MAR 16 1967

RANDOM GENERATION OF SENTENCES FROM CONTEXT
FREE AND CONTEXT SENSITIVE PHRASE
STRUCTURE GRAMMARS

MASTER

by

Martin Minow

October 1, 1966

DEPARTMENT OF COMPUTER SCIENCE · UNIVERSITY OF ILLINOIS · URBANA, ILLINOIS

PATENT REVIEW APPROVES RELEASE. PROCEDURES
COVERING PATENT REVIEW AND RELEASE ARE ON
FILE IN RECORDING SECTION.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

ERRATA SHEET

Report No. 215

RANDOM GENERATION OF SENTENCES FROM CONTEXT FREE AND CONTEXT SENSITIVE PHRASE STRUCTURE GRAMMARS

by

Martin Minow

<u>Page</u>	<u>Line</u>	<u>Correction</u>
2	7	Change "report" to " <i>repeat</i> "
7		Add braces around the rule between lines 6 and 7.
8	18	Change A to <i>X</i>
15	19	Should read: "theory (not allowing for complex symbols and, in the expansion program, for sentences embedded more"
17	last line	Should read: "least one family of computers, the Burrows B5000 series."
23	4	Change dash to arrow under Linguistic Usage
24	last line	Should read: $NP(I) = D(I) + ADJS + NOUN(I)$
29	4	Should read: "identical to $X(J, I)$."
31	8	Change 1965 to 1964 after Iverson.
36	11-12	Change arrows to equal signs
43	last line	Should read: "The pattern says that there must be an A which is first in the tree (see"
49	3	Should read: (V) ARD 1
52	Sect. 4.5 Line 8	Delete "and last" and change rules to rule
53		Add before first paragraph the following: "The asterisk appearing on all lines signals the beginning of the rule. There must be two asterisks in the flag section to define a new cycle."
55	2-3	Delete: "and where in the tree it failed."
56	12	Change 1966 to 1965 after Hutchison.

Report No. 215

RANDOM GENERATION OF SENTENCES FROM CONTEXT FREE
AND CONTEXT SENSITIVE PHRASE
STRUCTURE GRAMMARS*

by

Martin Minow

RELEASED FOR ANNOUNCEMENT
IN NUCLEAR SCIENCE ABSTRACTS

October 1, 1966

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

*This work was submitted in partial fulfillment of the Master of Arts degree in Descriptive Linguistics and supported in part by Atomic Energy Commission Contract AT(11-1)-1469 under the direction of Professor C. W. Gear.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

TABLE OF CONTENTS

	Page
1. INTRODUCTION	1
1.1 Forms of Grammar	1
1.2 Simplicity Assumption.	3
1.3 Linguistic Assumptions	5
2. THE CONTEXT-FREE GENERATOR	17
2.1 Form of the Grammar.	20
2.2 Program Conventions.	27
2.3 Recursion Marking.	31
2.4 Capacity of the CF Grammar	32
3. CONTEXT-SENSITIVE PROGRAMS	35
3.1 Representation of Trees.	39
3.2 The Initial Symbol	40
3.3 Use of the Programs.	40
4. TRANSFORMATION PROGRAM	43
4.1 Pattern.	43
4.2 Restrictions	46
4.3 Transformation Operations.	47
4.4 Overall Action of the Program.	49
4.5 Cycling.	52
4.6 Rule Execution	53
5. SOME CONCLUSIONS AND DISCUSSION.	56
APPENDIX 1. The Context Free Generator.	57
APPENDIX 2. Notes on Implementation	60
APPENDIX 3. CS and Transformation Programs.	66
REFERENCES	67

ACKNOWLEDGEMENT

I wish to thank Professor Arnold Zwicky for his assistance in performing this research. All errors are the responsibility of the author.

1. INTRODUCTION

This paper describes a set of computer programs which generate sentences from context-free (CF) and context-sensitive (CS) phrase-structure grammars. The paper is divided into three main sections: introductory comments on the linguistic assumptions reflected in the CS programs, descriptions of the overall structure of the programs and their use, and a series of appendices giving more detailed information. Future publication of flow diagrams, program listings, and sample output is planned.

The discussion which follows deals only with the CS and transformation programs. The CF program will be dealt with separately as it is completely independent of the CS programs and models a different type of grammar.

1.1 Forms of Grammar

The particular form of grammar implemented here is that proposed by Chomsky in several publications (especially 1957, 1963, 1964, and 1965). This paper also owes a deep debt to similar work done by the MITRE Corporation and printed in several internal documents, notably the English Preprocessor Manual (1965)* and as reported by Zwicky, et al. (1965).

These programs ignore several important aspects of Chomsky's work, the most obvious being their failure to offer mechanisms for the proper description of complex symbols as discussed in Chomsky (1965) and their inability to allow coordinate constructions to be correctly described.

*The MITRE Corporation has granted permission for the reprinting of portions of the English grammar developed there. Work on the MITRE grammar was supported by Contract No. AF19(628)-2390.

The latter difficulty arises from the fact that a coordinate sentence requires a tree structure of the form

(where S is "sentence"). However, the rules usable by the program in this paper will always insert excess structure. What is needed, and supplied by the MITRE grammar, is a rule schema such as

$$S \rightarrow \# \ S \ (and \ S)^* \ \#$$

where " $(x)^*$ " has the meaning "repeat x zero or more times." It is not difficult to implement this in the program. However, it should be noted that this schema cannot be formalized within the restrictions of Chomsky's definition of a context-sensitive grammar, as it is an abbreviation for an infinite number of rules.

All of these programs, the CF included, take as input a grammar and a set of random numbers:

They generate sentences, all of which are selected from the language defined by the grammar. The random number generator is used by the CF program to

select among subrules, by the CS program to allow certain portions of the grammar to be unordered, and by the transformation program to apply optional rules conditionally.

Since these programs are not intelligent, they produce sentences from the language actually defined rather than from that which the user intended to define. Often, these are not identical. By comparing output from these programs against the language under examination, it is hoped that the writing of grammars will be made somewhat easier in that (assuming that the programs are error-free) the linguist can easily see the effect of changing rules or their ordering on the sentences produced.

These programs define, in a very obvious sense, the well-formedness of a grammar. They permit certain constructions and forbid others. In generating sentences, they expand symbols and modify the output string in a certain manner. Except for the action of the random number generator, the process of expansion is completely determined by the grammar.

Every effort was made when writing these programs to make as few restrictions on the form of the grammar as possible. Those restrictions that are made are only those which are necessary for the program to interpret the rules. It is not suggested that the flexibility permitted is either necessary or desirable for grammars of natural or artificial languages.

1.2 Simplicity Assumptions

The mechanisms by which a sentence is generated may be modelled as follows

In this research, the rules for interpretation (ordered vs. unordered rules, etc.) were embedded in the production mechanism. Given a fuller understanding of language, however, it should be possible to define a production mechanism capable of interpreting the grammar in any of a number of ways. Assuming such separation is possible, we may discuss notions of simplicity as they affect the grammar, the rules for interpretation, and the production mechanism.

Simplicity in the production mechanism (computer program) is linguistically uninteresting as it is solely a matter of economy. We may say that a physically short, fast running program is to be preferred to one which is cumbersome and slow but, as long as both programs have a "black-box" equivalence, there is no theoretical reason to prefer one against the other.

Included in the simplicity notions for the grammar are the notational conventions available to the user. Chomsky (1965, p. 42) discusses the analysis of the verbal auxiliary, presenting the rule

Aux → Tense (Modal) (Perfect) (Progressive)

The parenthesis notation permitted by the grammar allows the rule to abbreviate several subrules with a great saving in the number of symbols. This saving is a demonstration of the linguistically significant generalizations inherent in the rule. However, this automatic expansion of abbreviated rules is independent of an algorithm for interpretation of the rules and is relevant only to the evaluation of different grammars.

Note that in the programs as written all such mechanisms for abbreviation are completely separate from the production procedure itself.

Justification for the particular rules of interpretation is both interesting and important to the study of language. Certain choices are motivated by the fact that they allow one to provide better descriptions of natural languages. It would be naive to assume that grammars for all natural languages will require only the types of analyses as are permitted here. What is permitted (or required or forbidden) by this theory will have to be modified as our understanding of language deepens.

1.3 Linguistic Assumptions

The most important linguistic assumption made is that rules are ordered. That is, in examining the tree against the grammar, the contexts will be examined in a particular order -- that in which they were punched. Thus, after examining subrule $\underline{m}$ of rule $\underline{n}$, the next to be examined is rule $\underline{n,m+1}$. If there is no such subrule, the program examines rule $\underline{n+1,1}$. If a particular rule contains sub-subrules^{*}, created by including optional

^{*}Using the conventions for writing rules for the CS program, the rule punched

$$\begin{aligned} X &= A / B \\ &= C / D \end{aligned}$$

has two subrules, each of which has only one sub-subrule. The rule

$$Y = [A / B (C)] / [E / F]$$

has only one subrule and the following six sub-subrules

$$\begin{aligned} &A / E \\ &A / F \\ &B / E \\ &B / F \\ &B C / E \\ &B C / F \end{aligned}$$

elements in the subrule cards, these will be examined in an order determined by the random number generator. Thus, rules and subrules are ordered and sub-subrules are not. Any time a subrule applies to a tree, the rule applies again to the tree just created. Only when all sub-subrules of a rule fail, or when the symbol being expanded is no longer a terminal node of the tree, is the next rule applied. The program goes through the grammar once only. When one cycle is finished, if the initial symbol SS is a terminal node of the tree (that is, if there was a successful application of a rule having SS as part of its output, such as $X \rightarrow Y(SS)$), then the entire grammar cycles to expand SS. The second cycle is special in that any sub-subrule having SS as part of its output will be automatically ignored. This was done in order to prevent the program's requiring an inordinate amount of time to expand one sentence. This second cycle is also different in that the tree found in the first pass through the grammar will not be examined during the second pass. Thus, except for the special treatment of the initial symbol, anything unexpanded at the end of the grammar is defined as terminal. In keeping with standard linguistic practice, the symbols appearing in the context of a sub-subrule are checked only against the symbols currently terminal.

Another assumption is that strings (i.e., trees) are examined left-to-right. Thus, if the rule

$$X \rightarrow Y / X \text{ -- } X$$

is applied to the tree

it would yield

rather than

or

There may be languages for which different conventions are needed; such languages will have to be handled by cleverly writing the grammar around the program's restrictions. The above rule must then be written

$X \rightarrow Y /$	$X \text{ -- } X$
	$Y \text{ -- } X$
	$X \text{ -- } Y$
	$X X \text{ -- } X$

Restrictions such as this are, of course, undesirable because they prevent the grammar acceptable to this program from correctly describing the structure of the language.

The treatment of the node S by the transformation program, to be discussed in detail in Chapter 4, is the most striking example of the program's assuming a particular conception of language. It is not at all obvious that a natural language can best be described by stating that transformations act on a tree in just the sequence defined here.

While the programs make certain assumptions about the form of rules, they do not mark as unusual or incorrect many things which may not be needed or desired in writing a grammar. For example:

$X = X X$ is legal, although the program will go into an infinite loop until stopped by the length of the string formed exceeding a system limit (5641 characters).

$X = A / B C$ is legal even though "context" is normally defined as including the symbol -- . The rule will be interpreted by the program as requiring that B and C be terminal symbols in the tree in the order given, but not that B or C be necessarily adjacent to X.

$X = D / A X C$ is also legal. Here, however, X in the context does not necessarily represent the symbol being expanded and the context may be nonlocal. When this rule is applied to the tree

it will yield

$X = Y / (A) \text{ -- } B$ is legal although there is no need here for optional elements in the context. Writing this will merely decrease efficiency as the program must create the sub-subrules

$Y / A \text{ -- } B$

$Y / \text{ -- } B$

and will not note that the second will succeed whenever the first does.

The transformation program makes several assumptions of linguistic interest. Following Chomsky (1965), we assume that the phrase structure component of a grammar generates a set of base trees which are interpreted by the semantic component of the language. If these are input to the transformation rules, the output is a set of surface trees which may be "pronounced" by the phonology.

The capabilities of the program in marking tree structure are also relevant. When scanning through the tree, the program will always skip over the least amount of tree structure possible. Thus if the pattern element being examined is

$(A \ B)$

and the tree at this point is

or

Then node B will always be chosen. The purpose of this convention is to allow the program to be able to continue scanning through the entire tree even though the scan may fail temporarily. Thus, in marking the tree

with the pattern

... (A B) (D)

(where "... " says that the match may be made anywhere in the tree and the lack of same means that the element being searched for must be in immediate right neighbor of the last element found) the match attempt will temporarily fail twice, giving

(where the numbers are node numbers assigned by the pattern).

Since it always scans for the least amount of tree possible, the program can continue scanning when a later test fails. The failure of a scan causes the program to restore the pattern to the last point at which the scan succeeded and restore the tree to its form after the successful scan. It then continues from the point where it ended before.

Once the tree has been marked, the program begins evaluating the restrictions and operations. In order to simplify the program, there is no requirement that these be written in any particular order. If a restriction fails or if an operation cannot find a needed node, the program returns to the element scan and attempts to re-mark the tree.

The restrictions which are allowed are minimal in that they permit only certain types of comparison of subtrees and tests for terminal or missing nodes. The program also permits a subtree to be compared against a literal string. The simple manner in which the program operates allows some rather strange restrictions to be formulated, for example:

- 1 DOM VERB is normal, meaning that the symbol VERB is included in the subtree which received node number 1.
- 1 DOM (VERB) states that VERB has neither daughters or sisters.
- 1 DOM (VERB(VTR)) states that VERB has no sisters and has only one daughter, VTR.

Since the program removes leading and trailing blanks, it is impossible to test whether a symbol has a sister except by using the negative tests. The test that VERB must have either a right or left sister or have a daughter would be

- 1 DOM VERB checks that VERB is dominated by node 1.
- 1 NDOM (VERB) fails if VERB has neither daughters or sisters.

It is also possible to confuse the transformation program by generating symbols containing node numbers or other symbols used internally by the program. If a rule such as

$X = '3'ABC$

applies, ABC will be generated with the node number 3 attached. Depending on the action of the scanning procedure, this ABC might be preserved until the operation is executed, and the result of the operation might be quite

difficult to predict. One possible use for such a rule might be to allow certain transformations to be signalled by generating (in the phrase structure component) certain node numbers that will never be assigned by the element scan. After the element scan has marked the tree, a restriction of the form

999 NEQU 0

would be written (where node number 999 was created by a phrase structure rule or by being inserted into the tree by another transformation). The transformation would then apply only if node number 999 was in the tree. By creating and deleting such symbols, extremely complicated sequencing could be effected.

The operations permitted by the program are interesting in that they make very few demands on its ability to examine the tree. All operations require the program to be able to locate a particular node. ERASE will remove this node and its daughters. Additionally, if this node was the only daughter of some node, ERASE will continue to remove structure. Thus, after ERASE 1, the tree

will become

The move operations require identification of two nodes, with the consequent movement of one of them into any one of four positions relative to the second. In the tree

we may move a node into the following places relative to B

where ls is the immediate left sister, rs is the immediate right sister, ld is the first (left-most) daughter, and rd is the last (right-most) daughter. More complex (in the sense of requiring more knowledge of the

tree) operations such as permute* were not implemented. The permitted

*Permute may be defined as "interchange two nodes and reattach them at the lowest node which dominates them both." Thus, permuting F and H in

will give

Permuting E and J in the original tree would give

as I would implement the operation (since the Polish representation of the string states that E follows C). However

is equally reasonable.

operations may move literal strings into the tree. The operation

(ABC) ALS 1

moves the symbol ABC into the tree as the left sister of node 1,

(X(Y '999'Z)) ARS 1

moves the tree

into the tree as node 1's right sister.

Few of the assumptions made by these programs about the form of language were conditioned by the mechanics of writing the program. A preference for scanning to the right and down in the tree is obvious and stems from the Polish representation chosen for trees and from the structure of SNØBØL. The desire to examine as little of the tree as possible when moving subtrees around is only incidentally motivated by the programmer's laziness. The crucial factor in not implementing such things as "add left aunt" or "add left sister skipping over the current left sister" is a requirement that the conceptual mechanism (theory of language) that these programs model be as simple as possible.

While these programs were written with a particular conception of language in mind, they make only one claim as to its validity: that it is possible to formulate a sufficiently detailed definition of that theory to permit its mechanization. While these programs do not model all of that theory (not allowing for complex symbols and for sentences embedded more than once, for example), they do model the greater part of that theory.

By noting where these programs are unsuccessful in permitting the formulation of the correct description of a language, it may be possible to discover where the theory itself must be extended.

2. THE CONTEXT-FREE GENERATOR

This program produces random sentences from an unordered context-free phrase structure grammar. The technique used is quite similar to that described by Yngve (1960, 1961). In order to describe it, some preliminary definitions will be made.

We first distinguish between terminal and nonterminal symbols. Terminal symbols are strings of characters which appear in the final output. By contrast, nonterminal symbols appear only in the grammar. "NOUN" would be an example of a nonterminal symbol, while "boy" would be terminal. In a rule of the form $X \rightarrow Y$, X will always be nonterminal while Y may be either terminal, nonterminal, or a string of symbols (which in turn may be either terminal or nonterminal). We will on occasion use string to refer to terminal symbols, while symbol will refer indifferently to terminals and nonterminals.

During production of a sentence, this program requires a vector for temporary storage of elements. Unlike the vectors used in most numeric computation, the maximum length of the vector cannot be determined in advance of the computation. Symbols and strings are entered and removed from this vector in what is called last-in-first-out order. Here, the program refers to only the top element of the vector. When the program removes this element, everything below "pops up" one space. When storing elements, whatever is put in the top of the vector pushes the remainder of the vector down. This type of storage device is commonly called a "push-down stack." It has been implemented as the main storage device of at least one family of the Burrows B5000 series.

As we do not have a wired-in stack at our disposal, the push-down device has been implemented in the program by assigning an index register as a pointer to a fixed-length vector. Assuming I to be the pointer, X the base address of the register, and MAX the maximum length of the stack, loading of data into the stack proceeds as follows:

The procedure requires a maximum of 4 cycles (or about 7μ sec.) on the IBM 7094, twice the time required to store the data. If the data is to overwrite the current entry in the top of the stack, the procedure would be entered at α . To restore data, we must test first for an empty stack (which may or may not be an error):

These procedures assume one-origin indexing -- the first entry into the stack being made at X_1 , the next at X_2 , etc.

Since the IBM 7094 is a fixed word-length computer, each word having a maximum of six characters, a way of handling strings of more than six characters is needed, for it is unreasonable to limit the length of strings so severely. The program uses a string reference word having the following characteristics:

where the numbers refer to bit positions, the left-most being the sign bit. When a string is read in, it is stored in a vector, six characters per word. The string reference word notes where the string begins in the vector and the length (in characters) of the string. Bit 1 is set to 1 to distinguish a string from a nonterminal symbol. A string can thus be manipulated by referring only to the string reference word. Only the output printing routine need refer to the string storage itself.

The overall production procedure is shown in Figure 1. As in the charts above, $\underline{I}$ will be used as the stack pointer, the stack will be named $\underline{X}$, the current top-of-stack, ($\underline{TOS}$) being $X_{\underline{I}}$. Note that rewriting elements will increase the stack length and outputting terminal strings will decrease the stack.

It is quite simple to modify the program to produce the tree structure (labelled bracketing) in addition to the terminal strings - either by changing the program to insert data into the stack as

"symbol("rewrite ")"

where the double quotation marks delimit terminal strings, or by changing the portion of the program which reads rules to prepare the necessary strings.

2.1 Form of the Grammar

A CF grammar may be characterized as follows (see Chomsky, 1964, for a more formal discussion):

- (1) Each rule is of the form

$$X \rightarrow Y$$

where X is a single symbol and Y is a nonvoid string of symbols. $\rightarrow$ has the meaning "rewrite X as Y ."

- (2) There is a specific symbol, the initial symbol, which defines the first rule in the grammar.
- (3) All symbols appearing on the left side of a rule are defined as nonterminal. Any symbol appearing on the right side of a rule which does not also appear as the left side of some rule is defined as a terminal symbol. The initial symbol is by definition nonterminal.

(4) A sentence is produced by the following procedure:

- a. Write the initial symbol.
- b. If there is a nonterminal symbol not yet expanded, expand it and continue at b.
- c. If there are no nonterminal symbols left, stop.

Except for stylistic variations, this grammar with its computer implementation is identical to Backus Normal Form as used in the description of ALGØL (Naur, 1960). The necessity of transferring the grammar to a computer-readable form requires a number of card-punching conventions, which are illustrated in the following table.

Of the above, only tags are peculiar to this grammar. They do not add any power to the grammatical form but rather make the writing of rules slightly easier. The use of tags to simplify the rules may be illustrated by noting that in English, the NOUN and VERB must agree in number. The validity of a grammar that produced

The boy run to school.

should be immediately suspect. Most rules in English are identical for singular or plural elements; but, unfortunately for the writer of a CF grammar, the particular plurality must be marked early in the production. In this grammar, we can write

$$\text{SENT} = \text{NP}(\text{I}) + \text{VP}(\text{I})$$

and let I range over singular or plural. This is identical to writing

$$\text{SENT} = \text{SSING} / \text{SPLUR}$$

$$\text{SSING} = \text{NPSING} + \text{VPSING}$$

$$\text{SPLUR} = \text{NPPLUR} + \text{VPPLUR}$$

Since rules rewriting NP and VP must transmit the plurality constraint to their daughters, etc., the advantages of the shorthand notation are

	<u>Linguistic Usage</u>	<u>Backus Normal Form</u>	<u>This Program</u>
Initial symbol	S	program	blank
Rewrite	-	::=	=
Concatenate	space	space	+ or context
Form of nonterminal symbol	X	<X>	X
Form of terminal symbol	x	x	\$X\$
Subrules	$\begin{Bmatrix} X & Y \\ A & B \end{Bmatrix}$	<X> <Y> <A> 	X + Y / A + B
Optional nonterminals	(X)	undefined	undefined
Ordering	usually ordered (note that this differs from the above definition)	unordered	unordered, except that the initial production must be physically first in the grammar deck
Tags	none	none	act to abbreviate rules

Figure 2. Notational Conversion.

obvious. This program allows for three independent tags, written as I, J, and K. The tags themselves are rewritten as single character nonterminal symbols as

$$I = S / P$$

There are times when the same constraint should be independently selected from the same values at different points in a grammar. For example, the object of an English sentence need not agree in number with the subject and predicate, thus we should write

$$VP(I) = VERB(I) + NP(J) \quad (1)$$

where J varies independently of I but over the same range. This program handles this by allowing one "tag covariation" rule of the form

$$(I,J) = S / P \quad (2)$$

With only three tags available, it is hardly necessary for more than one rule of this form. The covariation rule is equivalent to

$$I = S / P$$

$$J = S / P$$

Every time during a production when a tagged symbol is found on the right side of a rule and when that symbol has no match in the list of symbols appearing on the left sides of the rules, the tag equivalences are substituted. If the symbol is then found, the original tag is substituted for the equivalence when the rule is produced. For example, given a rule of the form.

$$NP(I) = D(I) + ADJS + NOUN(J) \quad (3)$$

When NP(J) in rule (1) is expanded, the search routine will not find any match. It will then substitute I for J and try again. Now, it will find rule (3). When (3) is entered into the stack, it will go in as though it were written

$$D(J) + ADJS + NOUN(J) \quad (4)$$

thus transmitting the correct constraint.

The small sample grammar in Figure 3 illustrates the points above. The first rule expands the initial symbol. It is one of the peculiarities of this program that the rule expanding the initial symbol must be the first rule in the deck; certain error checking procedures are thereby simplified. Rule 1 states that sentence is rewritten as a tagged Noun Phrase followed by a tagged Verb Phrase followed by the string consisting of the single character ".". Rule 2 is the tag equivalence rewrite discussed above. Note that Rule 3 does not exist in the printed text. The section of the program which handles tag equivalences produces

$$2* \quad I = S / P$$

$$3* \quad J = S / P$$

internally. Rule 4 expands NP, carrying the constraint over into the determiner and noun.

There is no rule whose left side is D(I). During production then, the tag is rewritten as S or P (randomly chosen before beginning each production). The particular character selected is appended to the symbol in place of the tag and the lookup routine is reentered. Rule 6 can rewrite plural determiner as either "the" or nothing -- i.e., plural determiners are optional. Since there is no provision for optionality in

- 1* = NP(I) + VP(I) + \$. \$
- 2* (I, J) = S / P
- 4* NP(I) = D(I) + ADJS + NOUN(I)
- 5* DS = \$ A\$ / \$ THE\$
- 6* DP = \$\$ / \$ THE\$
- 7* ADJS = ADJ / ADJS + ADJ
- 8* ADJ = \$ GOOD\$ / \$ BAD\$
- 9* NOUNS = \$ BOY\$ / \$ GIRL\$
- 10* NOUNP = NOUNS \$\$
- 11* VP(I) = VERB(I) + NP(J)
- 12* VERBS = \$ CHASES\$ / \$ CATCHES\$ / \$ KISSES\$
- 13* VERBP = \$ CHASE\$ / \$ CATCH\$ / \$ KISS\$

Figure 3. Sample Grammar.

this program, we are forced to produce a null terminal string. As noted in the above table, this is normally done in linguistic grammars by enclosing the optional symbols in parentheses. The plural noun phrase would be rewritten

(DP) ADJS NOUNP

As long as the rewriting of a symbol as \$\$ occurs immediately after it becomes optional and as long as at least one other subrule exists (i.e., the symbol does not actually disappear), it is possible to recover the linguistic form from that given here.

Rule 11 illustrates the use of tag equivalences. Sample output from this grammar is shown in Figure 4.

2.2 Program Conventions

The following should be observed when preparing input to this program:

1. Blanks are completely ignored except in strings.
2. Nonterminal symbols have a maximum of five meaningful characters.

Anything following will be ignored. Thus, ABCDEF and ABCDEXYZ are identically identified as ABCDE.

3. When tags are used, each tag, when it is rewritten, is added to the right of the symbol, possibly overwriting existing symbols. If the following tag rewriting rules exist

I = 1

J = 2

K = 4

- 1* Good bad good girls chase the good boys.
- 2* Bad girls chase the good good boys.
- 3* A bad good boy catches good girls.
- 4* The bad good girl kisses the good good good bad girl.
- 5* A good boy chases the bad boy.
- 6* A good good bad bad bad good boy kisses the good girls.
- 7* The bad bad girls kiss a good girl.
- 8* Good bad girls catch a good boy.
- 9* Bad boys chase a good good boy.
- 10* The bad good boy kisses a good girl.

Figure 4. Output from the Test CF Grammar.

ABCDE(I, J, K) will be searched for in the following order

ABCDE(I, J, K)

ABCD1(J, K)

ABCD2(I, K)

ABCD12(K)

ABCD4(I, J)

ABCD14(J)

ABCD24(I)

AB124

Note that tags are enclosed in parentheses and separated by commas. A string of tags is alphabetically ordered by the program. Thus X(I, J) is identical to (J, I).

4. When the character dot "." appears outside a string, it determines the end of the card, anything following being taken as comment, thus

X = Y + Z .REWRITE X

is identical to

X = Y + Z

If the first card in the rule deck contains a dot in column 1, making the entire card a comment, this card will be taken as a heading for the grammar (if the program is assembled within the monitor system it was designed for).

5. Rules may extend over as many cards as necessary. The program will scan until either column 72 or a dot is read, then continue on to the next card. Unless the first break character found on a card is an equality sign (parentheses and commas being part of the symbol), the scan for the

current rule will continue. Symbols and strings must be complete on one card. If a symbol is spread over more than one card, the program will consider the two parts as though they were concatenated (separated by +). If a string is not terminated by column 72, the program will print an error message.

6. The concatenation operator is optional next to other break characters. If desired, it must be punched after the right parenthesis ending a tag. Writing

$$X(I) \quad Y + Z$$

will cause Y to be ignored.

7. There are a few restrictions concerning strings:

- (a) As noted above, they must be complete on one card. If a string is too long, it must be broken into two or more concatenated pieces.
- (b) Any character, including other reserved characters, may appear within a string. However, if a dollar sign is desired within a string, it must be the only member of that string: \$\$\$ is legal while \$IT COST \$1.98\$ is not. The latter will result in an error message unless the error occurs twice on one card.
- (c) Since the dollar sign when punched in column 1 will cause the read procedure to assume that this card follows the grammar, the dollar sign beginning a string should start no earlier than column 2. The maximum length of a string is then 69 characters, because column 72 must contain a dollar sign to end the string.

8. All rules rewriting a specific symbol must appear together separated by slashes. If the same symbol is used as the left side of more than one rule, one of these will always be chosen. The specific rule

cannot be easily predicted because the choice is a complicated function of the number of names, the physical makeup of the rule deck, the sorting routine and the table lookup procedure. Choice among subrules is made by a random number generator. All subrules are equally probable.

2.3 Recursion Marking

After this program was written, the author attempted to generate random sentences (programs) from ALGØL, using a grammar published by Iverson (1965). The program produced 20-page-long programs of quite complicated structure which were impossible to examine or to check on a computer (by inputting them to an ALGØL compiler). The problem obviously is that ALGØL is highly recursive.* The difficulty was solved by writing a recursion marking procedure.

The algorithm used is similar to that presented by Trakhtenbrot (1963) and is shown in Figure 5. If requested, this procedure is executed after the grammar has been read but before the first sentence is generated. For a slight modification of Iverson's published grammar, containing 109 rules and 53 recursive subrules, the program required 3-15/60 seconds to mark recursion (compared with 1 second to read rules). It should be noted

*For the purposes of this discussion, we may define a recursive rule as one which has itself as one of the possible outputs. Thus, in Figure 4, Subrule 2 of Rule 7 is recursive. Another example would be a grammar printed as

$$1* = X$$

$$2* X = Y + Z$$

$$3* Z = X$$

The recursion marking procedure would flag rule 3 as recursive since when it analyses the output of this rule, it finds a symbol, X, which dominates Z in the tree of all possible productions. See Trakhtenbrot (1963) for details.

that in Figure 5 each stack entry consists of two 7094 words, the first (X1) containing the rule name, the other (X2) having a pointer to the rule and subrule currently being examined. I again is the stack pointer (here, though, stepped by 2).

When recursion marking is desired during sentence production, whenever the stack pointer is increased the pointer is checked against a stack limit. If this is exceeded, an appropriate comment is printed and the second part of the recursion process begins. Here, every subrule is checked for the presence of the recursion flag. If there is no flag, this subrule's output is entered into the stack as before. If the subrule is recursive, the current contents of the stack are examined. If the symbol is not already in the stack, its output is entered; if it is already in, all subrules of the rule being produced are examined. If one of the subrules is not flagged for recursion, that one is chosen. If, however, all subrules of the current rule are recursive, another flag is examined. If this flag is on, the recursion is considered nonfatal, the original choice being used; if it is off, the offender is deleted. The recursion finding process, its fatality, and the stack limit are all set by parameter cards for each particular grammar.

2.4 Capacity of the CF Grammar

This program operates in a finite device, simulating the potentially infinite-length stack with a fixed length vector. It is thus obvious that not all possible sentences that can be defined by any CF grammar can be generated. Additionally, some CF grammars cannot be accepted by the program because they exceed the current limits of 1000 subrules and 4000 symbols (nonterminal and string reference words). Strings are stored in whatever space remains; currently about 5000 locations are available.

After the rules are read, the unused space is compressed, and whatever remains becomes the push-down stack. After the ALGØL grammar was read, about 9000 locations were available. These limits seem sufficiently large that it is unlikely that any reasonable grammar will exceed them.

3. CONTEXT-SENSITIVE PROGRAMS

A context-sensitive phrase structure grammar differs from a context-free grammar only in that rules may have the form

$$A X B \rightarrow A Y B$$

where A and B are possibly null strings. The above has the interpretation "rewrite the single symbol X as the nonnull string of symbols Y in the context A -- B.". Thus rule would normally be written

$$A \rightarrow Y / A \text{ -- } B$$

In the tree

only DET, NOUN, VERB, and the NP dominated by VP can be rewritten or used to determine contexts. Note the following

$NP \rightarrow X / NP$	will succeed.
$NP \rightarrow X / \text{ -- } NP$	will fail.
$NP \rightarrow X / VERB \text{ -- }$	will succeed.
$NP \rightarrow X / VP \text{ -- }$	will fail, as VP is not terminal.
$NP \rightarrow X / \text{ -- } VERB$	will fail, as VERB precedes NP in the tree.
$VP \rightarrow X$	will fail, as VP is not terminal.
$NP \rightarrow X / NOUN \text{ -- }$	will fail, as NOUN is not immediately to the left of NP.

The CS generator consists of two programs. The first reads rules, expanding optional and choice elements and punching these. The second

reads the rules prepared by the first program and then reads trees for expansion.

The CS grammar accepted by these programs has the following form: each rule consists of an optional rule name or number enclosed in parentheses, the symbol to be expanded, an equality sign, and the expansion, optionally followed by a context punched after a slash. The program recognizes two types of subrules. The first type, prepared explicitly by the user, is recognized by a card beginning with an equality sign (without any preceding rule name or symbol). This means that what follows is a subrule of the last rule read. For example,

$$\begin{array}{l} X \rightarrow A / B \text{ -- } C \\ \rightarrow D \end{array}$$

is a rule with two subrules, the first expanding X as A in the context B -- C, the second expanding all other occurrences of X as D.

The second type of subrule, created by the program, will be called a sub-subrule. Sub-subrules allow one subrule either to have varying output or to succeed in different contexts. For example,

$$X \rightarrow \left\{ \begin{array}{l} \left\{ \begin{array}{l} A \quad B \\ C \end{array} \right\} \quad D \\ (E \quad F) \quad G \end{array} \right\} / \left\{ \begin{array}{l} \text{-- } Y \\ Z \end{array} \right\}$$

means that X is rewritten either as

$$A \quad B \quad D$$

or

$$C \quad D$$

or

G

or

E F G

in either of the contexts

-- Y

or

Z

Thus, we have the following eight sub-subrules:

A B D / -- Y

C D / -- Y

G / -- Y

E F G / -- Y

A B D / Z

C D / Z

G / Z

E F G / Z

While rules and subrules are expanded in numeric order, the sub-subrules will be examined in an unordered manner.

Although the two-dimensional form of writing rules produces an extremely readable grammar, it is a bit difficult to input to the computer. The above rule may be typed on one line as

$$X = [[A \ B \ / \ C] \ D \ / \ (E \ F) \ G] \ / \ [\ -- \ Y \ / \ Z]$$

where brackets take the place of the braces used in the two-dimensional representation and slashes within brackets separate choices. As before, optional material is enclosed within parentheses. Note that

$$(A \ ([B \ / \ C \ (D)]))$$

is legal, expanding to

A

A B

A C

A C D

Unfortunately, the card punch equipment available does not allow brackets to be punched directly. They must be punched '(' and ')' for left and right bracket respectively. The prime must precede the parenthesis in order to prevent ambiguities. The above rule will actually be punched

$X = '(' ('(A B / C ') D / (E F) G ') / '(' -- Y / Z ')$

In this program blanks are used to separate symbols. If we use the character ϕ to represent "blank", the string NO ϕ UN would be interpreted as two concatenated symbols. Blanks are optional before and after other break characters, any number of blanks being read as one. Note that the dash used in contexts is not a break character. Any number of minus signs will be read as one dash (so long as no blanks intrude). AB-C is one legal symbol; AB ϕ --- ϕ C is three. Cards with an asterisk in column 1 are treated as commentary. A single rule may spread over more than one card.

A formal definition of "rule" is given below. Terminal symbols are enclosed in double quotes.

$$\text{RULE} \rightarrow \text{SYMBOL "=" ELEMENT ("/" } \left\{ \begin{array}{l} \text{ELEMENT} \\ \text{MSTRING} \end{array} \right\})$$

$$\text{ELEMENT} \rightarrow \left\{ \begin{array}{l} \text{SYMBOL ("} \phi \text{" ELEMENT)} \\ \text{"(':" CSTRING ("/" ELEMENT "')"} \\ \text{"(" ELEMENT ")"} \end{array} \right\}$$

CSTRING $\rightarrow$ ELEMENT ("/" CSTRING)

MSTRING $\rightarrow$ "-" (MSTRING)

SYMBOL $\rightarrow$ any string of letters, numbers and minus signs

3.1 Representation of Trees

Since this program manipulates trees, they must have some sort of internal representation. Two possible forms suggest themselves.

LISP, a programming language specifically designed to handle trees, creates chained lists after the following examples (taken from Haines, 1965)

where represents the empty list. This is a very convenient and natural way to handle trees. Unfortunately, LISP was not available to the author. SNØBØL, a string manipulation language, was used instead. It was decided to represent trees in SNØBØL by using a fully parenthesized Polish notation. For example,

becomes

(S(NP(DET(the) NOUN(man)) VP(VERB(hit) NP(DET(the) NOUN(ball))))).

Blanks are meaningful; they separate elements dominated by the same node. In the preparation of trees for this program, redundant blanks may be inserted. Blanks should not go to the left of a left parenthesis without malice aforethought: ($\emptyset$ S(NP $\emptyset$)) is identical to (S(NP)), however, (S $\emptyset$ (NP)) is undefined.

3.2 The Initial Symbol

The initial symbol plays a unique role in these programs. Normally, the program executes the rules once (possibly cycling at individual rules). After the last rule has been applied, the output tree is checked to determine if the initial symbol is currently terminal in the tree. For any occurrence of the initial symbol, the entire grammar will cycle in order to expand this subsentence. The second cycles are unique in that (1) only the tree beginning with the inner SS will be expanded, the program ignoring the structure already found, and (2) any sub-subrule having SS as part of its output will be automatically deleted.

3.3 Use of the Programs

After a grammar is prepared, it is inputted to the CS rule reader, which will punch a deck of cards containing the internal form of the grammar. Each subrule will be expanded so that all parentheses and braces are removed (as discussed above). A series of lists will also be prepared containing such rule names as were mentioned and a list containing the number of sub-subrules for each rule/subrule. While this two-step process has many disadvantages, it has one interesting advantage: the grammar deck may be edited and only a subset inputted to the expansion program. By choosing only such sub-subrules as are needed, the user can prevent the program's generating much undesired optional structure. The ability to

change the rule numbering also may be seen as an advantage. If a rule is discovered to be in error, instead of recompiling the entire grammar, one need only expand the particular rule, using the numbering feature to correctly assign rule numbers. For example, if Rule 17 must be recompiled, the input might be

(17) $X = Y / Z$

The program can punch various cards; of interest here are the following two: (1) the card giving the symbol to be expanded by this rule and the number of subrules, punched as

$\text{RL}17(X.1)$

(where R stands for the record mark, punched as 0-2-8). The dot separates the symbol and the number of subrules. (2) Each sub-subrule would be punched as

$\text{RL}r\text{nbr}.\text{sr}.\text{ss}(\text{rewrite/context})$

or, using the above example

$\text{RL}17.1.1(X/)$

The rule (R) and left-hand side (L) cards may be added to the end of the grammar, where they will overwrite the original definition. If the number of subrules or the number of sub-subrules is changed, appropriate entries in the NAMES list must be changed. Entries are punched in this list as

$\text{r}\text{nbr}.\text{sr}.\text{ss}$

where rnbr and sr are known, the program searching for ss. To delete a

rule from the grammar deck, it is only necessary to remove the L_1 entry or to add a $\backslash L_1 \text{rnbr}(0)$ card to the end of the grammar. The last card of the grammar deck must be

~~\END\OFT\~~PUNCHING(X)

After reading the rule deck, the program begins to read and expand trees. Each tree card contains an asterisk in column 1, followed by an optional rule name or number at which to begin computation, followed by an optional tree enclosed in parentheses. If the tree must be punched on more than one card, successive cards should not have asterisks in column 1. The following are legal tree cards:

```
*
*17
*NPEXP (SS)
*(SS(S(NP VP))
*(SS(S
(NP VP))
```

The first is the "normal" expansion request, stating that the first rule is the first to be examined; the tree will be

(initial symbol)

where the initial symbol is that appearing on the left side of Rule 1.

Note that here rule names and rule numbers are not enclosed in parentheses.

The potential user should be warned that this program is rather slow when the amount of tree that it must examine becomes large.

4. TRANSFORMATION PROGRAM

Like the CS expansion program, the transformation program is divided into two parts--a rule reader and the transformation program itself. Transformation rules consist of a pattern, which labels nodes on a tree, followed by a series of tests and operations to be performed on the labelled nodes.

4.1 Pattern

Pattern cards are punched with an asterisk in column 1. If the pattern must spread over more than one card, each card must have an asterisk in the first column. The pattern consists of a set of optional flags followed by a series of pattern elements each enclosed in parentheses. The possible flags are as follows:

- C states that this rule is cyclical.
- ∅ states that this rule is optional.
- * is used to define the beginning of cycling group.
- G states that this rule is an embedding, or generalized, transformation. G may be program-generated.
- A states that this rule is a conjunction rule

The asterisk used for cycling groups is independent of that used to define the card as being a pattern card. These flags are all optional and unordered.

The pattern itself follows this example

* (A) (B C) ... (D E) ... S((F) (G O H))

The pattern says that there must be an A which first in the tree (see

Figure 6a for possible positions), immediately followed by either B or C (whichever is found first--see Figure 6b). B or C must be followed by D or E. Here, however, "... " implies that these may appear anywhere in the portion of the tree being examined after finding B or C. It should be noted that while the scan without "... " will not examine the daughters of the node just found, the scan for D or E will begin with the first daughter of B or C. The character S, when punched before a left parenthesis, signals that the transformation being read is an embedding transformation. It will force the program to add the character G to the flag string. The portion of the pattern between "S(" and its matching right parenthesis will mark structure within the inner S. If 0 (zero, not the letter Ø) appears as a node within an element, it indicates that this element is optional. Figure 6c shows a tree which successfully matches the above pattern. Note that the last node does not appear in the tree.

After successfully reading the pattern, the program will print the pattern and flags (including the G flag if produced). It will then number the nodes and print the numbering found. The above example will be printed

```

      PATTERN = G (A) (B C) ... (D E) ... '(F) (G 0 H)'
ELEMENT NUMBERS =   1   2       3       4       5

```

The primes signal the beginning and end of that part of the pattern which is to match the embedded S. The program is set up to allow only one set of elements to be dominated by S. Thus

```
* S((X) (Y)) S((Z))
```

is illegal, as is

```
* S((X) S(Y))
```

Figure 6a. Possible Positions for Node A.

Figure 6b. Possible Positions for Node B.

Figure 6c. A Marked Tree.

If only one node is to be dominated by S, the extra set of parentheses is not needed. Thus

* S(A)

is legal.

The restriction of the pattern as to the number of times the node S may be mentioned prevents the program from effectively handling conjunctions. It was discovered that the implementation of a program which successfully executed embedding and singular transformations would be a sufficiently difficult task. It would appear that in order to handle conjunctions it would be necessary to prevent the program from ignoring daughters of embedded sentences other than the one currently being transformed. While the A flag allows this type of scan, it has not been checked.

4.2 Restrictions

The restriction cards, as well as the transformation operations themselves, are punched one per card with blanks separating necessary elements. These cards must not have an asterisk in column 1. The possible restrictions are:

<u>n</u> DOM <u>x</u>	The node which received number <u>n</u> dominates <u>x</u> , where <u>x</u> is either a node (having at least one non-numeric character) or a literal node or subtree. As mentioned in Chapter 1, it is possible to require that node <u>n</u> be terminal by writing
-----------------------	--

n DØM 0

In Figure 6c, the following will succeed:

1 DØM (X Y)

2 ~~DOM~~ 3

4 ~~DOM~~ 0

3 ~~DOM~~ S

Node n may dominate other things along with x
n EQU x n and x are as discussed under ~~DOM~~. What n
 dominates must be exactly x.

n EQU 0

succeeds if node n is not in the tree as marked.

Note the following:

2 EQU (D(S(F)))

succeeds because the node numbers created by the
 pattern scan will be erased before any comparison
 takes place.

5 EQU 0

also succeeds.

~~NDOM~~ and ~~NEQU~~ are defined as the complements of ~~DOM~~ and EQU respectively.

Thus

1 ~~NEQU~~ 0

succeeds because there is a node 1 in the tree.

4.3 Transformation Operations

The transformation operations that are allowed are:

ERASE n Removes node n and whatever it dominates.

If the node is absent from the tree, a comment is
 generated and the tree is not affected.

ERASE 1

when applied to Figure 6c would yield Figure 6d.

If the operation

ERASE 0

is executed, the program will erase the entire tree without explicitly including it in the scan.

If the application of an ERASE instruction removes all daughters of a node, the node itself is erased. If applied to Figure 6c, ERASE 3 would yield the tree shown in Figure 6e. Note that the deletion of nodes without daughters continues, erasing W as well as C.

x ALS n Copy the subtree x into the tree as the immediate left sister of node n. If x is a node number which is missing from the tree, the program will comment and continue. If n is missing, the program will act as though the scan failed, reentering the scan routine to attempt to find another numbering. In addition to being a node number, x may be a literal subtree enclosed in parentheses. If the following is applied to Figure 6c, the result will be in Figure 6f.

1 ALS 4

(ABC) ALS 1

(P(Q R)) ALS 2

Note that node numbers will be removed from nodes named by x; however if x is a string, node numbers will not be removed (thus, the user can insert flags into the tree).

x ARS n This is identical to ALS except that x is added as the right sister.

x ALD n x is added as the left-most (first) daughter of n.

x ARD n x is added as the right-most (last) daughter of n.

(U) ALD 1

(V) ALD 1

will yield Figure 6g if applied to Figure 6c. If n has no daughters, ALD and ARD have the same effect.

These are all of the operations that are defined. In order to save user time, the following abbreviations are available:

n1 XCH n2 The subtree headed by n1 is exchanged for that headed by n2. It is compiled by the program as

n1 ALS n2

n2 ALS n1

ERASE n1

ERASE n2

n1 Mpq n2 requests a move operation. After the appropriate Apq, node n1 is erased. It compiles as

n1 Apq n2

ERASE n1

x SUB n Substitute. The subtree headed by x replaces that headed by node n. This is compiled as

x ALS n

ERASE n

4.4 Overall Action of the Program

After reading the transformation rule deck, the program begins reading trees. These are punched in the Polish notation already described. Tree cards should not have any asterisk or other identification character in column 1. The tree begins with a left parenthesis. The program assumes

Figure 6d. Operation of Erase 1.

Figure 6e. Operation of Erase 3.

Figure 6f. Tree after Several ALS Instructions.

Figure 6g. Tree After Adding Literal Elements.

that it ends with the matching right parenthesis. Trees may spread over more than one card. The program then numbers occurrences of the node S, beginning from the most embedded and continuing to the least embedded.

The tree

would be punched

(S(S(S S) S(A B)))

and then numbered

(S/3(S/1(S S) S/2(A B)))

The two S's on the left of the tree were not numbered because they are terminal.

4.5 Cycling

Before describing the action of the program for an individual rule, we will describe the use of the asterisk to mark cycling groups. There are times when a single rule should apply as many times as possible to a given sentence. These rules are marked with the letter C in the flag string. Other times, one may wish to apply a group of rules to a sentence until they all fail before continuing in the grammar. A series such as this is defined by punching an asterisk in the flag section of the first and last rules of such a group. Note that while a cyclical rule may appear within a cycling group, the groups themselves may not be nested. The last rule of the grammar will automatically end the last cycle. As an example of a series of rules which define cycling groups, note the following:

normal rule beginning a rule deck and a cycle.

cyclical rule

cyclical rule beginning a new rule cycle

normal rule within a cycle

cyclical rule within a rule cycle

end of cycle

beginning of another cycle

end of this second cycle and end of rule deck

The program operates independently for each cycle group. For each group, the rules are executed in order for the most deeply embedded S, then for the next most deeply embedded, etc. Finally, any structure above the least deeply embedded S is transformed. The program then passes to the next cycle, beginning again at the most deeply embedded S.

4.6 Rule Execution

When the program is about to begin a new rule, it checks the optionality flag. If there is a flag, the rule is accepted if a random number generator produces a certain value. Assuming that the rule actually will be executed, the program must define the amount of tree to be examined. Upon entering a new S-level, the tree is scanned to determine what S immediately dominates the current one. In the example above, both S/1 and S/2 are dominated by S/3. S/3 is not dominated by any S in the actual tree. The program, however, will cause it to be dominated by the special S/0. Dollar signs will be inserted into the tree to delimit the portion of the tree being scanned. Using the above example, the program will mark the tree

S/0(S/3\$(S/1(S S) S/2(A B))\$)

when transforming at the first and second S-level. At level 3 (when

examining the entire tree), the example will be marked

S/0\$S(S/3(S/1(S S) S/2(A B)))\$

If the pattern is not that of an embedding transformation, i.e. does not have the G flag, the program will act as though the pattern were punched

*G ... S(original pattern)

Whenever an embedding transformation begins operating on the inner S, the end of that S will be marked with another dollar sign. If a dollar sign is scanned, the program will immediately fail.

The pattern scan proceeds left to right, marking nodes as a successful match is developed. Whenever an element search succeeds, the current tree and pattern are saved in a push-down stack. The pattern saved is that at the beginning of the current element. The tree is positioned at the end of the current search but without having the number of the node just found inserted.

If the search is for an immediate match (i.e. does not have "... before the element), the program will skip over the daughters of the current node before beginning to search the next node. Whenever an S node (other than that for the embedded S) is scanned, if the flag string contains an A the scan continues into this sub-sentence; if the A is not present, the scan for an immediate match will fail, that for a skip match (with "...") will skip over this S and its daughters before continuing. If the embedded S is scanned, both procedures will fail.

When the scan fails, the program determines whether the element was optional. If so, the next element is attempted. If not, the program attempts to pop-up the stack in which the patterns are being saved.

Whenever the stack is emptied, a comment is printed and a new rule begun. The comment lists the first element which failed and where in the tree it failed. The failure procedure is also entered if any restriction fails or if an operation cannot find a needed node.

5. SOME CONCLUSIONS AND DISCUSSION

SNØBØL is an extremely powerful language. The IBM 7094 is an extremely inefficient device for performing character manipulation. The SNØBØL programs and what they seek to model are both extremely complicated. It must also be noted that the author is hardly experienced at writing efficient SNØBØL code. Consequently, the programs are slow for anything of interest.

The version of SNØBØL distributed by Bell Telephone Laboratories has facilities for implementing machine language functions in the SNØBØL programs. The implementation of SNØBØL on which these programs were run did not include these features. Therefore, such things as the random number generator had to be written in SNØBØL itself. The particular generator used (Hutchison, 1966, modified for a 1024 bit accumulator) required about ten conversions from string to integer and vice versa and therefore considerably increased execution times. Since much of the work of the program is done by functions which could be rewritten in machine language, the conversion to machine language would allow a great increase in speed. The main barrier to quick execution times, however, remains the necessity of examining the tree character by character. This was forced on the program by the Polish notation used.

In the near future, computers such as the IBM 360 and ILLIAC IV will be available which can handle character strings in a very natural manner. By then rewriting SNØBØL for these machines, or by writing the program itself in a hardware language for one of these machines, it should be possible to realize great gains in efficiency.

APPENDIX 1

THE CONTEXT FREE GENERATOR

The CF generator is currently assembled to run within a minimum monitor system, MINSYS. MINSYS reads parameter cards and calls on the specified programs. All MINSYS parameter cards have a dollar sign in column 1 followed by the name of the subsystem desired and any parameters needed. Blanks are totally ignored. The systems available in this assemble of MINSYS are:

\$ MESSAGE

The cards that follow are listed on the off-line printer.

\$ COMMENT

Same as MESSAGE, except that they will also be printed on-line.

\$ HALT

The computer stops for operator action.

\$ SYSERR

The program immediately returns to the system monitor, getting a core dump if system control cards specify one. Similar to FORTRAN CALL ERROR

\$ END

MINSYS returns to whatever it is embedded in. Currently this is the system monitor. No dump will take place.

\$ DUMP

Sets a flag within MINSYS so that all exits from MINSYS (END card or end-of-data deck) will go through SYSERR.

The above control cards are relics of an elder day when MINSYS was used

as the execution-time monitor for a set of bookkeeping programs. Except for DUMP, it is unlikely that any of the above would be of great use.

\$ HEADING

The following card is used as a primary heading for all successive printing.

The following routines allow flexible timing of programs.

\$ CLOCK ON

initializes an internal clock.

\$ CLOCK PRINT

prints the internal clock and resets it.

\$ TIME ON

sets a flag so that programs called by MINSYS may independently time their procedures.

\$ TIME OFF

The timing flag is cleared.

Note that CLOCK ON and CLOCK PRINT are independent of all other clocks within MINSYS. Of the above control cards, only HALT is automatically timed. After the following subsystems, the program will print the time used by that segment. After reading an END card or the end-of-data, the program will print the total MINSYS time used. The following two control cards allow the program to run on "quick turn-around" categories where the amount of printing allowed is limited.

\$ SHUTTLE

deletes all carriage controls and headings.

\$ NORMAL

restores carriage controls, etc.

\$ PØUTRAN, number of productions, (PUNCH), (RECURSIØN), (stack limit),
(NØN-FATAL)

PØUTRAN is the name of the CF generator. The optional parameters are:

number of productions, an integer giving the number of sentences to be generated.. If no number is given, no sentences will be generated.

PUNCH, if present, will cause the sentences to be punched in columns 1-72 in addition to being listed on the off-line printer.

RECURSIØN, if present, will cause the recursion procedure to be entered as described in the body of this paper.

stack limit is an integer giving the depth of the stack at which recursive subrules will be eliminated. It must follow immediately after RECURSIØN. If missing, 10,000 will be used.

NØN-FATAL, if present, sets a flag which is tested upon encountering a rule all of whose subrules are recursive. Its use is given in the body of the paper.

NOTES ON IMPLEMENTATION

60

where FWA is the first (low in core) word where the BCD or binary card image should be stored and EOF is an optional end-of-file exit. Control goes to EOF upon trying to read the end-of-file. The logical accumulator is set to zero if the next card on the tape is BCD. Called only by SCARDS. The University of Michigan System, from which PORTHOS is descended, uses TIX to transmit parameters where two are being sent in one parameter word, TXH where there is only one, thus minimizing the transfers into data or halts common in FMS programs.

SYSLIT Identical to SYSRIT, except the next card is only looked at. Multiple calls to SYSLIT will get the same card image. (Called only by SIMAGE--assembled with SCARDS.)

SYSSIT Skips a card on the input tape, calling sequence:

CALL SYSSIT

TXH , , EOF

This is not used but is included for completeness.

SYSWOT Prints an 133 column line image on the off-line printer (IBSYS SYSWOT). The first column is detached for carriage control. IBSYS and FMS users should note that their systems only allow 132 characters to be transmitted. (Annoying as it is, this restriction adhered to.) Calling sequence:

CALL SYSWOT

TXH FWA, , length

where FWA is, as before, the base address of the BCD line

and length is the number of words to be transmitted. Unlike the equivalent IBSYS routine, .MWR, this system does not allow parts of a line to be built up out of various strings. SYSW~~OT~~ is called only by SPRINT. SPRINT, among other things, simulates the 1401 carriage tape, allowing headings to be printed when desired. The interested reader is referred to SPRINT's listing for details. The following description of the carriage control codes is reprinted from the P~~ORT~~H~~OS~~ manual with permission.

The first character of each record determines the carriage control for printed records. This character is detached from the remainder of the record and is not printed. The code for the carriage control character under the P~~ORT~~H~~OS~~ system is as follows:

	<u>Left-most character for pre-print skip</u>
Single space	blank or +
Double space	0
Triple space	-
Sheet eject (skip to next page)	1
Skip to next half page	2
Skip to next quarter page	4
Skip to next sixth page	6

The 1401 peripheral processor program operates in the following manner for the above control character; a discussion of the carriage tape channels mentioned below may be found in Reference Manual, IBM 1401 Data Processing System, A24-1403-5.

- blank - Spaces one line before printing.
- 0 - Spaces two lines before printing.

- * - Spaces one line before printing.
- - Spaces three lines before printing. The blank, O, and - are changed to J, K, and L respectively for proper carriage movement within the 1401.
- 1 - Immediate page eject; skip to channel 1.
- 2 - Skips to channel 2 and spaces one line before printing.
(Note: The spaces following the skips are necessary for proper positioning since the carriage tape can have only one punch on each horizontal line).
- 4 - Skips to channel 4 and spaces two lines before printing.
- 6 - Skips to channel 6 and spaces three lines before printing.

All other characters are simply moved into a pre-set control carriage order to allow for additional spacing required by special jobs. Pre-print space suppression is not possible.

The carriage tape used in the P~~ORTH~~ØS system is punched as follows:

- Channel 1 --- punches in 13, 79
- Channel 2 --- punches in 12, 42, 78, 108
- Channel 3 --- punches in 14, 27, 43, 57, 80, 93, 109, 123
- Channel 4 --- punches in 11, 26, 41, 56, 77, 92, 107, 122
- Channel 5 --- punches in 15, 28, 44, 58, 81, 94, 110, 124
- Channel 6 --- punches in 10, 20, 30, 40, 50, 60, 76, 86, 96, 106,
116, 126
- Channel 7 --- punches in 9, 25, 39, 55, 75, 91, 105, 121
- Channel 8 --- punches in 8, 24, 38, 54, 74, 90, 104, 120
- Channel 9 --- punches in 6, 23, 37, 53, 72, 89, 103, 119
- Channel 10 --- punches in 5, 22, 36, 52, 71, 88, 102, 118
- Channel 11 --- punches in 4, 21, 35, 51, 70, 87, 101, 117
- Channel 12 --- punches in 7; 73

Only the characters listed previously and their corresponding channels (1, 2, 4, 6, 12) have any significance when operating under the PØRTHØS system with a system carriage tape. The remaining punches in the tape check any skipping that would result from the use of characters other than those listed.

SYSPCD Punch a decimal (BCD) card. The calling sequence is identical to that of SYSWØT. It is called from various places, including the generator itself.

SYSPCB Punch a binary card. Again, the same call as SYSWØT. It is not used but included for completeness.

SYSPPD Punch and print decimal. The first column is used for carriage control, the next 80 are both punched and printed. Called from SPRINT, not used by the generator.

SYSCAP Same calling sequence, the line is printed on-line.

SYSSAP This routine performs a page skip on the on-line printer.

The two subroutines above are entered only by HALT and COMMENT control cards.

SYSPAN Prints a panel dump (not a core dump also) and returns to the next location, destroying XR4 only. This may be assembled by the generator.

SYSERR The error exit to the monitor system (similar to CALL ERRØR in Fortran. Called here and there.

SYSTEM The normal end-of-job exit. Similar to CALL EXIT.

SYSHPR System Halt and proceed. Similar to IBSYS .PAUSE, called by HALT.

The macro-assembly language in which these programs were written is a modification of FAP. No attempt will be made to detail differenced

between this language and other assembly codes. The MAP user should be aware of differences in the CALL pseudo-op. Several routines in this deck transfer to 1,4 upon finishing their action. In this system, as mentioned above, nothing untoward should happen. In FMS or IBSYS, all kinds of troubles should result.

APPENDIX 3

CS AND TRANSFORMATION PROGRAMS

The CS and transformational programs were written in SNØBØL 3, a string manipulation language written at Bell Telephone Laboratories. In the CS expansion program, the tree will be printed with the terminal elements enclosed in primes. At the conclusion of the grammar, if the initial symbol is currently terminal, a flag will be set and the grammar re-begun for every instance of a terminal initial symbol. This flag causes the program to ignore (with comment) any sub-subrule having the initial symbol as part of its output. While this prevents the program from generating deeply embedded sentences, it also prevents it from taking an inordinate amount of time to expand one sentence.

In the transformation program, the pattern and tree will be printed (without any special marking of terminal nodes) along with the marked tree if the rule succeeds. If the operations are begun, they will be printed in their internal form. ALS, ARS, ALD, ARD will print as before, MLS, etc. and SUB will print in their expanded forms and the restrictions will be replaced by a single letter and EQU becoming D and E respectively. If a restriction is negative (NEQU or NDØM) the N will follow the D or E.

REFERENCES

1. Chomsky, N. Syntactic Structures. 's-Gravenhage: Mouton, 1957.
2. _____, "Formal Properties of Grammars". In Luce, R. D., Bush, R. R., and Galanter, E. (eds.), Handbook of Mathematical Psychology, Vol. II. New York: Wiley, 1963, pp. 323-418.
3. _____, Current Issues in Linguistic Theory, 's-Gravengage: Mouton, 1964.
4. _____, Aspects of the Theory of Syntax. Cambridge, Massachusetts: M.I.T. Press, 1965.
5. "English Preprocessor Manual." SR-132, MITRE Corporation, 1964, 1965.
6. Haines, E. C., Jr., "The TREET List Processing Language," SR-133, MITRE Corporation, 1965.
7. Hutchinson, D. W., "A New Uniform Pseudo-Random Number Generator," File No. 65, Dept. of Computer Sciences, Univ. of Ill., 1965.
8. Iverson, K., "A Method of Syntax Specification," Comm. ACM 7:588-9, Oct. 1964.
9. Naur, P. (ed.), "Revised Report on the Algorithmic Language ALGØL60," Comm. ACM 6:1-17, Jan. 1963.
10. Trakhtenbrot, B., Algorithms and Automatic Computing Machines. Boston: D. C. Heath, 1963.
11. Yngve, V., "A Model and An Hypothesis for Language Structure." Technical Report 369, Research Laboratory of Electronics, M.I.T., 1960.
12. _____, "Random Generation of English Sentences." Memo 1961-4, Mechanical Translation Group, Research Laboratory of Electronics, M.I.T., 1961.
13. Zwicky, A., Friedman, J., Hall, B. C., and Walker, D., "The MITRE Syntactic Analysis Procedure for Transformational Grammars," MTP-9, MITRE Corporation, 1965.