations and words, doing ambiguity detection by comparisons with previously generated words.

Tapes 1 and 2 will be used alternately to read incompletely produced derivations and words at level n (i.e. those still containing α_i as components) and to write the same type at level $n+1$, respectively. Tape 3 will contain in successive blocks the derivations and words of levels 0, 1, 2, $\cdots$, n. The *complete* words will be written on it and will be compared with the previous words read off it to produce the inscriptions on the output tape, tape 4. We would expect to see on the output tape, for example in the language of Figure 2:

level 0, O.K.; level 1, O.K.; level 2, ab: $\vee_4^3 C_2 0 \vee_4^2 0$, $\vee_4^4 C_2 \vee_4^1 00$;

level 3, etc.

It is clear that phase 1 (described above), which takes the given production system, checks it for connectivity and for generative admissibility and which orders the productions, can easily be mechanized. We therefore assume that the properly ordered productions are in the internal storage of the machine.

We begin by placing α_1 on tape 1 and by reading out the results of the ad hoc productions from the derivation language and the object language onto tape 3 and the results of the other productions onto tape 2 in a single pass through the pair of production systems. Beginning with the second derivation and word pair on tape 3, we compare each new word with all preceding words. If the whole level is completed without a repeated word, we print out the indication that that level has passed the test. Otherwise, for each ambiguous word we print out the word, its derivation and the first derivation on tape 3 which produced the same word.

Now we iterate this process, alternately switching the roles of tapes 1 and 2. When one of them has on it, at the end of level n, the set of all incomplete derivations and words from level n, we make one pass through it replacing the first α_i-symbol in each word and its derivation by each of the replacements for α_i and the corresponding $\delta\alpha_i$ appearing in the production systems, after which we move a marker up to the next α_i-symbol in that word; we also set a flip-flop bit to 1 as soon as a production is used which is not ad hoc (i.e. has a new α_i in it). When a word has been completely scanned whose flip-flop is still 0, we place it and its derivation on tape 3 and commence a comparison scan with tape 3 for possible items in the output tape. When a word has been completely scanned whose flip-flop is at 1, we mark a second flip-flop bit with 1 to so indicate. In any event each substitution in the word of the tape, when the result is incomplete, is placed with its marker and two flip-flop bits on the other of tapes 1 and 2. When this first tape is complete and every word of the second of the pair has its marker appearing up front (with the two flip-flops reset to 0), the level $n+1$ is finished and the appropriate indication is written on the output tape. If not all words have been completely passed through (a flip-flop in the internal memory can tell us this), we erase the first tape and switch the roles of the two tapes for the next pass of level $n+1$.

This completes the description of the processor.

If for each n we know a good lower limit $p_n < n$ such that no word of level less than p_n can be ambiguously derived at level greater than $n-1$, the comparison phase through tape 3 can be made considerably more efficient.

In general, for each n we may have a lower bound p_n and an upper bound q_n, where $p_n \leqq n \leqq q_n$, such that no word produced at level less than p_n or greater than q_n can be identical with a word produced at level n. Such bounds are not intrinsic properties of the language, depending, as they do, on the particular production system. But there are indications that their growth properties as functions of n are intrinsic properties of the language.

Such bounds can, as we have remarked, make the detector more efficient.

8. *Conclusion*

It seems clear that the detector just described is fairly efficient for the general context-free language. It certainly is not very efficient for the example of Figure 2, which is equivalent to a finite state language; for in such a case it is easier to generate the words and seek all derivations of the words as they are produced (see Gorn [3] where a two-tape generator is described for the finite state case).

However, it is hard to see how this detector could be improved, for example, for the language specified by:

$$\langle\alpha_0\rangle ::= \langle\alpha_1\rangle | \langle\alpha_2\rangle | \langle\alpha_3\rangle, \quad \langle\alpha_1\rangle ::= 1 \mid \langle\alpha_1\rangle\langle\alpha_2\rangle\langle\alpha_3\rangle,$$
$$\langle\alpha_2\rangle ::= 2 \mid \langle\alpha_2\rangle\langle\alpha_3\rangle\langle\alpha_1\rangle, \quad \langle\alpha_3\rangle ::= 3 \mid \langle\alpha_3\rangle\langle\alpha_1\rangle\langle\alpha_2\rangle$$

In this language, the first word of α_1 which is ambiguous is 123123123 with derivations at level 5:

$${\vee_3}^1{\vee_2}^2C_3{\vee_2}^1 0{\vee_2}^1 0{\vee_2}^2C_3{\vee_2}^1 0{\vee_2}^1 0{\vee_2}^2C_3{\vee_2}^1 0{\vee_2}^1 0{\vee_2}^2C_3{\vee_2}^1 0{\vee_2}^1 0{\vee_2}^1 0 \quad \text{and}$$

$${\vee_3}^1{\vee_2}^2C_3{\vee_2}^1 0{\vee_2}^2C_3{\vee_2}^1 0{\vee_2}^1 0{\vee_2}^2C_3{\vee_2}^1 0{\vee_2}^1 0{\vee_2}^2C_3{\vee_2}^1 0{\vee_2}^1 0{\vee_2}^1 0{\vee_2}^1 0.$$

(There is a third derivation of this word, also at level 5.)

Even for finite state languages it is easy, for any number n, to exhibit one in which the first ambiguity appears between level 0 and level n. The following is an example: $\langle\alpha_1\rangle ::= ab^n | \, a \, | \langle\alpha_1\rangle b$.

Designers of mechanical languages for use in programming would be interested either in methods which would assure unambiguity or in limited ambiguity detectors. Either would permit them to certify that their candidate can be made independent of the processors dealing with them. Clearly, the detector here described—no matter how efficient it is made—cannot be considered practical. The lesson seems to be that such mechanical "certifications" should make essential use of semantic and pragmatic properties, or should at least transcend the context-free conditions. Parikh [5] shows the existence of *essentially ambiguous* context-free languages; any context-free specification of them must be generatively ambiguous.

REFERENCES

1. Bar-Hillel, Y., Perles, M., and Shamir, E. On formal properties of simple phrase structure grammars. *Zeit. Phonetik, Sprachwiss. Kommunik. 14*, 2 (1961), 143–172.
2. Gorn, S. An axiomatic approach to prefix languages. Paper, Symp. Symbolic Languages in Data Process., Mar. 1962, Rome, Italy [Gordon & Breach].
3. Gorn, S. Processors for infinite codes of the Shannon-Fano type. Paper, Symp. Math. Theory of Automata, (Apr. 1962), Polytech. Inst., Brooklyn.
4. Schützenberger, M. P., and Chomsky, N. The algebraic theory of context-free languages. In *Studies in Logic*, North Holland Publ. Co., Amsterdam; in press.
5. Parikh, R. J. Language generating devices. Quart. Prog. Rep. no. 60, Res. Lab. Electronics, MIT, Jan. 1961, 199–212.
6. Kantor, D. G. On the ambiguity problem of Backus systems. *JACM 9*, 4 (Oct. 1962), 477–479.
7. Floyd, R. W. On ambiguity in phrase structure languages. *Comm. ACM 5*, 10 (Oct. 1962), 526, 534.