CS - 84
AF - 15

A COMPUTER SYSTEM
FOR
TRANSFORMATIONAL GRAMMAR

by

JOYCE FRIEDMAN

This research was supported in part by the United States Air Force Electronic Systems Division, under Contract F19628-68-C-0035.

STANFORD UNIVERSITY COMPUTER SCIENCE DEPARTMENT

COMPUTATIONAL LINGUISTICS PROJECT

JANUARY 1968

35

COMPUTER SYSTEM FOR TRANSFORMATIONAL GRAMMAR

by

Joyce Friedman

AF - 21
CS - 84

January 1968

A Computer System for Transformational Grammar

by

Joyce Friedman

Abstract

A comprehensive system for transformational grammar has been designed and is being implemented on the IBM 360/67 computer. The system deals with the transformational model of syntax, along the lines of Chomsky's Aspects of the Theory of Syntax. The major innovations include a full and formal description of the syntax of a transformational grammar, a directed random phrase structure generator, a lexical insertion algorithm, and a simple problem-oriented programming language in which the algorithm for application of transformations can be expressed. In this paper we present the syst as a whole, first discussing the philosophy underlying the development of the system, then outlining the system and discussing its more important special features. References are given to papers which consider particular aspects of the system in detail.

Table of Contents

	Page
Introduction	1
A metalanguage for transformational grammar	4
Basic concepts	7
Tree	7
Analysis	9
Restriction	11
Analysis algorithm	12
Complex symbol	13
Complex symbol operations	14
Components	16
Phrase structure	16
Lexicon	17
Transformations	18
Component algorithms	19
Phrase structure generation	19
Lexical insertion	20
Control of transformations	22
The Program	24
Directions for future work	25
Other transformational grammar systems	27
Acknowledgment	28
References	29

INTRODUCTION

The computer system for transformational grammar presented in this paper is the outcome of an attempt to write computer programs as aids to research in transformational grammar, in particular, as aids to writing grammars.

In the course of this work it soon became apparent that an essential prior task was the formalization of a general and inclusive notion of transformational grammar. The basic model is that of Chomsky's Aspects of the Theory of Syntax [3]; we have extended this model to fill in the many missing details and have formalized it to make it precise.

The system is implemented by a FORTRAN program on the IBM 360/67 computer. However, as a formal statement of transformational grammar, it can be considered independently of the program. We have therefore relegated to one section and to occasional footnotes all matters related directly to the program.

This paper may be considered as both a summary of and an introduction to the system. We have stressed the ways in which the system is new, and have left the details for other papers, which will be cited.

In developing the system our primary examples have been the MITRE grammar [18], the IBM Core Grammar [13] and the UCLA work on syntax [17].[1] However, we have not limited the system to matters treated in these examples, but have tried to be comprehensive.

---

[1] The UCLA work has kindly been made available to us in its preliminary stages through unpublished working papers and memoranda. We wish also to thank Barbara Hall Partee of UCLA for numerous discussions which have helped to clarify our ideas about transformational grammar.

A transformational grammar may be sketchily described as follows. The components of a transformational grammar are phrase structure rules, a lexicon, and a set of transformations. The process of generating a sentence consists first of the generation of a base tree using the phrase structure rules. Lexical items are then attached appropriately by a lexical insertion algorithm. Finally, the base tree with its lexical items is mapped by application of the transformations in some order into a surface tree. The terminal string of the surface tree represents the sentence.

From the outset we have felt that it was essential to consider a transformational grammar as a whole. A rule of a grammar may behave as intended in isolation, but in the grammar its interaction with other rules is crucial. It is precisely these interrelations which are most difficult to control, and we believe it is here that a computer system can be most helpful.

We did not wish to try to guess the exact amount of power required to describe the syntax of natural language, nor to be normative in our approach. Our aim is to handle as uniformly and simply as we can the sorts of things which do appear in the current work on transformational grammar. The formalism has been made general enough so that most of the formal grammars and rules which we have seen can be expressed naturally. On the other hand, there are some devices in the literature which appear to us to be so different in character from the rest of the material as to be unacceptable in anything like their present form, and we have not included them.[2/]

[2/] As an example we might cite the distance measure included in the Identity Erasure Transformation of [13]. This appears to us to be more properly considered as a linguistic rule, which should be expressible, but which should not appear as part of a particular transformation. Further comments on linguistic rules of this type appear below.

It is quite likely that at least some linguists will feel that the generality of the system is excessive. But there is no need for any one user to employ its full power. In the metalanguage of this system, a linguist may easily define his own subset of the syntax; we believe such formalization will make it easier for him to adhere to his conventions. Although we have not done so, it would be possible to provide user-oriented subroutines to verify that the user's additional constraints are not violated.

The traditional description of a transformational grammar can be given an alternative presentation in terms of basic concepts, components, and component algorithms. The basic concepts of a grammar are trees, analyses, restrictions, and complex symbols, with their corresponding algorithms. The components are phrase structure, lexicon, and transformations. The component algorithms are phrase structure generation, lexical insertion and control of transformations. Viewing a grammar in this way, we are able to see more clearly the basic problems to be treated. It is this breakdown which will be used in the subsequent description.

We assume that the reader is familiar with transformational grammar. The presentation is incomplete; we omit standard items and emphasize the ways in which this system differs from others. While the discussion below is largely informal, it is important that it is based on the completely formal syntax of [21].

A METALANGUAGE FOR TRANSFORMATIONAL GRAMMAR

To describe the syntax of a transformational grammar one must first choose a metalanguage. The usual choice by linguists has been English. The metalanguage used here is a modification of Backus Naur Form (BNF), familiar to computer scientists as the language used in the description of Algol 60. As we will use the symbols | , < and > in transformational grammars, we modify the usual BNF by replacing angular brackets by underlining, e.g. "transformation" rather than "<transformation>", and using "or" in place of "|" .

For linguists unfamiliar with BNF it should suffice to say that (1) the modified-BNF production " A ::= B C or D or E " expresses the context-free rewriting rule " $A \rightarrow \left\{ \begin{matrix} B\ C \\ D \\ E \end{matrix} \right\}$ ", (2) the nonterminal symbols of modified-BNF are denoted by the underlined name of the construct, viz. transformational grammar ::= phrase structure lexicon transformations (3) symbols not underlined are used autonymously, and (4) juxtaposition in the object language is indicated by juxtaposition in the metalanguage.

We refer to the constructs of the metalanguage as "formats", because they are in fact the free-field formats of the computer system. We have carried the underlining of format names into the text of the paper.

Basic to the syntax are the two formats word and integer. A word is a contiguous string of letters and digits beginning with a letter; integer is a contiguous string of digits. Except in these two formats, spaces may be used freely.

If a BNF description is to elucidate a language, it should not introduce names for intermediate formats which do not have meaning. In order to avoid additional formats where possible, and to simplify the description, we have introduced into the metalanguage the five operators list, clist, opt, booleancombination and choicestructure. In each case the operand is given within square brackets following the operator. Only the first three of these operators are used in this paper. They are:

1. list

a ::= list [integer]

allows a to be

1 2 6 9171 3 20

2. clist (comma list)

a ::= clist [integer]

allows a to be

1, 2, 6, 9171, 3, 20

3. opt (option)

a ::= opt [integer] word

allows a to be either

3 NP or NP

It is clear that any occurrence of an operator in a production could be deleted by the introduction of intermediate formats and corresponding additional productions. This would not change the object language.

A full description of the syntax of transformational grammar is given in [21]. In this paper we shall give only a few of the productions, as needed to describe special features of the system.

BASIC CONCEPTS

Each of the basic concepts is used throughout a grammar; they are defined recursively in terms of one another.

Tree

The format for a tree is

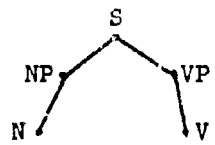
tree ::= node opt { complex symbol } opt [< list [tree] >]

where

node ::= word or sentence symbol or boundary symbol

The optional list of trees is the list of daughter subtrees of the node in left-to-right order. For example, the tree

S < NP < N > VP < V > > represents:



Because a bracketed representation of a tree can easily become cumbersome and unreadable, a substitution capability is provided by the production:

tree specification ::= tree opt [,clist [word tree]]

A tree is read and then searched for an occurrence[1] of the first word in the list. Then the tree following the word is substituted for that occurrence of the word. The process is repeated until the list is exhausted. For example, the tree specification S < S1 S2 > , S1 NP < N > , S2 VP < X > , X V results in the same tree shown above.

[1] In this and other similar substitutions for a word, it is intended that the word have exactly one occurrence in the tree.

Occasionally a tabular representation of a tree is preferable, and one is available in the system. It is used for inputs to the random generation routine, and as the output format.

For a detailed discussion of internal and external formats for trees used in the system see [26].

Tree operations

The basic operations for trees are comparisons and changes. The basic tree comparison is equality. The test for equality of trees can be combined with a test for either equality or nondistinctness of their corresponding complex symbols (see below). Trees may also be tested to see if they include a specified node (dominance).

Changes to trees include the elementary operations of the MITRE grammar and the IBM Core grammar. They also include the operation (tree) SUBST word which substitutes the tree for an occurrence of word. This can be used to allow a change to refer to a node inserted by a previous change in the same se[1]

[1] The MITRE programs [5] and Londe and Schoene [10] handle this same problem in other ways.

Analysis

Analyses occur in two places in the grammar: in the structural description for a transformation and as contextual features.

The syntax for an analysis is a strong generalization of the notion of proper analysis originally given by Chomsky. A proper analysis is given by a list of nodes which are to occur in a left to right cut across a tree. The syntax of an analysis here is fully recursive; the terms of the analysis are not simply nodes but structures which may contain further analyses.

analysis ::= list [opt [integer] term]

Note that this labelling of terms of an analysis allows the linguist to number only those terms to which he will refer.

term ::= structure or skip or (choice)

choice ::= clist [analysis]

Any member of the clist will satisfy the choice.

structure ::= element opt [complex symbol]
opt [opt [¬] opt [/] < analysis >]

A structure is an element which may optionally have a complex symbol and may optionally have a further analysis. The analysis of the element may be negative ("not analyzable as", denoted by ¬). The optional slash indicates that the analysis is not necessarily immediate. Its absence indicates an immediate analysis.

element ::= node or * or _

An element may be a specific node (see definition above) or simply an unspecified single word indicated by the definite node * . The underline symbol occurs only in analyses which are contextual features,

and indicates the location for lexical insertion. A complex symbol in an analysis always directly follows an element.

skip ::= $ opt [< structure >]

The use of skips rather than variables follows the MITRE grammar.

It may be noted that a tree is simply a subcase of structure in which no integers and none of the special symbols (,) , ¬ , / , * , and __ occur.

Restriction

A restriction may occur only in association with an analysis. It may be a proper part of a transformation, or may be part of a contextual feature or it may define the test for a conditional change in the structure change of a transformation.

Analysis algorithm

The analysis algorithm will be described in detail in [24]. The one linguistic rule so far incorporated in the system occurs here. A search is not allowed to go below a sentence symbol unless either the analysis is part of a transformation which has the parameter which specifically allows this, or the analysis itself contains a sentence symbol for which a further analysis is given. Thus there are two ways to specify the depth of a search.

Another interesting feature of the analysis algorithm is the provision for handling the associated restriction. A three-valued logic is used and the value of the restriction is "undefined" until the search has proceeded far enough to determine a value of "true" or "false" for the whole restriction. As the search proceeds or backtracks the value of the restriction is continually set and unset.

Complex symbol

Complex symbols occur in trees, in analyses and restrictions, in the structural change of a transformation, and in the lexical entries and the redundancy rules of the lexicon.

We distinguish between a feature specification and a feature:

feature specification ::= value feature

Feature specifications occur only in complex symbols.

A complex symbol is a list of feature specifications enclosed in vertical bars and is interpreted as a conjunction. A lexical entry contains a list of complex symbols which is interpreted as a disjunction.

Only the three values + , - and * are allowed.[1] Following UCIA [17] a feature specification with the indefinite value * means that the feature is "marked", without specifying whether it is + or - . The value * never appears in a complex symbol in a tree, and is never used with a contextual feature.

A contextual feature is an analysis structure which contains precisely one underline symbol __ and whose head element is a node. It optionally has an associated restriction. The underline indicates the node where the lexical insertion will occur. A user who adheres to Chomsky's "principle of strict local subcategorization" will use as the head element of each contextual feature the node which immediately dominates the one for which the lexical insertion is to be made. A user who disavows the principle may choose any dominating node for the head element. Contextual features appear only in the lexicon and are used solely in the lexical insertion process.

[1] Gross [6] allows arbitrary words to be declared as values.

Complex symbol operations

The basic operations for complex symbols are comparisons and changes.

The comparisons are for equality, non-distinctness, and two types of inclusion. The result of the comparison of two feature specifications A and B is shown in the tables below, where T represents true and F represents false and abs indicates that the feature is absent altogether. For the test to be true for complex symbols it must be true for all their feature specifications.

EQUALITY

A \ B	+	-	*	abs
+	T	F	F	F
-	F	T	F	F
*	F	F	T	F
abs	F	F	F	T

NONDISTINCTNESS

A \ B	+	-	*	abs
+	T	F	T	T
-	F	T	T	T
*	T	T	T	T
abs	T	T	T	T

INCLUSION-1

A \ B	+	-	*	abs
+	T	F	T	F
-	F	I	I	F
*	T	I	T	F
abs	T	T	T	T

INCLUSION-2

A \ B	+	-	*	abs
+	T	F	F	F
-	F	T	F	F
*	F	F	T	F
abs	T	T	T	T

The basic changes of complex symbols include merging A into B moving the features of A to B, erasing all the features of A from B, and saving in B only the feature specifications which are included in A. The results of these operations are shown in the tables below. It is to be expected that other operations will be added later as required.

MERGE

A \ B	+	-	*	abs
+	+	+	+	+
-	-	-	-	-
*	+	-	*	*
abs	+	-	*	abs

ERASE

A \ B	+	-	*	abs
+	abs	-	-	abs
-	+	abs	+	abs
*	abs	abs	abs	abs
abs	+	-	*	abs

SAVE

A \ B	+	-	*	abs
+	+	abs	+	abs
-	abs	-	-	abs
*	+	-	*	abs
abs	abs	abs	abs	abs

A redundancy rule A => C applies to a complex symbol B only if A is included-1 in B . If so, then C is merged into B .

COMPONENTS

The three components of a transformational grammar are phrase structure, lexicon, and transformations.

Phrase structure

The phrase structure of the system is a conventional context-free grammar. Complex symbols do not appear in the phrase structure: they are introduced during lexical insertion (see below). Rules are accepted in a linearization of the standard linguistic form and are immediately expanded.[1] For example, the rule

$$VP \rightarrow \left\{ \begin{array}{l} AUX \left\{ \begin{array}{l} MV\ (NP) \\ COP\ (\left\{ \begin{array}{l} NP \\ AP \end{array} \right\}) \end{array} \right\} \\ \\ S \end{array} \right\} (ADV)$$

is represented as

$$VP \rightarrow (AUX(MV(NP),COP(NP,AP)),S)(ADV)$$

The expression of rule schemas by use of the Kleene star * has not been included.[2]

---

[1] Black [3] also expands from a compact form.

[2] Londe [10] accepts the Kleene star.

Lexicon

A lexicon contains a preliminary part, or prelexicon, which contains feature definitions and redundancy rules. The feature definitions include a list of categorys in the order of lexical insertion. One may also give names to contextual features to avoid having to write them in full in the lexical entries. A redundancy rule is of the form:

redundancy rule ::= complex symbol = > complex symbol

The interpretation is that if a complex symbol includes all the feature specifications of the complex symbol to the left of the arrow (= >) of a redundancy rule then it implicitly contains those of the complex symbol to the right of the arrow. Explicit expansion of complex symbols by the redundancy rules can be carried out in the system.

In a lexical entry the set of possible complex symbols for a vocabulary word are given. If several vocabulary words have the identical set of complex symbols, the vocabulary words appear in a single lexical entry. Each complex symbol corresponds to a sense of the word. The set of complex symbols is regarded as a disjunction. Since the complex symbol itself is a conjunction of feature specifications this is in effect a normal form. Thus the system has the same power as one which allows arbitrary boolean combinations of features, (see Lakoff [7]), without their complexity. For example, to say that a verb must have both an animate subject and an inanimate object, one may use either one or two feature specifications in the same complex symbol. To say that it must have either an animate subject or inanimate object, two complex symbols are needed.

Transformations

The final component of a grammar consists of a list of transformations and a control program. The discussion of the control program will be deferred to the section on the algorithm for control of transformation .

A transformation consists of a transformation identification, a structural description, and (optionally) restrictions and structural change. The transformation identification may include, in addition to the transformation name, a group number and various parameters. A transformation may be referenced either by its transformation name or by the group number. The parameters indicate whether or not the transformation is optional, whether (and how) it is to be repeated after a successful application, and whether or not the analysis algorithm may march below an unmentioned sentence symbol. Keywords are also given here.

The structural change is expressed, as in the MITRE grammar [18], by a list of operations. A new feature of the system is the conditional change.

conditional change ::= IF < restriction > THEN
< structural change > ELSE
< structural change >

The basic operations for trees and complex symbols have already been discussed.

COMPONENT ALGORITHMS

The three main algorithms of a transformational grammar correspond to the three components and are phrase structure generation, lexical insertion and control of transformations. Our implementation of the first process is designed to be useful in the testing of a grammar. The second has not previously been fully described and we give for the first time an explicit algorithm. Various proposals have been made for the third algorithm; rather than choosing one of them we include the specification of the algorithm as part of the grammar.

Phrase structure generation

The system can be started with a base tree input by the user. However, it also has the capability of "directed random" generation of trees from the phrase structure grammar. This scheme, which is described in detail in [20], allows the user to specify a "skeleton" around which a tree is generated at random. The skeleton may also bear constraints of dominance, nondominance and equality. The scheme was designed to make it possible for the user to generate trees which are "interesting" rather than simply random; in particular, which will test a specific transformation. It should be noted that there is a restriction on the phrase structure grammars which can be handled by the algorithm: the rules must be ordered so that no symbol is introduced below the rule which expands it, with the exception of course of the sentence symbol.

Lexical insertion

The algorithm for lexical insertion is an interpretation of one of the two alternatives presented by Chomsky in Aspects. Complex symbols are introduced from the lexicon only after the phrase structure generation of the base tree is completed. In order to formalize the process, we have had to make decisions on many points not treated explicitly by Chomsky. The details are presented in [22]; we note here some of the salient features.

A contextual feature is simply a special case of analysis; thus much of the work in lexical insertion is done by the same analysis algorithm used for transformations.

Lexical insertion begins with the lowest embedded sentence, and works upward.[1] Within a sentence the order of lexical insertion is determined by the list of categorys in the prelexicon. This order may have considerable effect on the efficiency of the process. However, from a formal point of view, all categories are alike.

The basic criterion for lexical insertion is non-distinctness: the tree may already contain a complex symbol; a word and its complex symbol can be inserted only if the complex symbol is non-distinct from the one already in the tree. But this is only a necessary condition; each feature specification for a contextual feature must be checked by the analysis algorithm. If the value is + the analysis algorithm must succeed, and if - it must fail.

[1] Although complex symbols are not introduced in the phrase structure, it is possible that a skeleton input to the phrase structure generation routine already contains some words of the lexicon. In this case, the complex symbols for those words are looked up in the lexicon and inserted prior to the process described here.

Once a vocabulary word and complex symbol have been selected (at random from those meeting the above tests), one additional step is necessary before lexical insertion takes place. The possible side effects of the contextual features must be taken care of. If, for example, a verb has been selected which takes animate subject and inanimate object, feature specifications may need to be added to the complex symbols for the subject and object. Then contextual features are dropped from the complex symbol, since they have served their function, a + or - value replaces the indefinite value * , and the vocabulary word and complex symbol go into the tree.

Control of transformations

Each transformational grammar that has discussed at all the matter of order and point of application of transformations has presented a slightly different algorithm. From the available examples, it was possible to abstract the basic ideas involved and to write a simple programming language in which the linguist can express the algorithm for a particular grammar.[1] The control program refers to transformations either individually by transformation name or by group number. The language contains a repeat-instruction which allows a list of control instructions to be repeated either for a fixed number of times or until they all fail. One innovation is the IN-instruction. The statement

IN transformation name (integer) DO

causes the integer-th term of the transformation to be used as the starting point for the search algorithm. Such notions as "highest sentence", "lowest sentence", etc. can be expressed by the IN construct.

The notion of keyword has also been implemented.[2]

The control language allows branching on the success or failure of a transformation. The use of this conditional instruction makes it possible to write transformations with less attention to certain types of interaction. For example, suppose transformation T2 is to apply only if T1 has failed to apply. Then the instructions

[1] In addition to controlling the grammar, the control language also provides TRACE instructions which govern the amount of output.

[2] Keywords were first used in the MITRE programs [5]. They were implemented in a slightly different form by IBM [9].

IF T1 THEN GO TO A ELSE GO TO B,

A: T2,

B: ...

will cause T2 to be bypassed if T1 fails. This instruction may be considered excessively powerful. It is available because the alternatives frequently seem to be either to alter artificially the structural description of T2 or to include a restriction on T2 such as: "applies only if T1 has failed to apply".[1/]

For a detailed discussion of the control language and examples of control programs see [23].

We have not attempted to deal with the notion of implicit ordering of transformations.

[1/] The use of the conditional instruction will of course speed up the processing of a tree.

THE PROGRAM

The system is written as a collection of subroutines which can be called in various orders. A table of the subroutine structure is included in the Programmer's and User's Guide to the System [24].

A MAIN program consists of a sequence of subroutine calls. Typically a run begins with a call to the initialization subroutine, followed by calls to input routines for the components of the grammar. Then either a base tree is input, or a skeleton is input and the generation routine called. Lexical insertion is optional at this point. Then the transformation routine is called, and the program executes the user's control program. The process can be repeated with a new tree from the skeleton or with a new tree input.

Alternative MAIN programs to test individual components of the grammar can easily be constructed. For example, to test the phrase structure one might simply generate trees at random. Or, to test lexical insertion one could start with base trees containing incomplete complex symbols and investigate how they were completed. Transformations can be tested beginning from base trees with (or without) lexical items already included.

MAIN programs for a variety of purposes are also given in [24].

The system is implemented in FORTRAN IV (H) on the IBM 360/67. To the user, however, the system does not look like FORTRAN. All of the formats are free-field and, externally, words may be up to 40 characters long. See [19] for a description of the free-field input/output subroutine package.

DIRECTIONS FOR FUTURE WORK

There are many ways in which the work which has been done can be extended. Some of these correspond to interesting open questions in the transformational theory of syntax. We mention here some areas in which we plan to begin work soon. We think that the generality of the system will give us a strong starting point in these investigations.

Conjunction

No means of handling transformational schemas such as conjunction has been provided. In the earlier programs at MITRE a conjunction algorithm due to Schane [16] was included and we plan to carry this over into the present system as its first version of conjunction. We hope then to investigate the alternatives considered in the literature.

Idioms

A common proposal for the treatment of idioms is that an idiom occurs as a tree in the lexicon. We foresee only minor difficulties in incorporating idioms in this way, and plan to do so when time allows.

Linguistic rules

The current trend in transformational linguistics includes a search for linguistic rules which would apply to all grammars. Ross [14, 15], in particular, has been working along these lines. We hope later to investigate this work by devising means of incorporating

proposed rules into the system.[1]

Lexical derivation

The recent work by Chapin [2] and Chomsky [4] on lexical derivation has opened up some interesting lines of investigation which we are now beginning to explore within the system. A preliminary study of Chapin's early work was made prior to the development of the system and is reported in [30].

Dependency grammars

Jane Robinson [12] has recently offered a proposal for transformational grammars in which the underlying structure is a dependency grammar. The present system allows complex symbols to be associated with any node of a tree, but we do not now associate lexical words with higher nodes as would be required by the "projectivity" of dependency grammars.

[1] Ross's rule of tree-pruning has been incorporated by Gross [6].

OTHER TRANSFORMATIONAL GRAMMAR SYSTEMS

The earliest computer systems for transformational grammar were those of Petrick [11] and MITRE [18]. The system here is an outgrowth and extension of this early work at MITRE. Naturally it embodies a more recent version of transformational theory.

The partial system of Lieberman and Blair [8, 1] represents an early attempt to deal with the model of Aspects. A lexicon was defined, and phrase structure programs and some transformational programs were written.

Systems developed concurrently with this one include the console-controlled grammar testers of Gross [6] and of Londe and Schoene [10].[1/] The problems best treated by a system designed for immediate response to a user at a console differ from those appropriate to an off-line system such as ours. While there is some overlap in these systems, we believe ours is the first to consider all phases of transformational grammar in a unified system. For example, the three component algorithms have no correspondents in other systems and neither has included a lexicon. Various differences in common areas have been noted above.

[1/] We wish to thank both Dave Londe and Lou Gross for many pleasant and fruitful discussions, and for a free exchange of ideas from which our work has benefitted.

ACKNOWLEDGMENT

The system described in this paper was developed with Robert W. Doran (metalanguage, basic syntax, free-field input/output, analysis algorithm), Thomas H. Bredt (lexicon and lexical insertion), Theodore S. Martner (analysis algorithm), and Bary Pollack (restrictions, control language). We have worked closely and well together; while the primary areas of responsibility are as shown above, there is no part of the system that has not been helped by ideas from others in the group.

REFERENCES

[1] F. Blair, Programming of the grammar tester, in [9].

[2] Paul Chapin, On the Syntax of Word Derivation in English, MIT Thesis, 1967.

[3] Noam Chomsky, Aspects of the Theory of Syntax, M.I.T. Press, Cambridge, Massachusetts, 1965.

[4] Noam Chomsky, Nominalization, to appear in Peter S. Rosenbaum and Roderick Jacobs, eds., Readings in English Transformational Grammar, Blaisdell Publishing Co.

[5] J. Friedman, SYNN, an experimental analysis program for transformational grammars, WP-229, The MITRE Corporation, 1965.

[6] L. N. Gross, On-line programming system user's manual, MTP-59, The MITRE Corporation, 1967.

[7] George Lakoff, On the nature of syntactic irregularity, NSF-16 The Computation Laboratory, Harvard University, 1965.

[8] D. Lieberman, Design of a grammar tester, in [9].

[9] D. Lieberman, ed., Specification and Utilization of a Transformational Grammar, AFCRL-66-270, 1966.

[10] D. L. Londe and W. J. Schoene, TGT: Transformational Grammar Tester, Systems Development Corporation, 1967.

[11] Stanley R. Petrick, A recognition procedure for transformational grammars, M.I.T. Thesis, 1965.

[12] Jane J. Robinson, A dependency-based transformational grammar, IBM Research Report RC-1889, Yorktown Heights, N. Y., 1967.

[13] P. Rosenbaum and D. Lochak, The IBM Core Grammar of English, in [9].

[14] John R. Ross, A proposed rule of tree-pruning, paper presented to the Linguistic Society of America, 1965.

[15] John R. Ross, Constraints on variables in syntax, M.I.T. Thesis, 1967.

[16] Sanford A. Schane, A schema for sentence coordination, MTP-10, The MITRE Corporation, 1966.

[17] R. Stockwell, P. Schacter, B. Partee, et. al., Working Papers of the English Syntax Project, UCLA, 1967.

[18] A. M. Zwicky, J. Friedman, B. Hall, and D. E. Walker, The MITRE Analysis Procedure for Transformational Grammar, Fall Joint Computer Conference 1965, 27, 317-326. See also MTP-9, The MITRE Corporation, 1965.

The following references are working papers and reports of the Computational Linguistics Project, Computer Science Department, Stanford University.

[19] Robert W. Doran, 360 O.S. FORTRAN IV Free Field Input/output Subroutine Package, CS-79, AF-14, October 1967.

[20] Joyce Friedman, Directed Random Generation of Sentences, CS-80, AF-15, October 1967 (submitted for publication).

[21] Joyce Friedman and Robert W. Doran, A Formal Syntax for Transformational Grammar, AF- , forthcoming.

[22] Joyce Friedman and Thomas H. Bredt, Lexical Insertion in Transformational Grammar, AF- , forthcoming.

[23] Joyce Friedman and Bary Pollack, A Control Language for Transformational Grammar, AF- , forthcoming.

[24] Joyce Friedman, ed., Users' and Programmers' Guide to a Transformational Grammar System. This document is not yet complete but the following sections are available as working papers:

[25] J. Friedman, Subroutine Structure, AF-17, November 1967.

[26] J. Friedman, Trees, AF-1, September 1966.

[27] J. Friedman, Input routine for transformations, AF-16, October 1967.

[28] J. Friedman, Input routine for structural change, AF-18, November 1967.

[29] Bary Pollack, Routines for restrictions, AF-19, December 1967.

[30] Joyce Friedman, Programming lexical grapho-morphemic analysis, AF-3, November 1966.