CS - 115

PROGRAMMER'S MANUAL
FOR
A COMPUTER SYSTEM FOR TRANSFORMATIONAL GRAMMAR

by

Joyce Friedman
Thomas H. Bredt
Robert W. Doran
Theodore S. Martner
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 - 36

CS - 115

PROGRAMMERS MANUAL

FOR

A COMPUTER SYSTEM FOR TRANSFORMATIONAL GRAMMAR

by

Joyce Friedman*

Thomas H. Bredt

Robert W. Doran**

Theodore S. Martner

Bary W. Pollack

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

** Present address: Department of Mathematics
City University, London, ENGLAND

Abstract

This volume provides programming notes on a computer system for transformational grammar. The important ideas of the system have been presented in a series of reports which are listed in Appendix B; this document is the description of the system as a program. It is intended for programmers who might wish to maintain, modify or extend the system.

PROGRAMMING CREDIT

The principal programmers for each set of programs are listed below.

MAIN - Friedman and Martner

Free-field input/output - Doran

Trees - Doran and Friedman

Grammar input - Bredt, Doran (PSGINN), Friedman

Phrase-structure generation - Bredt, Friedman, John H. Gilman, Alan C. Tucker

Lexical insertion - Bredt

Analysis - Doran and Friedman (CXIN), and Martner (ANTEST - replacing an early version by Doran)

Restrictions - Pollack

Structural change - Bredt, Friedman, Barbara Jackson

Complex symbol operations - Friedman

Control program - Pollack

Table of Contents

Table of Contents (Continued)

Table of Figures

1. INTRODUCTION

This Manual is written by and for programmers. Its purpose is to make the code of the computer system for transformational grammar more readily understandable to programmers who wish to maintain and use the system, or to modify and extend it. Section 2 is a short outline of the subroutine structure of the system. It is followed in Section 3 by more detailed descriptions of the subroutines. Sections 4 and 5 are listings of the COMMON blocks and BLOCK DATA statements, respectively. Section 6 discusses possible extensions to the system.

The programs are written in FORTRAN IV for the IBM 360/67 compiled under FORTRAN H, OPT=2, under O.S. There are approximately 9000 lines of FORTRAN code; the compiled code, with storage areas, requires approximately 300,000 bytes of storage.

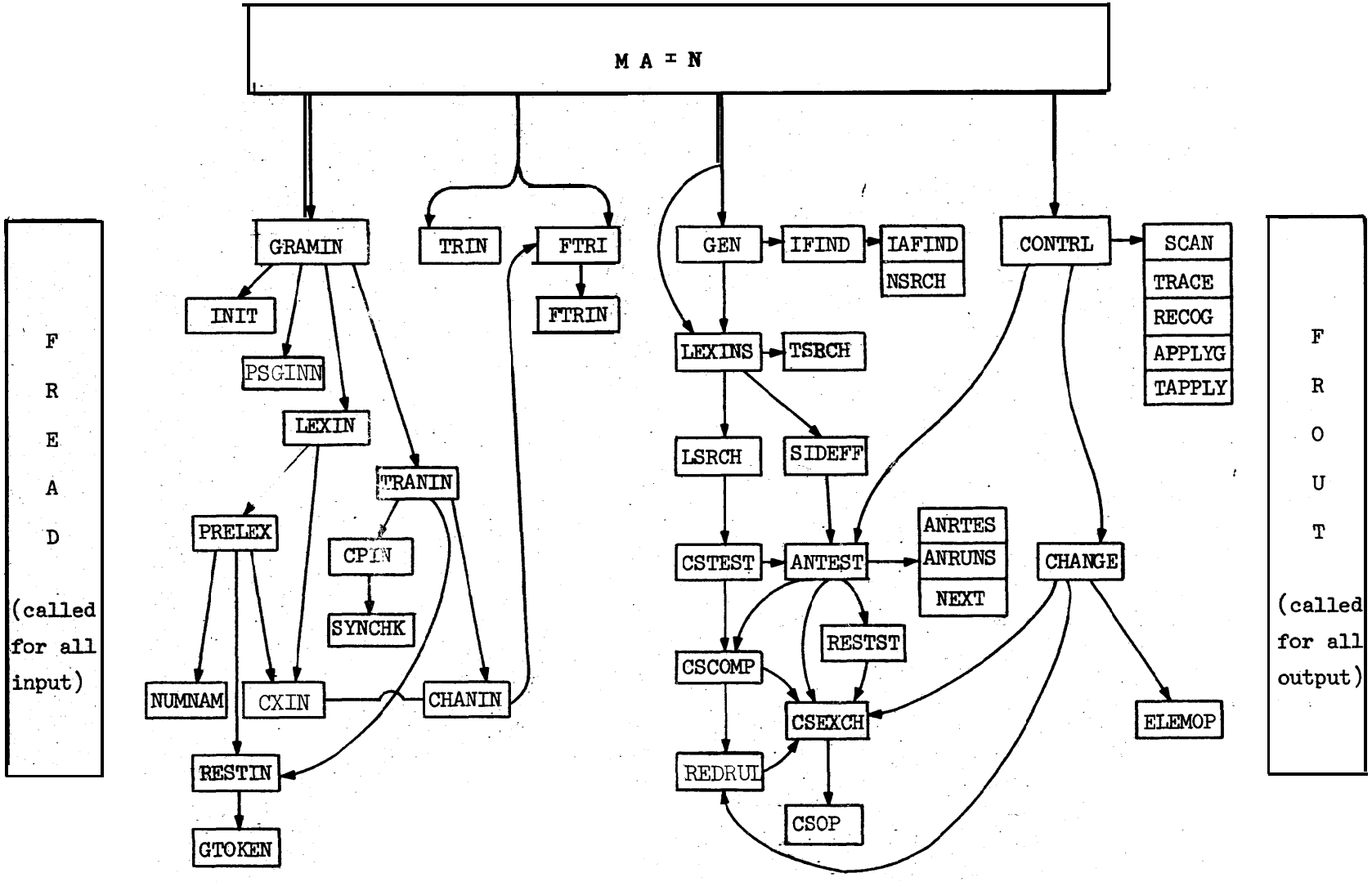
The inputs to the system consist of

1. a grammar (described by the formal syntax of AF-24*)
2. a one-line driver for the MAIN program (see Section 2.1)
3. input trees or skeletons (see 2.3 and 2.5).

Extended examples are given in AF-33 (CS-108).

A simplified schematic diagram of the basic structure of the system is given in Figure 1.1. Arrows go from calling routine to called subroutine.

* References on the system are listed in Appendix B below.



(Figure 1.1 Schematic Program Structure)

Programming Conventions

Input/Output

Almost all of the input to the system is handled by the free-field input/output package (FREEIO). The only exception to this is the alternative fixed-field tree input format. Likewise, most of the output is handled by FREEIO, with exceptions in certain cases of tabular debugging output and fixed-field trees.

Error messages

A uniform convention for error messages is used throughout the system. The standard form is

ERROR. Subroutine name. Message

Messages of the form

WARNING. Subroutine name. Message

are occasionally issued when a strong possibility of error exists, but an internal correction has been made.

Output files

System output is written on several different logical units. The minimum output for a standard run is placed on unit 6. Unit 7 contains additional general output useful for a more detailed study of the run. Units 8, 9 and 10 contain output for programmers concerned with ANALYSIS, RESTRICTIONS, and CONTROL, respectively.

2. SUBROUTINE STRUCTURE

In this section we list the subroutines of the system. For each subroutine a brief discussion is given of its role. Further discussion of each subroutine is given in the corresponding parts of Section 3.

2.1 Main program

Routine	Type	Role
MAIN	main	MAIN reads the directions for the current run. The input is in the form $\$MAIN\ \{\begin{matrix}TRIN\\FTRIN\end{matrix}\}((n)\{\begin{matrix}GEN\\LEX\end{matrix}\})(TRAN)$.

2.2 Free field input-output

Routines	Entries	Type	Role
FREAD(ØNE)		R*8	Free-field read. Returns a word or special character.
	INITLZ		Initializes FREAD, KØUTWD and /MAINCM/ .
FRØUT(ISTART,L1,...,L6)		SR	Free-field output of KSUMP from ISTART on.
KØUTWD(WØRD,LENGTH)		R*8	Returns an abbreviated word to FREAD
	EXPND(KTS,WØRD LENGTH)		Expands an abbreviated word to a long word.
	KEYØUT		Puts abbreviated words into KSUMP.
	LNGØUT		Outputs table of abbreviated words and long words.
/ CNSTCM/			INTEGER*2 constants.
/FCSTCM/			REAL*8 constants

2.3 Trees

Routine / Entries	Type	Role
TRIN	SR	Inputs fixed-field tree
TROUT(NQ1,NQ2)	SR	Outputs TREE starting at node NQ1 . If NQ2 = -1, outputs the number for each node.
/Z/		Short, miscellaneous block data, includes NS, NERROR.
FTRI(ARG)	I*2	Calls FTRIN with arguments for input to TREE if ARG = 1, or for addition to CHTREE if ARG = 2 .
FTRIN(FTREE,TREE,CLIST, MTREE,MCLIST,KA,KB, ISTART,FWORD)	I*2	Inputs free-field tree. Returns pointer to root of tree.
FTROUT(TOP,PJ)	SR	Free-field output of subtree headed by TOP. PJ = 1 punches output.

2.4 Grammar input

Routine / Entries	Type	Role
INIT	SR	Initializes everything.
GRAMIN	SR	Reads in the phrase structure, lexicon and transformations.
PSGINN	SR	Inputs phrase structure rules. Expands, orders, and stores them.
PSGSMP		Puts expanded phrase structure rules into KSUMP.
PSGOUT	SR	Outputs tables of the phrase structure rules.
LEXIN	SR	Reads in a lexicon - calls PRELEX.
PRELEX	SR	Reads in the prelexicon.

2.4 Grammar input (continued)

Routine	Entries	Type	Role
NUMNAM(FWORD,ARG) NAMEIN(FWORD,ARG)		I"2	Returns the number for the feature FWORD. Stores FWORD as the name of the contextual feature with the number ARG.
LEXSMP		SR	Copies the lexicon into KSUMP.
	LEXOUT		Outputs the internal tables for the lexicon.
TRANIN		SR	Reads in the transformations.
	TRANØU		Outputs the table of transformations.

2.5 Phrase structure generation

Routine	Entries	Type	Role
GEN		SR	Generates a directed random tree,
IFIND(M,N)		I*2	Subroutine for GEN. Returns 1, 0, -1 if M must, may, or cannot dominate N . IFIND(N,N)=1 .
IAFIND(I)		R*8	Called by IFIND. Returns I, if I is a terminal symbol. Otherwise, returns position of first rule which expands I .
NSRCH(N)		I*2	Called by IFIND. Returns position of last rule which introduces symbol N, 0 if none.

2.6 Lexical insert ion

Routine	Entries	Type	Role
LEXINS		SR	Does lexical insertion.

2.6 Lexical insertion (continued)

Routine / Entries	Type	Role
LSRCH(CATNØ,NØDE,WØRD,TCS)	SR	Finds entry of category CATNØ suitable for insertion at NØDE which has complex symbol TCS and WØRD (if non-blank).
TSRCH(CAT,NØDE)		Searches tree for lexical category (CAT) - returns node number in NØDE. Returns NØDE = 0 if there are none. Keep calling, TSRCH keeps searching.
CSTEST(NØDE,M,N)	I*2	Returns number of compatible complex symbol if complex symbol N is suitable for insertion at NØDE which already has complex symbol M .
SIDEFF(NODE,N)		Does side-effects for each contextual feature in complex symbol N .
CSCØMP(M,N,IND)	I*2	Compatibility test for complex symbols. If M or N > 0 they are node numbers. If M or N < 0, they are complex symbol numbers, If IND = 1, use nondistinctness test. If IND = 2, use inclusion-l test, If IND = 3 return pointer to compatible complex symbol found for node M .

2.7 Analysis

Routine / Entries	Type	Role
CXIN(KDUMMY)	I"2	Reads in a complex symbol and returns its number. If KDUMMY = 1, the complex symbol is first expanded by the redundancy rules.
SLFEAT(KDUMMY)	I*2	Reads in a contextual feature and returns its number.
ANALIN(KDUMMY)	I*2	Reads in a structural analysis and returns its number.
ANALØU(I)		Writes out the internal representation of structural analysis I.

2.7 Analysis (continued)

Routine / Entries	Type	Role
CSSUMP		Copies a complex symbol into KSUMP.
CSOUT		Outputs the interval tables for complex symbols.
ANTEST(TRANNØ,TREETP,ANALNØ)	L*1	Evaluates the structual description of transformation TRANNØ or the structural analysis ANALNØ in the subtree headed by TREETP.
ANRTES(PØSN)	L*1	Tests restrictions on the node assigned to PØSN. If PØSN > 0, test complex symbol also.
ANRUNS(PØSN)	SR	Unsets restrictions on node PØSN. Also sets NUMNØD and ANNØDE to zero.
NEXT(HERE,TØP,SIGN)	SR	Resets HERE to the next node after HERE.

2.8 Restrictions

Routine / Entries	Type	Role
RESTIN(ØNE)	I*2	Reads restriction . or restriction > ; returns its number.
RESTST(I,PØSN)	L*1	Tests and sets restriction designated by I or CREST. If PØSN = 0 resets the restriction first.
RESTUN(I,PØSN)	SR	Unsets restriction I . If PØSN = 0, sets CREST = I and completely resets restriction I .
RESTPR(I)	SR	Outputs tables for restriction I . I = 0 outputs all.
GTØKEN(SYM)	SR	Returns a token, i.e.,, a logical operator or condition.
/RESTCM/		Constants and storage.

2.9 Structual change

Routine / Entries	Type	Role
CHANIN	I*2 . .	Reads a structural change and returns a pointer to it.
CHANTY		Tidies up after all changes read,
CHANØU		Outputs the table of structural change.
ELEMØP(NWØRD,NQ1,NQ2)	SR	Applies operator NWØRD to arguments NQ1, NQ2 .
ERASE(NQ2) SUBSE(NQ1,NQ2) ALADE(NQ1,NQ2) AFIDE(NQ1,NQ2) ARIAE(NQ1,NQ2) ALESE(NQ1,NQ2) ARISE(NQ1,NQ2) SUBST(NQ1,NQ2) ADRIS(NQ1,NQ2) ADLES(NQ1,NQ2) ADLAD(NQ1,NQ2) ADRIA(NQ1,NQ2) ADFID(NQ1,NQ2)		Entries for specific changes. IBM operations are also done by ELEMØP, but do not have individual entries.
CHANGE(ID,CNRNUM)	SR	Performs the structural change of transformation ID using the CNRNUM-th analysis found by ANTEST.

2.10 Complex symbol operations

Routine / Entries	Type	Role
REDRUL(M)	I*2	Returns the number of the complex symbol obtained on expansion of complex symbol M using the redundancy rules.
CSØP(TYPE,A,N,M)	I*2	If TYPE = 1 returns pointer to new complex symbol created by doing operation A on complex symbols N and M . If TYPE= 2 returns value of test A on complex symbols N, M ,

2.10 Complex symbol operations (continued)

Routine	Entries,	Type	Role
CSEXCH(N,M)		I*2	. . Sets up calls to tests and operations in CSØP.
	CSEQ(N,M)		
	{CSINC1(N,M) CXINC1(N,M)		
	CSINC2(N,M)		
	CSNDST(N,M)		
	{CSMERG(N,M) MERGE1(N,M)		
	CSMERR(N,M)		
	CSERAS(N,M)		
	CSSAVE(N,M)		

2.11 Control program

Routine	Entries	Type	Role
CPIN		SR	Inputs a control program. Check syntax.
CØNTRL		SR	Interprets the control program.
SCAN(DMY)		I*2	Inputs next symbol and generates token.
	SCAN1(DMY)	I*2	Inputs next symbol and generates token.
TRACE(TNØ,TIM,ANFG)		SR	Generates TRACE output,
TAPPLY		SR	Invokes a transformation.
	APPLY1(TNØ)	SR	For IN-transformations,
	APPLY(TNØ)	SR	General
	OUTTRN	SR	Outputs the list of tranformations which have applied.
	APPLYI(TNØ)	SR	When inside an IN construct.

2.11 Control program (continued)

Routine / Entries	Type	Role
SYNCHK	L*1	Checks syntax of the control program.
RECOG	L*1	Generates token and recognizes symbols.
APPLYG(GNØ)	SR	Invokes transformations of group GNØ.
/CPCØM/		Storage.
/SYNCM/		Storage.

3. SUBROUTINE DESCRIPTIONS

In this section we describe individually each subroutine of the system. The reader will immediately notice that the level of detail in the program descriptions varies greatly. In general, where the programming is straight-forward we have simply described inputs, internal storage, and outputs, On the other hand, when more complicated algorithms are involved we have gone into considerable detail in order to try to make the programs easy to follow.

3.1 Main program

The subroutines.of the system may be combined in various ways by changing the main program (MAIN) . The current main program is given below. It accepts an input in the form

$$\$MAIN\{\begin{matrix}TRIN\\FTRIN\end{matrix}\}((n)\{\begin{matrix}GEN\\LEX\end{matrix}\})(TRAN) \quad .$$

The program first reads in a grammar. Then a tree is read by TRIN or FTRIN . The integer n controls the number of times this tree will then be used. If GEN is specified, the input is treated as a skeleton to be expanded by the generation routine GEN (which then calls the lexical insertion program (LEXINS)) . If LEX is specified, the input is assumed to be a complete phrase structure tree and lexical insertion is called directly. If TRAN is specified, the transformations will then be applied.

After n (or 1 if n is not specified) iterations a new tree is input. The program terminates when there are no more inputs.

MAIN PROGRAM

```
C      *****     MAIN II  *****    SUBROUTINE *****      *****
C      MAIN PROGRAM FOR TESTING GEN - 360/67  -  7/19/67 -
C      VERSION FOR TESTING GEN - II   8/18/67
C      INPUT CARD SEQUENCE
C              INITIALIZATION IS DONE BY GRAMIN
C              PSG, LEXICON, AND TRANSFORMATIONS ARE READ IN ARBITRA
C              ORDER BY GRAMIN
C              GRAMIN RETURNS WHEN $      IS READ.
C              THEN ...
C              READ IN (WHETHER TO CALL FTRIN OR TRIN),
C              ITERATIONS FOR GEN, CALL TO GEN, LEX, CONTRL.
       IMPLICIT INTEGER*2 (A-Z)
       COMMON /Z/ LANK,NXXX,NSS,NS,NAND,NOR,NANDOR,NERROR
       REAL*8     LANK,NXXX,NSS,NS,NAND,NOR,NANDOR
       COMMON /TREECM/ FTREE,TREE,CLIST,MTREE,MCLIST,NCODE
       REAL*8 FTREE(400)
       INTEGER*2 TREE(400,6),CLIST(400),MTREE,MCLIST,NCODE(10)
       COMMON/SKELCM/FISKEL,ISKEL,SKLIST,ISKELT,MSKLST
       REAL*8 FISKEL(200)
       INTEGER*2 ISKEL(200,6),SKLIST(200),ISKELT,MSKLST
       COMMON /CSCM/
      1 ANALWD,CSLIST(4,2000),ANALPT(500),ANALWP(2000),ANALST(2000),
      2 TEMPAN(2000),SLCTPT(400),ANALTP,SLCTTP,CSFG,CSFRPT,ANALWT
       REAL*8 ANALWD(200)
       COMMON/MAINCM/ CHRTR,KSUMP,ISUMP,NCHRTR
       REAL*8 CHRTR,KSUMP(2000)
       COMMON/ORDCM/ NUM,ISPEC,ORDFL,NUMFL
       LOGICAL*1 ORDFL,NUMFL
       REAL*8 ZMAIN/'$MAIN'/,ZFTRIN/'FTRIN'/,ZTRIN/'TRIN'/,
      1  ZGEN/'GEN'/,ZLEX/'LEX'/,ZTRAN/'TRAN'/,FREAD
       INTEGER*2 ONE/1/
       N1=1
       N2=-1
       N20=20
       WRITE(6,1700)
       NERROR=0
       CALL GRAMIN
       CSFSAV=CSFRPT
C  WE HAVE THE $. IF IT ISN'T US, QUIT NOW.
       IF (CHRTR .NE. ZMAIN)  STOP
```

```
C  IT IS US.  NOW IS IT TRIN OR FTRIN FOR TREES?
       NGEN = 1
       KKTREE = 0
       KKTRAN = 0
       KKGEN = 0
       CHRTR = FREAD(ONE)
       IF (CHRTR .EQ. ZFTRIN)  GO TO 4
       IF (CHRTR .EQ. ZTRIN)  GO TO 8
       GO TO 9
  4    KKTREE = 1
C  NOW LOOK FOR GEN FACTOR OR MARKERS FOR CALLS TO GEN, LEXINS, CONTRL.
  8    CHRTR = FREAD(ONE)
  9    IF (NUMFL) GOTO 14
       IF (ISPEC. NE. 0) GOTO 15
  10   IF (CHRTR .EQ. ZGEN)  GO TO 11
       IF (CHRTR .EQ. ZLEX)  GO TO 12
       IF (CHRTR .EQ. ZTRAN)  GO TO 13
       WRITE (6,1800) CHRTR
       GO TO 8
C  REMINDER TO CALL GEN
  11   KKGEN = 1
       GO TO 8
C  REMINDER TO CALL LEXINS
  12   IF (KKGEN .NE. 1)  KKGEN = 2
       GO TO 8
C  REMINDER TO CALL CONTRL
  13   KKTRAN = 1
       GO TO 8
  14   NGEN = CHRTR
       GO TO 8
  15   CSFRPT=CSFSAV
       IF (KKTREE .EQ. 0)  CALL TRIN
       IF (KKTREE .EQ. 1)  CALL FTRI(ONE)
       SAVCSF = CSFRPT
       IF (ORDFL) RETURN
       IF(NERROR.NE.0) WRITE(6,1600) NERROR
       NERROR=0
       DO 20 I=1,MTREE
       FISKEL(I)=FTREE(I)
       DO 20 K=1,6
20     ISKEL(I,K)=TREE(I,K)
       ISKELT=MTREE
       DO 30 I=1,MCLIST
30     SKLIST(I)=CLIST(I)
       MSKLST=MCLIST
```

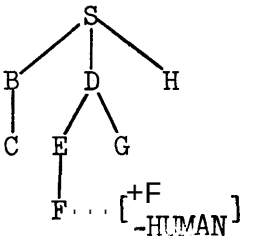
3.2 Free field input-output

A full description of the free-field input/output subroutine package is given by R. W. Doran in AF-14 (CS-79) to which the reader is referred. These programs are independent subroutines and may be used outside of the present system.

3.3 Trees

```
COMMON/TREECM/FTREE,TREE,CLIST,MTREE,MCLIST
REAL*8      FTREE(400)
INTEGER*2 TREE(400,6),CLIST(400), MTREE,MCLIST
```

Example



	FTREE	TREE col. 2	3	4	6	CLIST	
1	S	0	1	3	0	1	2
2	B	1	5	5	0	2	5
3	E	5	6	6	0	3	4
4	H	1	0	0	0	4	
5	D	1	8	9	0	5	6
6	C	2	0	0	0	6	7
7	F	3	0	0	10	7	
8	G	5	0	0	0	8	3
						9	8

MTREE=8

MCLIST=9

Discussion

FTREE is a list of the labels of the nodes of the tree. The numbering of the nodes is arbitrary except that the root of the tree is always node 1. TREE is a six-column array parallel to @TREE. Columns 1 and 5 are used for work-space. Col. 2 is a pointer to the parent of the node (0 for the root). Cal. 6 is a pointer to the complex symbol attached to the node (in CSLIST), or 0 if none. Notice that the format thus allows complex symbols to be attached to any node of the tree. Columns 3 and 4 point to the first and last positions in CLIST which contain the daughters of the node, CLIST gives the daughters in left-to-right order. MTREE is the current length of FTREE and TREE; MCLIST is the current length of CLIST.

The format is a compromise between case of search and ease of change. The list of node names in FTREE allows a quick search for a particular node name. The entries in TREE and FTREE need not be contiguous and CLIST likewise can be expanded without recompression. (The example shows CLIST as it might look after various changes have taken place).

The COMMON block /SKELCM/ is structured like /TREECM/ ; in the common block /CHANCM/ FCHTRE, CHTREE, CHCLIS, NCHT, NCHCL correspond to FTREE, TREE, CLIST, MTREE and MCLIST .

/Z/

Block data /Z/ contains a few miscellaneous parameters used in the system. The most important of these are NSS and NS which both continue the sentence symbol 'S' and NERROR which can be used to communicate an error condition. Some of the other parameters in /Z/ are no longer used.

External formats

The system has both fixed field and free field external representations for trees. TRIN and TROUT are the fixed field input and output routines; FTRIN and FTROUT are the corresponding free field routines.

TRIN and TROUT, fixed-field tree I/O

TRIN and TROUT(I,J) input and output trees to and from the internal format described above. The external format is immediately readable and readily punched. output may be printed or punched and may begin at any selected node of the tree. A substitution feature allows subtrees to be treated separately.

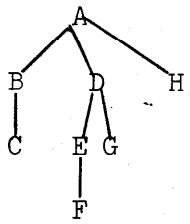
Figure 3.3.1 gives an example of the printed output of TROUT(1,0) . Figure 3.3.2 is a listing of cards produced by TROUT(1,1) . The input to TRIN is the same as the output of TROUT(1,1) .

Basic external format

The basic format is a representation in which the daughters of a node in field L appear in field L+1 . The first (left-most) daughter is in the same card as its parent. Daughters to the right appear on lower cards. Thus

```
A   B   C
    D   E   F
        G
    H
```

represents the tree



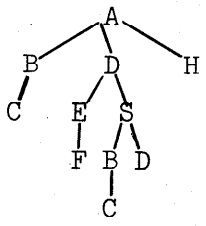
Substitution feature

A potential difficulty in the basic format is that the depth of a tree may exceed the maximum number of fields allowed. A substitution feature avoids this by replacement of a dummy node by a subtree. This is indicated by the use of a substitution card with XXX in the first field and the dummy node in the second. Thus, the input cards

EXAMPLE

```
A B      C
     D   E   F
         G
     H
-XXX G
S B      C
     D
(blank)
```

represent the tree



```
                                . . .
BASE  25      IS THE AUTO THE CONVEYANCE THAT THE HORSE WAS AT THAT TIME
SS    +
      S   PRE   Q
          NP  DET   ART   THE
                        PSAR  ADM
                  N     NCM   NCT
                  NU    SG
              AUX AUXA  TNS   PRES
              VP  BE
                  PRED  NP    OET   ART   THE
                                    PSAR  AOM
                              N     NCM   NCT
                              NU    SG
                              SS    +
                                    S     NP    OET   ART   THE
                                                      PSAR  AOM
                                          N     NCM   NCT
                                          NU    SG
                                    AUX   AUXA  TNS   PST
                                    VP    BE
                                          PREO  NP    OET   ART   OEM   WH
                                                                    THAT
                                                                    NBR
                                                            PSAR  AOM
                                                      N     NCM   NCT
                                                      NU    SG
                                          TIM   TM    AT
                                                      NPI   OET   ART   OEM   THAT
                                                                          NBR
                                                                    PSARAOM
                                                            TIME
                                                      NU    SG
                                    +
      +
+     Q     THE   ADM   NCT   SG    PRES BE   THE   AOM   NCT   SG    +     THE   AOM   NCT   SG    PST   BE    WH
THAT  NBR   ACM   NCT   SG    AT    THAT  NBR   AOM   TIME  SG    +     +
```

Figure 3.3.1 Example of Printed Tree Output

```
BASE  25      IS THE AUTO THE CONVEYANCETHAT THE HORSE WAS AT THAT TIME
SS    +
      S       PRE    Q
              NP     DET    ART    THE
                                   PSAR   ADM
                     N      NCM    NCT
                     NU     SG
              AUX    AUXA   TNS    PRES
              VP     BE
                     PRED   NP     DET    ART    THE
                                                PSAR   ADM
                                   N      NCM    NCT
                                   NU     SG
                                   SUB01
      +
XXX   SUB01
SS    +
      S       NP     DET    ART    THE
                                   PSAR   ADM
                     N      NCM    NCT
                     NU     SG
              AUX    AUXA   TNS    PST
              VP     BE
                     PRED   NP     DET    ART    DEM    WH
                                                THAT
                                                NBR
                                          PSAR   ADM
                                   N      NCM    NCT
                                   NU     SG
                     TIM    TM     AT
                                   NPI    DET    ART    DEM    THAT
                                                        NBR
                                                 PSAR   ADM
                                          TIME
                                   NU     SG
      +
                                                                BLANK
```

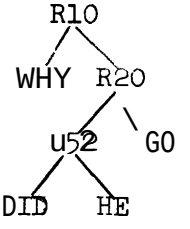
FIGURE 3.3.2 LISTING OF PUNCHED TREE OUTPUT

The only restriction on the use of the substitution feature on input is that a unique name be given to the dummy node for which the subtree is to be substituted. Substitution will be made only for the first occurrence of that name.

In output, substitution is made for all occurrences of the sentence symbol which occur at or beyond the field MAXSS . Thus, MAXSS should be set, on the basis of the grammars being processed, so that the maximum depth of a kernel tree does not exceed MAXJ - MAXSS, where MAXJ is the number of fields. If MAXSS is set too high to avoid overflow, substitution will be made for the rightmost field. For the MITRE Junior grammar the values of MAXSS = 5 and 13, for punch and print respectively, are acceptable for all but a few trees.

Alternative formats

Jane Robinson's PARSE program* uses an output format for binary trees in which the first daughter appears to the right and the second daughter, if any, appears below. Robinson's trees contain numbers associated with each node and the lines of the tree are put in. A simple example is the tree



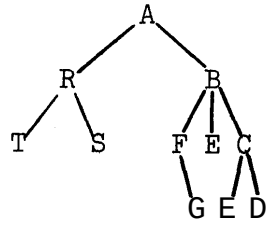
* J. Robinson, Preliminary codes and rules for the automatic parsing of English, RAND RM-3339-PR, 1962.

which is output as

```
1108    ****    0130
R10             WHY
*
*
1107    ****    9327    ****    9321
R20             U52             DID
*               *
*               *
*               2005
*               HE
*
*
1001
GO
```

F. Blair* uses an input form which is inverse to ours since the rightmost daughter occurs on the highest card. His input is free field except that all daughters of a given node must begin in the same column.

As an example, the tree



can be input as

```
A   B   C   D
            E
        E
        F   G
    R       S
            T
```

* D. Lieberman, Design of a grammar tester, and F. Blair, Programming of the grammar tester, in D. Lieberman, Ed. Specification and Utilization of a Transformational Grammar, AFCRL-66-270, 1966.

This limited use of free field seems to us to be no easier to punch than fixed field. Its major advantage is that, since his program is in LISP, atoms.of arbitrary length can be used. Blair's output is the standard LISP S-expression form.

Discussion of the format

The printed version of this format is easy to read; it seems to us at least as intuitive as the alternatives discussed above. It is not hard to punch an input tree directly from the graphic representation, although it may be easier to use coding sheets.

Corrections and modifications to a tree are very simple to make.

An interesting by-product of the form is that a small set of card types can be used to obtain all the trees possible within a given grammar. For the IBM Core Grammar* a set of 42 basic card types would suffice to give all the possible kernel trees. About ten additional card types would suffice to take advantage of the substitution feature for embedded sub-trees. Additional punching would be required only for input of lexical items.

TROUT

Output of trees is controlled by the two parameters of TROUT(I,J) . The first parameter controls the starting point of the output. If I=1 the entire tree will be output, preceded by its title and followed by the terminal string. If I is not 1, the sub-tree headed by node number I will be output. This feature can be useful in testing transformations, with I set in turn to each of the nodes of the proper

*P. Rosenbaum and D. Lochak, The IBM Core Grammar of English, Ibid.

analysis. If I is negative, an error indication is given; if 0, $NIL is output; if greater than MTREE, it is reset to 1 .

The parameter J controls the punch option and numbering. If J = 0 the tree is printed only; if J = 1 it is printed and punched offline; if J = -1 each node name will be preceded by the node number.

The parameters are protected so that the call is essentially by value.

TRIN

For input by TRIN the tree must be preceded by a title card. The first card must have a node in field 1 . The format is 12A6 . The tree is terminated by a blank card.

Conversion of decks

Conversion to this format of trees in another format is simple. They can be read in by the old input routine and then punched out by TROUT(1,1) . The output deck is ready for input to TRIN .

Error checks

If TROUT is called with I negative, an error (301) results,

In TRIN error 210 occurs when the dummy node for which a substitution is to be made cannot be found in the tree. A final check on the input tree detects trees in which the root is not the sentence symbol (error 90), or which have multiple roots (error 93). Otherwise the routine assumes that the input tree is good. It is therefore recommended that TRIN be immediately followed by a call to a checking routine to verify that the tree is in fact a correct tree of its grammar.

The programs are set up for 6-character words. If 8-character words are desired, the format statements, as well as the values of MAXSS and MAXJ must be changed. In the case of a-character words, it would be desirable to use the full 80-column card, so the format statements must be changed accordingly. The word BLANK can then no longer be written on the final blank card as it is in the 72-column version of TROUT(1,1).

FTRIN, FTRI, and FTROUT, free-field tree I/O

Free-field tree inputs are primarily used to read into /TREECM/ and /CHANCM/ . To avoid the necessity for specifying all the parameters in these cases, FTRI can be used. FTRI(l) calls FTRIN with the correct parameters for reading a tree into /TREECM/ . FTRI(2) calls FTRIN to add a tree to CHTREE in /CHANCM/ .

FTRIN(FTREE, TREE, CLIST, MTREE, KA, KB, ISTART, FWORD) reads a free-field tree into a block strutured like /TREECM/ in which KA is the maximum size of FTREE and TREE, and KB the size of CLIST . If ISTART = 0, an entire tree will be read, if ISTART = 1, FWORD will be taken as the root of the tree.

In the FTRIN code a single subtree is stored using a recursive algorithm with a pushdown. KNPUSH(I) contains a pointer to the parent of the Ith level of the subtree in array TREE and the daughters of this parent so far found are from MPUSH(KMPUSH(I)) to MPUSH(KMPUSH(I+1)-1). The recursion is depth first and whenever it is known that all the daughters of a given node have been found they are dumped into CLIST . Substitution is done by finding the node to be substituted for (pointers to terminal nodes are stored in NODES(50)) and then initializing the pushdown by retrieving the left sisters of the substituted node and placing them on

the pushdown. The substituted sub-tree is then expanded until a period or comma is encountered whence the right most sisters of the substituted node are retrieved from CLIST and then all of the new list of daughters stored back in CLIST . This causes waste space in CLIST and TREE, but there is no waste space if there is no substitution.

FTROUT(TOP,PJ) outputs the subtree of TREE which has root TOP . PJ = 1 causes it to also punch the output. The code for FTROUT is a very simple recursion. KMPUSH(I) tells us where in TREE the Ith level of the tree is and. KNPUSH(I) points to the daughter of KMPUSH(I) in CLIST with which we are dealing.

3.4 Grammar input

This section discusses the input routines for grammars and for the three components of a grammar.

INIT, initialization

Subroutine INIT initializes everything in the system, including the free-field input routine. It is called by GRAMIN .

GRAMIN, grammar input

GRAMIN first initializes the system by calling INIT and then reads in a grammar, Since each of the major components begins with an identifying word and ends with $END, GRAMIN is able to read either a full grammar or just one or two components. GRAMIN returns when it encounters the order $ which ends the grammar, leaving the order itself to be read by the MAIN program.

PSGINN, phrase structure grammar input

PSGINN reads compactly written context-free phrase structure rules from the input stream, expands and orders them and stores them in the rule storage area /PSGCM/ .

Storage of phrase structure rules

```
COMMON/PSGCM/NSGA1, NSGC, NSGA2, NSGB, KA, KB, KC
REAL*8 NSGA1(200), NSGC(2000)
INTEGER*2 NSGA2(200), NSGB(300), KA, KB, KC
```

Example

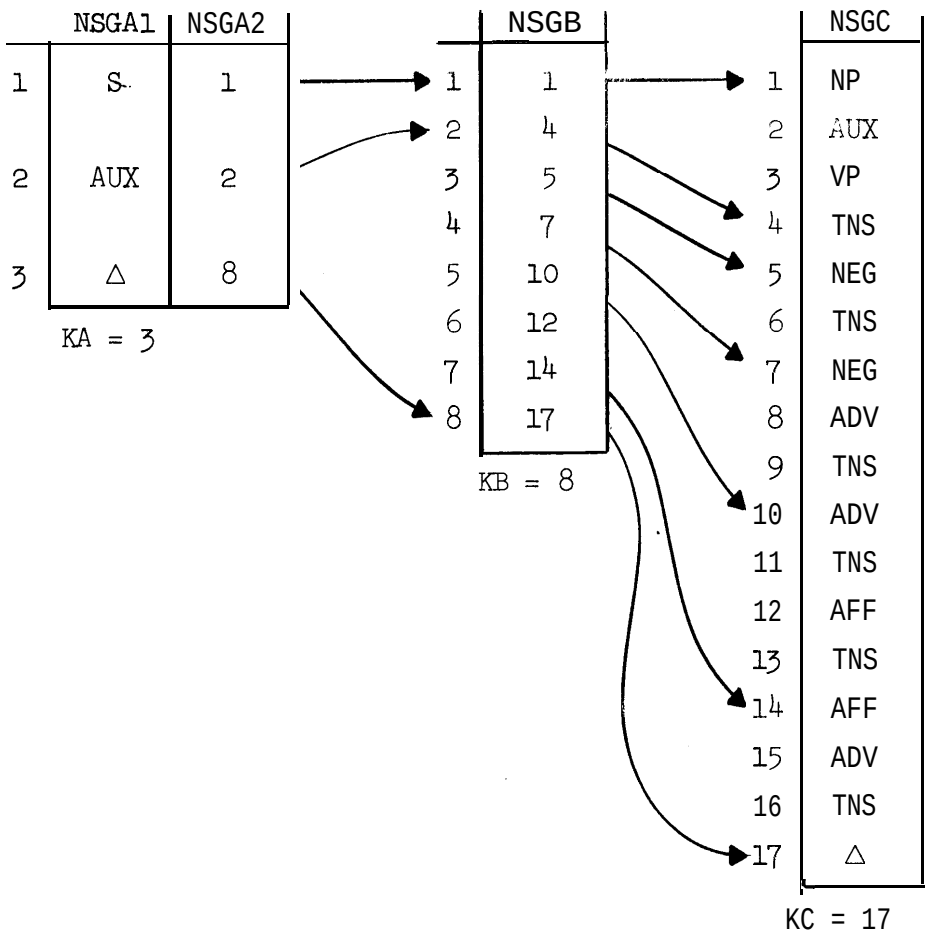
Input

```
  S = NP AUX VP.
AUX = ((NEG, AFF))(ADV)TNS.
```

Expanded form

```
      S =NPAUXVP.
AUX = TNS,
      NEG TNS,
      NEG ADV TNS, --
      ADV TNS,
      AFF TNS,
      AFF ADV TNS.
```

Internal form



Discussion of internal form

NSGA1 contains left-hand sides of rules.

NSGC contains right-hand sides of the (expanded) subrules,

NSGB(j) contains a pointer to the position in NSGC of the first word of the jth subrule.

NSGA2(i) contains a pointer to the position in NSGB which points to the beginning of the first subrule of rule i .

KA is the current length of NSGA1 and NSGA2

= number of rules + 1

KB is the current length of NSGB

= number of subrules + 1

KC is the current length of NSGC

= total number of words on RHS's + 1

Algorithms for Expanding and Ordering P.S. Rules

Task

To read a set of compactly written Phrase Structure Rules, to expand, order, and store them,

e.g., the rule Aux = ((NEG,AFF))(ADV)TNS. will be expanded to

AUX = NEG ADV TNS,
NEG TNS ,
AFF ADV TNS,
AFF TNS ,
ADV TNS ,
TNS .

then ordered algebraically to AUX = TNS ,

NEG TNS ,

NEG ADV TNS,

ADV TNS ,

AFF TNS ,

AFF ADV TNS .

and then stored as described above.

The overall logic of the program PSGINN is illustrated in Fig. 3.4.1.

The main (numbered romantically) steps are now described.

I/. The expansion of rules was broken down into 2 steps. An 'abbreviated node list" (i.e., a compactly written part of a rule, e.g., "(PAST, PRES)" in the rule "TNS = (PAST, PRES) is first of all scanned and a table of linkages built up and then expanded using the linkage table. Nodes are stored in array "NODES" and linkages in the 2 dimensional "LINKS" e.g., (NEG , AFF))(ADV)TNS is firstly converted into:

NODES	LINKS	1.2.3.4
1.	,	2 3 4 5
2. NEG	2	4 5
3. AFF	3	4 5
4. ADV	4	5
5. TNS	5	0

Every expanded node list may be obtained by chasing pointers until a 0 is found.

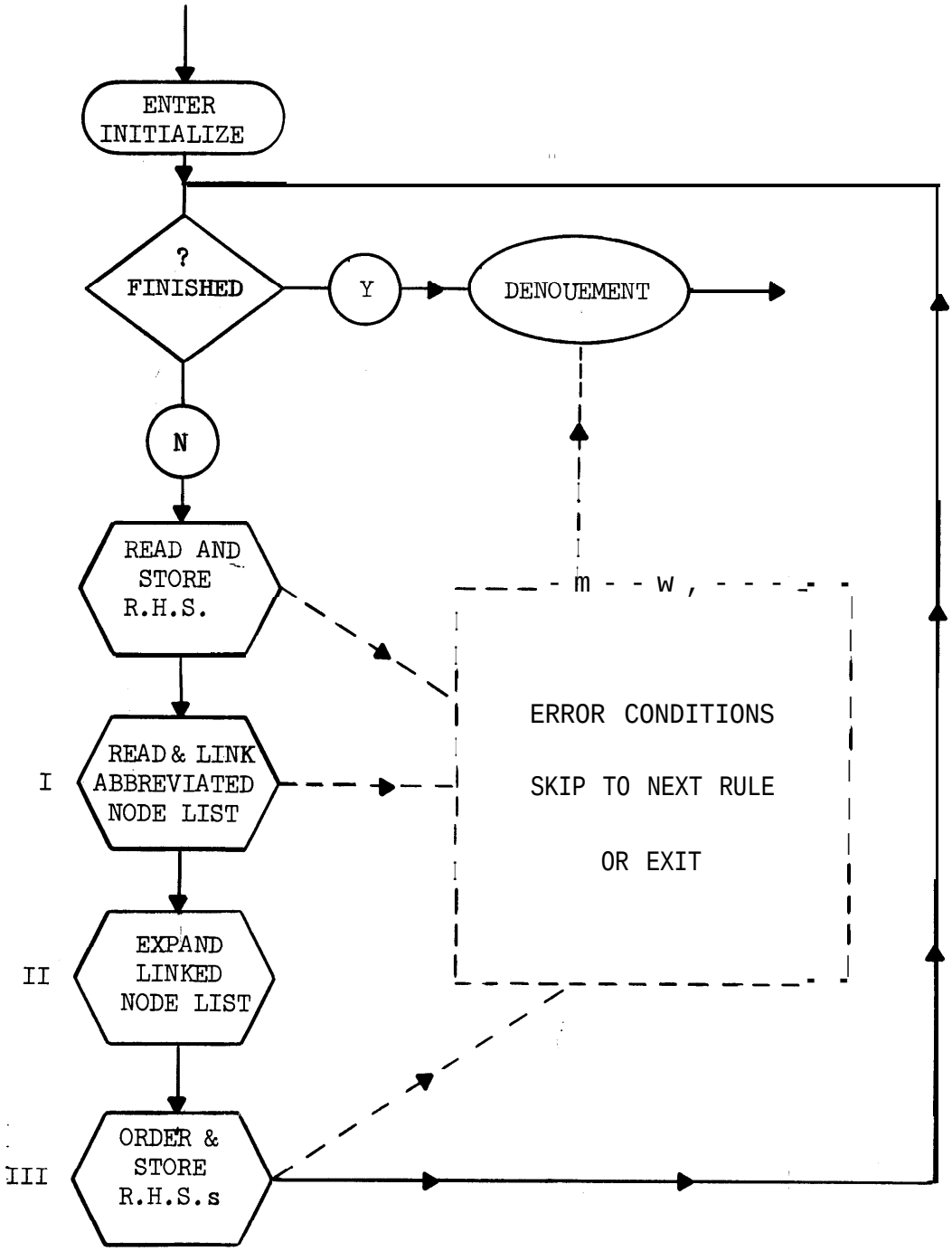


Figure 3.4.1 Macro-flow Diagram of Expansion, Order and Storage Algorithm.

e.g. LINKS(1,2) → NODES(3) = AFF

LINKS(3,1) → NODES(4) = ADV

LINKS(4,1) → NODES(5) = TNS

LINKS(5,1) → 0

so 'AFF ADV TNS' is one of the expanded node lists.

This first linkage section is the most complex. It was found possible to expand an abbreviated node list using a simulated pushdown stack, only having immediate knowledge of the character being scanned at present and the one previous.

There are 2 basic types of linkage between nodes in an abbreviated node list: -

a/. A-links as between A and B, A and C of A(B,C)D

b/. B-links as between B and D, C and D of A(B,C)D

A-links are links into parentheses, B links are links out of parentheses.

The idea of the algorithm of part I is then to scan the abbreviated node list, when parentheses are opened storing the A-type links for that level of the pushdown and when closing parentheses fixing the B-type links. Of course, links are also stored and -fixed when commas or nodes are encountered.

Nodes are stored linearly in NODES(I) when they are encountered, INODES points to the last node stored. LINKS are stored in LINKS(I,J), there being KLINKS(I) links in the Ith row.

The push down is rather complex. IPUSH indicates the level of operation. At level I the A-links are stored in MPUSH from

KMPUSH(I) to KMPUSH(I+1)-1 and the B-links in NPUSH from KNPUSH(I) to KNPUSH(I+1)-1 . IMPUSH and INPUSH point to the tops of MPUSH and NPUSH respectively.

KTR holds the character being scanned.

ISPEC indicates the type of the scanned character, ILAST the type of the previous character scanned.

we will go through the linkage of our example "((NM;, AFF))(ADV)TNS" describing what occurs at each stage. The internal configuration of the system at each stage is illustrated in Fig. 3.4.2.

Stage 1

The system is initialized as if the last character was a common (ILAST = 2) and an A-link from the 1st node (there is no first node, but a link from the first node indicates the beginning of an expanded node list) is placed into MPUSH at the IPUSH = 1 level. KNPUSH(1) = KNPUSH(2) indicates that level 1 of NPUSH is empty.

Stage 2

A parenthesis is scanned and causes the pushdown to be pushed down (IPUSH is increased by 1) and the links in MPUSH for the last level are copied into this level. NPUSH is also empty for this level.

Stage 3

Similar to stage 2.

Stage 4

"NEG " is entered into the table of nodes at NODES(2) and the A-links in this level of MPUSH are fixed onto "NEG" i.e., a pointer "2"

FIGURE 3.4.2 STAGE 1 - INITIALIZE

KTR _____ ISPEC _____ ILAST 2

INODE	NODES	KLINKS	LINKS	1	2	3	4	5	6
1	1. _____	1. 0	1.						
	2. _____	2. _____	2.						
	3. _____	3. _____	3.						
	4. _____	4. _____	4. _____						
	5. _____	5. _____	5.						

IPUSH	IMPUSH	IPUSH	INPUSH
1	1	1	0

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1.
2. 2	2.	2. 1	2.
3.	3.	3.	3.
4.	4.	4.	4.
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

is placed in LINKS(1,1) indicating that the first expanded node list starts with the contents of NODES(2) .

Stage 5

A comma preceded by a word causes a B-link from the word to be placed into NPUSH at this present level. In this example, INPUSH is increased by 1 to 1, KNPUSH(IPUSH+1) becomes INPUSH+1 (i.e., KNPUSH(4) becomes 2) and KNPUSH(IPUSH) has "2" placed in it.

Stage 6

The word "AFF" is placed in NODES(3) . A word preceded by a comma is much the same as a word preceded by a left parenthesis so the MPUSH link is fixed -"3" is placed in LINKS(1,2) .

Stage 7

A right parenthesis is preceded by a word (like a comma, slash, or period preceded by a word) causes a B-link from the word to be placed into NPUSH for this level,, The pushdown is popped (IPUSH is decreased by 1), but the links of the old level are still current, the next character determines the action to be taken.

Stage 8

Another right parenthesis.,

Firstly as at this level (IPUSH=2) we have KNPUSH(IPUSH)=KNPUSH (IPUSH+1) it follows that there have been no commas at this level and consequently the nodes of this level are optional. So the A-links into this level (just "1") become B-links out of this level (i.e., the A-links skip over the contents of this level). A transfer is made from MPUSH into NPUSH,

FIGURE 3.4.2 STAGE 2

KTR _____ ISPEC 1 ILAST 2

INODE	NODES	KLINKS	LINKS	1	2	3	4	5	6
1	1. _____	1. 0	1.						
	2. _____	2. _____	2.						
	3. _____	3. _____	3.						
	4. _____	4. _____	4.						
	5. _____	5. _____	5.						

IPUSH	IMPUSH	IPUSH	INPUSH
2	2	2.	0

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1.
2. 2	2. 1	2. 1	2.
3. 3	3.	3. 1	3.
4.	4.	4.	4.
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

FIGURE 3.4.2 STAGE 3

KTR (ISPEC 1 ILAST 1

INODE	NODES	KLINKS	LINKS 12 3 4 5 6'
1	1.	1. 0	1.
	2.	2.	2.
	3.	3.	3.
	4.	4.	4.
	5.	5.	5.

IPUSH	IMPUSH	IPUSH	INPUSH
x	x	x	0

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1.
2. 2	2. 1	2. 1	2.
3. 3	3. 1	3. 1	3.
4. 4	4.	4. 1	4.
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

FIGURE 3.4.2 STAGE 4

KTR NEG ISPEC 0 ILAST 1

INODE	NODES	KLINKS	LINKS 1 2 3 4 5 6
2	1.	1. 1	1. 2
	2. NEG	2. 0	2.
	3.	3.	3.
	4.	4.	4.
	5.	5.	5.

IPUSH	IMPUSH	IPUSH	INPUSH
3	x	x	0

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1.
2. 2	2. 1	2. 1	2.
3. 3	3. 1	3. 1	3.
4. 4	4.	4. 1	4.
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

FIGURE 3.4.2 STAGE 5

KTR _____ ISPEC 2 ILAST 0

INODE
2

	NODES	KLINKS	LINKS	1	2	3	4	5	6
1.		1	1.	2					
2.	NEG	0	2.	-	-	-	-	-	-
3.			3.						
4.			4.						
5.			5.						

IPUSH	IMPUSH	IPUSH	INPUSH
3	3	3	1

	KMPUSH	MPUSH	KNPUSH	NPUSH
1.	1	1	1	2
2.	2	1	1	
3.	3	1	1	
4.	4		2	
5.				
6.				
7.				
8.				
9.				
10.				

FIGURE 3.4.2 STAGE 6

KTR AFF ISPEC 0 ILAST 2

INODE	NODES	KLINKS	LINKS	12	3	4	5	6
x	1.	1. 2	1.	-2	3-	m	-	-
	2. NEG	2. 0	2.					
	3. AFF	3. 0	3.					
	4.	4.	4.					
	5.	5.	5.					

IPUSH	IMPUSH	IPUSH	INPUSH
3	3	3	1

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1. 2
2. 2	2. 1	2. 1	2.
3. 3	3. 1	3. 1	3.
4. 4	4.	4. 2	4.
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

Secondly, as the preceding character was a right parenthesis, the B-links for the preceding level are added to the B-links of this level. We now have links from 2, 3, 1 in NPUSH for this level.

Thirdly, the pushdown is popped again (IPUSH=1) .

Stage 9

A left paren. is scanned so the pushdown is again pushed. Now all the B-links out of the previous parenthesis level become A-links into the new parenthesis level., So NPUSH for this level is transferred to MPUSH and is itself eliminated by putting KNPUSH(IPUSH+1) = KNPUSH(IPUSH) .

Stage 10

"AD"" is entered into NODES(4) and the A-links for this level are fixed to "4" .

Stage 11

As in stage 7, the MPUSH A-links become B-links in NPUSH . First of all a B-link is entered from "ADV" in NODES(4) . The push down is popped to level 1 .

Stage 12

"TNS" is entered in NODES(5) . As the preceding character was a right parenthesis the B-links in NPUSH for the preceding level are fixed to "5".

Stage

A period firstly causes a link from "TNS" in NODES(5) to be placed in NPUSH . Then links in NPUSH for this first level are fixed to "0" indicating the end of an expanded node list. Control is passed to the expansion section,

FIGURE 3.4.2 STAGE 7

KTR 1 ISPEC 6 ILAST 0

INODE	NODES	KLINKS 1	LINKS	1	2	3	4	5.	6
3	1.	2	1.	2	3				
	2. NEG	2. 0 !	2.						
	3. AFF	3. 0	3.						
	4.	4.	4.						
	5.	5.	5.						

IPUSH	IMPUSH	IPUSH	INPUSH
2	2	2 .	0

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1. 2
2. 2	2. 1	2. 1	2. 3
3. 3	3. 1	3. 1	3.
4. 4	4.	4. 3	4.
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

FIGURE 3.4.2 STAGE 8

KTR) ISPEC 6 ILAST 6

INODE	NODES	KLINKS	LINKS	1	2	3	4	5	6
3	1.	1. 2	1.	2	3				
	2. NEG	2. 0	2.						
	3. AFF	3. 0	3.						
	4.	4.	4.						
	5.	5.	5.						

IPUSH	IMPUSH	IPUSH	INPUSH
1	1	1	0

	KMPUSH	MPUSH	KNPUSH	NPUSH
1.	1	1	1	2
2.	2	1	1	3
3.	3	1	4	1
4.	4		3	
5.				
6.				
7.				
8.				
9.				
10.				

FIGURE 3.4.2 STAGE 9

KTR (_____ ISPEC 1 ILAST 6

INODE	NODES	KLINKS	LINKS	1	2	3	4	5	6
3	1. _____	1. 2	1.	2	3	—	—	—	—
	2. NEG	2. 0	2.	—	—	—	—	—	—
	3. AFF	3. 0	3.	—	—	—	—	—	—
	4. _____	4. _____	4.	—	—	—	—	—	—
	5. _____	5. _____	5.	—	—	—	—	—	—

IPUSH	IMPUSH	IPUSH	INPUSH
2	4	2	0

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1. 2
2. 2	2. 2	2. 1	2. 3
3. 5	3. 3	3. 1	3. 1
4. 4	4. 1	4. 3	4. _____
5. _____	5. _____	5. _____	5. _____
	6. _____		6. _____
	7. _____		7. _____
	8. _____		8. _____
	9. _____		9. _____
	10. _____		10. _____

FIGURE 3.4.2 STAGE 10

KTR ADV ISPEC 0 ILAST 1

INODE
4

NODES
1.
2. NEG
3. AFF
4. ADV
5.

KLINKS
I . 3
-2. 1
3. 1
4. 0
5.

	1	2	3	4	5	6
LINKS 1.	2	3	4			
2.	4					
3.	4					
4.	-	-	-	-	-	-
5	.	-	-	-	-	.

IPUSH
2

IMPUSH
4

IPUSH
2

INPUSH
0

KMPUSH
1. 1
2. 2
3. 5
4. 4
5.

MPUSH
1. 1
2. 2
3. 3
4. 1
5.
6.
7.
8.
9.
10.

KNPUSH
1. 1
2. 1
3. 1
4. 3
5.

NPUSH
1. 2
2. 3
3. 1
4.
5.
6.
7.
8.
9.
10.

Following through the above example should give the reader a good feel for the algorithm*

During this stage a number of errors such as "(" followed by "," are checked for. If an error is encountered, the rule or context being expanded is skipped entirely.

II/, Expansion Algorithm

This is a straight-forward chasing of links and can best be understood by reading the appropriate section of the program. The Ith expanded node list is stored in MEXPND from KEXPND(I) to KEXPND(I+1)-1 . IEXPND points to KEXPND, JEXPND points to MEXPND . KMPUSH and KNPUSH are used during the expansion to keep track of how much has been expanded so far, The Ith word of an expanded node sublist at a given time is in NODES(LINKS(KMPUSH(I),KNPUSH(I))) .

III/. Ordering Sections

The ordering algorithm is simple. The smallest expansion is taken out of MEXPND and stored, being replaced by a large non-word (") . " in this case) and then the smallest expansion removed again and so on.

Duplicate expansions are removed. (The procedure is complicated by the requirement that "A B" when compared with "A" actually has to be compared with "A blank".)

IV/. Context Checker

Foul contexts like "-" or "A " or "A - B - " are removed and null contexts are accounted for, Error messages are issued.

FIGURE 3.4.2 STAGE 11

KTR 1 ISPEC 6 ILAST 0

INODE 4

	NODES	KLINKS	LINKS 1	2	3	4	5	6
1.		3	2	3	4			
2.	NEG	1	4					
3.	AFF	1						
4.	ADV	0						
5.								

IPUSH	MPUSH
1	1

IPUSH	INPUSH
1	0

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1. 4
2. 2	2. 2	2. 1	2. 2
3. 5	3. 3	3. 5	3. 3
4. 4	4. 1	4. 3	4. 1
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

FIGURE 3.4.2 STAGE 13

KTR ____ ISPEC 7 ILAST 0

INODE	NODES	KLINKS	LINKS	1	2	3	4	5	6
5	1.	1. 4	1.	2	3	4	5		
	2. NEG	2. 2	2.	4	5				
	3. AFF	3. 2	3.	4	5				
	4. ADV	4. 1	4.	5					
	5. TNS	5. 1	5.	0					

IPUSH	IMPUSH	IPUSH	INPUSH
1	1	1.	1 -

KMPUSH	MPUSH	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1. 5
2. 2	2. 2	2. 2	2. 2
3. 5	3. 3	3. 5	3. 3
4. 4	4. 1	4. 3'	4. 1
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

FIGURE 3.4.2 STAGE 12

KTR TNS ISPEC 0 ILAST 6

INODE	NODES	KLINKS	LINKS	1	2	3	4	5	6
5	1.	1. 4	1.	2	3	4	5		
	2. NEG	2. 2	2.	4	5				
	3. AFF	3. 2	3.	4	5				
	4. ADV	4. 1	4.	5					
	5. TNS	5. 0	5.						

IPUSH	IMPUSH	IPUSH	INPUSH
1	1	1	0

KMPUSH	M P U S H	KNPUSH	NPUSH
1. 1	1. 1	1. 1	1. 4
2; 2	2, 2	2. 1	2. 2
3. 5	3. 3	3. 5	3. 3
4. 4	4. 1	4. 3'	4. 1
5.	5.	5.	5.
	6.		6.
	7.		7.
	8.		8.
	9.		9.
	10.		10.

V/. Error Recovery

The general philosophy has been to try and continue after an error is found so as to check for further blunders. In later models, expanded and non-expanded node lists will be mixable so partial expansions will be valuable.

VI/. Denouement

When all rules have been read, the expanded rules are listed or hunched if desired and other odds and ends tidied up.

PSGSMP

The entry PSGSMP of PSGINN places the expanded phrase structure rules into KSUMP, which can be printed by calling FROUT .

PSGOUT

PSGOUT is a short subroutine which prints out the phrase structure rule tables.

LEXIN, lexicon input routine

Internal Formats

We describe here only the storage of category features as used by lexical insertion and the storage of the lexical entries. The storage of inherent features, contextual, feature labels and descriptions, and redundancy rules are treated elsewhere.

Lexicon data is stored in the common block labeled /LEXCM/ defined as below.

```
      COMMON/ LEXCM/

     1  LEXWD,LEXWDS,LEXCS,LEXCSS,LXCPTR,CATLST,NLXC,
     2  NLEX,NLEXW,NLEXCS,NCATL
        REAL*8 LEXWD(500),CATLST(20)
        INTEGER*2 LEXWDS(300),LEXCSS(300),LEXCS(500),
     1  LXCPTR(100,20),NLXC(20),NLEX,NLEXW,NLEXCS,
     2  NCATL
```

The category feature list is stored in the order input in the array CATLST . The parameter NCATL gives the number of entries in the category list.

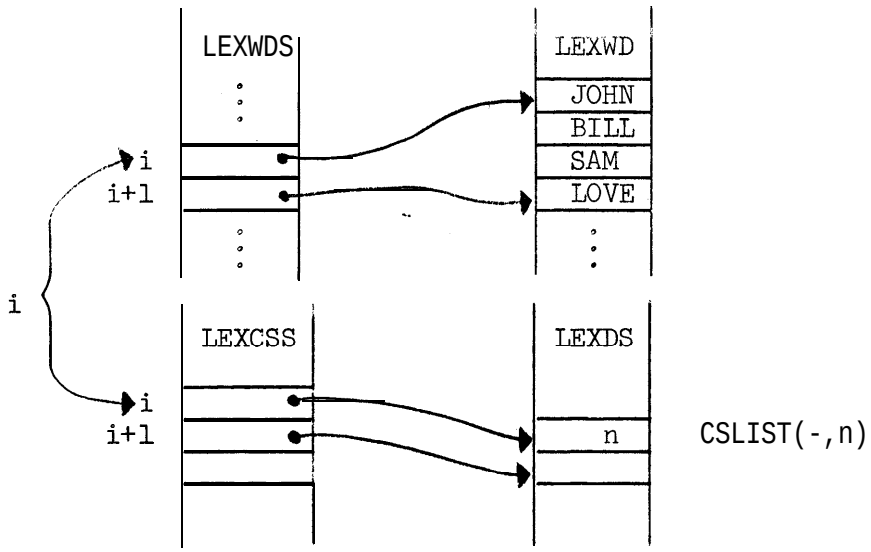
A lexical entry is defined as a list of vocabulary words and a list of complex symbols. Internally each entry is composed of two lists of pointers. One list (LEXWDS) contains pointers to the array LEXWD where the vocabulary words for the entry are stored. The other list (LEXCSS) contains pointers to the array LEXCS where numbers of the complex symbols for tne entry are stored (these numbers are pointers to the array CSLIST) .

To illustrate, if the ith and i+1st lexical entries are as defined below:

entry

i	JOHN BILL SAM |+N +HUMAN|
i+1	LOVE |+V + TRANS|

then the storage would be as shown below.



To simplify searching for lexical entries during the lexical insertion process, the entries are linked by lexical category; that is, all nouns are linked together, all verbs, etc.

For the jth entry in CATLST

NLXC(j) = number of lexical entries in that category

LXCPTR(k,j) = pointer (to LEXWDS and LEXCSS) for the kth lexical entry in category CATLST(j)

$(1 \leq k \leq NLXC(j))$

LEXIN. lexicon input

When LEXIN is entered, it immediately calls the subroutine PRELEX to read in the prelexicon portion of the lexicon. After PRELEX returns, the lexical entries are processed, The subroutine FREAD is used to read vocabulary words and special symbols ("22 "; "; "|") . Complex symbols are read by the subroutine CXIN . The flag ENTFL=.true. is used to indicate that an entry must be linked to the appropriate category list, The flag ENDEF=.true. indicates

that entry has been completed and thus the pointers in LEXCSS and LEXWDS must be specified. Error comments are produced by LEXIN if the array limits specified for the lexicon are exceeded. The array limits are specified in the common block /LEXCM/ .

PRELEX, prelexicon input

Calling sequence: CALL PRELEX

Description: the integer variable STAGE is used in a "computed goto" statement to transfer control to the appropriate place. If errors occur, such as illegal punctuation or the omission of punctuation, error comments are generated and recovery is attempted. The subroutines and functions called by PRELEX are

Type	Name and Args.	Purpose
FUNCTION	FREAD(ONE)	free field input
FUNCTION	NUMNAM(CHRTR,ZERO)	store category and inherent features
FUNCTION	SLFEAT(ONE)	read contextual feature description (not including restriction)
SUBR	NAMEIN(CWORD,I)	store contextual feature label
FUNCTION	RESTIN(ONE)	read restriction in contextual feature description
FUNCTION	CXIN(ONE)	read in complex symbol appearing in redundancy rules.

LEXSMP. lexicon output

calling sequence: CALL LEXSMP

description: This subroutine puts the lexical entries into the array KSUMP. They may then be printed or punched as desired by the

appropriate call to FROUT . The output format is suitable for use as input. The subroutine CSSUMP is used to put complex symbols into ..... If KSUMP becomes full the contents are printed by calling FROUT and then the remaining entries are stored. The lexical entries are put into KSUMP is category order as specified in CATLST .

LEXOUT, lexicon debugging

calling sequence: CALL LEXOUT

description: This subroutine generates a printout of the storage arrays for the lexicon described earlier. This printout is intended for debugging purposes only. The code for this subroutine is found in the subroutine LEXSMP .

NUMNAM, feature number and name

NUMNAM(FWORD,ARG) returns the feature number for the feature name FWORD . If ARG is nonzero, and FWORD has not previously been assigned a number, FWORD is assumed to be the name of an inherent feature, and a warning to that effect is printed.

The entry NAMEIN(FWORD,ARG) stores FWORD as the name of the contextual feature whose feature number is ARG .

The entry FEATOU prints out the internal tables for features, redundancy rules and calls CSOUT for the internal complex symbol tables.

TRANIN, input routine for transformations

TRANIN reads in a set of transformations and stores the information for later use. TRANIN is called by GRAMIN . TRANIN calls subroutines ANALIN (an entry to CXIN), RESTIN, CHANIN, and CPIN which read and store parts of the transformation specification, as shown in Figure 3.4.3.

Figure 3.4.3 Subroutines called by TRANIN

subroutine	result
ANALIN(ØNE)	returns pointer to-the structural analysis
RESTIN(ØNE)	returns pointer to the restriction
CHANIN(ØNE)	returns pointer to the structural change
CPIN	stores the control program for use by the control subroutine

The only information which is analyzed by TRANIN itself is the identification. The following comments explain the use of the parts of the identification. (1) The optional integer is solely for the convenience of the user and is ignored by the program; the transformation name is always used in referring to the transformation. (2) The group number is used by the control program to refer to a set of transformations. If the group number is omitted on input, it will be taken to be the same as the group number of the preceding transformation, or I in the case of the first transformation. (3) Repetition determines if and how the transformation will be reapplied to the same subtree. The choices are AC (analyze once and change), ACAC (analyze, change and repeat), AACC (find all analyses, then do all changes, and AAC (find all analyses, do one randomly-selected change). The null option is AC . (4) The choices for optionality are option (OP) and obligatory (OB) . The null option is OB . (5) The keywords must be present in the tree to which the transformation applies; this is a technical non-linguistic device to speed up the program by avoiding the analysis routine, (Remark: At some later time we may wish to expand the notion of keyword to allow Boolean combinations of keywords, or possibly even key-subtrees.) An embedding parameter which would allow a search to go below any sentence symbol was originally planned but has not been implemented; a tree will be searched below a sentence symbol only if the analysis explicitly mentions a sentence symbol and gives an analysis for it.

Internal storage. Transformations are stored in the common block /TRANCM/ . The present capacity is 100 transformations. The I-th transformation read is stored as follows:

FTRAN(I) = name of transformation

TRAN(I,1) = group number (stored as an integer 1-7)

TRAN(I,2) = repetition (1 for AC, 2 for ACAC, 3 for AACC and 4 for AAC)

TRAN(I,3) = optionality (0 for obligatory, 1 for optional)

TRAN(I,4) = (currently unused)

TRAN(I,5) = pointer to the structural analysis of structural description

TRAN(I,6) = pointer to the restriction of the structural description

TRAN(I,7) = pointer to the structural change

The keywords for the I-th transformation are stored in KEYS from KEYPT(I)+1 through KEYPT(I+1) . The number of transformations (hence the current length of both FTRAN and TRAN) is NTRAN . The total number of keywords is NKEYS .

Output of transformations. The transformations should be followed by the order $END . This causes the program to output FTRAN and TRAN in tabular form, followed by the list of keywords. Control is then returned to the main program. CALL TRANOU will also produce this output, which is illustrated in Figure 3.4.4.

FIGURE 3.4.4

TRANSFORMATIONS

	NAME	GROUP	CYCL	OPT	EMB	SO	RES	SC
1	DUMMY	1	0	0	0	2	0	0
2	CP1	1	1	0	0	3	0	1
3	CP2	1	1	0	0	4	2	4
4	CP3	1	1	0	0	5	0	5
5	IF	1	1	0	0	0	0	0
6	IOI	1	1	1	0	6	0	6
7	TO	1	1	0	0	7	0	7
8	PASSIVE	1	1	0	0	8	3	8
9	EYTR4	1	1	1	0	10	0	12
10	PROREP	1	1	0	0	11	0	13
11	WHA	1	1	0	0	12	0	14
12	RELPLACE	1	1	0	0	13	0	15
13	AUXFILL	1	1	0	0	14	0	16
14	AG	1	1	0	0	15	0	17
15	EVER	1	1	1	0	16	0	20
16	REGDEL 1	1	1	1	0	17	5	21
17	REGDEL 2	1	1	0	0	18	6	22
18	DEFI	1	1	0	0	19	0	23
19	WHAG	1	1	0	0	20	7	24
20	PROGDEL	1	1	0	0	21	9	27
21	RELDEL	1	1	1	0	22	0	28
22	ADJPLACE	1	1	0	0	23	0	31
23	CDUP	1	1	0	0	24	0	32
24	CNEG	1	1	1	0	25	0	33
25	CTENSE	1	1	0	0	26	0	34
26	TS	1	1	0	0	27	0	36
27	CD	1	1	1	0	28	0	37
28	TAG	1	1	1	0	29	0	38
29	NEGPLACE	1	1	0	0	30	0	41
30	NEGTAG	1	1	1	0	31	0	42
31	NEGAUX	1	1	1	0	32	0	43
32	QUES	1	1	0	0	33	0	44
33	YESNO	1	1	0	0	34	0	45
34	AF	1	1	0	0	35	0	46
35	PREPDEL	1	1	0	0	36	10	47
36	PD	1	1	0	0	37	0	48
37	AGDEL	1	1	1	0	38	0	49
38	THAT	1	1	1	0	39	0	50
39	VPCOMP	1	1	1	0	40	0	51
40	BEDEL	1	1	0	0	41	0	54
41	MC DEL	1	1	0	0	42	0	55
42	QDEL	1	1	0	0	43	0	56
43	ERASE	1	1	0	0	44	0	57

FIGURE 3.4.4 (part 2)

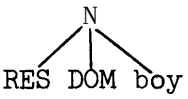
44	PAST	2	1	0	0	45	0	59
45	MTDEL	2	1	0	0	46	0	60
46	PLUDEL	2	1	0	0	47	0	61
47	NUM	2	1	0	0	48	0	62
48	NUAG	2	1	0	0	49	0	63
49	CONTR	2	1	0	0	50	0	64
50	NEGSPELL	2	1	0	0	51	0	65
51	DO1	2	1	0	0	52	0	66
52	DO2	2	1	0	0	53	0	67
53	DO3	2	1	0	0	54	0	68
54	BE1	2	1	0	0	55	0	69
55	BE2	2	1	0	0	56	0	71
56	BE3	2	1	0	0	57	0	73
57	BE4	2	1	0	0	58	0	75
58	HAVE1	2	1	0	0	59	0	77
59	HAVE2	2	1	0	0	60	0	79
60	HAVE3	2	1	0	0	61	0	81
61	WHPD1	2	1	1	0	62	0	83
62	WHPD2	2	1	0	0	63	0	54
63	WHDEL	2	1	1	0	64	0	85
64	DEFTHAT	2	1	0	0	65	0	86
65	WH1	2	1	0	0	66	0	57
66	WH2	2	1	0	0	67	0	89
67	WH3	2	1	0	0	68	0	91
68	PLADEL	2	1	0	0	69	0	93
69	C1	2	1	0	0	70	0	94
70	c2	2	1	0	0	71	0	95
71	C3	2	1	0	0	72	0	96
72	c4	2	1	0	0	73	0	97
73	BY	2	1	0	0	74	0	98
74	INDEF	2	1	0	0	75	0	99
75	DEF	2	1	0	0	76	0	100

TRANS KEYWORDS

3.5 Phrase structure generation

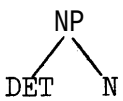
This section describes the routines GEN,IFIND,IAFIND, and NSRCH which are used to expand tree skeletons into base trees. A general description of this process is given in CS-80 (AF-15).

The main program is GEN and is called each time a skeleton is to be expanded. The skeleton is expanded, starting with the sentence symbol S, by selecting at random from the set of applicable phrase structure rules. The skeleton may contain restrictions which require dominance (DOM), nondominance (NDOM), equality (EQ), and special node symbols, null expansion (NL), or variable numbers of daughters (X or Y) . Restrictions DOM, NDOM, and EQ appear in the skeleton as daughters of nodes as shown in the example below.



The special node symbols NL, X, and Y appear directly in the skeleton., The appearance of NL as the leftmost daughter of a node indicate that no daughters are to appear to the left of the daughter to the right of NL, that is the skeleton

could <u>not</u> be expanded to

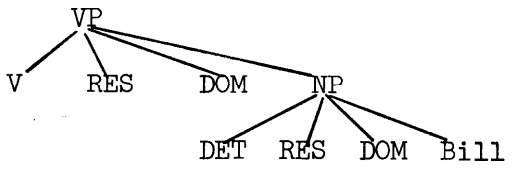


but could be expanded to

The appearance of NL as the rightmost daughter in the skeleton similarly limits expansion to the right.

The special node X indicates that 0 or more daughters must appear in its place and the special node Y indicates that 1 or more daughters must appear,

The restrictions DOM,NDOM,EQ may refer to nodes or vocabulary words (vocabulary words are handled during lexical insertion),, DOM and NDOM may also refer to complex symbols with the node symbol a blank. GEN uses the subroutine LSRCH (described in Section 3.6) with a node number of 0 to determine if there is a compatible complex symbol in the lexicon, The DOM restriction allows subtrees to be specified as well as single nodes and these subtrees may contain further restrictions. For example



The function'subprogram IFIND(M,N) is used by GEN in testing DOM and NDOM restrictions to determine if a node of type M must, might, or cannot dominate a node of type N . IFIND calls IAFIND(N) to find the first PSG rule (in PSGA1) which expands a node of type N . IFIND calls NSRCH(N) to find the last PSG rule which could expand to a node of type N .

GEN

The following storage arrays are used by GEN in building the base tree and handling restrictions

	type
STRING(200,2)	R*8
ITRACK(lOO)	I*2
NEQLST(20)	I*2
EQTRAN(20)	R*8
NRESRS(10)	I*2
NRES(10,3)	R*8
NOK(20)	I*2
NTEM(50,2)	R*8
NTEM3(50)	I*2
NTEMCS(50)	I*2
NOK(20)	I*2

STRING

The array STRING contains a parallel list of the terminal node symbols and the terminal node numbers for the tree e.g., for the I-th element in terminal string

STRING(I,NSTA) = node number

STRING(I,NSTB) = node symbol

During expansion the new elements are inserted in the appropriate side of the array. After expansion of an element the rest of the string is copied over,

The pointers NSTA and NSTB are then reversed which in effect "flips" the array.

ITRACK

This array contains pointers to the acceptable rule expansions of a given node, The number of entries in ITRACK is given by the parameter MTRACK.

NEQLST

This array is used for handling equality restrictions, The first appearance of a node with an equality restriction is expanded and the node number is placed in NEQLST(I) where I is the equality restriction number, For each succeeding appearance of a node with the same equality restriction, the restriction number is saved in TREE(NODE,5) and the node is not expanded further at that time, When the base tree has been completely specified (including lexical insertion) then the subtree headed by the original node (with restriction I) replaces each appearance of a node with TREE(NODE,5) = I .

EQTRAN

The routine TRIN does not convert integers to integer format, This conversion is performed by table look-up in the array EQTRAN.

NRESRS

If a DOM restriction contains a subtree (more than a single node), or if a complex symbol appears in a restriction then the tree node which is the top of the subtree (or has the complex symbol) is saved in NRESRS(MRES) where MRES is the number of restrictions (DOM or NDOM) detected thus far., Otherwise NRESRS(MRES) = 0 .

NRES

For each DOM or NDOM restriction

NRES(___,1) = DOM or NDOM

NRES (___,2) = top node symbol of the restriction

NRES(___,3) = tree node of the first daughter making up the restriction (RES) .

NOK

NOK(MK) = result of IFIND for the MKth daughter of a node with DOM or NDOM restriction.

NTEM, NTEM3, NTEMCS

For each possible (MKth) daughter of a node with a DOM or NDOM restriction

NTEM(___,1) = node symbol

NTEM(___,2) = DOM or NDOM

NTEM3(_) = NOK(MK)

NTEMCS(_) = complex symbol pointer if a complex symbol was specified in the restriction.

The actual operation of GEN may be summarized as follows.

1. If there is a skeleton, store it in TREE .

   If there is no skeleton, start with SS(SS=S) in TREE .

2. Pick a PSG rule (linear pass through NSGA1) .

   If no more rules, go to step 11.

3. Match element of STRING with the left part of the rule.

   If no more entries in STRING, go to step 2.

4. If there are no daughters specified in the skeleton, put pointers to the possible rule expansions in ITRACK and go to step 6.

5. If there are daughters in the skeleton

5a. Search daughters for a restriction. If there is a RES and

- if it is an EQ
  - put first node number in NEQLST and continue with step 5a.
  - for subsequent occurrences of the restriction put the restriction number in TREE (node, 5) and go to step 3.
- if it is a DOM restriction with daughters or a complex symbol, save the node number in NRESRS
- if it is a DOM or NDOM of an SS put this as the first entry in NRES; otherwise put restriction in next NRES entry.,

5b. Put pointers to the possible rule expansions in ITRACK replacing X and Y nodes and treating the NL . If there were DOM or NDOM restrictions, use IFIND to determine the effect of the rules in ITRACK for each restriction and save the results in NTEM . If the restriction contained a complex symbol, consult LSRCH as well. Delete those rules from ITRACK which are not desired.

6. Pick a rule from ITRACK entries.

7. Put rule expansion in TREE (if not already there) and in STRING. Unlink RES, X, Y, and NL daughters.

8. Save the remaining STRING entries.

9. If there were restrictions, then for each restriction
   - test each daughter and find the one corresponding to the NTEM entry.
   - for a DOM restriction
     - if satisfied, set indicator (MK) and if DOM has daughters or a complex symbol attach them
     - if not satisfied yet, put the DOM restriction (with complex symbol or daughters, if any) on one of the new daughters which could possibly meet the restriction.
   - for a NDOM restriction, if still could be violated, add the restriction to each new daughter which could be expanded into a node of the type not desired.

10. Fill and flip the STRING array and go to step 3 (pick up where we left off in STRING).

11. If there are any SSes or Ses in STRING and if an SS or S appeared in a DOM restriction, then put the leftmost S in the first entry in STRING and go to step 2 (this wipes out the old STRING).

12. Do lexical insertion (call LEXINS),

13. If there were EQ restrictions, search tree for nodes marked with a restriction number (in TREE (___,5)) and substitute the subtree headed by NEQLST(TREE(___,5)) .

IFIND (M,N)

IFIND is an INTEGER*2 function with two REAL*8 arguments M and N which are node symbols that appear in the tree and in a DOM or NDOM restriction, respectively.

IFIND(M,N) = -1 if a node of type M never can dominate a node of type N

IFIND(M,N) = 0 if a node of type M might dominate a node of type N.

IFIND(M,N) = 1 if a node of type M must dominate a node of type N.

The result of IFTND is obtained by examination of the phrase structure rules stored in the arrays NSGA1, NSGA2, NSGB, and NSGC . The array ISTACK is used for pushdown to save intermediate parameters and the array CATRES is used to save intermediate results.

A heuristic has been introduced to increase the efficiency of the search process. Any node symbol examined to see if it dominates another node symbol is tested only once. If the search is performed exhaustively a given category may be examined several times if it appears more than once in the phrase structure rules. For example:

1. S → NP VP
2. VP → V NP
3. NP → N S

For IFIND(S,N), the NP will be examined only once even though it appears on the right-hand side of rules 1 and 2. The array BADST is used to remember rules which have been previously examined.

IAFIND(I)

IAFIND is a REAL*8 function with a REAL*8 argument. The value of I is a node symbol. If I is the sentence symbol (SS=S) or a terminal symbol of the phrase structure grammar then the value of IAFIND(I) is I . Otherwise the value of IAFIND is the index to the first rule in the phrase structure grammar pointer to (NSGA1) that expands the symbol I .

NSRCH(N)

NSRCH is an INTEGER*2 function with a REAL*8 argument, N . N is a node symbol. The value of NSRCH is the index to the last rule in the phrase structure grammar that introduces the symbol N . If no rule introduces the symbol N, the value of NSRCH is 0 .

3.6 Lexical insertion

The main subroutine for lexical insertion is LEXINS. This program calls the subroutine TSRCH to locate lexical category nodes in the tree. LEXINS calls the subroutine LSRCH to locate complex symbols and vocabulary words that are suitable for insertion at a specified lexical category node. LSRCH calls the subroutine CSTEST to test if the lexicon complex symbols are suitable for insertion in the tree. CSTEST calls the subroutine CSCOMP to determine if a tree complex symbol and a lexicon complex symbol are compatible. CSCOMP will assign values (either to + or -) to all features with the value *. CSCOMP merges complex symbols and expands the result using the subroutine REDRUL. The result of CSCOMP is either a complex symbol number or the integer value zero to indicate that the two complex symbols are incompatible. If CSCOMP indicates that the complex symbols are compatible, CSTEST then calls the subroutine ANTEST to test each contextualfeature specification in the complex symbol returned by CSCOMP.

The following common blocks contain arrays and variables used in lexical insertion.

```
      COMMON/LINSCM/
     1  SRCHL,ELIST,NSRCHL,NELIST
      INTEGER*2 SRCHL(2,50),ELIST(2,50),NSRCHL,NELIST
```

where

SRCHL(2,50) is a stack of parent and daughter pointers that is used by the subroutine TSRCH in searching for nodes of a particular lexical category. This array is initialized by the subroutine LEXINS. The number of entries in SRCHL is given by the parameter NSRCHL.

ELIST(2,50) is an array used to hold the lexical items found by the subroutine LSRCH

ELIST(1,-) = index to vocabulary word in the array LEXWD

ESIST(2,-) = pointer to the complex symbol in the array CSLIST

NELIST gives the number of items in ELIST. If more than one item is found, an item is selected at random for insertion,.

```
COMMON/CONFCM/
1   CFVALS(100)
```

The array CFVALS is used to save the value for a contextual feature when it has been determined by the subroutine ANTEST. This array is ihitialized by the subroutine LSRCH and data is entered into the array by the subroutine CSTEST. Before CSTEST calls ANTEST to analyze a contextual feature, it first checks to see if the value has already been 'obtained.

if M is the feature number (CSLIST(l,))

$$\text{CFVALS(M-MXEXP)} = \begin{cases} 0 \Rightarrow & \text{no value determined for this feature.} \\ 1 \Rightarrow & \text{feature is positively specified.} \\ 2 \Rightarrow & \text{feature is negatively specified.} \end{cases}$$

LEXINS, lexical insertion

calling sequence: CALL LEXINS

description: lexical insertion is performed in two passes. On the first pass restrictions and vocabulary words introduced by the directed random generation of the tree-are considered. On thesecond pass, the remaining lexical category nodes are treated. The operation of the

program may be summarized by the following sequence of actions.

1. initialize EQLST for equality restrictions
2. search tree, breadth first (right to left) and top down. Make a list (SSLIST) of the appearances of the sentence symbol (SS).
3. do pass 1

   3.1 take SS from SSLIST (last first) if no more go to step 4.

   3.2 search tree for category symbols in the order specified in CATLST if no more entries in CATLST go to step 3.1.

   3.3 call TSRCH(CA,T,CNODE) to get next category node.

   CNODE = 0 ⇒ no more nodes in this category so get next CATLST entry and go to step 3.2.

   CNODE = 0 ⇒ if not a restriction or vocabulary word go to step 3.3.

   3.4 if equality restriction

   -- convert restriction number to integer (TRIN doesn't do this). We require $1 \leq$ restriction number < 20.

   -- if EQLST(restriction no.) = 0 then

   EQLST(restriction no.) = node of lexical category (CNODE)

   CALL LSRCH (LC,CNODE, vocabulary word, tree complex symbol)

   LC = lexical category number in CATLST

   CNODE = node of lexical category symbol

   if there is an entry - attach the complex symbol and treat the side effects CALL SIDEFF (CNODE,LECS)goto step 3.3.

   if no entry - error comment go to step 3.3.

-- if EQLST(restriction no.) ≠ 0 then

substitute vocabulary word and complex symbol

CALL SUBST(EQLST(I),CNODE)

go to step 3.3.

3.5. If dominance restriction, erase RES and DOM daughters and go to step 3.6.

3.6. Vocabulary word is specified

CALL LSRCH(LC,CNODE,WORD,TREE(CNODE,6))

if there is an entry

attach new complex symbol- treat side effects

go to step 3.3.

if no entry

write error comment

go to step 3.3.

4. do pass 2.

4.1. take SS from SSLIST (last first)

if no more RETURN

4.2. get entry in CATLST

if no more go to step 4.1.

4.3. search tree for node in proper category

CALL TSRCH(CAT,CNODE)

return

CNODE = 0 ⇒ no more in this category; go to step 4.2.

CNODE ≠ 0 ⇒ if daughter on node (from pass 1)

go to step 4.3.

4.4. search lexicon for vocabulary word and complex symbol

```
CALL LSRCH(LC,CNODE,FBLANK,TREE(CNODE,6))
if no entry
     error comment
     go to step 4.3.
if entry
     attach complex symbol
     attach vocabulary word
          CALL ALADE(MTREE,CNODE)
     treat side effects
          CALL SIDEFF(CNODE, complex symbol no.)
     go to step 4.3.
```

TSRCH, tree search for lexical category nodes

calling sequence: CALL TSRCH(CAT,NODE)

where CAT(REAL*8) = node type desired

NODE(INTEGER*2) = return parameter

return parameter:

NODE = 0 ⇒ no more nodes in the category CAT

NODE ≠ 0 ⇒ number of category rule in TREE.

description:

The initial tree top and first daughter are stored in SRCHL(1,1) and SRCHL(2,1) respectively by LEXINS. The search is depth first and left to right in the tree but never goes below any SS or below a lexical category node. The depth of search is recorded by the parameter NSRCHL. On subsequent calls to TSRCH, the search is resumed where it left off.

LSRCH, search lexicon

calling sequence:

CALL LSRCH(CATNO,NODE,WORD,TCS)

where CATNO = number of category of interest (pointer to CATLST)

NODE = { 0 special call by GEN
tree node location for lexical item

WORD = { blank if no vocabulary word is yet associated with node vocabulary word
if a particular vocabulary word has already been specified in the tree

TCS = { 0 if no complex symbol is defined in tree
pointer to complex symbol in CSLIST if lexical category node in tree has a complex symbol attached.

description:

LSRCH has several modes of operation depending on the values of its operands. In all cases the basic function is to find a lexical item (vocabulary word and complex symbol) which are suitable for insertion in the tree. The acceptable item is returned in the COMMON array ELIST.

ELIST(1,NELIST) = pointer to vocabulary word in array LEXWD

ELIST(2,NELIST) = pointer to complex symbol in CSLIST.

if there is no lexical item suitable for insertion then NELIST = 0 on return to the calling program (LEXINS).

If a vocabulary word has been specified LSRCH searches the lexicon in the appropriate category for that word. If the word is found the complex symbols associated with the entry are tested by using the function CSTEST(NODE,TCS,LCS) just as in the case when no vocabulary word is specified.

If no vocabulary word has been specified, then lexical entries are examined in the proper category but in a random manner (random selection without replacement). If the entry selected contains acceptable complex symbols (this is determined by using the function CSTEST(NODE,TCS,LCS)) then the search terminates. If the entry does not contain acceptable entries, then the entry is marked as unacceptable (LBAD(j) = true where j = jth entry in the category specified by CATNO) and a new random entry is selected.

We illustrate this process below. N is the number of entries remaining to be tested. I is the increment used to obtain an entry. Initially, N equals the number of entries in the category.

compute I (I = random integer, $1 \leq I \leq N$)

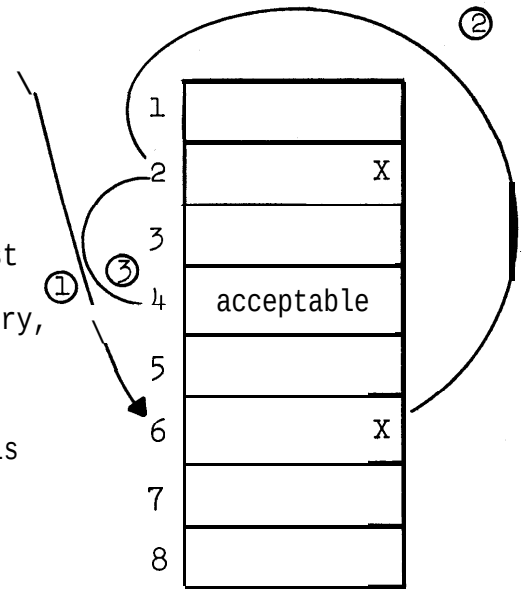
get I^{th} entry not yet tested

test the entry--if acceptable, then exit

mark entry not acceptable

N = N - 1 if N = 0, exit--no entry is acceptable

Suppose N = 8, compute $I_1 = 6$ and test entry 6 . Entry 6 is not acceptable, so mark entry, decrement N, compute $I_2 = 4$. Step 4 untested entries to entry 2. Test entry 2. Not acceptable. Mark entry, decrement N, compute $I_3 = 2$. Step 2 entries. Test entry 4. This entry acceptable, so exit.



This method of selection weights lexical entries equally. Since an entry may have more than one complex symbol, complex symbols do not have exactly equal probabilities of being selected. If this is an important consideration, the lexicon should be defined so that each entry consists of a single complex symbol with its associated vocabulary words. If lexical items are to receive equal probability of selection, the lexicon should be defined so that each entry is a single vocabulary word and a single complex symbol.

CSTEST, test complex symbol for lexical insertion

CSTEST is a function subprogram the value of which is an integer variable (INTEGER*2). The function is referenced as shown in the example below.

```
CSR = CSTEST(NODE,TCS,LCS)
```

where

NODE = { the node in the tree if doing lexical insertion
       { 0 for a test in the generation of the base tree by the program LSRCH-GEN.

TCS = { pointer to a complex symbol (usually the complex symbol associated with the tree)...

LCS = pointer to a complex symbol from the lexicon.

These results are

CSR = 0 if the complex symbol is not suitable for insertion in the tree.

CSR = complex symbol pointer if the complex symbol in the lexicon is suitable for insertion. In this case, the variable CSR points to the new complex symbol which is to be inserted in the tree.

description: The basic test used to determine if a complex symbol is suitable for insertion in the tree has two parts. The first part is the compatibility test. This test is performed by the program CSCOMP. If the tree complex symbol and the lexicon complex symbol are not compatible, the lexicon complex symbol is rejected and the value of CSTEST is zero. If the complex symbols are compatible, then the program CSCOMP returns a new complex symbol that is the result of the compatibility test. The second part of the test performed by CSTEST involves the analysis of each of the contextual features that appear in the new complex symbol. This part of the test is performed in two passes. On the first pass, the value of each contextual feature is compared with the entry in the array CFVALS. If the array entry is defined, then the value of this feature has already been determined by an earlier call to CSTEST (for a complex symbol which was rejected). We require that the values of the

contextual feature in the array and the value in the complex symbol be the same. On the second pass contextual features whose values do not appear in the array CFVALS are treated. The program ANTEST is used to analyze the tree for each such feature. Before ANTEST is called the node in the tree which corresponds to the top of the contextual feature description must be determined. This is done by getting the node type (e.g. S,VP) from the contextual feature description (in ANALWD). The tree is then searched up from the node for which lexical insertion is to be performed and the first occurrence of a symbol of the proper type is used as the top node for the analysis process. Of course if the node is not found the tree does not match the feature description and the value of the feature is minus (2). If the tree matches the feature description (ANTEST returns the logical value true), the value of the feature is + (1). If tree does not match the feature description (ANTEST returns false.) the value is minus (2). Again we require that the tree value and the lexicon value be the same. The value determined for the contextual feature is saved in the array CFVALS so that the analysis program will not be called twice for the same contextual feature in the search for a lexical item for the same tree node. If the tree meets all the contextual feature specifications, the complex symbol number is returned indicating that the complex symbol is acceptable for insertion in the tree.

SIDEFF (a separate entry point in CSTEST), side effects

When a vocabulary word and complex symbol are inserted in the tree, side effects must be considered and if necessary treated. For a definition and discussion of side effects see CS-103. The program which handles side effects is called SIDEFF(NODE,N) where NODE is the tree node where a lexical item was just inserted and N is a pointer

to the complex symbol just inserted. A rigorous treatment of side effects is not performed, however the cases which usually occur in practice are handled correctly. We describe here what is actually done by the program, not what should be done. The program performs as outlined below:

1. Side effects for negatively specified contextual features are ignored.
2. For each positively specified contextual feature.
   2.1 If the contextual feature description does not contain a complex symbol, there are no side effects for this feature so look for another contextual feature.
   2.2 If the contextual feature does contain a complex symbol, the analysis routine is called to examine the tree for the feature description. When the program ANTEST tests as a complex symbol embedded in a contextual feature description it uses the program CSCOMP. CSCOMP saves the result of the compatibility test. When ANTEST returns, SIDEFF examines the array ANNODE and retrieves the tree node which matched the node in the contextual feature description. SIDEFF then uses the function CSCOMP(NODE,0,3) to retrieve the complex symbol derived by CSCOMP during the analysis of the tree and attaches the complex symbol to the node. This is done for every complex symbol appearing in the contextual feature description. We use the result of the compatibility test rather than the complex symbol that appeared in the contextual feature description to insure that features

with the value *(3) will not appear in the tree.

CSCOMP(M,N,IND), compatibility test

This program is an integer*2 function whose basic task is to determine the compatibility of two complex symbols. The test for compatibility is described in general terms in CS-103. The parameter IND indicates the function CSCOMP is to perform

if IND = 1: perform compatibility test using nondistinctness as the subordinate test. (This mode is used in lexical insertion.)

IND = 2: perform compatibility test using inclusion as the subordinate test. (This mode could be used in the analysis process for transformations.)

IND = 3: then M is a tree node for which a compatible complex symbol has been obtained on a prior call to CSCOMP. The purpose of the current call is to retrieve the number of the compatible complex symbol from the array TREECS(M) where it has been saved. (Thismode is used in treating side effects.)

The parameters M and N point to either a tree node or to a complex symbol in the array CSLIST.

M,N > 0 ⇒ tree node

M,N < 0 ⇒ point'to a complex symbol

If M or N point to a tree node then CSCOMP uses the complex symbol attached to the tree node. If there is no complex symbol attached to the tree node, then CSCOMP creates a complex symbol which contains a single feature specification, a positive feature specification for

the category designated by the tree node. If the tree node has a complex symbol attached but this complex symbol does not contain a category feature specification, then CSCOMP creates the proper category feature specification and links it to the tree complex symbol. At this point CSCOMP has two complex symbols to test for compatibility, each of which contains a category feature specification if it is possible to determine one.

CSCOMP checks if each complex symbol has a category feature specification and, if both do, checks to see that the same feature appears in both complex symbols. This is necessary because category features are exceptions to the test for nondistinctness. If this check fails, CSCOMP returns the value 0 to indicate incompatibility.

If IND = 1, the nondistinctness test is made (CSNDST(MM,NN))
where MM and NN are the complex symbols derived from M and N .

If IND = 2, the inclusion test is mode (CSINC1(MM,NN)), MM and NN as above.

If these tests fail, the return value is 0 .

If the appropriate test succeeds, the complex symbols are merged to form a new complex symbol. (NNEW = CSMERG(MM,NN)).

Next asterisks which may appear in the complex symbol pointed to by NNEW must be considered. Asterisks appearing as values of features in a complex symbol indicates that the value of the feature may be either + or - with equal probability. Thus a complex symbol with k asterisks for feature values is really an abbreviation for 2^k complex symbols. The result of the compatibility test (if successful) is a complex symbol which does not contain the asterisk value. Therefore at this time we select at

random without replacement from the possible 2^k complex symbols and test each complex symbol for compatibility by expanding the complex symbol using the function REDRUL(NNEW') where NNEW' is the complex symbol NNEW with asterisk values changed to either + or - . If the redundancy rule expansion is successful, then the complex symbol pointer NNEW' is returned as the value of CSCOMP. If the expansion is unsuccessful, a new value assignment is computed and the expansion repeated with the new complex symbol NNEW'. The process continues until either a successful expansion is obtained or until all value assignments are exhausted. In the latter case, the return value of CSCOMP is 0 to indicate incompatibility. Value assignments are computed so that each possible value assignment has equal probability of selection. The limit on the number of asterisks that appear in a complex symbol has been arbitrarily set at four. To increase this limit, increase the size of the array ASTLST and increase the test value at statement label 121.

If the complex symbol pointed to by NNEW does not contain any asterisk values, then the complex symbol is expanded by the redundancy rules (REDRUL(NNEW)) and the appropriate result generated.

In summary, the value of CSCOMP is zero if the complex symbols are incompatible. If the complex symbols are compatible the value is a pointer to a new complex symbol obtained by merging the originals, selecting a value assignment for any asterisk values, and expanding by the redundancy rules.

If parameter M pointed to a tree node then the result of the compatibility test is saved in the array TREECS(M) ($1 \leq M \leq 400$) so that it may be retrieved on a later call to CSCOMP with IND = 3.

3.7 Analysis

CXIN, input routine for complex symbols, structural analyses and contextual features

CXIN is a somewhat involved input routine which is used to read three different types of objects. Since complex symbols may contain contextual features, and structural analyses may contain complex symbols, this program would have been much easier in any language which allowed recursive subroutine calls.

There are three entry points to the subroutine. The normal entry CXIN is for complex symbols, SLFEAT is for selectional features and ANALIN is for structural analyses. At each entry logical flags are set so that it is always possible to tell which entry was called.

Data Storage

Complex symbol storage

The function of CXIN is to store the input object in the appropriate arrays for later use. We first describe these arrays.

Complex symbols are stored in CSLIST which is a 4-row, 2000 column array, with current length = CSFRPT - 1. (This structure was chosen so that a single feature specification would be looked at either as four INTEGER*2 entries, or as one REAL*8 entry). Each column of CSLIST contains a feature specification consisting of feature number, feature type, and feature value and also a pointer to the next feature specification. A complex symbol number is a pointer to the first feature specification of the complex symbol, Subsequent feature specifications in the complex symbol are found by following the

pointers. The last feature specification of the complex symbol has a pointer of 0. The feature specifications of a single complex symbol are ordered by feature number. (The list structure of CSLIST was not exploited in most of the code; actually the feature specifications for any one complex symbol form an adjacent block in CSLIST.)

The feature number of a category or inherent feature (herein called explicit features) is obtained by a call to NUMNAM. Numbers are assigned in the order in which the features were first encountered. Explicit features not given in the prelexicon, but encountered later, are assumed to be inherent features.

The feature number of a contextual feature is 100 plus its position in the list of contextual features.

The type of a feature is 0 for category features, 1 for inherent features, and 3 for contextual features.

The value of a feature is 1 (+), 2 (-),3(*).

Storage of structural analyses

Structural analyses are stored in the parallel arrays ANALWP and ANALST with subsidiary arrays ANALWD (current length = ANALWT) and ANALPT (current length = ANALTP). ANALPT contains pointers to ANALWP and ANALST. The current length of ANALWP and ANALST is ANALPT(ANALTP). Structural analysis I is stored from ANALPT(I-1)+1 through ANALPT(I), ($I \geq 2$). ANALWD is a REAL array containing first "*" and " " and then the words which occur in analyses. Each symbol or group of symbols in the structural analysis goes into an entry in ANALPT. ANALWP contains pointers and other information:

Analysis item	ITYPE	ANALST	ANALWP
word	1	pointer to ANALWD	pointer to preceeding integer or 0
—	8	2	
*	1	1	
%	10	0	
integer	2	-(100+integer)	
complex symbol	3	-5	pointer to CSLIST
<	4	-1	pointer to >
—<	18	-2	
/<	17	-3	
—/<	16	-4	
>	5	-6	pointer to word preceding <
(of an option	6	-7	pointer to)
) of an option	7	-8	pointer to (
(of a choice	19	-9	pointer to ,
, of a choice	13	-10	pointer to , or)
(of a choice	20	-11	pointer to (

STORAGE OF A STRUCTURAL ANALYSIS

The values in ANALST and ANALWP are chosen for the convenience of the analysis routine (ANTEST). In CXIN the values for ANALST are stored in the array TTYPE, which is indexed by ITYPE, the internal numbers in CXIN for the symbols. This allows CXIN to be changed with relative ease.

Storage of contextual features

A contextual feature is a (special) structural analysis, enclosed within angular brackets. Contextual features are stored in SLCTPT(200,2), in which the first column contains a pointer to the structural analysis.* The second column is set to 0 by CXIN, but will be used by PRELEX to store a pointer to the restriction on the structural analysis. The current length of SLCTPT is SLCTTP. In order to be able to use the same sequence of numbers for all features, the feature number of a contextual feature is its position in SLCTPT + MXEXP (the maximum allowable number of explicit features).

(Names of contextual features appear only in the array SLNAME which is internal to subroutine NUMNAM.)

Initialization of Storage

The storage arrays of CXIN are initialized by the subroutine INIT which is at the beginning of every run. INIT does the following for CXIN:

```
ANALWD(l) = FSTAR    "*"
ANALWD( 2) = FLINE   "_"
ANALWT = 2
ANALPT(l) = 0
ANALTP = 1
SLCTTP = 0
CSFRPT = 1
```

Temporary storage areas in CXIN

Entities read by CXIN are not stored in the above arrays until they have been completely read in. Temporary storage is used during the read-in process.

*The mnemonics for contextual feature storage were created when we were calling them "selectional features", hence the "SL".

LEVEL and SLEVEL are used to record the current levels of complex symbols, and of analysis and contextual features. (Recall that the basic difficulty is that complex symbols may contain structural analyses which may contain complex symbols.) Initially both LEVEL and SLEVEL are set to 1. SLEVEL is incremented by 1 when the left bracket of a new contextual feature is encountered; it is decremented by 1 when a contextual feature is finished and stored into SLCTPT. LEVEL is incremented by 1 when a complex symbol is encountered in a contextual feature; it is decremented by 1 when a complex symbol is finished and stored into CSLIST.

SLPUSH and SLPUSN hold the values which will go into ANALST and ANALWP. The SLEVEL-th analysis is stored in' SLPUSH and SLPUSN from SLPHPT(-1) + 1 through SLPHPT(SLEVEL).

CSPUSH holds the values which will be stored in CSLIST. The LEVEL-th complex symbol is stored in CSPUSH from CSPHPT(LEVEL-1) + 1 to CSPHPT(LEVEL).

PUSH is a two-column array used as a push-down for terms in an analysis which will be needed to set up the backwards pointers in ANALWP. For the SLEVEL-th contextual feature, PUSHPT(SLEVEL) points to the first entry in PUSH for that feature, VLPUSH(SLEVEL) is the value of the feature,

Reading in a complex symbol

The above explanation of storage is intended to help explain how the three uses of CXIN for complex symbols, contextual features and analysis are interrelated. We now describe the behavior of the subroutine in each of these uses.

When CXIN is called, initialization steps set up the temporary arrays. SLFLAG and ANALFL are both set to false, so that the entry point

can be recalled. The parameter STAGE indicates what is expected next from FREAD. When STAGE = 1, the routine expects either a feature value, or a "|" which will terminate the complex symbol When STAGE = 2, a feature is expected. If the feature is a word, the associated feature number is retrieved from NUMNAM. The feature type is computed and the value, type and number are stored in CSPUSH. If the feature is a contextual feature, STAGE is set to 3 and a contextual feature is read (see below). (STAGE = 4 is an error skip.) When the complex symbol has been terminated by a "|", the feature specifications are sorted on feature number, a check is made to see that there is only one category feature, and the-complex symbol is moved into CSLIST. LEVEL is reduced by 1. If LEVEL = 1, ANALFL is FALSE and SLFLAG is FALSE, the parameter of CXIN is tested, and if = 1, the complex symbol is expanded by a call to REDRUL. CXIN then returns control. If the triple test above is not met, then we have just completed a complex symbol within an analysis, so the routine continues.

Reading in a structural analysis

In reading a structural analysis (either on a call of ANALIN or SLFEAT or within a complex symbol), SLEVEL is first increased by 1, STAGE is equal to 3, and then FREAD is used to read the entities of the analysis. As each entity is read a branch is made on the value of ISPEC returned by FREAD, and ITYPE is set to the internal number for the entity (in sorting it in CSPUSH and later in ANALST, TTYPE (ITYPE) will be used).

For each value of ITYPE the process is essentially the same. A check is made to see that-the entity can correctly follow the

previous ITYPE (now stored as NLAST). The value of SLPUSH (and hence of ANALST) is computed, from ANALWD for ITYPE=1, (100 + the integer) for ITYPE=2, and otherwise TTYPE(ITYPE). The value of SLPUSN (and hence of ANALWP) is computed by backing up in PUSH for entities which point backwards. Entities which are to be pointed to by subsequent entries are stored in PUSH, which contains in the first column a pointer to SLPUSH and in the second column a code for the type, KEEP(ITYPE). The entity is then stored in SLPUSH and the pointer in SLPUSN.

If a complex symbol is encountered, it is read as described above.

An analysis is terminated when either the ">" which corresponds to the initial "<" of a contextual feature or a period is found. The analysis is then compared with previous analyses so that it will not be stored twice. If it is new it is stored in ANALST and ANALWP and the routine continues if within a complex symbol, or otherwise terminates.

ANTEST, analysis

ANTEST is the subprogram which performs analysis (see AF-34 for a description of the analysis procedure; equivalent knowledge will be assumed in the current description). It is called with three arguments: TRANNO, TREETP, and ANALNO. Either TRANNO (for transformations) or ANALNO (for contextual features) is used to locate a structural analysis which has been coded into ANALST and ANALWP by subroutine CXIN; this structural analysis is copied into arrays ANLIST and ANWDPT in positions 1 to TPOSN. The method of finding the structural analysis and a pointer to its associated restriction is diagrammed in Figure 3.7-1. TREETP is a number indicating the location in TREE/FTREE of the top node of the sentence tree which is to be tested for analyzability.

ANTEST returns the value TRUE if the given sentence tree is analyzable as the given structural description, and FALSE if not. For a TRUE return, it further supplies (in the 50-position array NUMNOD) the positions of tree nodes which have been associated with numbered structural description nodes. Since some transformations require that all possible analyses be found, NUMNOD is dimensioned 50x10 so that ANTEST can return up to ten different analyses; in this case, NUMCNT will be the number of analyses actually found.

To simplify this description, we will use several words in unusual senses. A defnode will be anything in a structural description —— word, underline, asterisk, or boundary symbol —— which matches a single sentence node. This will free the word node to refer to only sentence-tree nodes. An option will be a choice with only one structural analysis in its clist of structural analyses; from here on, a choice will be a

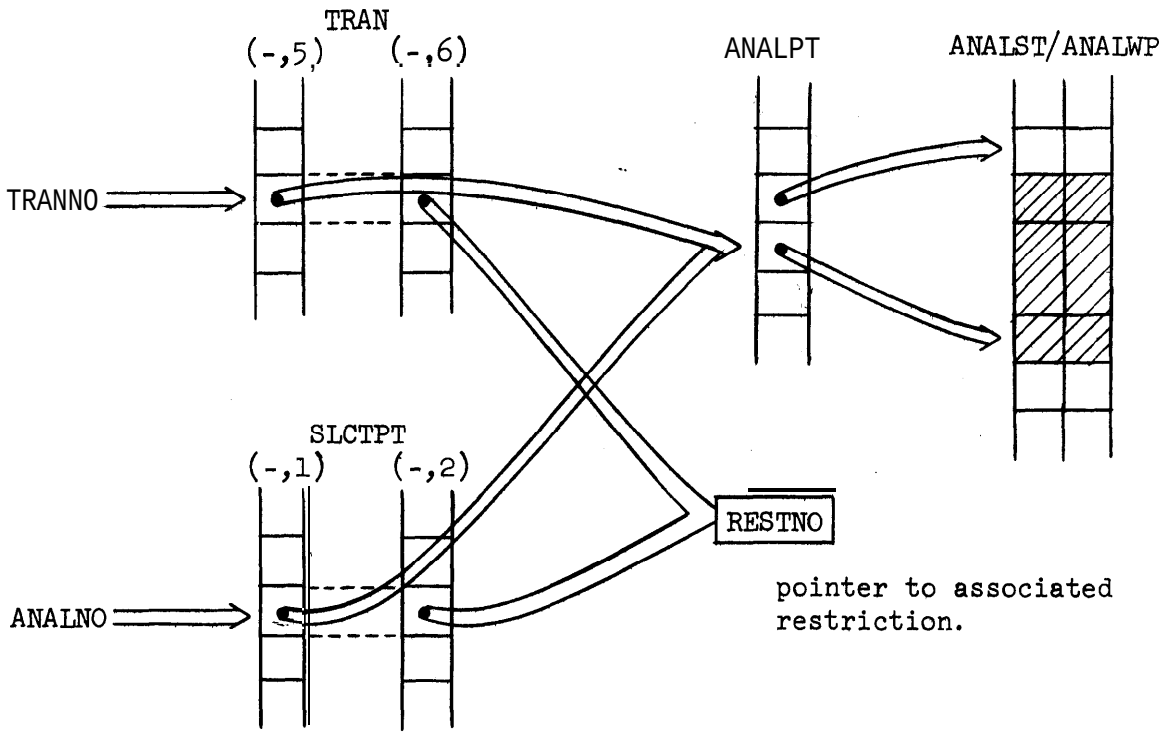

TRAN
(-,5)
(-,6)
ANALPT
ANALST/ANALWP
TRANNO
SLCTPT
(-,1)
(-,2)
RESTNO
ANALNO
pointer to associated
restriction.

Figure 3.7-1

choice (in the usual sense) which is not an option.

Analysis is probably the most complex single operation performed in this program system, because of the elaborate procedure that must be followed for matching and for backtracking when no match is possible. To simplify this procedure as much as possible, an elaborate system of pointers is set up during analysis. The backbone of this system is the five vectors ANSKIP, ANNODE, ANPREV, ANNEX, and ANPAR, which parallel the two vectors ANLIST and ANWDPT in which the structural analysis is stored.

Skips are ignored when first encountered; after the next defnode has been matched, a range must be assigned to preceding skips. For this purpose, the variable SKIP and vector ANSKIP are used. SKIP indicates the position of the last bypassed skip; ANSKIP points back to other preceding skips. ANSKIP is defined for all bypassed skips, matched defnodes, and the (of options (in case it is later decided that the option should not be taken) and choices. It always points to a preceding skip, and equals zero if there is no preceding skip. The skip routine (statement numbers 500-599) uses ANSKIP to find skips after a node has been matched, or at the end of an analysis level,

For backtracking when a match cannot be found, the variable PREV and vector ANPREV are used. PREV points to the previous significant item —— defnode or (or < —— preceding the current item. ANPREV continues the chain. ANPREV is defined for each defnode and (and < and > which is currently active (e.g., an option which has been bypassed is not active). It equals zero for the first significant item of the structural analysis and for every < . Using ANPREV, the backtracking routine (statement numbers 700-799 in the program) can thus easily find where it-is to restart the search after a mismatch.

The matching of a choice is aided by the vectors ANNEX and ANPAR and the variable PAR. PAR points to the opening (of the choice while a search is made for a match to the first defnode of the choice, and is zero otherwise; its major use is as a flag. ANNEX is used to chain together those defnodes which are possible first-defnodes of a choice. (See Figure 3.7-2 for a sample use of ANNEX and ANPAR). ANNEX of the (points to the first possible first-defnode, ANNEX of this defnode points to the next, and ANNEX of the last points to the) of the choice. If there is a choice within the choice which may be first, ANNEX points to the (of this inner choice, which is then chained as usual; the) of this choice then continues the chain. ANPAR is used for skips, since ANSKIP does not sufficiently define skips within choices; note, for example, that in Figure 3.7-2 the defnode C may be preceded by no skip or by the skip in position 1, depending on whether or not defnode B has been matched. ANPAR is defined for a defnode or (or , or skip, and points to the chain of skips which will precede a defnode if it is first in a choice. It is set negative when pointing to a (or , and positive when pointing to a skip. When a choice is encountered, the choice-setup routine (statement numbers 300-399) sets PAR; if this choice has not previously been seen, it also sets up the ANNEX and ANPAR chains. The skip routine uses ANPAR as well as ANSKIP to find skips; the matching routine (statement numbers 400-499) uses ANNEX to move through the chain of defnodes for a choice, and ANPAR to set up the proper pointers in ANSKIP. The backtracking routine uses ANPAR to aid the restart when it backs up into a choice.

The correspondence between defnodes and tree nodes is handled by the

Sample Use of ANNEX and ANPAR

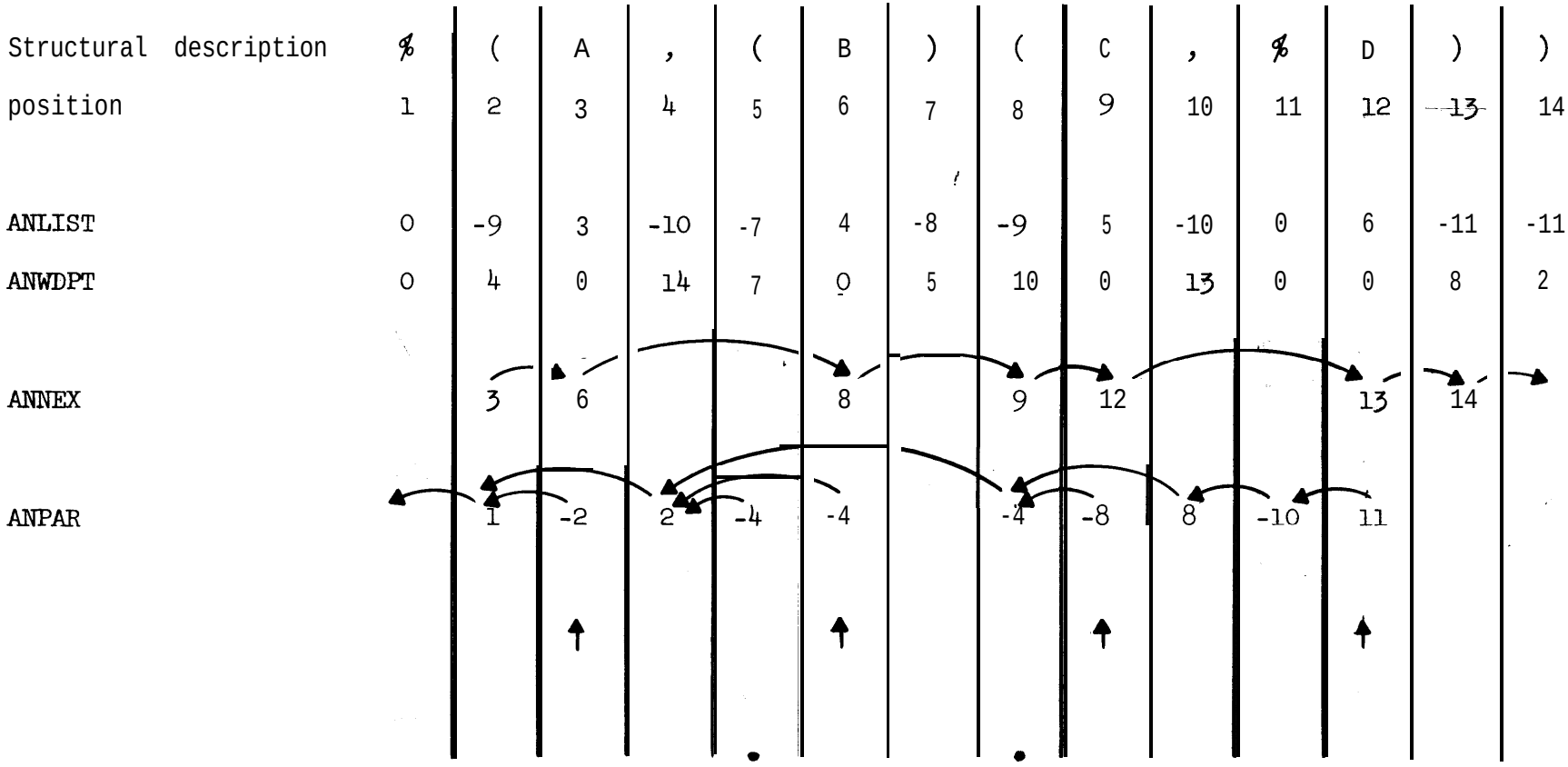


Figure 3.7-2

matching routine and by subprograms ANRTES, ANRUNS, and NEXT. The arrays NUMNOD and SKPNOD, vector ANNODE, and variables HERE and LAST contain pointers to tree nodes. ANNODE is defined for defnodes and skips. For defnodes, it contains the position of the matching tree node; it is assigned on a match, and reset during backtracking. For skips, it points to SKPNOD, a 200x2 array which points to the preceding and following matched tree nodes, or is set to -1000 for a null skip; it is assigned in the skip routine and reset during backtracking. HERE points to the tree node currently being tested for a match; it is advanced in NEXT (see below) and reset on a backtrack. LAST is the last matched tree node, used only to appropriately -set the first column of SKPNOD.

Subprogram ANRTES is called after a match has been found. It checks restrictions and complex symbols, and sets NUMNOD. The first step is to move through the chain of pointers to numbers set in ANWDPT by the input routine CXIN. For each number, it copies the tree node pointer into the number-th position of NUMNOD and calls RESTST to test any restrictions associated with that number. If all succeed, and if there is a complex symbol, it calls CSCOMP to check that it corresponds to the tree complex symbol (inclusion for transformations, compatibility for contextual features). ANRTES then returns TRUE if all tests succeed and FALSE on failure.

Subprogram ANRUNS reverses the procedure of ANRTES; it calls RESTUN instead of RESTST and restores NUMNOD. It is called by ANRTES on failure and by the backtracking routine. The miscellany with-1000 in ANRTES and ANRUNS is occasioned by the problem of telling whether a defnode has simply not yet been reached, or has been bypassed (and thus made explicitly null). ANNODE will be zero for not-yet-reached defnodes and skips, -1000 for null skips, -2000 for bypassed skips (set during the

final routine (statement numbers 800-899)), and 1000 for bypassed defnodes. NUMNOD combines the -1000 and -2000 into -1000. Since a number may be assigned to several defnodes or skips (particularly in the case of choices), no testing or reassignment in NUMNOD should take place during the final routine if NUMNOD has already been set. Finally, in ANRUNS it is impossible to tell, if NUMNOD and ANNODE are both -1000, whether NUMNOD should be reset; for this reason, ANNEX of a skip (not otherwise used, thus usually zero) is set to 1 if this skip's ANNODE is not to unset NUMNOD.

Levels should be discussed before proceeding to NEXT. For these, the vector LEVTOP and variables LEVEL, TOP, and HERE are used. LEVEL is initially zero; one is added to it every time the program enters an angle-bracketed subanalysis, and one is subtracted at the end of processing the subanalysis. TOP is initially set equal to TREETP (the top node of the tree under consideration); every time a subanalysis is entered, the current TOP is saved in LEVTOP and a new TOP is created pointing to the tree node which matches the defnode heading the subanalysis; at the end of the subanalysis, the old TOP is restored. TOP is set-negative if the < of the subanalysis does not have a / preceding it (an immediate constituent analysis). HERE is set negative at the beginning of a subanalysis to flag the beginning. This processing takes place in the levels routine (statement numbers 600-699). The levels routine also checks success or failure of subanalyses. If the < was preceded and analysis reaches the > either at the righthand side of the subtree or with a skip preceding, the subanalysis succeeds and analysis continues. Otherwise, the subanalysis fails, and backtracking is begun at the defnode heading the subanalysis.

Subprogram NEXT finds the next node in the tree. It has three arguments: HERE, the previous node, TOP, the top node of the tree under consideration, and SIGN, a flag which takes on the values -2, -1, 0, 1, 2. The new node is returned as a new HERE; HERE is set to zero if there is no next node. NEXT is complicated by several features. The very first node that should be examined is the topmost node in the tree; this is indicated by HERE=0. This, however, is true only for the first level of analysis; for subanalyses, the first daughter of the top node is the first node to be tried. Thus all subanalyses commence with HERE set negative and equal in absolute value to TOP. Thereafter, the procedure depends on whether there was a / preceding the subanalysis. If so, or if this is the topmost level, TOP will be positive and a search will be made for daughters of HERE (the leftmost of which will be taken as a new HERE), and if there are no daughters the search will continue to the right of HERE, but not going above TOP. If there was no / then TOP is negative; in this case, the search will be immediately for right sisters of HERE. If no next node exists, HERE is set to zero.

Skips introduce a further complication. If there is a skip preceding the current defnode, it is all right to leave dangling tree branches behind, but not if there is no skip. SIGN is used for this purpose. When matching is first attempted, SIGN is set to zero and remains zero; after a failure to match, SIGN is set to 1 and changes to -1 when a branch is skipped (that is, when the old HERE has no daughters). Also, the fact that SIGN=0 indicates that no attempt should be made to find daughters of HERE; HERE has already been matched, so its daughters are unavailable. SIGN is set to 2 when no next node exists.

Choices require more machinery. After skipping a branch during a choice search, it is fairly easy to thereafter examine only those first-defnode candidates which are preceded by skips; however, it would be much nicer to quit immediately if none of them are preceded by skips. For this reason, SIGN is set to -2 if it is -1 and a first-defnode preceded by a skip is about to be tested; if SIGN is still -1 after ckecking all candidates, the match routine exits immediately to backtrack instead of fruitlessly advancing HERE through the rest of the tree.

The end of a level requires a check to see if any more nodes exist to the right of the last-matched one, plus an assignment of range if a skip is rightmost in the level. For this purpose, NEXT is entered with SIGN=2. If no next node exists, SIGN will still be 2 and HERE will be zero; any skip will have null range. If one exists, SIGN will be set to -1 and HERE will be set to minus the absolute value of TOP, which is the appropriate value to insert into SKPNOD to indicate a level-ending skip.

Minor points not yet covered include POSN, which points to the current position of the AN---- arrays. The scan section (statement numbers 200-259) decides which other section of the program is to be called next, on the basis of what kind of (WASFUR) thing is at the current POSN. DEFNOD, set to ANLIST of the current defnode in the match section, points to ANALWD, which contains names of defnodes; the first two entries in ANALWD are permanently set to be * and — . CYCL indicates cyclicity of transformations; as used by ANTEST, a value of 0 (contextual feature) or 1 or 2 means to find at most one analysis,

while a value of 3 or 4 means to find all possible analyses. If the parameter TRANNO is negative, it means that the current structural description and top tree node are the same, but the tree has been shuffled around since last time; this is simply to save setup time, since vectors ANLIST, ANWDPT, ANNEX, and ANPAR are already in place.

3.8 Programs for Restrictions

This section describes a set of subroutines (RESTIN, RESTST, RESTUN, and RESTPR) which manipulate restrictions. They input, test and set, unset and print restrictions, respectively; GTOKEN is a "work routine" for RESTIN. Input includes translation into internal format and storage of the restrictions.

The description of a restriction is given in Figure 1* below. The description of the internal format and the composition of restriction storage (/RESTCM/ - restriction common block) is given below.

RESTIN is called by any routine requiring a restriction input. The primary routine calling RESTIN is TRANIN. Input to RESTIN is completely free field and is read by FREAD. RESTIN calls GTOKEN and CXIN which generate the next token and read in a complex symbol, respectively. RESTIN returns the number of the restriction it just read in.

RESTST is called by a routine which needs to know if an analysis satisfies a particular restriction or not. It returns true or false, although internally it may find a restriction to be "undefined' (as a result of a reference to a node which has yet to be assigned). "Undefined" values are interpreted as true. RESTST calls CXEQ and CXINC to determine if complex symbols are equal or included in one another, respectively.

*The references to figures in section 3.8 are to figures 3.8.1 to 3.8.10. These figures are found at the end of this section.

RESTUN is called by any routine which needs to unset (reset) a node without unsetting (resetting) the whole restriction. Uneetting means . setting to undefined all conditions which refer to the given node. RESTUN may also be used to completely reset a restriction.

RESTPR will print a given restriction or print all the restrictions, It is essentially a dump of /RESTCM/ .

GTOKEN is described in the description of the operation of RESTIN below.

- INTERNAL STORAGE -

Restrictions are stored in the common block /RESTCM/ given in Figure 4. The present capacity is about 150 restrictions. Capacity is determined by the size as well as number of restrictions. The I-th restriction read is stored as follows:

RESTS(I*4-3) = value of restriction I:
0 = false, 1 = true, 2 = undefined

RESTS(I*4-2) = pointer to first entry in RESTR

RESTS(I*4-1) = pointer to last entry in RESTR

RESTS(I*4) = pointer to first entry in CONDS

RESTR(J) = if > 0: a pointer to first entry in CONDS
if < 0: -1 = logical OR
-2 = logical AND
-3 = logical NOT

CONDS(K) = value of the condition:
0 = false, 1 = true, 2 = undefined

CONDS(K+1) = coded type of condition:
type = N*100+L, for the L-th N-ary restriction
(at present there are only 1-ary and 2-ary restrictions, so all types are in the range:
100 < type < 300)

CONDS(K+2) = first argument of restriction, it is always a number > 0 which refers to a particular node

CONDS(K+3)	= if > 0:	the number of a particular node
. . .	if< 0:	a pointer into WD (designating a
CONDS (K+1+N)		word e.g., "S", "PRED", or "ADS")(in
		particular, to the -CONDS(M)'th word)
	if < NWD:	a pointer to a complex symbol which has
		been read in by CXIN . (In particular,
		to the -CONDS(M)+NWD'th complex symbol.)
		NWD is currently 100.

1) RESTS (restrictions) is always a multiple of 4 times the number of restrictions in length; i.e., each restriction takes exactly four locations in RESTS.

2) RESTR (restriction tree) is of arbitrary length for each restriction, but will always be at least one location long. The contents of RESTR is the Polish postfix for the restriction read in. It is composed completely of references to conditions (the basic primitives, e.g., "TRM", "NUL" , "NDOM", etc.) and references to logical operators (e.g., "AND", "OR", "NOT") .

3) CONDS (conditions) is the list of primitives which comprise the restriction. Each condition is always at least 3 locations long, and 'in general will be N+2 locations long for each N-.ary condition.

- RESTIN: OPERATION -

CALL: REST IN (ONE)

RESTIN is self-initializing: it initializes/RESTCM/ the first time it is called. RESTIN provides checks to see that the capacity of /RESTCM/ is not exceeded.

RESTIN utilizes the "railway shunt algorithm" to create a restriction in storage, It runs using a token generator to provide the next token. Tokens are of two types: conditions and operators. Conditions are returned by the token generator in an array called "TOK". Operators are returned as numbers > 0 in "OPFG".

The effect is to "compile" a logical expression composed of conditions: the logical relations and pointers to the conditions appear in RESTR in Polish postfix, and the conditions themselves appear in CONDS.

RESTIN will input an arbitrarily complex logical combination of restrictions.

GTOKEN generates the next token for RESTIN. It operates as follows:

1) Read the next symbol (by calling FREAD)

2) Test to see if it is a number, if so, go to (6)

3) Test to see if it is a "reserved word", if so, go to (5)

4) Find which logical operator it is and return

5) Find which 1-ary operator this is, generate the array TOK: containing the index of the condition followed by its argument; return

6) Find which N-ary operator this is, generate the array TOK: containing the index of the condition followed by its N arguments; return

Every time GTOKEN encounters a "reserved word" (e.g., "ADJ", "PRED", "NOUN", etc.) it searches the ar-ray WC for a copy. If it finds one, it uses the negative of the index into WD. If it does not find a copy, it generates one and uses the negative of the new index into WD. Every time GTOKEN encounters a complex symbol, it calls CXIN to input it, . CXIN returns the number of the complex symbol. GTOKEN fills in TOK with minus this number minus NWD. NWD is presentiy set at 100. If there is any possibility of there being more than 100 "reserved words", NWD should be made larger: the test routine discriminates between words and co-mplex symbols by comparing their indices with NWD.

The lengths of RESTS, RESTR and CONDS are currently 500 each, LRESTS, LRESTR and LCONDS (data initialized varia'bles in RESTIN) should always be set to the lengths of their respective arrays at compile time. Ths capacity of /RESTCM/ is approximately LCONDS/3.5 restrictions.

- RESTST: OPERATIONS -

CALL: RESTST(I, POSN)

RESTST both sets and tests the restriction designated 'by I or CREST (CREST = current restriction; this variable is in /RESTCM/), and it returns true or false accordingly.

Every time the current restriction is changed, RESTST automatically resets it before testing, If the current restriction is the same as the one at the last call to RESTST, RESTST saves time by not resetting the restriction first.

If POSN is zero, CREST is set to I and restriction I is completely reset before it is tested, If POSN is non-zero, RESTST tests restriction CREST.

RESTST interprets the Polish postfix in RESTR: it acts like a Polish postfix machine. Each reference to a condition is interpreted to mean:

1) If the condition 'has value true or false, load the value stack with this value.

2) If the condition is undefined, evaluate the condition.

3) Load the stack with the value of the expression,

RESTST condenses each pair of conditions whose names are (NAME) and N(NAME) (referring to the normal and negative forms of a condition) into one evaluation via the varia'bie "'NORMAL". The value of a condition winds up in the variable "CVAL".

RESTST will evaluate an arbitrarily complex logical combination of conditions. It uses the truth tables in Figure 6 to evaluate the restriction. If the final value is undefined, RESTST will return true. Figure 7 gives a ta'ble of arguments and actions for RESTST.

- RESTUN: OPERATION -

CALL: RESTUN(I,POSN)

RESTUN unsets the condition designated by I . If POSN is zero, it will set CREST to I and then completely reset restriction I . RESTUN does this by setting RESTS(CREST*4-3) to undefined, and then setting each component condition of the restriction to undefined.

If POSN is greater than zero, the argument refers to a node in the analysis using the current restriction, All. component conditions which reference this node are set to undefined,

RESTUN accomplishes the unsetting by going down CONDS and utilizing the coded information therein.

Figure 8 gives a table of arguments and results for RESTUN.

- RESTPR: OPERATION -

CALL: RESTPR(I)

RESTPR prints the restriction designated by the parameter I . If this parameter is zero, it prints all the restrictions. See Figure 9 for sample output.

- FIGURE 3.8.1 -

SYNTAX OF RESTRICTIONS

RESTRICTIONS ::= *RES RESTRICTION .

RESTRICTION ::= BOOLEANCOMBINATION (CONDITION)

CONDITION ::= CONDITION1|CONDITION2|CONDITION3

CONDITION1 ::= RELATION1 RIGHT-PART

CONDITION2 ::= INTEGER RELATION2 RIGHT-PART

CONDITION3 ::= INTEGER RELATION3 RIGHT-PARR, RIGHT-PART

RIGHT-PART ::= INTEGER|WORDY|COMPLEX O L

RELATION1 ::= TRM|NTRM|NUL|NNUL

RELATION2 ::= EQ|NEQ|DOM|NDOM|HAS|NHAS|EQCS|NEQCS

RELATION3 ::= EMPTY (There are no 3-ary conditions as yet.)

NOTE 1: The definitions of the relations are found in Figure 3.8.10.

NOTE 2: Although the syntax will allow the creation of almost arbitrary constructs, not all of them have meaning, The input routine (RESTIN) will not detect any meaningless constructs: it will accept any syntactically correct restriction, Only during the evaluation of the restriction (via RESTST) will the error be detected,

NOTE 3: Additional relations will be included as they are found to be useful.

- FIGURE 3.8.2 -

TABLE OF ALLOWABLE ARGUMENTS

RESTRICTION	CODE	# ARGS	ARGUMENT 1			ARGUMENT 2		
			INTGR	WORD	C-SYM	INTGR	WORD	C-SYM
TRM	101	1	x					
NTRM	102	1	x					
NUL	103	1	x					
NNUL	104	1	x					
EQ	201	2	x			x		
NEQ	202	2	x			x		
DOM	203	2	x				x	
NDOM	204	2	x				x	
HAS	205	2	x			x		x
NHAS	206	2	x			x		x
EQCS	207	2	x			x		x
NEQCS	208	2	x			x		x

- FIGURE 3.8.3 -

SUBPROGRAM CALL/RESULT TABLE

SUBPROGRAM	TYPE	CALLS	RESULTS
RESTIN(ONE)	I*2	GTOKEN	Returns number of the restriction it reads in and stores restriction.
RESTST(I,POSN)	L*1	CSEQ CSINC	Returns true/false depending on whether the restriction is satisfied or not.
RESTUN(I,POSN)	S	---	Unsets a restriction; returns nothing,
RESTPR(I)	S	---	Prints a restriction; returns nothing.
GTOKEN(SYM)	S	FREAD CXIN	Returns a token: logical operator or condition.

NOTE 1: TYPE: I*2 - INTEGER*2 function
L*1 - LOGICAL*1 function
S - subroutine

NOTE 2: ONE = dummy argument
1 = restriction number
POSN = position in an analysis
SYM = array internal to RESTIN

- FIGURE 3.8.4 -

COMMON BLOCK FOR RESTRICTIONS

```
COMMON /RESTCM/ WD,CREST,PS,PR,PC,PW,RESTS(500),RESTR(500),     RESTCM1
                CONDS (500)                                     RESTCM2
REAL*8 WD(100)                                                  RESTCM3
```

COMMON BLOCK FOR GTOKEN

```
COMMON /RTOKEN/ OPFG,LTH,TOK(10)                                RTOKEN
```

DATA VARIABLES IN RESTIN

```
INTEGER*2 LRESTS/500/,LRESTR/500/,LCONDS/500/,NWD/100/
```

- FIGURE 3.8.5 -

SAMPLE RUN

EXAMPLE 1:

```
            NUL  5  .
RESTS()     2    1  1  1     undf'd,ptr1,ptr2,ptr3
RESTR()     1                ptr4
CONDS()     2  103  5        undf'd,NUL,5
WD()                         empty
```

ptr1 - points at the 1 in RESTR()

ptr2 - points at the 1 in RESTR()

ptr3 - points at the 2 in CONDS()

ptr4 - points at the 2 in CONDS()

EXAMPLE 2:

```
            3  DOM  4
RESTS()     2    2  2  4     undf'd,ptr1,ptr2,ptr3
RESTR()     4                ptr4
CONDS()     2  203  3  4     undf'd,DOM,3,4
WD()                         empty
```

ptr1 - points at the 4 in RESTR() .

ptr2 - points at the 4 in RESTR()

ptr3 - points at the 2 in CONDS()

ptr4 - points at the 2 in CONDS()

EXAMPLE 3:

```
                TRM  7  &  6  HAS  |=+HUMAN|  .
RESTS( )        2    3     5     8    undf'd,ptr1,ptr2,ptr3
RESTR( )        8    11    -2         ptr4,ptr5,&
CONDS( )        2  101     7     2   205  6  -101
                                      undf'd,TRM,7,undf'd,HAS,6,|=+HUMAN|
WD( )                                 empty
```

ptr1 - points at the 8 in RESTR( )

ptr2 - points at the -2 in RESTR( )

ptr3 - points at the first 2 in CONDS( )

ptr4 - points at the first 2 in CONDS( )

ptr5 - points at the second 2 in CONDS( )

EXAMPLE 4:

```
          ¬ ( ( 6 DOM PRED |5 NDOM VP ) &8 NEQCS |=-HUMAN| ) .
RESTS( )    2    6   11   15   undf'd,ptr1,ptr2,ptr3
RESTR( )   15   19   -1   23   -2   -3
                                 ptr4,ptr5,| ,ptr6,&,¬
CONDS( )    2' 203   6   -1    2 204 5    -2    2    208 8   -102
                                 undf'd,DOM,6,PRED,undf'd,5,VP,
                                 undf'd,NEQCS,8,|=-HUMAN|
WD( )       PRED  VP
```

ptr1 - points at the 15 in RESTR( )

ptr2 - points at the -3 in RESTR( )

ptr3 - points at the first 2 in CONDS( )

ptr4 - points at the second 2 in CONDS( )

ptr5 - points at the third 2 in CONDS( )

Note that positive numbers in CONDS() refer to nodes in the analysis, and that negative numbers are pointers. Pointers of magnitude less . than 100 refer to WD(); pointers of magnitude greater than 100 refer to complex symbol storage.

All restrictions go in with value undefined,

- FIGURE 3.8.6 -

TRUTH TABLES--FOR RESTST

	NOT
F	T
T	F
U	U

AND	F	T	U
F	F	F	F
T	F	T	U
U	F	U	U

OR	F	T	U
F	F	T	U
T	T	T	T
U	U	T	U

F = false (coded as 0)

T = true (coded as 1)

u = undefined (coded as 2)

- FIGURE 3.8.7 -

RESTST: TABLE OF ARGUMENTS AND RESULTS

I \ POSN	=0	>0	<0
>0	CREST=I test & set	test & set using current	illegal
=0	illegal	illegal	illegal
<0	illegal	illegal	illegal

- FIGURE 3.8.8 -

RESTUN: TABLE OF ARGUMENTS AND RESULTS

I \ POSN	=0	>0	<0
>0	CREST=I reset restrct I	reset all refs to node POSN in restrct I	illegal
=0	illegal	illegal	illegal
<0	illegal	illegal	illegal

- FIGURE 3.8.9 -

RESTPR: SAMPLE OUTPUT

```
RESTRICTIONS IN


                    6 ROOM SS | 6 EQ 5.

JUST READ RESTRICTION      9


                  - RESTS -
REST #  VALUE   PTR 1   PTR 2   PTR 3

   9       2       24      26      48


                  - RESTR
REST #  TREE

   9      48    52    -1


                  - CONDS -
REST #  VALUE   TYPE   ARGUMENTS

   9       2      204    8    -4
           2      201    6     5


                  - WORDS -
/S        /ALPHA    /FOO      /SS       /


                  - RESTS -
REST #  VALUE   PTR 1   PTR 2   PTR 3

   1       2        1       4       1
   2       2        5       5       8
   3       2        6      10      12
   4       2       11      12      24
   5       2       13      15      28
   6       2       16      21      32
   7       2       22      22      40
   8       2       23      23      44
   9       2       24      26      48
```

- RESTR -

KESJ #	TREE					
1	1	0	-2	-1		
2	8					
3	12	10	-1	20	-1	
4	24	-3				
5	28	-3	-1			
6	-3	32	-3	36	-3	-1
7	40					
8	44					
9	48	52	-1			

- CONDS -

REST #	VALUE	TYPE	ARGUMENTS
1	2	201	1 -1
	2	103	-1
2	2	205	5 -101
3	2	204	6 -2
	2	207	7 -102
	2	203	17 -3
4	2	205	4 -103
5	2	206	8 -104
6	2	207	4 -105
	2	207	5 7
7	2	203	3 5
8	2	203	7 -3
9	2	204	8 -4
	2	201	6 5

- WORDS -

/S /ALPHA /FOO /SS /

- Figure 3.8.10 -

DEFINITION OF RELATIONS

FORMS

RELATION1 INTEGER1

INTEGER2 RELATION2 INTEGER3

INTEGER2 RELATION2 WORD1

INTEGER2 RELATION2 COMPLEX SYMBOL

where RELATION1 is one of the unary relations and

RELATION2 is one of the binary relations

DEFINITION

NAME	RESULT
TRM	true if node INTEGER1 is terminal
	false if node INTEGER1 is not terminal
NTRM	same as ¬ TRM
NUL	undefined if node INTEGER1 has yet to be assigned
	false if node INTEGER1 has been assigned
	never true
NNUL	undefined if node INTEGER1 has yet to be assigned
	true if node INTEGER1 has been assigned
	never false
EQ	true if node INTEGER2 is equal to node INTEGER3:
	has same substructure and complex symbols are equal
	(uses CSEQ to test complex symbols)
	false if not equal

NEQ same as ¬ EQ

DOM true if substructure of node INTEGER2 includes a WORD equal to WORD1. Does not search below an S

false if not equal

NDOM same as ¬ DOM

HAS true if node INTEGER2 and node INTEGER3 have non-conflicting complex symbols or if node INTEGER2 and COMPLEX SYMBOL are non-conflicting. (Uses CSINC to test complex symbols)

false if not

NEQCS same as ¬ EQCS

NOTE: All relations (except NUL & NNUL) are undefined if at least one operand is undefined

3.9 Structural change

CHANIN, Input routine for structural change.

CHANIN is an INTEGER*2 function of one dummy INTEGER*2 argument. CHANIN reads in the instruction part of a structural change, stores it, and returns a pointer to the instruction.

The syntax of structural change is given in Appendix A (5.01 - 5.10). The formats restriction, tree, and complex symbol are given elsewhere in the descriptions of the subroutines which read and store them. CHANIN is called by TRANIN after it reads an SC . CHANIN reads the structural change and stores it, and returns after reading a period. CHANIN calls RESTIN(ONE), FTRI(TWO), and CXIN(ONE) to read a restriction, tree, or complex symbol.

Internal storage

A pointer to the structural-change for the J-th transformation is stored by TRANIN in TRAN(J,7) . The instruction is stored by CHANIN in the COMMON block /CHANCM/.

Initialization: /CHANCM/ is initialized by a BLOCK DATA program given in section 5. CHAN, CHWORD, FCHTRE, CHTREE, and CHCLIS are initially empty. OPLIST is initialized to contain the list of operators and complex operators. The current sizes of CHAN, CHWORD, CHTREE, CHCLIS, and OPLIST are NCHAN, NCHW, NCHT, NCHCL, and NOPL; the maximum sizes are MXCHAN, MXCHW, MXCHT, MXCHCL and MXOPL.

Each change instruction is stored in a line of CHAN. A change is stored in CHAN as follows:

CHAN(I,1) = type of first argument

0 if none
1 if integer
2 if word
3 if '(tree)
4 if complex symbol

CHAN(I,2) = first argument

if type 1 then the integer
if type 2 then a pointer to CHWORD
if type 3 then a pointer to CHTREE
if type 4 then a pointer to CSLIST

CHAN(I,3) = index of the operator or complex operator in OPLIST

CHAN(I,4) = type of first second argument 0, 1, or 2 as above

CHAN(I,5) = second argument

as for first argument

CHAN(I,6) = pointer to next instruction to be done (0 if none)

A conditional change is also stored in a line of CHAN, but the allocation is different:

CHAN(I,1) = 6

CHAN(I,2) = pointer to the restriction

CHAN(I,3) = pointer to the next instruction to be done if the restriction is met (0 if none)

CHAN(I,4) = pointer to the next instruction to be done if the restriction is not met (0 if none)

CHAN(I,5) = CHAN(I,6) = 0

CHWORD is simply a REAL*8 list of words. FCHTRE, CHTREE, and CHCLIS store trees as described in section 3.3.

The final setting of the pointers to the next instruction is not done by CHANIN proper, but by the entry CHANTY, which tidies up the table CHAN. This entry is called by TRANIN after all the structural changes for the grammar have been read. A call to CHANOU causes the structural change tables to be output.

The output of CHANOU is shown in Figure 3.9.1.

Figure 3.9.1

STRUCTURAL CHANGES

STRUCTURAL CHANGES

I	TYPE	ARG	OP	ARGT	ARG	NEXT
1	6	1	?	3	0	0
2	2	1	1	1	4	0
3	2	2	1	1	4	0
4	2	1	1	1	5	0
5	2	3	1	1	4	0
6	1	5	4	1	3	0
7	0	0	11	1	4	0
8	1	3	5	1	10	9
9	1	7	6	1	3	10
10	2	4	1	1	5	11
11	2	5	1	1	5	0
12	1	4	2	1	5	0
13	1	6	6	1	3	0
14	1	6	2	1	4	0
15	1	4	4	1	5	0
16	1	4	4	1	3	0
17	6	4	18	19	0	0
18	4	9	12	1	4	0
19	4	10	12	1	4	0
20	2	6	3	1	6	0
21	0	0	11	1	3	0
22	0	0	11	1	3	0
23	2	7	5	1	5	0
24	6	8	25	26	0	0
25	4	17	12	1	4	0
26	4	18	12	1	4	0
27	0	0	11	1	5	0
28	0	0	11	1	4	29
29	0	0	11	1	5	30
30	0	0	11	1	6	0
31	1	4	2	1	3	0
32	1	3	3	1	4	0
37	1	5	2	1	3	0
34	2	5	3	1	3	35
35	2	8	3	1	3	0
36	0	0	11	1	5	0
37	0	0	11	1	5	0
38	1	3	2	1	7	39
39	1	4	1	1	7	40
40	1	5	1	1	7	0
41	1	2	4	1	5	0
42	1	4	4	1	9	0
43	1	4	8	1	3	0
44	1	4	2	1	2	0
45	1	5	2	1	3	0
46	1	3	4	1	4	0
47	0	0	11	1	3	0
48	0	0	11	1	3	0
49	0	0	11	1	3	0
50	0	0	11	1	5	0
51	1	4	?	1	3	52
52	1	5	2	1	3	53

53	1	6	2	1	3	0
54	2	0	11	1	4	0
55	1	0	11	1	4	0
56	2	0	11	1	3	0
57	2	0	11	1	1	58
58	2	0	11	1	3	
59	2	9	5	1	2	
60	2	0	11	1	3	0
61	2	0	11	1	3	0
62	2	10	3	1	2	
63	2	10	5	1	3	
64	2	11	5	1	3	0
65	2	11	5	1	2	
66	2	12	5	1	7	0
67	2	13	5	1	2	0
68	2	14	5	1	2	
69	0	0	11	1	3	70
70	2	15	5	1	3	0
71	0	0	11	1	7	72
72	2	16	5	1	3	0
73	0	0	11	1	7	74
74	2	17	5	1	7	0
75	0	0	11	1	2	76
76	2	18	5	1	3	0
77	0	0	11	1	7	78
78	2	19	5	1	3	0
79	0	0	11	1	7	80
80	2	8	5	1	3	0
81	0	0	11	1	2	82
82	2	20	5	1	7	0
83	0	0	11	1	3	0
84	0	0	11	1	3	0
85	0	0	11	1	3	0
86	2	3	5	1	2	0
87	0	0	11	1	2	88
88	2	21	5	1	7	0
89	0	0	11	1	7	90
90	2	22	5	1	3	0
91	0	0	11	1	2	92
92	2	23	5	1	3	0
93	0	0	11	1	2	0
94	2	24	5	1	2	0
95	2	25	5	1	2	0
96	2	10	5	1	3	0
97	2	26	5	1	2	
98	2	27	5	1	2	
99	2	28	5	1	2	0
100	2	29	5	1	7	

DIGRAPHS

1 COMMA	2 CAPNUM	3 THAT	4 RE	5 EN	6 EVER	7 OFF	8 HAVE	9 ED
10 S	11 NOT	12 DID	13 DOES	14 ON	15 IS	16 ARE	17 WAS	18 WERE
19 HAS	20 HAD	21 WHO	22 WHICH	23 WI-AT	24 FOR	25 TO	26 ING	27 RY
28 A	29 THE							

Figure 3.9.1 (cont.)

CHANGE, control program for structural change

When a transformation is to be applied, CONTRL calls CHANGE. CHANGE makes a subroutine call for each of the change operations in the structural change. If the operation is a tree operation, ELEMOP is called; if it is a complex symbol operation, CSEXCH is called. When all of the change operations have been performed, CHANGE relinquishes control to CONTRL.

ELEMOP, elementary tree operations

ELEMOP performs one tree operation for each call from CHANGE, and then returns. ELEMOP also contains separate entries for a subset of the tree operations and is occasionally called by other subroutines. For example, GEN calls the entry ALADE in building a tree.

The elementary tree operations of ELEMOP are those of the MITRE grammars and those of the IBM core grammar. The MITRE operations are:

SUBST	SUBSE	substitution
ADRIS	ARISE	add as right sister
ADLES	ALESE	add as left sister
ADFID	AFIDE	add as first daughter
ADLAD	ALADE	add as last daughter
ADRIA	ARIAE	add as right aunt
	ERASE	erase

The operations in the left-hand column first make a copy of the subtree to be adjoined, and then adjoin the copy; those in the right-hand column move the original subtree to the new position, thus effectively erasing the original. The IBM operations are:

SUBSTI	SUBSEI
ADRISI	ARISE1

ADLESI ALESEI

ADLADI ALADEI

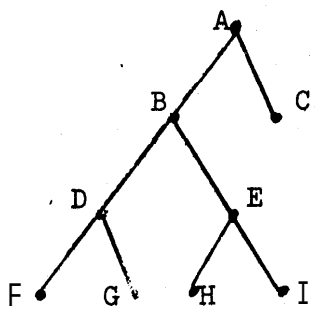
ERASEI

The IBM operations differ from the MITRE operations in that in general in the IBM operations chain upward from the named nodes. Description of the individual IBM operations follow.

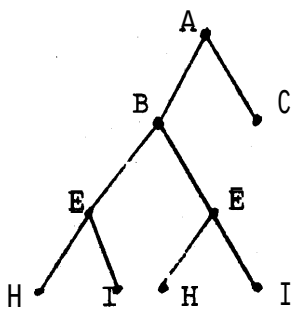
Substitute: SUBSTI (N1,N2) and SUBSEI (N1, N2)

Both SUBSTI and SUBSEI substitute the subtree headed by N1 for the subtree headed by N2; in addition, SUBSEI erases the original occurrence of N1.

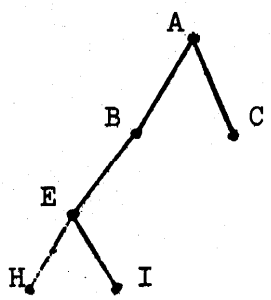
Given the tree:



SUBSTI (E,D) produces:

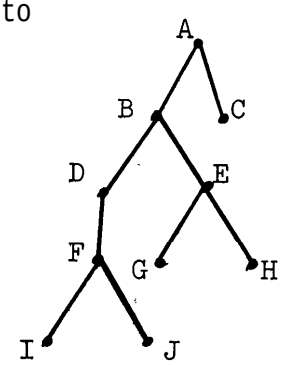


while SUBSEI (E,D) produces:

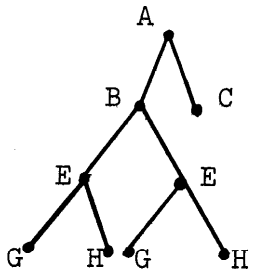


If immediately above N1 there is a non-branching chain of nodes, the top of that chain is used instead of N1; similarly, the top of any

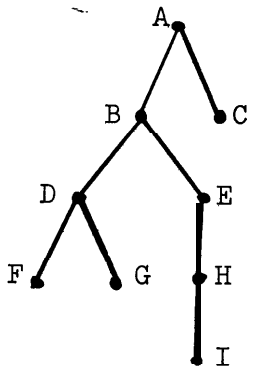
non-branching chain above N2 is used instead of N2. Thus, SUBSTI(E,F) applied to



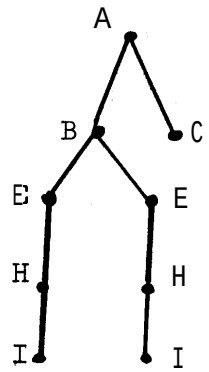
produces



and SUBSTI(I,D) applied to



produces

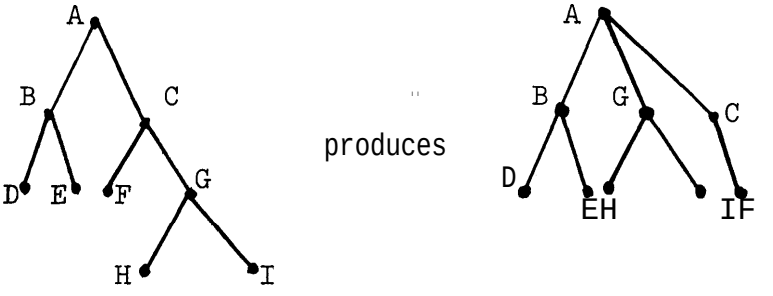


Add as right sister: ADRISI(N1,N2) and ARISEI(N1,N2)

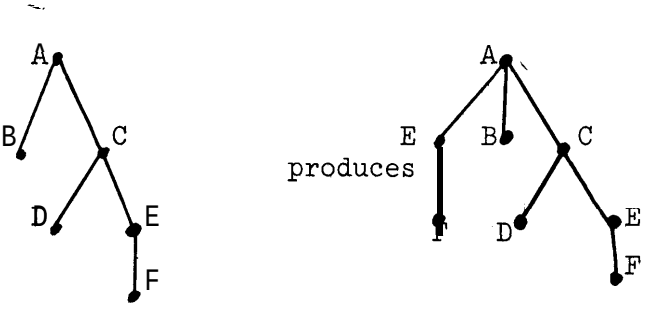
Add as left sister: ADLESI(N1,N2) and ALESEI(N1,N2)

These operations add the node N1 as the left or right sister of the node N2; in addition, ARISE1 and ALESEI erase the original occurrence of N1.

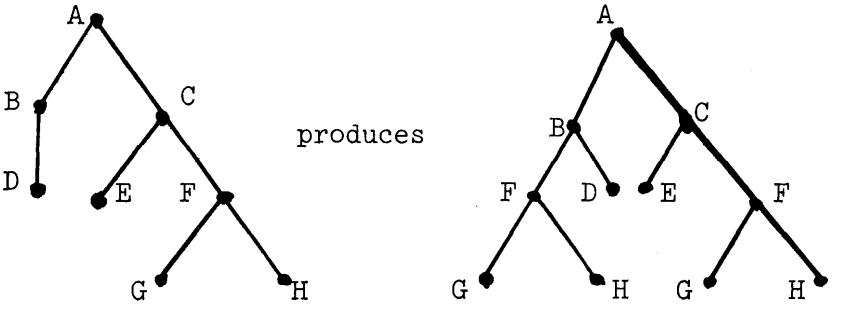
ARISEI(G,B) applied to



N1 will be replaced in these operations by the head of any non-branching chain above N1, but N2 will not be so replaced. Thus, ADLESI (F,B) applied to



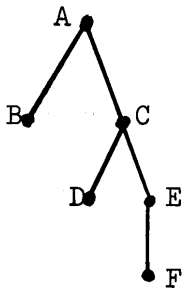
but ADLESI (F,D) applied to



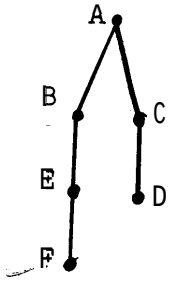
Add as daughter: ADLADI (N1,N2) and ALADEI (N1,N2)

ADLADI adds N1 as the only daughter of N2; be used if N2 already has descendants. ALADEI adds N1 as the only daughter of N2 and

also erases the original occurrence of N1. The operations chain upward from N1 but do not chain from N2. ALADEI (F,B) applied to



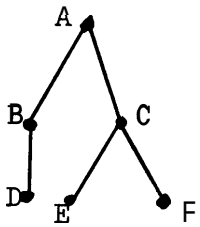
produces



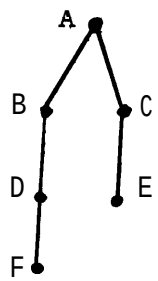
while

ALADEI (F,D)

applied to



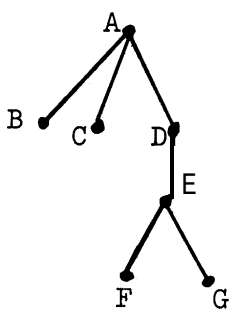
produces



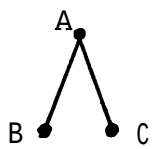
Erase: ERASE1 (N1)

This operation deletes from a tree the subtree headed by N1 as well as any non-branching chain above N1.

ERASE1 (E) applied to



produces



3.10 Complex symbol operations

CSEXCH(N,M)

CSEXCH (Complex Symbol Exchange)_ is an INTEGER*2 function which sets up calls to CSOP and returns the results of CSOP. The arguments N and M are complex symbol numbers (i.e. pointers to CSLIST).

CSEXCH is never entered from the beginning but always from one of the entries which determine the test or operation to be performed. The entries are:

For tests

CSEQ

CSINC1 (or equivalently, CXINC1)

CSINC2

CSNDST

For operations

CSMERG

CSMERR

CSERAS

CSSAVE

For each entryexcept CSMERR CSEXCH makes a preliminary test and an immediate return if the test or operation is trivial (N=M). Otherwise it makes the appropriate call to CSOP by setting TYPE to 1 for operations and 2 for tests, and selecting the array A which defines the test or operation as the second argument of CSOP.

Each of the matrices defines a function of feature values and is of the form:

N + + + + - - - - * * * * A A A A (A = absent)

M + - * A + - * A + - * A + - * A

A(N, M)

For tests the values in A(N,M) are 1 if true, and in general 2 if false. An exception is CSINC1 where 2 and 3 are both false, 2 in the case of noninclusion because of the absence of the feature in M and 3 if the values in N and M conflict. (This distinction is used by REDRUL.)

For operations the values in A(N,M) are the values to be given to the feature in the new complex symbol being constructed. They are 1 (+),2(-), 3 (*), 4 (absent) and 5 (a random choice between + and -).

It should be noted that new operations and tests may easily be defined, simply by adding to CSEXCH a new entry and a corresponding new value for the matrix A .

CSOP(TYPE,A,N,M)

CSOP is an INTEGER*2 function of four arguments which is called by CSEXCH. CSEXCH determines the function of a particular call to CSOP and sets up the arguments. The arguments are:

TYPE, an integer with value 1 is a new complex symbol is to be created, or value 2 if a test on complex symbols is to be evaluated.

A is an integer array which represents the 4x4 matrix which defines the operation or the test to be performed.

N and M are pointers to the complex symbols which are the arguments of the operation or test.

Each test or operation on two complex symbols is computed from the value of the test or operation on the individual feature specifications of the two complex symbols. The value of a test is the maximum of the values for individual feature specifications; thus, if a test is to succeed it must be true (=1) for all pairs of feature specifications. The value of an operation is the complex symbol resulting from pairwise application of the operation to the feature specifications in two complex symbols.

In order to understand the flow of CSOP it is necessary first to know the structure of the array CSLIST(4,2000) which contains the complex symbols. Each entry in CSLIST consists of (feature number, feature type, feature value, pointer to the next feature specification in the complex symbol (0 if none)). The entries in CSLIST for a particular complex symbol are sorted on the first column (feature number). This ordering is taken advantage of in going through the complex symbol.

The subroutine uses the same basic cycle to pick the current feature specification pairs for both operations and tests. The main difference in the treatment in operations and tests comes when a feature specification pair has been selected. Then a branch is made depending on TYPE and the operation or test carried out for the current pair. TYPE is also tested in initialization and in finishing up.

The discussion of how the matrix A determines the result of CSOP will be found in the writeup of CSEXCH.

REDRUL(M)

REDRUL is an integer*2 function of the complex symbol number M. It

returns the number of the complex symbol obtained after expansion of M by the redundancy rules, of 0 if a contradiction is found in doing the expansion.

The redundancy rule $A \Rightarrow B$ has been stored in RULE as the pair of complex symbols (A,B). Parallel to RULE is a temporary LOGICAL*1 array RULCHK.

First RULCHK is initialized to FALSE, and FLAG is set FALSE.

The main cycle is a pass through the rules in RULE. For the Ith rule, the computation is: If RULCHK is TRUE the rule is skipped. Otherwise, the two parts A and B of rule I are compared with the complex symbol M using CXINC1. If A is included in M and B is not included in M and B does not conflict with M , the new value of M is set to the result of merging (MERGE1) M and B , RULCHK (I) is set TRUE and FLAG is set TRUE. If A and B are both included in M , the RULCHK (I) is simply set TRUE. The next rule is then considered. If A is included in M , but B conflicts with M (i.e. CXINC1 returns 3), an error message is printed and the subroutine terminates. After all rules have been tried, FLAG is tested and if TRUE, it is reset to FALSE and the main cycle is repeated. If FLAG is FALSE, no changes have occurred on the last cycle, so the expansion process is complete.

After the iteration of the main cycle is completed a space recovery section of the code is executed to reduce waste space in CSLIST. All intermediate complex symbols created by merging are erased and only the final result is retained.

3.11 Control program

This section describes a set of subroutines (CPIN, SYNCHK, RECOG, CONTRL, SCAN, TAPPLY, TRACE, APPLYG) which input and interpret control programs.

CPIN inputs a control program, checks syntax, and checks block structure. It also builds the symbol table which associates symbols and locations in CPBUF (the control program main storage area).

SYNCHK is a general context free grammar recognizer. It is called by CPIN to check the syntax of the control program.

RECOG is a token generator/recognizer for SYNCHK. It uses SCAN to do the actual token generation.

CONTRL interprets a control program residing in CPBUF. It checks syntax as it executes. SCAN is a token generator for CONTRL.

SCAN is a token generator used primarily by CONTRL to determine what is the next item in the execution sequence.

TAPPLY is the subroutine (with four entry points) which determines whether a given transformation should be invoked and if so, invokes it. It is driven by CONTRL and APPLYC and itself drives TRACE.

TRACE is the routine which does the outputting *of* trace infoxmation during the execution of the control program.

APPLYG is driven by CONTRL. Its function is to invoke those transformations of a group (denoted by group number) which should be invoked. It hands the members of a group to TAPPLY one-by-one.

FORMAL, DEFINITION OF -A CONTROL PROGRAM

SYNTAX

CONTROL-PROGRAM : **:=**	CONTROL-PROGRAM1 .
CONTROL-PROGRAM : **:=**	SCLIST [CONTROL-INSTRUCTION]
CONTROL-INSTRUCTION **::=**	LABEL CONTROL- INSTRIJCTION OR **CONTROL-INSTRUCTION** LABEL I NSTRI JCTION
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-INSTRUCTON OR GO- INSTRIJCTION OR TRACE- INSTRIJCTION OR STOP-INSTRIJCTION
TRANSFORMATION-ELEMENT **::=**	TRANSFORMATION-NAME OR GROUP-NUMBER
REPEAT- INSTRUCTION **::=**	RPT INTEGER < CONTROL-PROGRAM1 > OR R P T < **CONTROL-PROGRAM1** >
IN-INSTRUCTION : **:=**	IN TRANSFORMATION-NAME (INTEGER) DO < CONTROL-PROGRAM >
IF-INSTRUCTION **::=**	IF INSTRUCTION THEN **GO-INSTRUCTION** OPT [ELSE **GO-INSTRUCTION**]
FLAG-INSTRUCTION **::=**	FLAG-NAME **TRASFORMATION-** LIST
FLAG-NAME **::=**	FLAG OPT [INTEGER]
CO-INSTRUCTION **::=**	GO TO WORD OR GOTO WORD

TRACE-INSTRUCTION	: :=	TRACE TRANSFORMATION-LIST SPECIFICATION OR UNTRACE TRANSFORMATION-LIST OR TREE
SPECIFICATION	::=	BEFORE TEST OR AFTER SUCCESS OR AFTER FAILIJRE OR AFTER CHANGE OR RESULT
STOP- INSTRUCTION	: :=	STOP OR .
TRANSFORMATION- LIST	: : =	TRANSFORMATION-ELEMENT OR < SCLIST [TRANSFORMATION-ELEMENT] >

SEMANTICS

A control program is a sequence of control instructions separated by semi-colons' and ending with a period.

Each control instruction may be labeled with an indefinite number of labels.

A label is a word, which is not a reserved word, followed by a colon. -All terminal symbols of the syntax and all transformation names are reserved words. Duplicated labels are not allowed.

There are two types of control instructions: those specifying control elements (instructions to the interpreter] and those designating transformations. Control elements may be thought of as operators and transformation elements as operands.

Instructions may be grouped for convenience by enclosing them in. angular brackets. Nesting within angular brackets may occur to any desired depth. Each pair of angular brackets serve to define a block (see Block Structure below).

There are seven types of control elements. Each differs in its effect on the interpreter and its effect on the tree.

A transformation element may be the name of a transformation or the name of a transformation group (denoted by a Roman numeral).

Two forms of repeat instructions exist: definite and indefinite. Both are similar in interpretation.

The definite repeat will execute the following control program INTEGER number of times or until the control program has no effect (i.e. has value false - see Values below), whichever occurs first.

The indefinite repeat will execute the following control program until it has no effect (i.e. has value false - see Values below). The number of iterations of the control program will not exceed INFNTY - a variable in CPCOM.

The value of the repeat is true if any transformation was succcessfully invoked and is false otherwise.

The IN-construct allows the user to fix a top node of the tree. This node may or may not be the mot of the tree, allowing the transformations in the control program following to operate on a subtree of the original tree if desired.

Execution of an IN-construct proceeds as follows:

Invoke the named transformation.

If successful, examine the node designated by the INTEGER.

If this node has never been used as the top node of the IN before, execute the control program using this node as the top of the tree.

If this node has been the top node of the IN before, find another top node by invoking the transformation again. If the invocation is unsuccessful, the IN terminates. If it is successful, examine the node designated by the INTEGER as above.

The value of the IN is true if any invoked transformation succeeds and is false otherwise.

The FLAG instruction provides the means by which a group of transformations may be monitored. Flagging both establishes the group and sets the flag to false. The value of a flag is true if any transformation in its group has been successfully invoked since the flag was last established and is false otherwise.

There are ten flags which may be referred to by number. The construct FLAG with no INTEGER following is taken to mean FLAG 0.

An IF-construct with FLAG means: if the current value of the designated flag is true then execute the first GOTO, if it is false then execute the second GOTO if it exists (otherwise control passes to the next instruction). If no INTEGER follows FLAG flag 0 is assumed. An IF followed by any other instruction means: if the value of the instruction is true then execute the first GOTO, if it is false then execute the second GOTO if it exists (control passes to the next instruction if it does not). Note that a group must be formed through an instance of a FLAG-construct before an IF-construct with FLAG has meaning.

Two forms of GOTOs exist. They are entirely equivalent in their effect. Both force the execution of the control program to continue from the point specified. Jumps into blocks are not allowed (see Block Structure below).

Three types of trace instruction exist: TRACE, UNTRACE and TREE. UNTRACE resets the trace operation (turns off the output). TRACE turns on a specified type of dump (see below). TREE outputs the whole current tree whenever it is executed.

Five types of dumps are provided. Any combination, including all, are possible. BEFORE TEST outputs the current tree before a call to **ANTEST** is made but after all keywords have been satisfied. AFTER SUCCESS outputs the current tree after **ANTEST** has returned **true** but before the tree has been changed. AFTER FAILURE outputs the current tree after **ANTEST** has returned false. AFTER CIIANGE outputs the current tree after a call to subroutine **CHANGE.** **RESULT** outputs one line telling whether or not the transformation was successfully applied.

The STOP instruction terminates execution of the control program. An alternate way to terminate the" **control** program is to **"run** off the **end", i.e.** to try to execute the period.

A TRANSFORMATION-LIST is either a transformation element or a list of transformation elements separated by semi-colons and enclosed. in angular brackets.

VALUES

Each INSTRUCTION of a **control** program has a value. This value **is** determined as follows **:**

CONTROL ELEMENTs:

RPT - true if any value of its control program is true; false otherwise.
IN - true if the IN-transformation has value true; false otherwise.
IF - true if its INSTRUCTION is true; false otherwise.
FLAG - has no value. A FLAG within an IF has the value of the FLAG designated.
GO - has no value.
TRACE - has no value.
STOP - has no **value.**

TRANSFORMATION-ELEMENTs:

TRANSFORMATION-NAME - true if the transformation is successfully applied (i.e. a structural change has been made or would have been if it were not void); false otherwise.
GROUP-NUMBER - true *if* any transformation in the designated group is successfully applied; false otherwise.

Lists :

CONTROL-LIST - true if any element of the list has value true;
false otherwise.

TRANSFORMATION-LIST - true if any element of the list has value true;
false otherwise.

CONTROL-PROGRAM1 - true if any element of the list has value true;
false otherwise.

An undefined value is taken to be false throughout.

In general angular brackets serve to combine many values into one. This combination is formed by taking a logical inclusive-OR of the values of the elements within the brackets.

BLOCK STRUCTURE

A block may be formed by the usage of angular brackets. The elements within a block form a unit and determine one value (see Values above). Control may pass to a block only by executing the angular bracket at its head. Control may pass fmm a block either by executing the angular bracket at its tail or by executing a GOTO. Any label within a block may be thought of as being local to that block. However, throughout an entire control program duplication of labels is'not permitted (even though the duplicates may be in different blocks).

A block is formed by any of the following constructs: RPT, IN and IF. The block formed is inherent to the construct - control may only pass into such-.a block through its head. Control may pass *from* such a block through its tail or by the execution of a COTO.

In fact the interpreter will allow GOTOs from one block into another so long as the level of the destination is as low or lower than the level at the GOTO. Thus it is possible to enter a block at a point other than its head (but the stack will not have been set up by the block entry bracket, so results other than those which were desired may be obtained).

Note that every entry into a block forces a push onto the stack, and every departure forces a pop.

STORAGE

The main storage area for the control program is CPCOM (see Figure 1*). Almost all communication is done via variables and arrays residing within this region. The initialization of CPCOM is done in three ways. Data which never changes is loaded via BLOCK DATA subprograms (see Figure 2). CPBUF and all symbol-related data are set up by CPIN. All other initialization occurs within the first few statements of CONTRL.

The following variables and arrays are all in CPCOM:

- CPBUF() - main control program buffer: contains the symbolic control program as read in by CPIN.
- CPPTR - pointer into CPBUF: used by CONTRL to indicate the current instruction being interpreted.
- SYTB() - symbol table: contains the alphameric symbols of the control program.
- SYTV() - symbol values: contains pointers into CPBUF. Each entry of SYTB has an associated SYTV entry showing at what point the symbol was defined.
- SYTL() - symbol levels: contains the level of each symbol in SYTB. Used to detect errors during execution.
- SYTN - the number of entries in SYTB.
- TERM() - contains the terminal symbols of the syntax (see Figure 4).
- STK() - execution stack (see Figure 3 and the description of CONTRL below).
- SPTR - points at the current first postion of STK.
- OTOP() - points into OTOPS (see description of TAPPLY).
- OTOPS() - contains old top nodes for a currently executing IN-instruction (see TAPPLY description).
- LVL - the current level
- ZSCAN - auxilliary output from SCAN (see below).
- ZINT . . . 2UNT - tokens for the terminal symbols in TERM().

*The references to figures in section 3.11 refer to figures 3.11.1 to 3.11.5. These figures appear at the end of this section.

INFNTY - the value taken to be infinity by RPT and TAPPLY.
FGN(,) - boolean array designating flagged transformations.
FGV(,) - boolean array containing the values of flagged transformations.
TRCF(,) - boolean array designating which transformations are being traced and by what type of trace.
APFG - a flag set by TAPPLY used by CONTRL. It is true when a transformation which has been invoked by a call to TAPPLY has successfully attempted to modify the current tree.
IFFG - used by CONTRL in the evaluation of IF-statements. True if the IF will take the first branch; false otherwise.
RFG -- true whenever the stack contains an IN-instruction; false otherwise.

The following variables and arrays reside in SYNCM:

SNTX(,) - base syntax used by SYNCK.
CUR - current goal symbol.
SCN - set to semi-colon: error recovery symbol.
SPT - stack pointer for STAK.
STAK() - stack used by SYNCK in recognition process.
IPT - pointer into CPBUF used by RECOG.
NEQN - number of syntax equations, set to 57.
TRCFG - trace flag for SYNCK, set to false.
DMPFG - dump flag for SYNCK, set to false.
RECFG - recovery flag for SYNCK, set to true.

The following variables and arrays reside in local storage:

BLOK(,) - contains a skeleton of the block structure of the program. **Used** by CPIN.
OPTR - points at the top element of **OTOP.** Used by CONTRL,
SLVL - used by CONTRL to keep track of levels when scanning for semi-colons or angular brackets.
TYP - indicates the type of instruction in the stack. May be set to zero, ZIF, ZIN or ZRPT only. Used by CONTRL.
TYP - indicates the type of tracing. Used by TRACE.
TRCTYP - indicates the type of tracing. Used by **CONTRL.**
FLGN - holds the flag **number.** Used by CONTRL.
TRM() - an **equivalenced** array allowing SCAN to reference the tokens parallel to **TERM** in **CPCOM.**
ROMAN() - holds the Roman numerals I through **X.** Used by SCAN.
S() - array holding the marked **S's** in **TAPPLY.**
TOP - contains the top node for **TAPPLY.**
CYC - contains the cyclicity *of* a transformation in **TAPPLY.**
OPT - contains the optionality of a transformation in **TAPPLY.**
GOAL - contains the goal S in **TAPPLY.**
ANFG - contains the result of having called **ANTEST** in **TAPPLY.**
TRNS() - contains the transformations which have **successfully** been invoked in order. Used by **TAPPLY.**
TIM - contains the time of call for TRACE,

DESCRIPTION OF CPIN

CPIN inputs a control program into CPBUF and checks its syntax and block structure. It also detects undefined and multiply defined labels and undefined symbols. The program operates in two passes.

Pass 1 inputs the control program up to and including the period at its end. It detects labels by the colon following them and enters them into SYTB along with their location and level. LVL contains the current level: it is increased by one each time a < is seen and decreased by one whenever a > is seen. Concurrently a skeleton of the block structure is built in BLOK.

The format of BLOK is:

```
BLOK(I,1)   - number of labels in this block
BLOK(I,2)   - parent of this block
BLOK(I,3)   - labels in this block
  ...              ...
BLOK(I,20)  - .
```

Example: for this control program:

```
L1:  TRAN1 ;
     < L2: TRAN2 ; L3: < L4: TRAN3 > ; L5: > ;
L6:  STOP .
```

BLOK would appear as:

```
2   0   L1   L6
3   1   L2   L3   L5
1   2   L4
```

SYTB	SYTV	SYTL
L1	3	0
L2	8	1
L3	12	1
L4	15	2
L5	20	1
L6	25	0

Pass 2 scans CPBUF checking to see that each GOTO refers to a defined symbol and that no jumps into blocks occur. It also checks to see that each GO is followed by a TO. The number of errors detected is output at the conclusion of Pass 2 if it is greater than zero.

CPIN then calls SYNCHK which checks the syntax of the input program (see SYNCHK below).

DESCRIPTION OF SYNCHK

SYNCHK is a top-down recognizer. It drives a stack attempting to recognize a CONTROL-PROGRAM. It uses the syntax in SYNCH which is equivalent to the syntax given above.

The program operates as follows:

Load the stack with CONTROL-PROGRAM.

If the stack's top element is non-terminal, find an equation with this non-terminal as left-most symbol and stack it. Examine stack's top element as above.

If the stack's top element is terminal, call RECOG to see if the first element of the input stream is the same as this symbol. If it is, continue examining this equation. Every time. a non-terminal is found it is pushed onto the stack. Each time a terminal is found a call is made to RECOG. If RECOG returns true it advances the input stream over the symbol it just recognized. If RECOG returns false it does not.

Each time the recognizer finds that the input stream and the current equation differ, it scans for another equation which has the same left-most symbol as the current one. If it finds one it tries to use it in the recognition process. If it is unable to find one or runs out of new equations it will pop the stack if it has not advanced the input stream and continue searching for valid equations. If it has advanced the input stream then there is a syntax error at the current position.

When a *syntax* error is encountered the recognizer begins scanning until it finds the symbol SCN (currently a semi-colon) in the input stream. It then assumes that it has successfully recognized the current equation and continues.

Two types of trace output are available. Both are normally off.

TRCFG - if true, the recognizer will output the name of each non- terminal symbol which it successfully recognizes.

DMPFG - if true, the recognizer will ouput the stack every time it begins to examine the current symbol.

RECFG is normally true - it provides the error recovery described above. If you wish to avoid the error recovery set RECFG to false.

The syntax used by **SYNCHK** is given in Figure **5.** This syntax is more general than the syntax given above and better reflects the actual operation *of* the control **program** although it is more complex.

DESCRIPTION OF RECOG

RECOG is the recognizer/token generator for SYNCK. It calls SCAN to generate the actual token and then compares this with the tokens in SNTX. The input stream pointer IPT is advanced if the desired symbol and the input symbol match, otherwise it is not.

DESCRIPTION OF CONTRL

The first few statements of CONTRL initialize the various stacks and variables which it uses.

The main loop begins at statement 10. It is at this point that the program has just finished recognizing and interpreting a control instruction and it is ready to look for the next. We now call SCAN for the next token and go to the appropriate part of the interpreter depending upon what SCAN returns. If the symbol is illegal or undefined, a branch is made to statement 9, the standard error recovery section.

Below is a description of the actions of the control program for each construct or symbol which it sees.

Syntax errors send the control to statement 9. At this point scanning fur the next semi-colon begins. The program keeps track of its level and will stop scanning when it finds the next semi-colon at the current level.

INTEGER Illegal syntax, control passes to Error Recovery.

T-NAME APPLY or APPLY1 is called depending upon whether or not the control program is currently in an IN-construct. The result of the invocation is inserted in the stack.

WORD words, that is labels, are passed over. They have no effect on the control program.

GROUP-NO APPLYG is called with this group number as argument. The result of invoking the transformations of this group is inserted in the stack.

< A left angular bracket signals the beginning of a TRANSFORMATION-LIST. The level is increased by one, the stack is pushed down, and the arrays OTOP and OTOPS are pushed down.

> A right angular bracket signals the end of some type of transformation- or control group. A test is made to see if the group is an IN, an IF, a RPT, or a TRANSFORMATION-LIST. If it is an IN, control passes to statement 7500 (described below). If it is an IF, control passes to statement 6500 (described below). If it is a RPT, the stack is tested. If the value is false, the RPT terminates and control passes to statement 10. If the value is true, the value is reset to false, the repeat counter is decremented (if it is zero, terminate the RPT), then the RPT is started again. If it is a TRANSFORMATION-LIST the stack is popped and the value of the list is inserted into it.

; Semi-colons have no effect.

(Parentheses are illegal. Control" passes to Error Recovery.

) Parentheses are illegal. Control passes to Error Recovery.

. A period terminates the execution *of* the control program (see STOP below).

: Colons are ignored.

. AFTER Illegal, control passes to Error Recovery.

BEFORE Illegal, control passes to Error Recovery.

CHANGE Illegal, control passes to Error Recovery.

DO Illegal, control passes to Error Recovery.

ELSE Illegal, control passes to Error Recovery.

FAILIJRE Illegal, control passes to Error Recovery.

FLAG Checks the stack to see if the control program is currently executing an IF-construct. If it is, then it returns the value of the designated flag. If an INTEGER follows the FLAG, that flag is used, if not then flag zero is used. If the control program is not executing an IF-construct, it checks the next symbol for INTEGER. If it finds one, FLGN is set to that number, if not, FLGN is set to zero. Then the following expression is read and each transformation designated (by name or group number) is flagged by setting the corresponding entry in FGN to true. The corresponding values in FGV are all reset to false.

GO Scan the next symbol for TO, if found go to GOTO; if not found, go to Error Recovery.

GOTO Scan the next symbol for WORD. If not found, go to Error Recovery. Look up the value of the label in SYTV and its level in SYTL. If the new level is higher than the current one an attempt is being made to jump into a block, complain and go to Error Recovery. Otherwise change CPPTR to the value looked up.

IF Push the stack and put IF into the top, push OTOP and OTOPS.

THEN (statement 6500) Pop the stack. If the old value is true go to statement 10. If the old value is false scan to an ELSE (if it exists or a semi-colon if it does not) and go to statement 10.

IN Scan *for* the name of the IN-transformation, left parenthesis, an integer, right parenthesis, DO. Then initialize OTOP and OTOPS. Attempt to invoke the IN-transformation by calling APPLY1. If it does not apply, go to statement 10. If it does then push the stack, enter ZIN into the stack and go to statement 10, Control passes to statement 7500 at the right angular bracket of an IN-construct. We then call APPLY1, again seeking a new S for top node. If APPLY1 returns true then the stack is pushed as above and control passes to statement 10 after resetting CPPTR to the left angular bracket of the IN. If APPLY1 returns false the IN terminates and conrol passes to statement 10.

RESULT Illegal, control passes to Error Recovery.

RPT Scan for an integer, if one is found enter it as the repeat counter value. If not, enter INFNTY. Scan to a left angular bracket and go to < above.

STOP Terminate the execution of the control program. Print the number of instructions executed and the transformations which hae applied, then return to the calling program.

SUCCESS Illegal, control passes to Error Recovery.

TEST Illegal, control passes to Error Recovery.

THEN Illegal, control passes to Error Recovery.

TO Illegal, control passes to Error Recovery.

TRACE Scan to tho specification after marking the current position. Then rescan the TRANSFORMATION-LIST setting the TRCF entry for *each* transformtion which is to be traced.

TREE Call TROUT *to* output the current tree.

UNTRACE Reset the TRCF entries for each transformation designated in the transformation list.

Note that the syntax which describes the operation of the control program is given in Figure 5 below.

DESCRIPTION OF SCAN

SCAN is the token generator for CONTRL. In addition it keeps track of the current position in CPBUF.

SCAN has two outputs: the first is the token for the current symbol (returned in SCAN). The second is ZSCAN - this is auxilliary information: the integer, label, transformation number, etc.

Values for SCAN and ZSCAN are as follows:

SYMBOL	SCAN	ZSCAN
undefined symbol	0	0
INTEGER	1	the integer
TRANSFORMATION-NAME	2	the transformation number
LABEL	3	pointer into SYTB for this label
GROUP-NUMBER	4	integer representing the Roman numeral
terminal symbol	token	----

Terminal symbols are all "reserved words". The tokens for the terminals are given in Figure 4. The tokens are referenced by index via the array TRM - an equivalenced array. No transfoxmatfon may have the same spelling as a terminal symbol.

The entry point SCAN1 is used by RECOG. It differs only in that no trace information i s printed during the program's execution.

DESCRIPTION OF TAPPLY

TAPPLY is called to invoke a transformation, If it has applied APFG is set to true and FGV is set to true. In addition, this transformation is entered into TRNS. A transformation is said to have applied if it successfully modified the tree (or at least called subroutine CHANGE) or if it would have called CHANGE if the structural change had not been void.

The subroutine has four entry points which are described below. Each of the first three are similar in all but minor details. We now give a brief description of the operation of APPLY.

First, all S's are marked in the current subtree. If no S's exist, we are done.

Then we find the keywords for this transformation. If none exist then we will be trying the transformation at every S, so set the appropriate flag.

Find the first S which has been marked which dominates all the keywords (or use the first S if there are no keywords). *If* no such S exists, then exit after updating TRNS *if* necessary.

Trace before ANTEST.

Call ANTEST (if the structural description is not zero, if it is, then set ANFG = true), put the result into ANFG.

If ANFG is false, unmark the current S (so it will never be tried again) and return to the process above.

If ANFG is true, then if the structural change is zero trace after change,. unmark the S, and return to the process above. If the structural change is non-zero, then branch to one of the four segments dealing with the. particular type of transformation,

Type 1 - AC - non-cyclic: call CHANGE, then unmark the S and proceed as above.

Type 2 - ACAC - cyclic: call CHANGE, then return to the above without unmarking the current S.

Type 3 - AACC: call CHANGE NUMCNT times, then unmark the S and return to the above.

Type 4 - MC: call CHANGE once picked from among the NUMCNT choices at random and proceed as above.

We have not discussed optionality in the above. Optionality tests are inserted before calling ANTEST for Type 1 (AC) transformations and within the three subparts for the other three.

APPLY is the entry point for general invocation of transformations. It proceeds as above.

APPLY1 is the entry point for the execution of the IN-transformation, It updates OTOP and OTOPS and uses APPLY or APPLYI depending upon whether the current IN-construct is within another IN or not. For APPLY1 to be successful the node found by ANTEST must be different from all nodes already in OTOPS.

The formats of OTOP and OTOPS are: entries in OTOP are pairs of pointers into OTOPS - there is one pair for each level of execution. Entries in OTOPS are the actual nodes which have been used as top nodes inside an IN-construct.

APPLY1 is the entry point for the execution of the program inside the angular brackets of an IN-construct. It differs from APPLY in that there is only one S which may be used as the goal and top node of the tree for ANTEST and CHANGE - the node specified by the INTEGER of the IN-construct.

OUTTRN is the last entry point - it is used by CONTRL to output the contents of TRNS - the transformations which have successfully applied in order of application.

Each of the entries to TAPPLY above will set APFG to true if the invocation is successful and enter the transformation into TRNS.

DESCRIPTION OF TRACE

TRACE outputs trace information which may be the tree (by a call to TROUT) or just the result *of* invoking the transformation (true or false depending upon whether or not the transformation was successful),

TRACE is called at three points during the inwcation of a transformation: before calling ANTEST (but after all keywords have been satisfied), after calling ANTEST but before calling CHANCE, and after calling CHANGE.

The type *of* TRACE (there are five types) is determined by use of the array TRCF (in CPCOM).

ANFG is a logical variable giving the value of the last call to ANTEST.

Values for TIM and TYP are:

TIM	=1 before ANTEST
	-2 after ANTEST, before CHANGE
	=3 after CHANGE
TYP	-1 BEFORE ANTEST
	-2 AFTER FAILURE of ANTEST
	=3 AFTER SUCCESS of ANTEST
	-4 AFTERCHANGE
	=5 RESULT

DESCRIPTION OF APPLYG

APPLYG invokes the transformations in the group designated by GNO one-by-one. If GNO refers to a non-existent group APPLYG comments to this effect and returns false. Otherwise the value of APPLYG is an inclusive-OR or the values of all transfonaations in the designated group. The value of APPLYG is returned in APFG (a variable in CPCOM).

APPLYG will use APPLY or APPLY1 depending upon whether or not the control program is currently executing an IN-construct. (It tests RFG to determine this.)

Figure 3.11.1

CPCOM, SYNCM

CPCOM:

```
      IMPLICIT INTEGER*2 (A-Z)                                          CPCOM0
      COMMON /CPCOM/ CPBUF,SYTB,TERM                                    CPCOM1
      REAL*8 CPBUF(500),SYTB(100),TERM(32)                              CPCOM2
      COMMON /CPCOM/ SYTV(100),STK(100),OTOP(20),OTOPS(50),             CPCOM3
     1 CPPTR,SPTR,LVL,SYTN,ZSCAN,SYTL(100),                             CPCOM4
     2 ZINT,ZTRN,ZWRD,ZGRN,ZLAN,ZSMI,                                   CPCOM5
     3 ZLPR,ZRPR,ZPER,ZCOL,ZAFT,ZBEF,ZCHN,                              CPCOM6
     4 ZDO,ZELS,ZFAL,ZFLG,ZGO,ZGOT,ZIF,                                 CPCOM7
     5 ZIN,ZRSL,ZRPT,ZSTP,ZSUC,ZTST,ZTHN,                               CPCOM8
     6 ZTO,ZTRC,ZTRE,ZUNT,INFNTY                                        CPCOM9
      COMMON /CPCOM/ FGN,FGV,TRCF,APFG,IFFG,RFG                         CPCOM10
      LOGICAL*1 FGN(100,10),FGV(100,10),TRCF(100,5),APFG,IFFG,RFG       CPCOM11
```

SYNCM:

```
      IMPLICIT INTEGER*2 (A-Z)                                          SYNCM0
      COMMON /SYNCM/ SNTX,CUR,SCN,SPT,IPT,NEQN,                         SYNCM1
     1 TRCFG,DMPFG,RECFG                                                SYNCM2
      REAL*8 SNTX(57,8),CUR,SCN                                         SYNCM3
      LOGICAL+1 TRCFG,DMPFG,RECFG                                       SYNCM4
```

Figure 3.11.2

BLOCK DATA STATEMENTS

SYNCM

```
 BLOCK DATA                                                          %SYNCM
 IMPLICIT INTEGER*2 (A-Z)                                            SYNCM0
 COMMON /SYNCM/ SNTX,CUR,SCN,SPT,IPT,NEQN,                           SYNCM1
1 TRCFG,DMPFG,RECFG                                                  SYNCM2
 REAL*8 SNTX(57,8),CUR,SCN/';'/                                      SYNCM3
 LOGICAL*1 TRCFG/.TRUE./,DMPFG/.FALSE./,RECFG/.TRUE./                SYNCM5
 INTEGER*2 NEQN/57/                                                  SYNCM6
 EQUIVALENCE
1 (SNTX(1,1),C1(1)),(SNTX(1,2),C2(1)),(SNTX(1,3),C3(1)),
2 (SNTX(1,4),C4(1)),(SNTX(1,5),C5(1)),(SNTX(1,6),C6(1)),
3 (SNTX(1,7),C7(1)),(SNTX(1,8),C8(1))
 REAL*8 C1(57)/
1 'CTLPGM. ','PGM.    ','PGM.    ','CP.     ','CP.     ',
2 'CP.     ','CI.     ','CI.     ','CI.     ','Cf.     ',
3 'SC.     ','SC.     ','LBL.    ','LBL.    ','INST.   ',
4 'INST.   ','INST.   ','CL.     ','CL.     ','CL1.    ',
5 'CL1.    ','CL1.    ','CE.     ','CE.     ','Cf.     ',
6 'CE.     ','CE.     ','CE.     ','CE.     ','TF.     ',
7 'TE.     ','RPT-I.  ','RPT-I.  ','IN-I.   ','DOP.    ',
8 'IF-I.   ','IF-I.   ','FLG-I.  ','FN.     ','FN.     ',
9 'GO-I.   ','GO-I.   ','TRC-I.  ','TRC-I.  ','TRC-I.  ',
1 'SPEC.   ','SPEC.   ','SPEC.   ','SPEC.   ','SPEC.   ',
2 'STP-I.  ','TL.     ','TL.     ','TL.     ','TL1.    ',
3 'TL1.    ','TL1.    '/
 REAL*8 C2(57)/
1 'PGM.',    'SC.'     'CP.',     'CI.',     'CI.',
2 'CI.',     'LBL.',    'LBL.',    'INST.',   'INST.',
3 ';',       ';',       'ZWRD.',   'ZWRD.',   'CE.',
4 'CL.',     'TE.',     '<',       '<',       'INST.',
5 'INST.'    'INST.'    'RPT-I.',  'IN-I.',   'IF-I.',
6 'FLG-I.', 'GO-I.',    'TRC-I.',  'STP-I.', 'ZTRN.',
7 'ZGRN.',   'RPT'      'RPT',     'IN',      'DO',
8 'IF',      'IF'       'FN.',     'FLAG',    'FLAG',
9 'GOTO',    'GO',      'TRACE',   'UNTRACE', 'TREE',
1 'BEFORE', 'AFTER',    'AFTER',   'AFTER',   'RESULT',
2 'STOP',    '<',       '<',       'TE.',     'TE.',
3 'TE.',     'TE.'/
 REAL*8 C3(57)/
1 '.',       'CP.',     ' ',       'SC.'      'SC.',
2 ' ',       'Cf.',     ' ',       'LBL.',    ' ',
3 'SC.'      ' ',       ':',       ':',       ' ',
4 ' ',       ' ',       'SC.',     'CL1.',    'SC.',
5 'SC.',     ' ',       ' ',       ' ',       ' ',
6 ' ',       ' ',       ' ',       ' ',       ' ',
7 ' ',       'ZINT.',   '<',       'ZTRN.',   '<',
8 'INST.',   'INST.'    'TL.',     'LINT.',   '',
9 'ZWRD.',   'TO',      'TL.',     'TL.',     ' ',
1 'TEST',    'SUCCESS', 'FAILURE', 'CHANGE',  '',
2 ' ',       'SC.',     'TL1.',    ' ',       'SC.',
3 'SC.'      ' '/
```

SYNCM, Cont'd

```
 REAL*8 C4(57)/3*' ','CP.',8*' ','LBL.',4*' ',
1 'CL1.','>','CL1.',11*' ','<','PGM.','(','PGM.','THEN',
2 'THEN',4*' ','ZWRD.','SPEC.',8*' ','TL1.','>',
3 ' ','TL1.',' ',' '/
 REAL*8 C5(57)/17*' ','>',13*' ','PGM.','>',
1 'ZINT.','>','GO-I.','GO-I.',14*' ','>',5*' '/
 REAL*8  C6(57)/31*' ','>',' ',')','','ELSE',21*''/
 REAL*8 C7(57)/33*' ','DOP.',' ','GO-I.',21*' '/
 REAL*8 C8(57)/57*''/
 END                                                                    %
```

CPCOM

```
 BLOCKDAT A                                                             %CONTROL
 IMPLICIT INTEGER*2(A-Z)                                                CPCOMO
 COMMON/CPCOM/ CPBUF,SYTB,TERM                                          CPCOM1
 REAL*8 CPBUF(500)/'STOP    '/,SYTB(100),TERM(32)/                      CPCOM2
1 'ZINT.    ','ZTRN.    ','ZWRD.    ','ZGRN.    ','<         ',        CPCOM2A
2 '>         ',';         ','(         ',')         ',',         ',    CPCOM2B
3 ':         ','AFTER     ','BEFORE   ','CHANGE    ','DO        ',    CPCOM2C
4 'ELSE      ','FAILURE ','FLAG      ','GO        ','GOTO      ',    C PCOM2D
5 'IF        ','IN        ','RESULT   ','RPT       ','STOP      ',    cPCOM2E
6 'SUCCESS','TEST      ','THEN     ','TO        ','TRACE     ',       CPCOM2F
7 'TREE      ','UNTRACE '/                                             CPCOM2G
 COMMON /CPCOM/ SYTV(100),STK(100),OTOP(20),OTOPS(50),CPPTR,SPTR,       CPCOM3
1 LVL,SYTN,ZSCAN,SYTL(100),                                             CPCOM4
2 ZINT,ZTRN,ZWRD,ZGRN,ZLAN,ZRAN,ZSMI,                                   CPCOM5
3 ZLPR,ZRPR,ZPER,ZCOL,ZAFT,ZBEF,ZCHN,                                   CPCOM6
4 ZDO,ZELS,ZFAL,ZFLG,ZGO,ZGOT,ZIF,                                      CPCOM7
5 ZIN,ZRSL,ZRPT,ZSTP,ZSUC,ZTST,ZTHN,                                    CPCOM8
6 ZTO,ZTRC,ZTRE,ZUNT,INFNTY                                             CYCOM9
 INTEGER*2                                                              CPCOM9A
1 ZINT/1/,ZTRN/2/,ZWRD/3/,ZGRN/4/,ZLAN/5/,ZRAN/6/,ZSMI/7/,              CPCOMYB
2 ZLPR/8/,ZRPR/9/,ZPER/10/,ZCOL/11/,ZAFT/12/,ZBEF/13/,ZCHN/14/,         CPCOM9B
3 ZDO/15/,ZELS/16/,ZFAL/17/,ZFLG/18/,ZGO/19/,ZGOT/20/,ZIF/21/,          CPCOM9C
4 ZIN/22/,ZRSL/23/,ZRPT/24/,ZSTP/25/,ZSUC/26/,ZTST/27/,ZTHN/28/,        CPCOM9D
5 ZTO/29/,ZTRC/30/,ZTRE/31/,ZUNT/32/,INFNTY/10/                         CPCOM9E
 COMMON /CPCOM/ FGN,FGV,TRCF,APFG, IFFG,RFG                             CPCOM10
 LOGICAL*1 FGN(100,10),FGV(100,10),TRCF(100,5),APFG,IFFG,RFG            CPCOM11
 ENG                                                                    %
```

Figure 3.11.3

STACK-

STK - CONTENTS:					
0	*POSN*	POSN	POSN	POSN	current position in CPBUF
1	VAL	VAL	VAL	VAL	current value *of* the evaluation
2	TOP	TOP	TOP	TOP	current top node
3	0	RPT	IN	IF	type *of* operation
4	-	RPT-CTR	T-NAME	-	see below
5	-	-	IN-NODE	-	see below

Where:

POSN	= some setting of CPPTR
VAL	= 0 for false > 0 for true
TOP	= either one (by default) or sane value inserted by an IN-construct
0	= flag indicating a CONTROL- or TRANSFORMATION-LIST
RPT	= flag for RPT-construct (ZRPT)
IN	= flag for IN-construct (ZIN)
IF	= flag for IF-construct (ZIF)
RPT-CTR	= repeat counter; set to INFNTY or the integer following the RPT and counted' down to zero
T-NAME	= number of the IN-transfomation
IN-NODE	= node number of the current top of the tree for an IN-construct

Note that any instruction containing an angular bracket always affects the stack. So does an IF-construct.

Figure 3.11.4

TERMINAL SYMBOLS

TERMINALS AND ASSOCIATED VALUES (ALL IN **/CPCOM/)** S-28-68

TERMINAL	VALUE #	STATEMENT
<UNDF'D SYMBOL>	0	9
<INTEGER>	ZINT 1	
<T-NAME>	ZTRN 2	1000
<WORD>	**ZWRD** 3	
<GROUP-NO>	ZGRN 4	**2000**
<	**ZLAN 5**	**2050**
>	ZRAN 6	**2060**
;	**ZSMI 7**	**10**
(	**ZLPR** 8	
)	ZRPR 9	
.	ZPER **10**	**3000**
:	**ZCOL 11**	
AFTER	ZAFT 12	
BEFORE	ZBEF **13**	
CHANGE	**ZCHN** 14	
Do	**ZDO 15**	
ELSE	ZELS 16	**6550**
FAILURE	ZFAL **17**	
FLAG	ZFLG 18	**4000**
GO	ZGO 19	**S000**
GOTO	ZGOT 20	**5500**
IF	ZIF 21	**6000**
IN	ZIN 22	**7000**
RESULT	ZRSL 23	
RPT	ZRPT 24	**8000**
STOP	**ZSTP** 25	**3000**
SUCCESS	ZSUC 26	
TEST	**ZTST 27**	
THEN	**ZTHN 28**	**6500**
TO	**ZTO 29**	
TRACE	**ZTRC 30**	**9000**
TREE	**ZTRE 31**	**9800**
UNTRACE	**ZUNT 32**	10000

Figure 3.11.5

SYNTAX FOR SYNCHK

1	CTLPGM.	::=	PGM.	.				
2	PGY.	::=	sC.	CP.				
3	PGM.	::=	CP.					
4	CP.	::=	c r.	SC.	CP.			
5	CP.	::=	CI.	SC.				
6	CP.	::=	CI.					
7	CI.	::=	LRL.	CI.				
8	CI.	::=	LRL.					
9	CI.	::=	INST.	LBL.				
10	CI.	::=	INST.					
11	SC.	::=	;	SC.				
12	SC.	::=	;					
13	LBL.	::=	ZWRD.	:	LBL.			
14	LRL.	::=	ZWRD.	:				
15	INST.	::=	CE.					
16	INST.	::=	CL.					
17	INST.	::=	TE.					
18	CL.	::=	<	SC.	CL1.	>		
19	CL.	::=	c	CL1.	>			
20	CL1.	::=	INST.	SC.	CL1.			
21	CL1.	::=	INST.	SC.				
22	CL1.	::=	INST.					
23	CE.	::=	RPT-I.					
24	CE.	::=	IN-I.					
25	CE.	::=	IF-I.					
26	CE.	::=	FLG-I.					
27	CE.	::=	GO-I.					
28	CE.	::=	TRC-I.					
29	CE.	::=	STP-I.					
30	TE.	::=	ZTRN.					
31	TE.	::=	ZGRN.					
32	RPT-I.	::=	RPT	ZINT.	c	PGM.	>	
33	RPT-I.	::=	RPT	<	PGY.	>		
34	IN-I.	::=	IN	ZTRN.	(	ZINT.	)	DOP.
35	DOP.	::=	DO	C	PGM.	>		
36	IF-I.	::=	If	INST.	THEN	GO-I.	ELSE	GO-I.
37	IF-I.	::=	IF	INST.	THEN	GO-I.		
38	FLG-I.	::=	FN.	TL.				
39	FN.	::=	FLAG	ZINT.				
40	FY.	::=	FLAG					
41	GO-I.	::=	GOTO	ZWRD.				
42	GO-I.	::=	GO	TO	ZWRD.			
43	TRC-I.	::=	TRACE	TL.	SPEC.			
44	TRC- I.	::=	UNTRACE	TL.				
45	TRC-I.	::=	TREE					
46	SPEC.	::=	BEFORE	TEST				
47	SPEC.	::=	AFTER	SUCCESS				
48	SPEC.	::=	AFTER	FAILURE				
49	SPEC.	::=	AFTER	CHANGE				
50	SPEC.	::=	RESULT					
51	STP-I.	::=	STOP					
52	TL.	::=	<	SC.	TL1.	>		
53	TL.	::=	<	TL1.	>			
54	TL.	::=	TE.					
55	TL1.	::=	TE.	SC.	TL1.			
56	TL1.	::=	TE.	SC.				
57	TL1.	::=	TF.					

4. COMMON BLOCKS

```
      IMPLICIT INTEGER*2 (A-Z)


C /CRDCM/
      COMMON /ORDCM/ NUM,ISPEC,CRDFL,NUMFL
      INTEGER*2 NUM,ISPEC
      LOGICAL*1 CRDFL,NUMFL
C /MAINCM/
      COMMON /MAINCM/ CHRTR,KSUMP, ISUMP,NCHRTR
      REAL*8 CHRTR,KSUMP(2000)
      INTEGER*2 ISUMP,NCHRTR
c /CNSTCM/
      COMMON/CNSTCM/ NBLANK,NLAND ,NMINUS,NSLASH,NCENT ,NSTOP ,NLESS
     1 NLEFTP,NPLUS ,NLOR  ,NXCLM ,NDOLLR,NSTAR,NRITEP,NSCOLN,NLNOT
     2  N1211,NCOMMA,NPERC ,NLINE ,NGREAT,NQUERY,NCOLON,NBOUND,NAT
     3 NQUOTE,NEQUAL,NDQUOT
      INTEGER*2      NBLANK,NLAND ,NMINUS,NSLASH,NCENT ,NSTOP ,NLESS
     1 NLEFTP,NPLUS ,NLOR  ,NXCLM ,NDOLLR,NSTAR ,NRITEP,NSCOLN,NLNOT
     2 N1211 ,NCOMMA,NPERC ,NLINE ,NGREAT,NQUERY,NCOLON,NBOUND,NAT
     3 NQUOTE,NEQUAL,NDQUOT
C /FCSTCM/
      COMMON /FCSTCM/
     1FBLANK,FLAND ,FMINUS,FSLASH,FCENT ,FSTOP ,FLESS ,FLEFTP,FPLUS  ,
     2FLOR  ,FXCLM ,FDOLLR,FSTAR ,FRITEP ,FSCOLN,FLNOT ,F1211 ,FCOMMA,
     3FPERC ,FLINE ,FGREAT,FQUERY,FCOLON,FBOUND,FAT   ,FQUOTE,FEQUAL,
     4FDQUOT,PAGE . ,RECORD
      REAL*8
     1FBLANK, FLAND ,FMINUS,FSLASH, FCENT ,FSTOP ,FLESS ,FLEFTP,FPLUS  ,
     2FLOR  ,FXCLM ,FDOLLR,FSTAR ,FRITEP,FSCOLN,FLNOT ,F1211 ,FCOMMA,
     3FPERC ,FLINE ,FGREAT,FQUERY,FCOLON,FBOUND,FAT   ,FQUOTE,FEQUAL,
     4FDQUOT,PAGE  , RECORD
C /MAINCM/
      COMMON /MAINCM/  CHRTR,KSUMP,ISUMP,NCHRTR
      REAL*8  CHRTR,KSUMP(2000)
      INTEGER*2 ISUMP,NCHRTR
c /PSGCM/
      COMMON /PSGCM/ NSGA1,NSGC,NSGA2,NSGB,KA,KB,KC
      REAL*8 NSGA1(200),NSGC(2000)
      INTEGER*2 NSGA2(200),NSGB(300),KA,KB,KC
c /T/
      COMMON/T/ TITLE(F)
      REAL*8 TITLE
```

```
c /TREECM/
      COMMON /TREECM/ FTREE,TREE,CLIST,MTREE,MCLIST
      REAL*8 FTREE(400)
      INTEGER*2 TREE(400,6),CLIST(400),MTREE,MCLIST
C /Z/
      COMMON /Z/ LANK,NXXX,NSS,NS,NAND,NOR,NANDCR,NERROR
      REAL*8     LANK,NXXX,NSS,NS,NAND,NOR,NANDOR
      INTEGER*2 NERROR
C /WORK1/
      COMMON /WORK1/ LINE,STRING,NSUBS2,NSUBS1,N1,N2
      REAL*8 LINE(14),STRING(120),NSUBS2(10)
      INTEGER*2 NSUBS1(10),N1(14),N2(14)
C /RESTCM/
      COMMON /RESTCM/ WD,CREST,PS,PR,PC,PW,RESTS(500) ,RESTR(500),
     1 CONDS(500 )
      REAL*8 WD(100)
C /FRCCM/
      COMMON /FRCCM/ OUTWDS,LNGPTS,LNGPT,PTPT,THDS,HDS,TENS,UNITS,WDF
     1 LNGWDS
      REAL*8 OUTWDS(100)
      INTEGER*4 LNGPTS( 130 )
      INTEGER*2 LNGPT,PTPT,THDS,HDS,TENS,UNITS,WDPT
      LOG ICAL*1 LNGWDS (2000)
C /CHANCM/
      COMMON/CHANCM/CHAN(400,6),CHWORD(100),OPLIST(50),
     1 FCHTRE (200), CHTREE(200,6),CHCLIS(200),
     2 NCHAN,MXCHAN,NCHW,MXCHW,NOPL,MXOPL,NCHT,NCHCL,MXCHT,MXCHCL
      REAL*8 CHWORD,FCHTRE,OPLIST
C /FEATCM/
      COMMON /FEATCM/ FTNAME,MXEXP,NBCAT,NBEXP,NRULE,RULE(2,200 )
      REAL*8 FTNAME(100)
```

```
c /CSCM/
      COMMON /C SCM/
     1 ANALWC,CSLIST(4,2000),ANALPT(500),ANALWP(2000),ANALST(2000),
     2 TEMPAN(2000),SLCTPT(200,2),ANALTP,SLCTTP,CSFG,CSFRPT,ANALWT
      REAL*8 ANALWD(200)
C /LEXCM/
      COMMON/LEXCM/
     1 LEXWD,LEXWDS,LEXCS,LEXCSS,LXCPTR,CATLST,NLXC,NLEX,NLEXW,NLEXCS,
     2NCATL
      REAL*8 LEXWD(500),CATLST(20)
      INTEGER*2 LEXWDS(300),LEXCSS(300),LEXCS(500),LXCPTR(100,20),
     1 NLXC(20),NLEX,NLEXW,NLEXCS,NCATL
C /LINSCM/
      COMMON/LINSCM/
     1 SRCHL,ELIST,NSRCHL,NELIST
      INTEGER*2 SRCHL(2,50),ELIST(2,50),NSRCHL,NELIST
C /TRANCM/
      COMMON/TRANCM/FTRAN(100),KEYS(200),TRAN(100,7),KEYPT(100),
     1      NTRAN,NKEYS
      REAL*8 FTRAN,KEYS
C /RTCKEN/
      COMMON /RTCKEN/UPFG,LTH,TCK(10)
c /SKELCM/
      COMMON/SKELCM/FISKEL,ISKEL,SKLIST,ISKELT,MSKLST
      REAL*8 FISKEL(200)
      INTEGER*2 ISKEL(200,6),SKLIST(200),ISKELT,MSKLST
c /CONFCM/
      COMMON /CONFCM/
     1 CFVALS(100)
```

```
C /CPCOM/
      CGMMCN /CPCOM/CPBUF,SYTB,TERM
      REAL*8 CPBUF(500),SYTB( 100),TERM(32)
      COMMON /CPCOM/SYTV(100),STK(100),OTOP(20),OTOPS(50),CPPTR,SPTR,
     1 LVL,SYTN,ZSCAN,SYTL(100),
     2 ZINT,ZTRN,ZWRD,ZGRN,ZLAN,ZRAN,ZSMI,
     3 ZLPR,ZRPR,ZPER,ZCOL,ZAFT,ZBEF,ZCHN,
     4 ZCO,ZELS,ZFAL,ZFLG,ZGO,ZGOT,ZIF,
     5 ZIN,ZRSL,ZRPT,ZSTP,ZSUC,ZTST,ZTHN,
     6 ZTC,ZTRC,ZTRE,ZUNT, INFNTY
      COMMON /CPCOM/ FGN,FGV,TRCF,APFG,IFFG,RFG
      LOGICAL*1 FGN(100,10),FGV(100,10),TRCF(100,5),APFG,IFFG,RFG
C /TRANCM/
      COMMON/TRANCM/FTRAN(100),KEYS(200),TRAN(100,7),KEYPT(100),
     1      NTRAN,NKEYS
      REAL*& FTRAN,KEYS
C /ANALCM/
      COMMON /ANALCM/NUMNOD(50,10),SKPNOD(200,2),NUMCNT,SKPTOP,
     1 ANLIST(100),ANWDPT(100),ANNODE( 100),TTPOSN,UNDNOD,TOPNOD,RESTNO,
     2 TNO,ANSKIP(100),ANPREV(100),ANPAR(100),ANNEX(100)
C /SYNCM/
      COMMON/SYNCM/ S N T X ,STAK,CUR,SCN,SPT,IPT,NEQN,
     1 TRCFG,DMPFG,RECFG
      REAL*8 SNTX(57,8),CUR,SCN
      INTEGER*2STAK( 1000 )
      LOGICAL*1 TRCFG,DMPFG,RECFG
```

5. BLOCK DATA SUBPROGRAMS

```
      BLOCK DATA
      IMPLICIT INTEGER*2(N)
      COMMON/CNSTCM/ NBLANK,NLAND ,NM INUS,NSLASH,NCENT ,NSTOP ,NLESS  ,
     1  NLEFTP,NPLUS ,NLOR  ,NXCLM ,NDOLLR,NSTAR ,NRITEP,NSCOLN,NLNOT ,
     2    N1211 ,NCOMMA,NPERC ,NLINE ,NGREAT,NQUERY,NCOLON,NBOUND,NAT ,
     3 NQUOTE,NEQUAL,NDQUOT
      DATA            NBLANK,NLAND ,NMINUS,NSLASH,NCENT ,NSTOP ,NLESS  ,
     1 NLEFTP,NPLUS ,NLOR  ,NXCLM ,NDOLLR,NSTAR ,NRITEP,NSCOLN,NLNOT  ,
     2   N1211 ,NCOMMA,NPERC ,NLINE ,NGREAT,NQUERY,NCOLON,NBOUND,NAT  ,
     3 NQUOTE,NEQUAL,NDQUOT
     4/Z0040,Z005C,Z0060,Z0061,Z004A,Z004B,Z004C,Z004D,Z004E,Z004F,Z005A
     5,Z005B,Z005C,Z005D,Z005E,Z005F,Z006A,Z006B,Z006C,Z006D,Z006E,Z006F
     6,Z007A,Z007B,Z007C,Z007D,Z007E,Z007F/
      RETURN
      END


      BLOCK DATA
      COMMON /FCSTCM/
     1 FBLANK,FLAND ,FMINUS,FSLASH,FCENT ,FSTOP ,FLESS ,FLEFTP,FPLUS  ,
     2 FLOR  ,FXCLM ,FDOLLR,F STAR,FRITEP,FSCOLN,FLNOT ,F 1 2 1 1 ,FCOMMA,
     3 FPERC ,FLINE ,FGREAT,FQUERY,FCOLON,FBOUND,FAT   ,FQUOTE,FEQUAL,
     4 FDQUOT,PAGE  ,RECORD
      REAL*8
     1 FBLANK,FLAND ,FMINUS,FSLASH,FCENT ,FSTOP ,FLESS ,FLEFTP,FPLUS  ,
     2 FLOR  ,FXCLM ,FDOLLR,FSTAR ,FRITEP,FSCOLN,FLNOT ,F1211 ,FCOMMA,
     3  FPERC ,FLINE ,FGREAT,FQUERY,FCOLON,FBOUND,FAT   ,FQUOTE,FEQUAL,
     4 FDQUOT,PAGE  ,RECORD
      DATA
     1 FBLANK,F L A N O ,FMINUS,FSLASH,FCENT ,FSTOP ,FLESS ,FLEFTP,FPLUS ,
     2 FLOR  ,FXCLM ,FDOLLR,FSTAR ,FRITEP,FSCOLN,FLNOT ,F1211 ,FCOMMA,
     3  FPERC ,FLINE ,FGREAT,FQUERY,FCOLON,FBOUND,FAT   ,FQUOTE,FEQUAL,
     4 FDQUOT,PAGE  ,RECORD
     5 /' ','&','-','/',' ','.','<','(','+',Z4F40404040404040,' ','$',
     6 '*',')',';','¬',' ',',','%','_','>','?',':','#','@',
     7  Z7D40404040404040,'=','"','P$$$$$$$','B$$$$$$$'/
      END


      BLOCK DATA
      COMMON /Z/LANK,NXXX,NSS,NS,NAND,NOR,NANDOR,NERROR
      REAL*8      LANK,NXXX,NSS,NS,NAND,NOR,NANDOR
      INTEGER*2 NERROR
      DATA        LANK,NXXX,NSS,NS,NAND,NOR,NANDOR,NERROR
     1/' ','XXX','S' ,'S','AND','OR','ANDOR',0/
      END
```

```
 BLOCK DATA
 COMMON/CHANCM/CHAN(400,6),CHWORD(100),OPLIST(50),
1   FCHTRE (200), CHTREE(200,6),CHCLIS(200),
2 NCHAN,MXCHAN,NCHW,MXCHW,NOPL,MXOPL,NCHT,NCHCL,MXCHT,MXCHCL,NCOP
 REAL*8 OPLIST/5HSUBSE,5HARISE,5HALESE,5HALADE,5HAFIDE,5HARIAE,
1  5HERASE,5HSUBST,5HADRIS,5HADLES,5HADLAD,5HADF ID,5HADRIA,
1  6HSUBSEI,6HARISEI,6HALESEI,6HALADEI,6HERASEI,6HSUBSTI,
1  6HADRISI,6HADLESI,6HADLADI,6HERASEF,5HSAVEF,6HMERGEF,
1  5HMOVEF/
 REAL*8 CHWORD,FCHTRE
 INTEGER*2 CHAN,CHTREE,CHCLIS
 INTEGER*2 NOPL/26/,NCOP/23/
 INTEGER*2 NCHAN/0/,NCHW/0/,NCHT/0/,NCHCL/0/
 INTEGER*2 MXCHAN/400/,MXCHW/100/,MXOPL/50/,MXCHT/200/,MXCHCL/200/
 END


 BLOCK DATA
 IMPLICIT INTEGER*2 (A-Z)
 COMMON /CPCOM/ CPBUF, SYTB,TERM
 REAL*8 CPBUF(500 )/'STOP     '/,SYTB(100),TERM(32)/
1 'ZINT.    ','ZTRN.    ','ZWRD.    ','ZGRN.    ','<         ',
2 '>         ',';         ','(         ',')         ','.         ',
3 ':         ','AFTER     ','BEFORE    ','CHANGE    ','DO        ',
4 'ELSE      ','FAILURE   ','FLAG      ','GO        ','GOTO      ',
5 'IF        ',' IN       ','RESULT    ','RPT       ','STOP      ',
6 'SCCCESS   ','TEST      ','THEN      ','TO        ','TRACE     ',
7 'TREE      ','UNTRACE   '/
 COMMON /CPCOM/ SYTV(100),STK(100),OTOP(20),OTOPS(50),CPPTR,SPTR,
1    LVL,SYTN,ZSCAN,SYTL(100),
2 ZINT,ZTRN,ZWRD,ZGRN,ZLAN,ZRAN,ZSMI ,
3 ZLPR,ZRPR,ZPER,ZCOL,ZAFT,ZBEF,ZCHN,
4 ZDO,ZELS,ZFAL,ZFLG,ZGO,ZGOT,ZIF,
5  ZIN, ZRSL,ZRPT,ZSTP,ZSUC,ZTST,ZTHN,
6 ZTC,ZTRC,ZTRE,ZUNT,INFNTY
 INTEGER*2
1 ZINT/1/,ZTRN/2/,ZWRD/3/,ZGRN/4/,ZLAN/5/,ZRAN/6/,ZSMI/7/,
2 ZLPR/8/,ZRPR/9/,ZPER/10/,ZCOL/11/,ZAFT/12/,ZBEF/13/,ZCHN/14/,
3 ZDO/15/,ZELS/16/,ZFAL/17/,ZFLG/18/,ZGO/19/,ZGOT/20/,ZIF/21/,
4 ZIN/22/,ZRSL/23/,ZRPT/24/,ZSTP/25/,ZSUC/26/,ZTST/27/,ZTHN/28/,
5 ZTC/29/,ZTRC/30/,ZTRE/31/,ZUNT/32/,INFNTY/10/
 COMMON /CPCOM/FGN,FGV,TRCF,APFG,IFFG,RFG
 LOGICAL*1 FGN(100,10),FGV(100,10),TRCF(100,5),APFG,IFFG,RFG
 END
```

6. POSSIBLE EXTENSIONS

There are certain extensions to the Transformational Grammar System which we have considered, but which have not been implemented. An informal discussion is given here of ways in which these extensions might be made. The additions considered are:

A. Rule features

B. Tree-pruning

C. n-ary features

D. Restrictions on skips

E. Analysis of skips,

6.1 Rule features

This section discusses the changes which would be necessary to include rule features. It is inconclusive in not defining where rule features will appear and where they will be looked for. This is an open linguistic question, as is the question of the need for rule features.

Input of rule features

A rule feature is simply a transformation name used as a feature. However, since the lexiconmay be read in before the transformations, the program cannot recognize rule features as such. The lexicon contains a list of category features, a list of inherent features, and a list of contextual feature definitions. Any feature which does not occur in those lists is now assumed by the program to be an inherent feature, and a message "WARNING. NUMNAM. FEATURE xxx ADDED AS INHERENT"

is printed, In the table of feature names (FTNAME) the entries from 1 to NBCAT are names of category features, and from NBCAT+1 to NBEXP names of inherent features. Names of contextual features are stored in SLNAME .

To modify the input to allow for rule features, the use of FTNAME would be modified slightly so that the entries from 1 to NBCAT were category features, the entries from NBCAT+1 to NBSPEC were inherent features given on the list of inherent features in thelexicon, and the entries from NBSPEC+1 to NBEXP were additional feature names encountered in reading the lexicon. At this point the program would not know if they were inherent features or rule features, so the message above would be altered to "WARNING. NUMNAM. FEATURE xxx ADDED AS INHERENT OR RULE?

After the transformations have been read in, the feature name table FTNAME could be searched from NBSPEC+1 to NBEXP to see which transformation names occur there. The number of the corresponding rule feature (i.e., the index in FTNAME) can be stored in TRAN(1,4) . This column contained the EMB parameter until EMB was abolished.

Effect of rule features

The possible cases in which rule features can affect the handling of a transformation are shown in the table on the next page, where 1 indicates apply, 0 don't apply, and .5 apply with probability 0.5 .

transformation type	value of rule feature		
	+	unmarked	-
OPC	1.0	0.5	0.0
OBmajor	1.0	1.0	0.0
OBminor	1.0	0.0	0.0
OPmajor	0.5	0.5	0.0
OPminor	0.5	0.0	0.0

(The use of major and minor rules is discussed in Lakoff*. The subdivision of each of these classes into OB and OP seems to be a natural extension.)

The system now allows OB and OP as the only two optional classes. The list could easily be extended to the five classes above by inventing suitable mnemonics.

Testing rule features

Currently no tests are made of features except as they occur within complex symbols. The best approach to rule features would seem to be to write an integer*2 subroutine FTINC(csno,featno) which would return the value of the feature specification for featno in complex symbol csno . Values are currently represented as 1 for + and 2 for - . 0 could thus represent the unmarked case. In testing the structural description of a transformation with a rule feature, ANTEST could call FTINC to obtain its value.

*Lakoff, G. On the Nature of Syntactic Irregularity. NSF-16, The Computation Laboratory, Harvard University (1965).

Where to look for rule features

The difficulty problem in incorporating rule features comes in deciding where to look for them. The following possibilities occur:

(1) look for rule features on every-node used in the analysis,

(2) look for rule features only on nodes corresponding to numbered terms,

(3) look for rule features only on nodes corresponding to numbered terms with a small subset of special numbers,

(4) look for rule features only after encountering some special symbol in the structural analysis.

Alternative (1) is bad because there is the possibility that the rule feature might be found more than once, with opposite values. Alternative (2) is bad for this same reason. The linguist must number terms for use in restrictions and structural change. He might thus be forced to number two which would have the rule feature with opposite values. Alternative (3) would notpresent any real problems, since the numbers are otherwise arbitrary (and between 1 and 50). Alternative (4) is unpleasant because it would require some changes to CXIN .

Lakoff has suggested that the rule feature should be looked for on the main verb. This does not solve our problem since we still need to indicate the main verb. However, it does lead us to think that alternatives (1) and (2) above are too broad,

Note that we cannot require that the rule feature be explicitly mentioned in the structural description because this would mean that it must always be present for the transformation to work. This would be acceptable only for OBminor and OPminor rules.

Suppose we were to create a new restriction RUL which would be true if the node were marked for the rule feature of the current transformation and false otherwise. Then the analysis would fail at that point if the value of FTINC(csno,featno) is - , but should it then proceed to look for another analysis?

When should rule features be tested

In order that the rule feature test tie in properly with the repetition parameters (AC, ACAC, AACC, AAC) and the optionality parameters (OB, OP) it would appear that the rule feature should be tested only after the analyses have been found. Otherwise an AC transformation, for example, would go on to find a second analysis, when the first fails, only because of the rule feature. This problem needs to be thought about carefully, since it is not clear what is linguistically correct in the various cases which arise.

6.2 Tree-pruning

Some linguists (notably Ross*) have discussed a notion of tree-pruning. Tree-pruning is essentially an obligatory transformation which must be applied whenever the structural description is met, and thus fails to fall into any linear ordering of transformations. Within the system as it stands tree-pruning could be handled by defining one or more tree-pruning transformations, PRUNE1, ..., PRUNEn, and writing the control program for the transformations so that these transformations

*Ross, J. R. A proposed rule of tree-pruning. Presented to the Linguistic Society of America (1965).

are invoked after every successful application of another transformation. This is somewhat awkward and it might be desirable to handle this automatically by an instruction in the control program itself, say TREEPRUN, which would automatically invoke the-tree-pruning transformations after each change. To do this one would simply add the new instruction to the control language, and then incorporate the calls to the tree-pruning transformations into the TRACE subroutine at the same point that TRACE . ..AFTER CHANGE is now tested. This would be more elegant; it would also be more time-consuming in execution.

6.3 n-ary features

The recent attention to case in grammars of English might be best handled by the use of features with more than 2 values. Fillmore* has proposed that case be handled within the phrase structure, but treatment by n-ary features is an alternative which should certainly be explored. To do this would require some fairly major changes in the system, both in the input routine for complex symbols and in the various tests and changes to complex symbols. One possibility for external format would be to allow small integers as values in addition to the + , - and * now allowed, viz., |+ N3 PREP - HUMAN| . There is no basic reason why this could not be done, but it would take some time to implement it well.

*Fillmore, C. J. A proposal concerning English prepositions, Georgetown Monograph Series on Language and Linguistics, 19(1966), pp. 19-34.

6.4 Restrictions on skips

Although the present syntax for structural analysis for the system does not allow the numbering of terms which are skips, the possibility of doing so might be considered. This would make it possible to test dominance and nondominance restrictions on skips. It is not clear what equality of skips should mean. The main question here is whether linguistically a skip should be treated as analyzable in any way. The system of transformational grammar as it now stands is cleaner than one which would allow this, and we have seen no examples in which it is required (although it might have been used in the definition of LOWESTS).

The problem could be handled by an integer function INSKIP*2(SKPPTR,WORD,ITEST) which would decide whether there was a node with the real*8 name WORD in the scope of the skip whose bounds are in the SKPPTR-th entry of SKPNOD . If so, it returns that node's position in the TREE; if not, it returns 0 . When ITEST = 0, the entire range is searched. When ITEST $\neq$ 0, the range beyond node number ITEST is searched.

This subroutine could be called by both the restriction tester (RESTST) and the analysis tester (ANTEST) . For RESTST the call would be INSKIP(NUMNOD(N,NUMCNT),word,0) . If the value is 0, there is no dominance; if nonzero, dominance.

6.5 Analysis of skips

The syntax for skips which was originally considered for the system was:

skip % opt[opt[¬]opt[&]⟨clist[structure]⟩]

The interpretation would be that the clist of structures referred to structures within the range of the skip. & would mean that all must be present; ¬ & would mean that none may be present; ¬ would mean that at least one must not be present; and no preceding symbol would mean that at least one must be present. For the reasons discussed in D above we have not felt that this strong a definition was necessary. To make the extension would require changes to the analysis routine, ANTEST . The subroutine INSKIP described above could be used here.

The call from ANTEST would be approximately

```
      1 = 0
a     I =  INSKIP(SKPPTR,ANALWD(ANLIST(POSN)),I)
      IF (IJ3Q.0) GO TO . . .
      IF ( . . .                          ) GO TO  α
      ...
```

MODIFIED 23 AUGUST 1968

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 BOUNDARY SYMBOL
1.04 SENTENCE SYMBOL ::= s
1.05 BOUND&Y SYMBOL ::= #

2.01 STRUCTURAL DESCRIPTION ::= STRUCTURAL ANALYSIS opt[, WHERE RESTRICTION] .
2.02 STRUCTURAL ANALYSIS ::= list1 TERM J .
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 I)
2.07 SKIP ::= %

3.01 RESTRICTION ::= booleancombination[CONDITION I
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 NUDE
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 V A L U E : : = + o r - o r *

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 ::= UNARYOPERATOR 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 BINARYTREE 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 L E F T::=NODE
6.04  RULF RIGHT::=NODE or l i s t f RULE RIGHT }o f( list{ RULE RIGHT } )o r( c l i s t f RULE RIGHT } )

7.01  LEX CON ::= LEXICON PRELEXICON LEXICAL ENTRIES $END
7.02  PRELEXICON ::= FEATURE DEFI NITIONS opt{ REDUNDANCY RULES }
7.03  FEA URE DEFINITIONS ::= ATEGORY DEFINITIONS opt{ INH INHERENTDEFINITIONS } opt{ CONTEXTUAL DEFINITIONS }
7.04  CATEGORY DEFINITIONS ::= CATEGORY list{ C A T E G O R Y FEATURE } .
7.05  INHERENT DEFINITIONS ::= INHERENT listf INHERENT FEATURE }.
7.06  CONTEXTUAL DEFINITIONS::= CONTEXTUAL cl istf C O NTEXTUAL DEFINITION } .
7.07  CON EXTUAIDEFINITION::= CONTEXTUAL FEATURE LABEL = CONTEXTUAL FEATURF DESCRIPTION
7.08  CON EXTUAL F E A T U R E LABEL ::=WORD
7.03  REDUNDANC Y RULES::=R U L E S 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  TRANSFORMAT I OYS : :=TRANSFORMATIONS 1 ist{ TRANSFORMATION } CP CONTROL PROGRAM . $END
8.02  TRANSFORMATION ::= TRANS IDENTIFICATION SD STRUCTURAL DESCRIPTION opt{ SC STRUCTURAL CHANGE . }
8.03  IDE TIFICATION : := opt{ INTEGER } TRANSFORMAT I ON NAME opt{ l i s t{ PARAMETER. }} opt{ KEYWORDS } .
8.04  PARAMETER ::= GROUP NUMBER or OPT I ONALITY OR REPETITION
8.05  GROUP NUMBER ::= I or II or III or IV or V or, VI or VI1
8.06  ONALITY ::=O B o r OP
8.07  REPETITION ::=A C o rA C A C o r AACC o r A A C
8.08  KEYWORDS ::= ( l i s t f NODE } )

9.01  CONTROL PROGRAM ::=s c l i s t f o p t f LABEL : }INSTRUCTION }
9.02  LABEL ::= WORD
9.33  STRUCTION ::= RPT I N S T R U C T I O N o r INSTRSTRUCTION o r IF INSTRUCTION
                  o r G O I NSTRUCTI O N o r TRACE INSTRUCTION o r STOP I NSTRUCT I ON
                  or T INSTRUCTIOII or <sclist{ INSTRUCTION } >
9.04  T INSTRUCTION::= TRANSFORMATOF GROUP GROUP NUMBER
9.05  R PTINSTRUCTION ::= RPT opt{ < CONTROD PROGRAM >
9.06  I N INSTRUCTI O N::=I N TRANSFORMAT I ON NAME (INTEGER ) D O < CONTROI PROGRAM >
9.07  I F INSTRUCT I ON ::=IF INSTRUCTION THEN GO INSTRUCTION opt{ E L S E GO INSTRUCTION }
9.08  GO INSTRUCTION ::=G O T O LABEI
9.09  TRACE INSTRUCTION ::=T R A C E T INSTRUCTION T R A C E SPECIFICATION or UNTRACE T INSTRUCTION o r TREE,
9.10  TRACE SPECIFICATION ::=BEFORE TEST or AFTER FAILURE or AFTER SUCCESS or AFTER CHANGE
9.11  S T O P INSTRUCTION::=STOP
```

APPENDIX B

Reports on the Computer System for Transformational Grammar

AF-14 CS-79	360 O.S. FORTRAN IV Free Field Input/Output Package	R. W. Doran October 1967
AF-15 CS-80	Directed Random Generation of Sentences (to appear, CACM)	J. Friedman October 1967
AF-21 CS-84	A Computer System for Transformational Grammar	J. Friedman January 1968
AF-24 CS-95	A Formal Syntax for Transformational Grammar	J. Friedman R. W. Doran March 1968
AF-25 CS-103	Lexical Insertion in Transformational Grammar	J. Friedman T. Bredt July 1968
AF-33 CS-108	Computer Experiments in Transformational Grammar	J. Friedman, Ed. September 1968
AF-34	Analysis in Transformational Grammar	J. Friedman T. Martner September 1968
AF-35	A Control Language for Transformations	J. Friedman B. Pollack September 1968