CS - 112

A CONTROL LANGUAGE FOR TRANSFORMATIONAL GRAMMAR

BY

Joyce Friedman and Bary W. Pollack

This research was supported in part by the United States Air Force Electronic Systems Division, under Contract F196828-C-0035.

STANFORD UNIVERSITY COMPUTER SCIENCE DEPARTMENT
COMPUTATIONAL LINGUISTICS PROJECT
AUGUST 1968

AF - 35

CS - 112

A CONTROL LANGUAGE FOR TRANSFORMATIONAL GRAMMAR

by

Joyce Friedman* and Bary W. Pollack

This research was supported in part by the United States Air Force Electronic Systems Division, under Contract F 196828-C-0035, at Stanford University.

* Present address: Computer and Communication Sciences Department
University of Michigan, Ann Arbor, Michigan

ABSTRACT

Various orders of application of transformations have been considered in transformational grammar, ranging from unorder to cyclical orders involving notions of '"lowest sentence'* and of numerical indices on depth of embedding. The general theory of transformational grammar does not yet offer a uniform set of '"traffic rules" which are accepted by most linguists. Thus, in designing a model of transformational grammar, it seems advisable to allow the specification of the order and point of application of transformations to be a proper part of the grammar.

In this paper we present a simple control language designed to be used by linguists for this specification.

In the control language the user has the ability to:

1. Group transformations into ordered sets and apply transformations either individually or by transformation set.

2. Specify the order in which the transformation sets are to be considered,

3. Specify the subtrees in which a transformation set is to be applied.

4. Allow the order of application to depend on which transformations have previously modified the tree.

5. Apply a transformation set either once or repeatedly.

In addition, since the control language has been implemented as part of a computer system, the behavior of the transformations may be

monitored giving additional information on their operation.

In this paper we present the control language and examples of its use. Discussion of the computer implementation will be found in Pollack [1].

The need for a language to express '"traffic rules"

The transformational component of a grammar consists primarily of a set of transformations; but it must also contain what Fillmore [2] has called "traffic rules'@ which specify the order in which the transformations are to be applied. These rules may be considered either as part of linguistic theory -- in which case there is one set of rules which applies to all grammars, or they may be considered as a proper part of a grammar. In this paper we take the position that the traffic rules, or "control program'", of a grammar are a part of the transformational component. In doing so we do not wish to maintain anything at all about the possibility that a universal set of traffic rules will someday be found. Our position reflects the fact that at the present time linguists disagree on what the traffic rules are. So we start from the premise that each grammar contains its own set of traffic rules, and we define a language in which these rules can be expressed,

The suggestion that the metatheory of grammars contains some complex scheme for "traffic laws'" within a grammar, and a control unit which directs the order of application of rules, occurs in Lees [3] as one of three alternative plans for rule ordering. The language proposed in this paper may be viewed as a proposal for Lees' '"complex scheme".

The control language which we present was developed for a computer program which accepts and manipulates transformation grammars -- we required some decision as to the order of application of transformations. None of the specific plans which have been proposed seem to have any general acceptance. Thus we felt that our programs would be more

useful if the specification of the traffic rules were a user option. The programs may be used to investigate order of application of rules as one of the open empirical questions about grammars.

We hope that the control language will also be of interest per se, independently of the programs.

The control language operates by taking advantage of mechanisms which must already be in any system of transformational grammar. For example, the "IN-construct", used to determine the subtree for which a transformation is to be invoked, itself uses a transformation in this determination. Likewise, there is no provision for placing special indices on the sentence tree, but instead feature specifications, already in the system, are used. The decision to stay within the devices already available causes some difficulty in expressing some of the proposed cycling orders, as is apparent in Example 3 below. However, the alternative would be to program special devices specific to the various proposals in the literature, which we prefer not to do until some general ideas can be abstracted from them.

The purpose of the control program is to determine in what order and at what point a transformation is invoked. Thus, in the familiar control sequence: apply the cyclic transformations to the lowest sentence, the control program must select the lowest sentence subtree and then invoke the transformations in order for that subtree.

In this presentation of the control language we first discuss the transformation component as it relates to the control program, then what is meant by invoking a transformation -- this will be primarily a discussion of the meaning of the parameters of a transformation. Then

we shall discuss the control language itself and show how it provides a step-by-step selection both of the transformation to be invoked and the tree node which is to be the top of the subtree treated by the analysis algorithm,

The transformational component of a grammar

The system with which we are working contains a formal metasyntactic description of transformational grammar. The metasyntax is described in [4]. In this paper we will cite the formal descriptions, but will in every case also spell them out in English. We can use the syntax to show the position of the control program within the grammar:

0.01 transformational grammar ::= phrase structure lexicon
transformations $END

8.01 transformations ::= TRANSFORMATIONS list[transformation]
CP control program . $END

(Numbering here corresponds to the full syntax given in Appendix A.) The interpretation of these rules is:

a transformaional grammar consists of a phrase, structure followed by a lexicon followed by a set of transformations followed by the terminator $END.

transformations, the transformational component with which we are here concerned, consists of the identifier TRANSFORMATIONS followed by a list of transformations, followed by the identifier CP

and a control program terminated by a period. The transformational component is terminated by $END.

The important point here is that the control program is a proper part of transformations. It is needed to provide the ordering for the transformations.

Identification of a transformation

The control program must be able to refer to individual transformations and to recognize whether or not they are optional, and if and how they are to be repeated. The information specific to a single transformation is provided in the identification which is the first part of a transformation.

8.02 transformation ::= TRANS identification SD structural description opt[SC structural change .]

A transformation consists of the phrase TRANS, followed by an identification, followed by the phrase SD and the structural description, followed optionally by the phrase SC , a structural change, and a period. (The reason for allowing the structural change to be optional will be seen below in the discussion of the IN-construct.)

8.03 identification ::= opt [integer] transformation name opt [list [parameter]] opt [keywords].

The identification of a transformation consists of an optional integer, followed by the transformation name, followed optionally by a

list of parameters and optionally by keywords. The integer is for external identification only; within the grammar a transformation is referred to by its transformation name.

8.04 parameter ::= group number or optionality or repetition

There are three types of parameters: the group number (a roman numeral) identifies the transformation as part of a group. The group number may be used to refer to all of the transformations in the group.

Optionality (OB or OP) has the usual interpretation, obligatory or optional. Repetition includes four possibilities (AC, AACC, ACAC and AAC) -- which are more general than those which have previously been considered and will be discussed in detail below.

Although the list of parameters is optional, each transformation is in fact specified for group number, optionality and repetition, since for each there is a null option. If no group number is specified, it will be taken to be the same as that of the previous transformation (or I for the first transformation). The null option for optionality is obligatory (OB) and for repetition AC .

Invoking a transformation

By invoking a transformation we mean (in the simplest case of an OB AC transformation) that the analysis algorithm will be applied to determine if the structural description is met, and that if so, the structural change will be applied. However, this description is not yet complete, for the analysis algorithm is not always to be applied to the full sentence tree. It is certainly necessary to be able to specify

that the analysis algorithm is to be applied to a specific subtree. Therefore, we modify the definition above to state that in invoking a transformation, the analysis algorithm will be applied to a specified subtree.

Optionality and repetition

We have defined above what it means to invoke a transformation with optionality OB and repetition AC. In the tables below we extend the definition to cover the full range of cases for these parameters; in each case a single specification of the subtree is implicitly assumed. --.

The repetition parameter has four possible values, with mnemonics composed of the letters A (for "analyze") and C (for "change"). These mnemonics AC, ACAC, AACC, AAC were invented because the phrases "cyclic", "noncyclic", "iterative", "recursive" etc. have by now had so many different interpretations that confusion can easily arise.

Repetition	To invoke the transformation
AC	The analysis algorithm is applied to find the first match for the structural description; the structural change is then carried out if one is found.
ACAC	The process just described for AC is repeated until no further match is found.
AACC	The analysis algorithm is applied to find all possible matches for the structural description; the corresponding structural changes are then carried out.
AAC	The analysis algorithm is applied to find all possible matches for the structural description; one of these is selected at random and the appropriate structural change applied.

Table I

Invoking an obligatory (OB) transformation

In the case of a control program which must run without human intervention, the natural way to decide in optional cases is by random choices; if the program interacts with an on-line user then the decisions in optional cases may be made by the user. (Our implementation of the control language is in an off-line environment; therefore a random choice is made. This is a characteristic of the implementation, not of the control language, which could be used in either type of environment.) Table II shows the process of invoking an optional (OP) transformation.

Repetition	To invoke the transformation
AC	Decide. If yes, proceed as for OB case.
ACAC	(α) Decide. If yes, proceed as for OB AC case. Repeat (from α) until either a negative decision is reached, or no match is found.
AACC	The analysis algorithm is applied to find all possible matches for the structural description. For each, a decision is made and if yes the corresponding change is applied.
AAC	Decide. If yes, proceed as for OB case.

Table II

Invoking an optional (OP) transformation

The repetition parameter AAC

The tables above define the meaning of the four possible repetition parameters. Some discussion is now in order to defend our choice of values for repetition. AC, ACAC and AACC are all cases which are commonly found in the literature, although some arguments have been given to show that ACAC is unnecessary. The case AAC is new and is suggested by difficulties found in the literature. Consider for example, the WH-Attraction transformation of Rosenbaum and Lochak [5], which we give here in their notation:[1/]

[1/] In the notation of our system, WH-Attraction would be written:

TRANS 10 WHA "WH-ATTRACTION" I OB AAC .

SD # % ART S/< 4 NP % 6 (* < PREP NP/< WH % > >, NP/< WH % >) % > % #.

SC 6 ALESE I 4.

All variables are replaced by % . Substructures are indicated by angle brackets. (A,B) is a choice. * is any one node. A full description of the format of structural description is given in [6].

10 WHAT WH-Attraction OB

#	U	ART	[NP	W	$\left\{\begin{array}{l}\text{PREP+[WH X]}_{NP}\\ \text{[WH X]}_{NP}\end{array}\right\}$ Y]$_S$	Z	#	
1	2	3	4	5	6	7	8	9 ===>
1	2	3	6+4	5	∅	7	8	9

The structural description above contains a choice; but notice that if the sentence is analyzable as the upper choice, then it is also analyzable as the lower one, The intention is that in the case where both structural descriptions can be matched, either one of the analyses is acceptable. This is precisely what the AAC parameter specifies. The same situation arises for their Question transformation.

We have also found the parameter AAC useful in the WH-Question transformation of Traugott's grammar of Old English [7]. There the problem is somewhat more difficult, since more than one element at a time may be questioned. The desired solution was achieved by the following pair of transformations:

```
TRANS WHA "WH-QUESTION" AACC OP.
SD % 1 Q % 2 NP % .
SC WH ALESE 2, ERASE 1.

TRANS WHA2 "WH-QUESTION" AAC OB.
SD % 1 Q % 2 NP % .
SC WH ALESE 2, ERASE 1.
```

The first of these transformations optionally inserts WH as the left sister of zero or more NP's in the sentence. If at least one WH is inserted the Q is erased so that WHA2 will fail. If no WH is inserted by the first transformation, then the OB transformation WHA2 will insert exactly one WH as left sister of a randomly selected NP .

The possibility of creating a special parameter so that this case could be handled by a single transformation was considered but was rejected since it seemed too special.

Keywords

The optional list of keywords which appears in the transformation identification is simply a technical device used to bypass applications of the analysis algorithm.[1/] Whenever a node is to be specified as the top of a search by the analysis algorithm it is first verified that all of the keywords are dominated by that node; if they are not, the analysis is assumed to have been tested and to have failed.

This completes the discussion of what is meant by invoking a transformation for a specified top node. We now discuss the specification of the top node for an analysis.

1/ This device was first used by Friedman in the SYNN programs at MITRE [8]; it was also used by IBM [5].

Specifying the top node for an analysis

The analysis algorithm which determines if the sentence tree matches the structural descriptionof a transformation is described in [6]. Before the analysis algorithm is applied, the control program must have determined both the transformation to be invoked and the top node of the subtree in which it is to be invoked.

Default option for top node

The sentence symbol (S) plays a special role in the specification of the top node. Unless the control program specifically calls for a top node which has some other label (which may be done using the IN-construct described below), the top node will always be a sentence symbol.

To illustrate this specification of top node, consider first a very simple control program consisting of one instruction:

(1) TRAN1

This program simply consists of the transformation name TRAN1. It is interpreted to mean that the transformation TRAN1 is to be invoked. Each time it is invoked the top node is (by default) a sentence symbol. The termination of the top node proceeds as follows:

1. A list is made of all the sentence symbols in the tree (see Diagram 1); this is the list of "marked S's". (If there are none, the program terminates.)

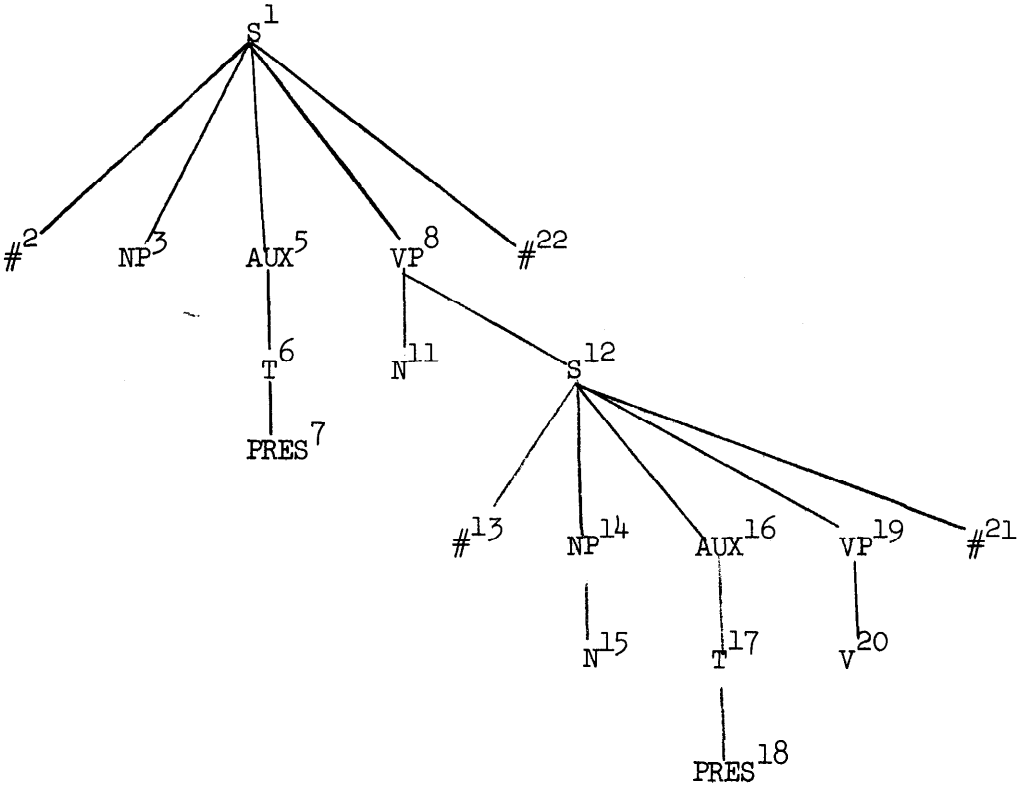
2. No special order is guaranteed for the marked S's.

3. Find the first marked S which dominates all the keywords of TRAN1.

4. Invoke TRAN1 using this S as top node.

5. Then, without repetition of any marked S , repeat this process until all marked S's have been used.

This completes the application of the control program (1).



For this tree the list of "marked S's is:

$\{S^1 \quad S^{12}\}$

The list of "marked S's" in Diagram 73 (below is:

$\{S_1 \quad S_2 \quad S_3\}$

Marked Sentence Symbols

Diagram 1

Invoking groups of transformations

We have seen above a program which invokes the single transformation TRAN1 . We now show how groups of transformations may be invoked.

The simplest way to invoke more than one transformation is to invoke a group of transformations by group number:

(2) II

The control program (2) will invoke all of the transformations which have group -number II. The identical effect is achieved by control program (3), where the transformations of group II are listed by name:

(3) TRAN5; TRAN6; TRAN7

and no other transformations belong to group II.

The determination of the top node is done one transformation at a time. Thus if there are two sentence symbols S_1 and S_2, the order of application will be:

Invoke TRAN5 at S_1;

Invoke TRAN5 at S_2;

Invoke TRAN6 at S_1;

Invoke TRAN6 at S_2;

Invoke TRAN7 at S_1;

Invoke TRAN7 at S_2.

The IN-construct

The simple specification of top node described above is of course inadequate in many cases. It must be possible to select as top node sentence symbols with special characteristics; such as lowest sentence, next-to-lowest sentence, top sentence, and so on. The facility for doing this is provided by the IN-construct. The basic idea of the IN-construct is that the analysis algorithm itself can be used to determine the specification of top nodes.

The form of the IN-construct is given by:

9.06 IN-instruction ::= IN transformation name (integer)
DO < control program>

The transformation name which occurs here may be the name of a special transformation which is invoked only for this purpose. (In this case it need not have a structural change.)

As an example, consider the control program:

(4) IN LOWESTS(1) DO < TRAN1 >

where the transformation LOWESTS is given by:

TRANS 0 LOWESTS III.
SD 1 S ¬ / < # % S < # % # > % # > , WHERE 1 DOM # .

The structural description of LOWESTS will be matched if the tree contains an S which dominates a boundary symbol (#), but which does not (¬) dominate another S which dominates boundary symbols.

(This corresponds exactly to the definition of lowest sentence given in Rosenbaum and Lochak [5].) The integer 1 can now be used to refer to this lowest snetence. Notice that LOWESTS has been given the group number III -- this is chosen to be different from all other group numbers in the transformations so that LOWESTS will never be invoked except in the IN-construct.

The control program (4) operates as follows:

First, LOWESTS is invoked (with top node determined as in the case of the control program (1) above). If the analysis is successful, the node designated by 1 is taken as the sole marked S for application of the sub-control program TRAN1 . After this sub-control program is completed, LOWFSTS is again invoked. For each repetetion a new lowest sentence must be found to correspond to the integer 1 . If a new lowest sentence is found, the sub-control program is repeated. The control program (4) terminates when no new lowest sentence is found.

Notice that the application of TRAN1 may change the tree so that sentence symbols which did not previously satisfy LOWESTS now do so.

When the IN-construct is applied to a group of transformations as in:

(5) IN TRAN1(1) DO < TRAN2; TRAN3; TRAN4 >

The single top node determined by TRAN1 is used for the subsequent three transformations. Note that the effect is not necessarily the same as the sequence of instructions:

(6) IN TRAN1(1) DO < TRAN2 >;

IN TRAN1(1) DO < TRAN3 >;

IN TRAN1(1) DO < TRAN4 >

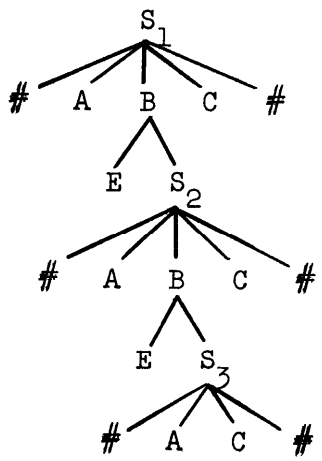
In (5) TRAN1 is invoked once, and the node corresponding to the integer 1 is taken as the top node for the three subsequent transformations, even though after TRAN2 the structural description of TRAN1 may no longer be satisfied by the tree. Thus the IN-construct allows us to select a top node on the basis of the tree structure at a particular time, and to continue to use this top node although the tree structure changes.

Example 1. The IBM Core Grammar

With the instructions which have been discussed so far we are able to construct a control program which corresponds to the IBM Core Grammar of Rosenbaum and Lochak [5, pages 28-32]. We first quote from the reference the description of the pattern cycling:

> The transformational component of the Core Grammar contains an ordered set of cyclic and post-cyclic transformational rules. The cyclic rules apply to a lowest sentence. A lowest sentence is an S boundary and X is a variable which does not contain #. In the diagram (73), S_3 meets the conditions of a lowest S .

(73)



> The cyclical transformational rules apply in sequence to lowest S's. Consider, for instance, the following set of cyclic rules in which the symbol X is a variable ranging over any structure at all.

(74)

a.

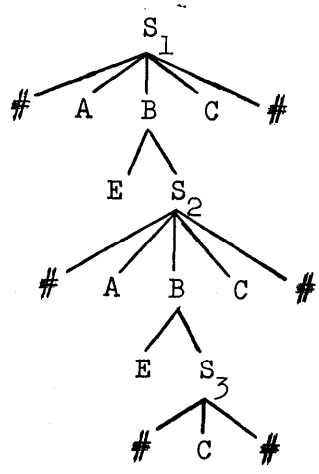
#	A	X	#	
1	2	3	4	==== >
1	∅	3	4	

b.

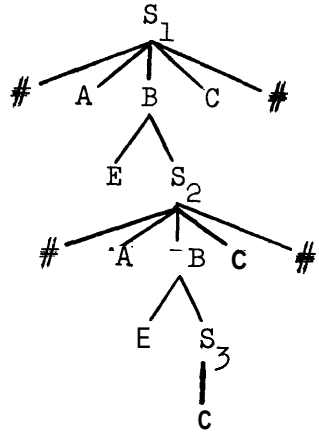
#	X	#	
1	2	3	mm-->
∅	2	∅	

Applying cyclically, the rules in (74) operate on S_3 in the P-marker given in(73) producing, sequentially, the P-markers(75) and(76).

(75)

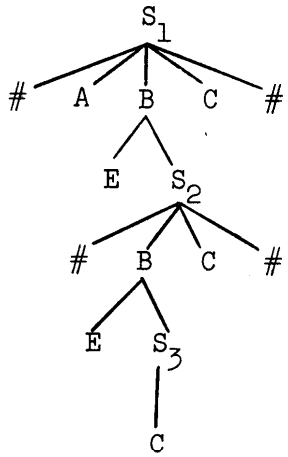


(76)

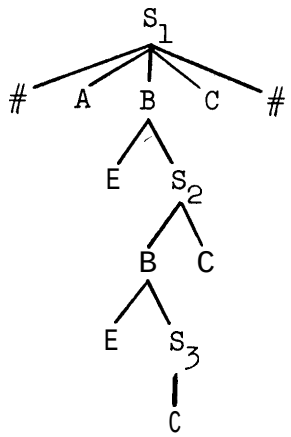


As the result of the application of the rules in (74), S_2 now meets the conditions of a lowest S and the cyclic rules apply again yielding the P-markers(77) and(78).

(77)

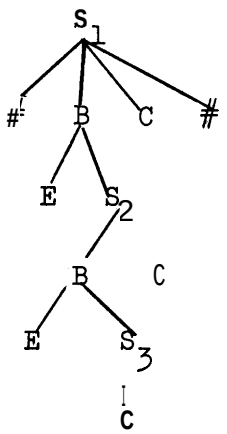


(78)

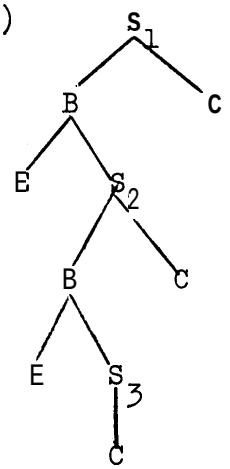


After this cycle Sl now meets the conditions of the lowest S and the cyclic rules apply once again yielding (79) and (80).

(79)



(80)



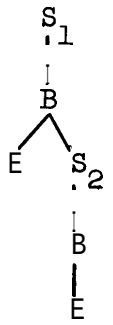
This application of the rules leaves a P-marker in which no S qualifiesas a lowest S . Hence, the transformational cycle is terminated.

The P-marker produced by the rules of the transformational cycle is the input to the post-cyclic transformational rules. A possible rule might be that in (81), which deletes every assistant of C in (80), thus generating the derived P-marker (82).

(81)

X	C	Y	
1	2	3	===>
1	∅	3	

(82)



The post-cyclic rules are ordered. The derivation is terminated after the applicability of the last post-cyclic rule has been tested.

The simple transformational component used in this example could be written in our system as:

```
TRANSFORMATIONS

    TRANS 0 LOWESTS III.

    SD 1 S ¬ / < # % S < # % # > % # > , WHERE 1 DOM #.

"CYCLIC  TRANSFORMATIONS"

    TRANSAI.

    SD  #  2  A  %  # .

    SC ERASE 2.

    TRANSBI.

    SD  1  #  %  3  # .

    SC ERASE 1, ERASE 3.
```

```
     "POST-CYCLIC TRANSFORMATION"
          TRANS C II.
          SD  %  2  C  % .
          SC ERASE 2.
CP   '*CONTROL PROGRAM"
          IN LOWESTS(l) DO < I >; II .
$END "END OF TRANSFORMATIONS"
```

When LOWESTS is invoked for (73) the top nodes S_1 and S_2 will fail to yield a successful analysis. S_3 will qualify as a lowest S . Transformation A produces (75); transformation B then produces (76). LOWESTS is then invoked again for each of S_1 which fails and then for S_2 which this time is successfully analyzed as a lowest S . Transformation A produces (77), transformation B produces (78), LOWESTS is again invoked for S_3 and fails. It is invoked again for S_1 and succeeds because S_1 is now the lowest S . Transformation A produces (79) and transformation B produces (80). Then LOWESTS is invoked for S_2 and S_3 and fails both times. LOWESTS is finally invoked for each of the three S's but no successful analysis is found. Hence the execution of the first instruction terminates.

The second instruction causes each of the transformations of group II (in this case there is only one) to be invoked once for each of the S's in the tree. Each time transformation C is invoked, one occurrence of C is deleted. The result is (82).

Order of instructions in a control program

In the example above it was sufficient to do the instructions in order without any branching in the program. The instructions are simply carried out in the order listed. It is clear, however, that one would like to be able to choose the next instruction on the basis of what has happened so far in the application of the control program. This facility is provided by if-instructions, go-instructions and labels.

CO-instructions and labels

The simplest change in the linear flow of control is provided by allowing transfer to a labeled instruction. Any instruction in the control program can be labeled by preceding it with a word (i.e., any sequence of letters and digits beginning with a letter) followed by a ":" . Control can be transfered to the instruction labeled say DO1 by a go-instruction GOT0 DO1. Thus, in the control program:

```
FIRST:  TRAN1; TRAN2; GOTOFIRST; TRAN3
```

The order of execution is TRAN1, TRAN2, TRAN1, TRAN2, TRAN1, ... This program is not recommended because it contains an infinite loop, but go-instructions can be combined with if-instructions to create sensible programs.

Conditional instructions

The form of a conditional instruction of if-instruction is given by:

9.07 if-instruction ::= IF instruction THEN go-instruction
opt [ELSE go-instruction]

or

IF TRAN2 THEN GOTO EMB ELSE GOTO CONJ

where EMB and CONJ are labels and TRAN1 and TRAN2 are transformation names.

The instruction between the IF and the THEN may be of any type. With each type of instruction there is an associated value. The simplest case is an instruction which is a transformation name: the value is true, if the transformation has been invoked and it applies (that is, the transformation's structural description is met); the value is false otherwise.

Table III below gives the value corresponding to each instruction type. (Some of these types have not yet been introduced.) The interpretation of an IF-instruction is that first the instruction (between the IF and the THEN) is performed. If the resulting value is true the GO-instruction after the THEN is performed, otherwise the GO-instruction after the ELSE is performed. (Just as in ALGOL.)

In using an IF-instruction it is important to note that in

IF T1 THEN T2

T1 will first be invoked for all of the current S's, and if it is successful at least once then T2 will be invoked for all current S's. Normally what is wanted is not the above, but conditional application within a given S . This can be achieved by using the IF-instruction

within an IN-construct, for example:

IN NEXTS(1) DO < IF T1 THEN T2 >

where the structural description for NEXTS is simply

SD $1 S % .

In this case the sentences will be considered one at a time, and the invocation of T2 in a particular sentence will be conditional on the previous success of T1 in that sentence.

A note on tree-pruning

Ross' "tree-pruning'" [9] is an example of a general convention for grammars which one might want to test in a computer system for transformational grammar. One way to handle tree-pruning is to include in the language an instruction which gives the list of node names for which it applies, as has been done by Gross [10]. If tree-pruning were to become generally accepted we would probably follow Gross' treatment of it. In the present system the tree-pruning convention can be simulated by constructing tree-pruning transformations, and inserting their calls at appropriate points in the control program, probably as conditional instructions as:

IF T1 THEN TREEPRUNE1

Example 2. A grammar of Swahili

In "A transformational grammar of Swahili" [11], Klevansky uses a control program in which each transformation is called by transformation name. The transformations QNANI, QNINI and QLINI are optional; conditional instructions are used to insure that at most one of them will be successfully applied.

```
CP   INSERTKU; FIXNEGCOP;
     PREAGV;
     NEGSUB;
     REL1; REL2;
     ANPRE1; ANPRE2;
     PREAGAV; FIXCOP;
     IF QNANI THEN GOTO E;
     IF QNINI THENGOTO E;
     IF QLINI THEN GOT0 E;
E:.
```

Example 3. Zwicky's proposal for control of cycling

Zwicky [12] has considered the following method of control of cycling:

a. Instances of S in a base tree are indexed as follows:
(1) Any instance of S that does not dominate an S receives the index 1 .
(2) Any instance of S that dominates other instances of S receives the index N+1 if (a) every dominated S is indexed, and (b) the maximum index of a dominated S is N.

b. On the N^{th} pass through the rules all subtrees dominated by an S with index N are operated upon, and no other subtrees are operated upon.

This control program can be expressed only with difficulty in our control language. The problem is that we have no convenient way of marking indices. The following program is an inelegant but accurate expression of Zwicky's scheme -- it uses inherent features INDEX1,..., INDEXN to mark indices. The maximum possible depth of a tree must be known beforehand; the program below works only up to depth 4 .

Four transformations are used to insert indices; four more are used in IN-constructs. Transformations INDEX1, ... INDEX4 insert feature specifications which correspond to the indices above:

```
TRANS INDEX1
SD % 1S %, WHERE 1 NDOM S .
SC |+ INDEX1| MERGEF 1 .

TRANS INDEX2
SD % 1S / < % S|+ INDEX1| % > .
SC |+ INDEX2| MERGEF 1 .

TRANS INDEX3
SD % 1S / < % S|+ INDEX2 | % > .
SC |+ INDEX31 MERGEF 1 , |+ INDEX21 ERASEF 1 .

TRANS INDEX4
SD % 1S / < % S|+ INDEX31 % > .
SC |+ INDEX41 MERGEF 1, |+ INDEX3 + INDEX2| ERASEF 1 .
```

Transformations FIRST, SECOND, ... FOURTH will associate the integer 1 with the appropriately indexed S's.

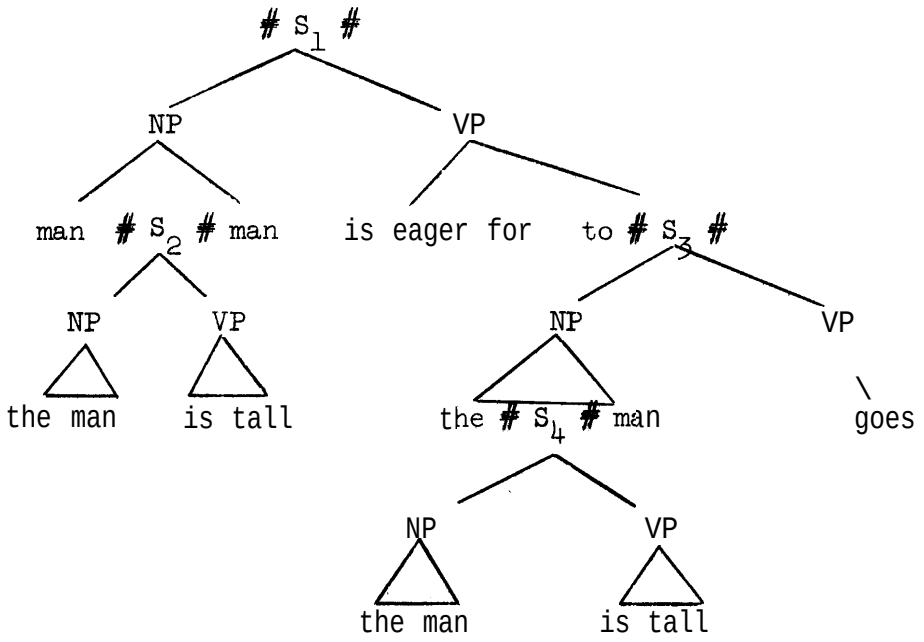
```
TRANS FIRST .
SD % 1 S |+ INDEX1| % .
SC |+ INDEX1| ERASEF 1 .

TRANS SECOND .
SD % 1 S |+ INDEX2| % .
SC |+ INDEX21 ERASEF 1 .

TRANS THIRD
SD % 1 S |+ INDEX3| % .
SC |+ INDEX3| ERASEF 1 .

TRANS FOURTH
SD % 1 S |+ INDEX4| % .
SC |+ INDEX41 ERASEF 1 .
```

If II is the group number for the embedding transformations, the control program can then be expressed as:

```
INDEX1; INDEX2; INDEX3; INDEX4;
IN FIRST(l) DO < II >;
IN SECOND(l) DO < II >;
IN THIRD(l) DO < II >;
IN FOURTH(l) DO < II >.
```

Note that the indices are erased when used; this will prevent them from interfering with other tests on features.

If we apply this to Zwicky's example:



the effect is as follows:

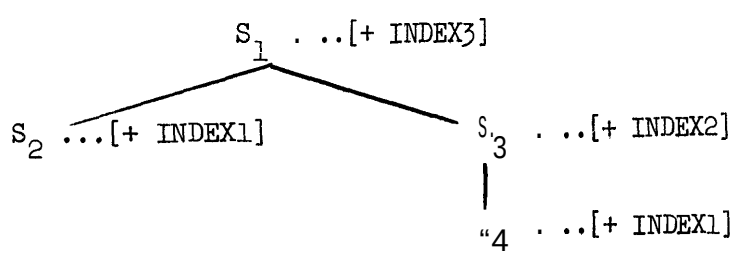
INDEX1 merges the feature specification + INDEX1 into the complex symbols associated with S_2 and S_4 .

INDEX2 merges the feature specification + INDEX2 into the complex symbols for S_1 and S_3 .

INDEX3 succeeds for S_1 only. It merges the feature specification + INDEX3 into the complex symbol and erases the feature + INDEX2 .

INDEX4 fails.

After these first four transformations the tree is (schematically):



The remaining instructions will now invoke the embedding transformations for S_2, S_4, S_3 and S_1 , as desired, and will delete all occurrences of the index features.

FLAG-instructions

FLAG-instructions provide a means for remembering which transformations and which groups have applied (i.e., which transformations and which groups have value true), In general, whenever a transformation name or group number appears the corresponding transformation or group is invoked and some value is produced. If the transformation or group appears within an IF-instruction this value may be tested, but the value is then discarded. The same holds true for the instructions within a RFT-instruction; the value is determined, the RPT-instruction is terminated or is continued, and the value is discarded. In order to be able to change the course of execution of the control program by remembering which transformations and which groups have the value true one FLAG's the appropriate transformations and groups.

The form of a FLAG-instruction is given by

flag instruction ::= flag name transformation list

flag name ::= FLAG opt [integer,]

transformation list ::= transformation element or < sclist [transformation element] >

transformation element ::= transformation name or group number

Flags are considered to be variables which may take on the values true and false. These variables are numbered from zero up. (Within the implementation of the control language ten flags numbered from zero to nine are provided.) Two type of FLAG-instructions exist: those with an integer following FLAG and those without. FLAG-instructions without an integer are taken to refer to FLAG 0.

The FLAG-instruction is interpreted as:

1. Discard any previous value or definition this flag may have had.
2. Define the named flag as consisting of the named transformations and/or groups.
3. Set the flag's value to false.

A flag's value remains false until one of the transformations which it represents or some transformation in one of the groups it represents is invoked. Then if the invoked transformation's structural description is met (and any structural changes are made) the flag's value is changed to true. The flag's value will remain true until the flag is redefined. For example, a flag (say Flag 5) which represents the transformations TRAN1, TRAN2 and TRAN3 might be defined by:

FLAG 5 < TRAN1 ; TRAN2 ; TRAN3 >

Flag 5 is set to false at the time this instruction is executed. Thereafter, if any of the above three transformations are invoked and do apply (that is, if any of the three transformations are invoked and their structural descriptions are met) Flag 5 will be set to true. Flag 5 will retain the value true until it is redefined. (If Flag 5 is redefined as TRAN1, TRAN2 and TRAN3 then its value is just reset to false.)

A flag which represents every transformation in group II and the transformation TRAN4 might be defined by:

FLAG 3 < II ; TRAN4 >

Flag 3 is set to false when this instruction is executed. Thereafter, if any transformation in group II or if TRAN4 is invoked and does apply Flag 3 will be set to true.

A flag which only represents the transformation TRAN5 is defined by:

FLAG 2 TRAN5

The value of a flag may be tested at any time through the use of an IF-instruction. For example,

IF FLAG 7 THEN GOT0 LABL3 ELSE GOT0 LABL4

If Flag 7 has value true then a transfer to LABL3 will be made; if Flag 7 has value false then a transfer to LABL4 will occur instead.

The RET-instruction

In order to repetitively invoke a transformation, group of transformations or control program one-may use a RFT-instruction. Two forms of RPT-instructions are provided: the first specifies a fixed maximum number of times the following instructions are to be executed; the second will continue to cycle through the instructions until each has the value false.

The form of the RPT-instruction is:

RPT opt [integer] < control program >

A RPT-instruction with an integer is interpreted as follows:

1. Set the RPT counter to the value of the integer.
2. Execute each element of the control program in the angular brackets in the normal manner.
3. If at least one of the elements had the value true, then decrease the RPT counter by one and if it is still greater than zero, go back to step 2.
4. If no element of the control program had the value true (or if the RPT counter has a value less than 1) terminate the RET-instruction.

For example:

RPT 5 < TRAN1 ; TRAN2 ; III >

will repeat the sequence: invoke transformation TRAN1, invoke transformation TRAN2, invoke every transformation in group III, until either

none of them apply or five iterations of the sequence have occurred.

An example of a RPT-instruction without an integer is:

RFT<II ; TRAN3 >

This instruction will invoke every transformation in group II and then invoke transformation TRAN3 and repeat until none of the transformations in group II apply and TRAN3 does not apply. Then the REP-instruction will terminate. Note that it is possible to create infinite loops with RPT-instructions.

RPT-instructions may include any arbitrary control program, and in particular they may include other REP-instructions, So,

RPT 4 < III ; RPT <IV > ; TRAN4 >

will invoke the transformations in group III, then invoke all transformations in group IV repetitively until none apply, then invoke transformation TRAN4, and this sequence will be repeated at most four times.

It is occasionally the case that transformations cannot be explicitly ordered. The following example is taken from Menzel, et. al. [13]. "The sentence John and Mary ran and will walk tomorrow, and sang and will dance tomorrow respectively. requires a derivation where secondary conjunction must precede the (conjunction transformation) schema, whereas John saw a movie and ate pizza and will run tomorrow. requires a derivation where the processes take place in the other order, the schema first and then secondary conjunction." The RF&instruction provides a means to specify both possibilities. If the conjunction schema is called group I and the secondary conjunction transformations are called

group II, then the following instruction will achieve the desired result:

RPT < I ; II >

This instruction will invoke all transformations in group I and then all transformations in group II and then will repeat the process until no transformations from either group apply. The linguist must insure, however, that the transformations in the two groups interact in such a way that if for a particular sentence the transformations in group II are to apply first, then no transformation in group I will apply.

The STOP-instruction

The STOP-instruction terminates the execution of a control program. STOP-instructions may appear at any point. In our implementation of the control language a STOP-instruction forces an output of the final tree, lists the transformations which have applied in the order in which they were invoked and reports how many instructions were executed.

A STOP-instruction need not appear within a control program -- when the terminal period of the control program is encountered the same effect is achieved.

Determining the value of an instruction

Each instruction of the control language has a value: true or false. These values are summarized in Table III below. In general, a value of true means that some change has been made to the tree (although not necessarily).

The simplest instruction is the name of a transformation. Such an instruction has value true if the structural description of the transformation is met at least once in the current tree and is false otherwise. If the transformation has a structural change, then a true value implies that this change has been made at least once.

Group numbers denote sets of transformations. The value of a group number is true if at least one of the transformations in its set has value true and is false otherwise.

Transformations and group numbers may be grouped together into a transformation list by enclosing them in angular brackets (see the syntax in Appendix B). Transformation lists may be used as the instruction part of an IF-instruction, for example. The value of a transformation list is true if any transformation or group number within the list is true and is false otherwise.

Control lists are lists of control instructions; their values are determined similarly to transformation lists: if any instruction within the list has value true then the control list also has value true, and the value is false otherwise.

CO-instructions, TRACE-instructions and STOP-instructions have no values. Within the computer implementation of the control language these instructions are given the value false.

REP-instructions take the value of the control program within them: if any instruction within the RPT-instruction is true then the value of the RPT-instruction is true, and it is false otherwise.

An IN-instruction takes the value true if the transformation named after the IN is true and is false otherwise. Note that this is equivalent to saying that an IN-instruction takes the value true if the list of instructions following the DO is executed at least once.

The FLAG-instruction has no value unless it is used within an IF-instruction. Then the value of the FLAG-instruction is the value of the flag denoted (see above).

The value of a control program is true if any instruction within the control program has value true and is false otherwise.

TRANSFORMATION ELEMENTS

transformation name
true if the structural description of the transformation is met
false if the structural description is not met

group number
true if any transformation within the group is true
false if all transformations within the group are false

transformation list
true if any transformation or group number within the angular brackets is true
false if all transformations and group numbers within the angular brackets are false

CONTROL ELEMENTS

repeat instruction
true if any -instruction within the angular brackets is true
false if all instructions within the angular brackets are false

in instruction
true if the transformation following the IN is true
false if the transformation following the IN is false

if instruction
true if any instruction between the IF and the THEN is true
false if all instructions between the IF and the THEN are false

flag instruction
has no value unless it is between the IF and THEN of an IF-instruction, then
true if the denoted flag currently has value true
false if the denoted flag currently has value false

go instruction
has no value

trace instruction
has no value

stop instruction
has no value

control program
true if any transformation element or control element within the list is true
false if all transformation elements and control elements within the list are false

TABLE III

Determination of the value of an instruction

Monitoring the application of transformations

Thus far the description of the control language has included only instructions which actually contribute to the determination of the output of the transformational component in the linguistic sense. The control language has been implemented as part of a computer system for transformational grammar; it therefore also contains instructions which monitor the application of transformations and control the amount and type of computer output which is produced.

Trace instructions

The simplest monitoring instruction is the trace-instruction TREE which causes the tree to be output. For example, if the control program for the IBM Core Grammar is changed to:

```
IN LOWESTS(1) DO < I ; TREE > ; II
```

the tree will be output after group I has been invoked for a lowest S . The results in the extended example given above will be that trees (76), (78) and (80) are output. (Tree (82), the final result, is automatically output without special instructions.) If we had wished to see only the final result(80) of the cyclic transformations, we could have written:

```
IN LOWESTS(l) DO < I > ; TREE; II
```

In testing a transformational grammar, one frequently is more interested in some transformations than in others. The language provides instructions which will-enable transformations and groups of

transformations to be "traced", so that more information is provided on their operation. The trace-instruction TRACE is followed by a transformation name or a group number and by a trace-specification which must be BEFORE TEST or AFTER FAILURE or AFTER SUCCESS or AFTER CHANGE. The trace begins when the TRACE is encountered and it is terminated at a corresponding UNTRACE. Whenever a transformation which is being traced is to be invoked, the corresponding trace-specification is examined and the sentence tree is output at the appropriate point in the invocation process.

TRACE BEFORE TEST will output the current tree after invoking the named transformation and after satisfying all specified keywords but before testing the structural description.

TRACE AFTER SUCCESS will output the current tree after invoking the named transfromation and finding the structural description (and keywords) met.

TRACE AFTER FAILURE will output the current tree after invoking the named transformation and finding the keywords met but the structural description not met.

TRACE AFTER CHANGE will output the current tree after invoking the named transformation and making the structural change (if one is specified).

Any number of transformations may be traced at one time, and any combination of trace types may be on for a given transformation at one time. For example,

TRACE TRAN5 AFTER SUCCESS

will force an output of the current tree each time TRAN5 is invoked and its structural description is satisfied. The output will be made just before making the structural change specified within TRAN5.

```
TRACE< I ;TRAN6 > BEFORE TEST ;
TRACE I AFTER CHANGE
```

will force an output of the current tree each time a transformation in group I is invoked and each time transformation TRAN6 is invoked. The output will occur just after the appropriate keywords are found but before the structural description is tested. In addition, each time a transformation in group I applies the current tree will again be output. This output will occur just after the structural change has been made.

ACKNOWLEDGEMENT

We wish to thank Thomas H. Bredt, Robert W. Doran, Theodore S. Martner and Barbara H. Partee for ideas which have been incorporated in the control language.

COMPLETE SYNTAX FOR TRANSFORMATIONAL GRAMMAR

0.01 TRANSFORMATIONAL GRAMMAR ::= PHRASE STRUCTURE LEXICON TRANSFORMATIONS $END

1.01 TREE SPECIFICATION ::= TREE opt[, clist[WORD TREE]]
1.02 TREE ::= NODE opt[COMPLEX SYMBOL] opt[[list[TREE]]]
1.03 NODE ::= WORD or SENTENCE SYMBOL or BOUNDARYSYMBOL
1.04 SENTENCE SYMBOL ::= S
1.05 BOUND&Y SYMBOL ::= #

2.01 STRUCTURAL DESCRIPTION ::= STRUCTURAL ANALYSIS, opt[, WHERE RESTRICTION] .
2.02 STRUCTURAL-ANALYSIS ::= list[TERM]
2.03 TERM ::= opt[INTEGER] STRUCTURE or opt[INTEGER] CHOICE or SKIP
2.04 STRUCTURE ::= ELEMENT opt[COMPLEX SYMBOL] opt[opt[¬] opt[/] ⟨ STRUCTURAL ANALYSIS ⟩]
2.05 ELEMENT ::= NODE or * or
2.06 CHOICE ::= (clist[STRUCTURAL ANALYSIS])
2.07 SKIP ::= %

3.01 RESTRICTION ::= booleancombination[CONDITION]
3.02 CONDITION ::= UNARY CONDITION or BINARY CONDITION
3.03 UNARY CONDITION ::= UNARY RELATION INTEGER
3.04 BINARY CONDITION ::= INTEGER BINARY TREE RELATION NODE DESIGNATOR or
INTEGER BINARY COMPLEX RELATION COMPLEX SYMBOL DESIGNATOR
3.05 NODE DESIGNATOR ::= INTEGER or NODE
3.06 COMPLEX SYMBOL DESIGNATOR ::= COMPLEX SYMBOL or INTEGER
3.07 UNARY RELATION ::= TRM or NTRM or NUL or NNUL or DIF or NDIF
3.08 BINARY TREE RELATION ::= EQ or NEQ or DOM or NDOM or DOMS or NDOMS or DOMBY or NDOMBY
3.09 BINARY COMPLEX RELATION ::= INC1 or NINC1 or INC2 or NINC2 or CSEQ or NCSEQ or NDST
or NNDST or COMP or NCOMP

4.01 COMPLEX SYMBOL ::= | list[FEATURE SPECIFICATION] |
4.02 FEATURE SPECIFICATION ::= VALUE FEATURE
4.03 FEATURE ::= CATEGORY FEATURE or INHERENT FEATURE or CONTEXTUAL FEATURE or RULE FEATURE
4.04 CATEGORY FEATURE ::= CATEGORY
4.05 CATEGORY ::= WORD
4.06 INHERENT FEATURE ::= WORD
4.07 RULE FEATURE ::= TRANSFORMATION NAME
4.08 CONTEXTUAL FEATURE ::= CONTEXTUAL FEATURE LABEL or CONTEXTUAL FEATURE DESCRIPTION
4.09 CONTEXTUAL FEATURE DESCRIPTION ::= ⟨ STRUCTURE opt[, WHERE RESTRICTION] ⟩
4.10 VALUE ::= + or - or *

5.01 STRUCTURAL CHANGE ::= clist[CHANGE INSTRUCTION]
5.02 CHANGE INSTRUCTION ::= CHANGE or CONDITIONAL CHANGE
5.03 CONDITIONAL CHANGE ::= IF ⟨ RESTRICTION ⟩ THEN ⟨ STRUCTURAL CHANGE ⟩
opt[ELSE ⟨ STRUCTURAL CHANGE ⟩]
5.04 CHANGE ::= UNARY OPERATOR INTEGER or
TREE DESIGNATOR BINARY TREE OPERATOR INTEGER or
COMPLEX SYMBOL DESIGNATOR BINARY COMPLEX OPERATOR INTEGER
or COMPLEX SYMBOL DESIGNATOR TERNARY COMPLEX OPERATOR INTEGER INTEGER
5.05 COMPLEX SYMBOL DESIGNATOR ::= COMPLEX SYMBOL or INTEGER
5.06 TREE DESIGNATOR ::= (TREE) or INTEGER or NODE
5.07 BINARY TREE OPERATOR ::= ADLAD or ALADE or ADLADI or ALADEI or ADFID or AFIDE or
ADRIS or ARISE or ADRISI or ARISE1 or ADLES or ALESE or ADLESI or ALESEI
or ADRIA or ARIAE or SUBST or SUBSE or SUBSTI or SUBSEI
5.08 BINARY COMPLEX OPERATOR ::= ERASEF or MERGEF or SAVEF
5.09 UNARY OPERATOR ::= ERASE or ERASE1
5.10 TERNARY COMPLEX OPERATOR ::= MOVEF

```
6.01    PHRASE STRUCTURE ::= PHRASESTRUCTURE list( PHRASE STRUCTURE RULE ) $END
6.02    PHRASE STRUCTURE RULE ::= RULE LEFT = RULE RIGHT .
6.03    RULE LEFT ::= NODE
6.04    RULE RIGHT ::= NODE or list( RULE RIGHT ) or ( list( RULE RIGHT ) ) or ( clist( RULE RIGHT ) )

7.01    LEXICON ::= LEXICON PRELEXICON LEXICAL ENTRIES $END
7.02    PRELEXICON ::= FEATURE DEFINITIONS opt( REDUNDANCY RULES )
7.03    FEATURE DEFINITIONS ::= CATEGORY DEFINITIONS opt( INHERENT DEFINITIONS ) opt( CONTEXTUAL DEFINITIONS )
7.04    CATEGORY DEFINITIONS ::= CATEGORY list < CATEGORY FEATURE > .
7.05    INHERENT DEFINITIONS ::= INHERENT list( INHERENT FEATURE ) .
7.06    CONTEXTUAL DEFINITIONS ::= CONTEXTUAL clist( CONTEXTUAL DEFINITION ) .
7.07    CONTEXTUAL DEFINITION ::= CONTEXTUAL FEATURE LABEL = CONTEXTUAL FEATURE DESCRIPTION
7.08    CONTEXTUAL FEATURE LABEL : := WORD
7.03    REDUNDANCY RULES ::= RULES clist( REDUNDANCY RULE ) .
7.10    REDUNDANCY RULE ::= COMPLEX SYMBOL => COMPLEX SYMBOL
7.11    LEXICAL ENTRIES ::= ENTRIES list( LEXICAL ENTRY ) .
7.12    LEXICAL ENTRY ::= list( VOCABULARY WORD ) list( COMPLEX SYMBOL )
7.13    VOCABULARY WORD : := WORD

8.01    TRANSFORMATIONS : := TRANSFORMATIONS list( TRANSFORMATION ) CP CONTROL PROGRAM . $END
8.02    TRANSFORMATION : := TRANS IDENTIFICATION SD STRUCTURAL DESCRIPTION opt( SC STRUCTURAL CHANGE . )
8.03    IDENTIFICATION opt( INTEGER ) TRANSFORMATION NAME opt( list( PARAMETER ) ) opt( KEYWORDS ) .
8.04    PARAMETER ::= GROUP NUMBER or OPTIONALITY OR REPETITION
8.05    GROUP NUMBER : := I or II or III or IV or V or VI or VII
8.06    OPTIONALITY ::= OB or OP
8.07    REPETITION ::= AC or ACAC or AACC or AAC
8.08    KEYWORDS ::= ( list( NODE ) )

9.01    CONTROL PROGRAM ::= sclist( opt( LABEL : ) INSTRUCTION )
9.02    LABEL ::= WORD
9.33    INSTRUCTION ::= RPT INSTRUCTION or IN INSTRUCTION or IF INSTRUCTION
                        or GO INSTRUCTION or TRACE INSTRUCTION or STOP INSTRUCTION
                        or T INSTRUCTION or < sclist( INSTRUCTION ) >
9.04    T INSTRUCTION ::= TRANSFORMATION NAME or GROUP NUMBER
9.05    RPT INSTRUCTION ::= RPT opt( INTEGER ) < CONTROL PROGRAM >
9.06    IN INSTRUCTION : := IN TRANSFORMATION NAME ( INTEGER ) DO < CONTROL PROGRAM >
9.07    IF INSTRUCTION ::= IF INSTRUCTION THEN GO INSTRUCTION opt( ELSE GO INSTRUCTION )
9.08    GO INSTRUCTION ::= GO TO LABEL
9.09    TRACE INSTRUCTION ::= TRACE T INSTRUCTION T R A C E SPECIFICATION or UNTRACE T INSTRUCTION or TREE
9.10    TRACE SPECIFICATION ::= BEFORE TEST or AFTER FAILURE or AFTER SUCCESS or AFTER CHANGE
9.11    STOP INSTRUCTION ::= STOP
```

APPENDIX B

CONTROL PROGRAM SYNTAX

The syntax given below is purely descriptive (as is the syntax given in Appendix A). However, the control program syntax has been translated into a precedence syntax suitable for use by a parser. The operation of the control program in our implementation is determined by this translated syntax (see Pollack [1]).

```
CONTROL-PROGRAM  ::=          CONTROL-PROGRAM1 .
CONTROL-PROGRAM  ::=          SCLIST [ CONTROL-INSTRUCTION ]
CONTROL-INSTRUCTION  ::=      LABEL CONTROL-INSTRTJCTION OR
                              CONTROL-INSTRUCTION LABEL
                              INSTRUCTION
LABEL ::=                     WORD : LABEL
                              WORD :
INSTRUCTION ::=               CONTROL-ELEMENT OR
                              TRANSFORMATION-ELEMENT OR
                              CONTROL-LIST
CONTROL-LIST  ::=             < SCLIST [ INSTRUCTION ] >
CONTROL-ELEMENT ::=           REPEAT-INSTRUCTION  OR
                              IN-INSTRUCTION  OR
                              IF-INSTRUCTION OR
                              FLAG-INSTRUCTION OR
                              GO-INSTRUCTION OR
                              TRACE-INSTRUCTION OR
                              STOP-INSTRUCTION
TRANSFORMATION-ELEMENT ::=    TRANSFORMATION-NAME OR
                              GROUP-NUMBER
REPEAT-INSTRUCTION  ::=       RPT INTEGER < CONTROL-PROGRAM1 > OR
                              RPT < CONTROL-PROGRAM1 >
```

```
IN-INSTRUCTION ::=          IN TRANSFORMATION-NAME ( INTEGER )
                                 DO < CONTROL-PROGRAM1 >
IF-INSTRUCTION ::=          IF INSTRUCTION THEN GO-INSTRUCTION
                                 OPT [ ELSE GO-INSTRUCTION ]
FLAG-INSTRUCTION ::=        FLAG-NAME TRANSFORMATION-LIST
FLAG-NAME ::=               FLAG OPT [ INTEGER]
GO-INSTRUCTION ::=          GO TO WORD OR
                            GOT0 WORD
TRACE-INSTRUCTION ::=       TRACE TRANSFORMATION-LIST SPECIFICATION OR
                            UNTRACE TRANSFORMATION-LST OR
                            TREE
SPECIFICATION ::=           BEFORE TEST OR
                            AFTER SUCCESS OR
                            AFTER FAILURE OR
                            AFTER CHANGE
STOP-INSTRUCTION ::=        STOP OR

TRANSFORMATION-LIST ::=     TRANSFORMATION-ELEMENT OR
                            < SCLIST [ TRANSFORMATION-ELEMENT ] >
```

REFERENCES

[1] Pollack, B. W. The Control Program and Associated Subroutines. m-28, Computer Science Department, Stanford University (June 1968).

[2] Fillmore, C. J. The Position of Embedding Transformations in a Grammar. Word, 19 (1963), 208-231.

[3] Lees, R. B. A Grammar of English Nominalizations. Supplement to International J. Amer. Linguistics, Baltimore (1960).

[4] Friedman, J. and Doran, R. W. A Formal Syntax for Transformational Grammar. CS-95, AF-24, Computer Science Department, Stanford University (March 1968).

[5] Rosenbaum, P. and Lochak, D. The IBM Core Grammar of English. in Lieberman, D. (Ed.), Specification and Utilization of a Transformational Grammar, AFCRL-66-270 (1966).

[6] Friedman; J. and Martner, T. S. Analysis in Transformational Grammar. AF-34, Computer Science Department, Stanford University (September 1968).

[7] Friedman, J. Computer Experiments in Transformational Grammar II: Traugott's Grammar of Alfredian Prose. AF-23, Computer Science Department, Stanford University (February 1968).

[8] Friedman, J. SYNN, an Experimantal Analysis Program for Transformational Grammars. WP-229, the MITRE Corporation (1965).

[9] Ross, J. R. A Proposed Rule of Tree-pruning. NSF-17, Computation Laboratory, Harvard University (1966), IV-i-18.

[10] Gross, L. N. A Computer Program for Testing Grammars On-Line. Mimeographed (1968).

[11] Klevansky, L. Computer Experiments in Transformational Grammar VI: Swahili. AF-32, Computer Science Department, Stanford University (June 1968).

[12] Zwicky, A. M. On the Ordering of Embedding Transformations. Mimeographed handout, meeting of the Linguistic Society of America (Summer 1966).

[13] Menzel, P., Shopen, T., and Partee, B. H. Rule Ordering: Preliminary Report. UCLA Working Paper #1 (October 1967).