Technical Report TR-65-23   September 1965

NsG - 398

TREETRAN - A FORTRAN IV Subroutine Package

for Manipulation of Rooted Trees

by

John L. Pfaltz

TABLE OF CONTENTS

Introduction

This report is designed to serve as a manual by which a user with only a working knowledge of FORTRAN may, by a system of subroutines, extend FORTRAN IV to manipulate data which is structured as rooted trees.

Intuitively, rooted trees may be considered as a formalization of the situation where the nature of an event (or entity, or piece of data) is dependent upon a single preceding event in space or time. Consequently rooted trees can serve as valuable models in many different fields.

The chain of command structure is a rooted tree that naturally occurs in management, as are various operational decision trees. In the theory of games the choice of alternative moves is dependent upon the preceding move of the opponent, and similar tree structures appear in economic theory. Rooted trees can serve as a model for asexual reproduction, and, under certain conditions, animal migration. The syntax of all natural language is basically tree structured. Consequently trees have proved useful in language translation, both to other natural languages as well as to artifical computer languages. As one example of such possible TREETRAN applications, a program for the generation of syntactically correct sentences will be given later in this report.

The list of rooted-tree applications above is meant to be suggestive, not exhaustive. It is felt that many problems in economics, management, decision theory, language, and statistics, which could profitably be modeled with tree structures are ignored for want of programmers who are conversant with the traditional list structured languages such as LISP, IPL-V, SLIP etc.

It is hoped that this package of rooted tree routines will help bridge this gap by permitting researchers in diverse fields to manipulate tree structures in FORTRAN, a widely known and easily learned compiler language. In addition as we acquire a larger body of experience with applications with tree-structured data, the true value of this tool may be more accurately assessed.

TREETRAN - A FORTRAN IV Subroutine Package for Manipulation of Rooted Trees

I. BACKGROUND:

Very often the pieces of data to be used in solving a problem are not independent, but rather stand in some sort of relation to one another. A familiar example from mathematics is the matrix, where one seldom considers the individual elements independently, but rather in their "rectangular" relationship to the other elements.

Graphs and trees give examples of more general collections of structured data. Consider the data to be stored at the vertices of the graph and the edges to represent structure relationships between the data. With this view a matrix is just a very special data structure; one where the graph representing the relational structure is "rectangular".

It is clear that the power of matrix theory lies in the ability to consider and manipulate the matrix as an entity in its own right, and FORTRAN incorporates this ability. Matrix arrays are named. From then on the matrix may be read in or out by name only, and its name alone is sufficient to provide the entire array as argument to manipulative subroutines.

This subroutine package seeks to extend this type of labeling and manipulative ability to a different class of structured data -- specifically a collection of data whose interrelationships can be represented by a rooted tree. While it would be desirable to develop routines to handle still more general data structures; one encounters tree structures sufficiently often to make even this extension worthwhile.

II. ROOTED TREES:

Consider a graph with a directed edge joining the node a_1 to the node a_2. We will call a_1 a precedent of a_2 and call a_2 an antecedent of a_1.

By a rooted tree is meant a directed graph without circuits such that every node has at most one precedent. Graphically

a rooted tree may be drawn as below:
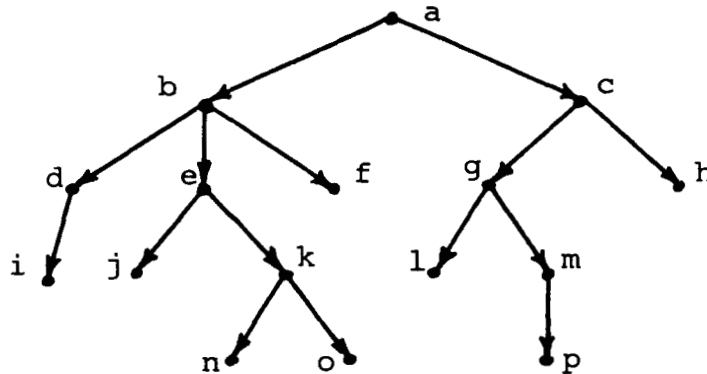
The root of the tree is the one node without a precedent
(node a in the figure). One may also consider subtrees and
their roots (perhaps principle node would be a better terminology).
For example, node e can be considered the root of the subtree
consisting of itself, j, k, n, and o.

A twig is a node without antecedents. Thus, i, j, n,
o, f, l, p, and h are twigs.

The depth of a node below a specified node is the number
of edges between them. Nodes d, e, f, g, and h are all of
depth 2 below the root a.

III. __FORTRAN IV IMPLEMENTATION__:

In FORTRAN a piece of data is represented by a sym-
bol, say "X". The symbol "X" represents the data itself
and may be used exactly as if it was a number or set of
alphabetic characters. The symbol is defined by an arithmetic,
input, or DATA statement.

In the rooted tree package we introduce symbols that do
not represent data, but represent a node of the tree where
data is stored. Such a symbol (for instance "A") we say
"names" a node. One must refer to the data associated with
node A indirectly via a DATA function. Thus,

$$X = DATA (A, FLAG)$$

makes X equal to the data stored at A.

A name symbol can be defined when a node is created. for

For example,

<div align="center">A = CREATE (1, ARG1)</div>

creates a node name "A" with one associated piece of data
ARG1. Every node is a given machine address name internally
by the system when it is created. However, the programmer
may also represent it with a symbol in case he wishes to
refer to it by name later in his program.

The arithmetic statement Y = X makes the symbol Y
represent the same value as X. B = A makes the symbol B
name the same node as A. For this reason we may call B
(and/ or A) an alias for the node. In order to avoid conflict
with the FORTRAN floating fixed point convention, we have
adopted the policy of denoting all node names and aliases
by floating point variables.

Since every tree (or subtree) has only one root there
is no ambiguity if we let the name of the root node also be
the name of the entire tree for which this is the root. In
all cases the context will make it clear whether the symbol
refers only to the node itself or to the entire tree for which
this node is the root.

Having symbolically named tree structures we may now
treat them as specific entities. In particular, we may attach
one tree to another; delete subtrees; compare trees; output
them; etc. These manipulations should become clear as
we discuss the actual TREETRAN subroutines.

IV. USE OF TREETRAN SUBROUTINES

Initialization:

Before executing any tree subroutines, space for the
tree structures must be reserved by the two statements

<div align="center">DIMENSION SPACE (n)
CALL INITAS (SPACE, n)</div>

Where n is approximately 6 times the number of nodes in use
at any one time. Note that when a tree structure is no longer
needed its memory locations can be returned to a common pool
for reassignment in succeeding tree structures.

Node Creation and Data:

Any number of words* of any type (fixed, floating, or

alphabetic) may be stored at a node. The function CREATE (n, ARG1, ARG2) creates, names and stores up to 2 words of data at a node simultaneously. n = 0, 1 or 2 denotes the number of data words to be stored at the node. ARG1 and ARG2 are the words of data to be stored (they need not be specified unless indicated by n). The function value is the name of the node.

For example, A = CREATE (1, 107.3) creates a node with name (or alias) A and stores the floating point number 107.3 there.

If more data is to be stored at a node the subroutine ADD (A, ARG) will add the datum ARG to any data already stored at node A.

Note that creating a node does not imbed it in a tree structure (except the trivial one consisting of only the node itself).

Three functions retrieve data that is stored at a given node DATA (A, FLAG) and IDATA (A,FLAG) both retrieve a single piece of data stored at a node A and deliver it as a function value. If more than one piece of data is associated with A these functions will return the first word that was stored there. If no data is associated with that node a very large number is returned and FLAG is set to TRUE. There are two functions solely to eliminate the difficulties of fixed floating point name conventions in FORTRAN.

With these two functions one can use data associated at a node in the conventional way in FORTRAN.
For example, the statements
IF (IDATA (C2, FLAG) . EQ. 7) GO TO n
and
Y = SIN (DATA(PLACE,FLAG)/2.0)
check whether the fixed point number stored at the node named C2 is equal to 7 and calculate the sine of one half the floating point number stored at the node called PLACE respectively.

---

* However, it is usually quite uneconomical to store large arrays in a tree structure.

It should be clear that
          X = DATA(CREATE(1,Y),FLAG)
merely makes X equal to Y; and that
          C = CREATE(2,DATA(A,FLAG),DATA(B,FLAG))
stores the data associated with nodes A and B at a new node
C as well.

If several words of data are stored at node A the subroutine
DLIST (A,N,ARRAY) puts the n words of data stored at A into
the vector ARRAY (which must have been dimensioned by the
calling program). The order of the words of data in ARRAY
will be the order they were stored at A.

So far only data consisting of a few words of numeric
or alphabetic data has been considered for storage at individual
nodes. However the data we wish to associate with a given
node might itself be tree structured. To accomplish this
we merely store the name of the data tree as the datum at the
node. In this manner we can quite easily build "trees of trees".

One question arises when retrieving data from nodes. Is
the word a piece of datum itself or is it just the name of
a still lower level data tree? Most programmers will probably
keep track within their program of the nature of the data they
expect to retrieve at any step. In cases of doubt, however,
a simple check is provided by the function NAMTST (DATUM).
This function returns zero if the datum is the name of a
subtree, non-zero otherwise.

Several further considerations involved in creating
"trees of trees" will be considered later in the section entitled
"Restrictions and Programming Considerations."

Tree Construction:

The function ATTACH (B, A) makes node B an antecedent
of node A. By using this function one can build trees link-
ing nodes to specified nodes. However because the name of a
node is also the name of the tree for which that node is
root, the ATTACH function can also attach entire tree structures
to other trees.

For example, if we have constructed two trees, TREE1 and TREE2 as in the following diagram,

Then the statement CALL ATTACH (TREE1, TREE2) creates the single tree shown in figure 2.

fig. 2
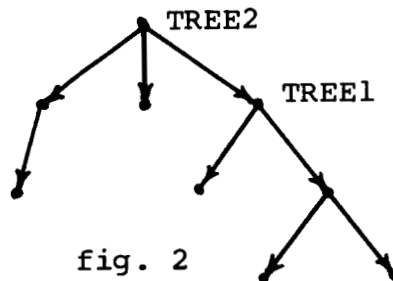
(Note that this structure is isomorphic to the subtree with root b in figure 1)

ATTACH can be called as either a subroutine or a function. When called as a function it delivers as its function value the name of the precedent node. This provides a nesting capability. If the program tries to attach a node that is already in a tree structure (i.e., has a precedent) a warning diagnostic is printed, the node (and corresponding subtree) is detached from its current structure and attached to the new structure.

As an example the reader may verify that the coding

```
DIMENSION SPACE (1000)
CALL INITAS (SPACE, 1000)
TREE = CREATE (1, 1)
X = TREE
DO 107 I = 2, 7
IF (MODF (I, 2) .EQ. 0) GO TO 106
X = ATTACH (CREATE 1, I), X)
GO TO 107
```

```
106 CALL ATTACH (CREATE (1, I), TREE)
107 CONTINUE
        .
        .
        .
```

generates the following tree where the numbers represent data at the nodes rather than their names.
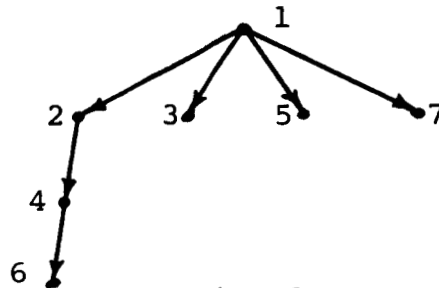
fig. 3

Tree Output:

A single print routine PRTREE (A,I) prints tree structures and their contents in an easily readable format. PRTREE represents the tree structure by varying the position of the node across the page corresponding to its depth below the root A. The second parameter controls the format, under which the contents of each node is printed, as follows:

$$
\begin{aligned}
I &= 1 \quad \text{Integer (I8)} \\
&= 2 \quad \text{Alphabetic (A6)} \\
&= 3 \quad \text{Floating Point (E11.4)} \\
&= 4 \quad \text{Octal (O12)}
\end{aligned}
$$

If PRTREE encounters the name of a tree stored as data it assigns and prints a label for the data tree. After printing the originally requested tree, it proceeds to print each of the data trees, together with the assigned label for identification.

Figure 4 is a graphic representation of a "tree of trees" where each node contains a single alphabetic letter as data. In addition the nodes with "C", "D", "M" and "W" also contain data which is itself tree structured. This relationship is indicated by the dashed lines. Figure 5 is the same tree structure as printed by PRTREE (the lines are drawn by hand to emphasize the tree structure).

Level 0

Level 1
Data trees
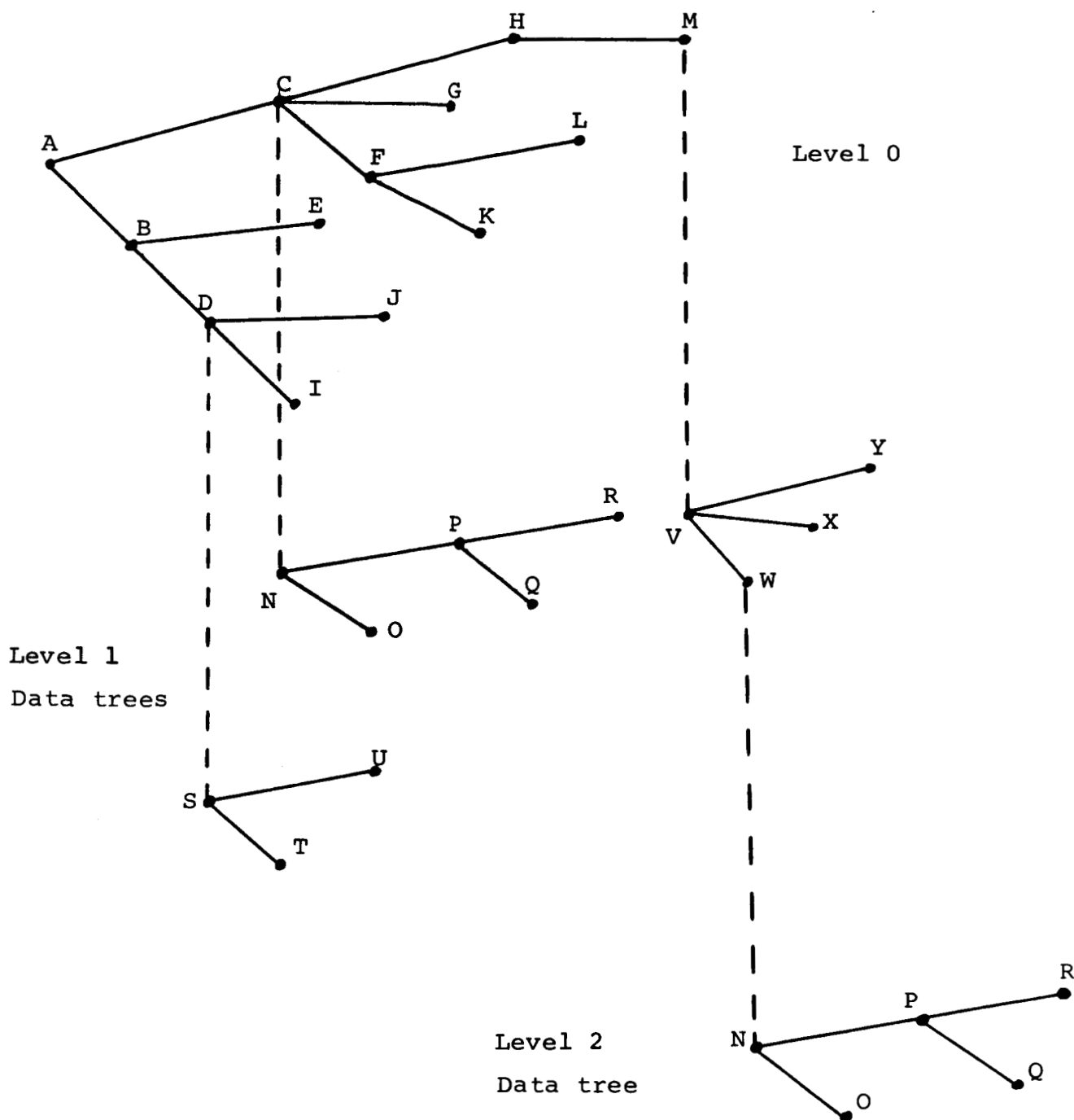
Level 2
Data tree

Tree structure in which single letters of the alphabet,
together with 4 lower level tree structures, are stored
as data at the nodes.

figure 4

ROOT 'NODE' . . . OF TREE STRUCTURE

A
      NODE
    B
        NODE
      D
      (DATA TREE
      LABEL 1)
           NODE
           I
           NODE
           J
        NODE
      E
    NODE
    C
    (DATA TREE
    LABEL 2)
        NODE
      F
           NODE
           L
           NODE
           K
        NODE
      G
        NODE
      H
           NODE
           M
      (DATA TREE
      LABEL · 3)
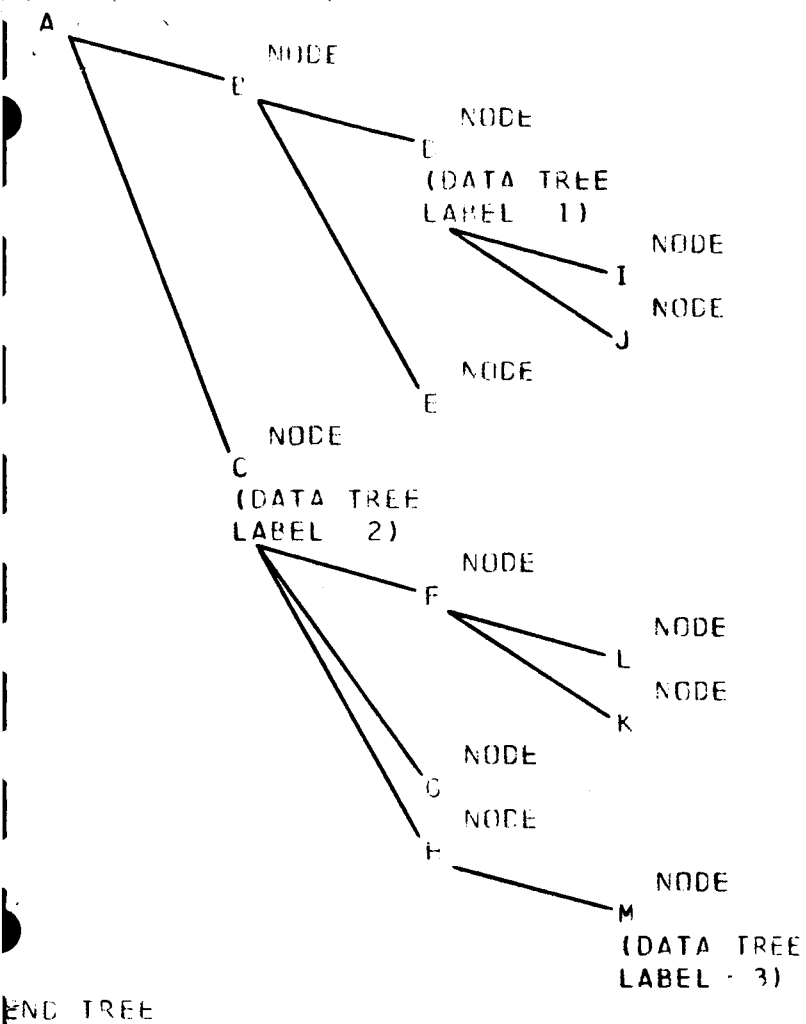
END TREE

ROOT NODE . . . OF DATA TREE, LABEL 1

S
      NODE
    T
      NODE
    U
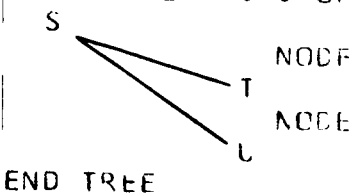
END TREE

figure 5

ROOT NODE . . . OF DATA TREE, LABEL 2

N

        NODE
      O
        NODE
      P

            NODE
           Q
            NODE
           R

END TREE

ROOT NODE . . . OF DATA TREE, LABEL 3

V

        NODE
      W
      (DATA TREE
      LABEL   2)
   NODE
      X
        NODE
      Y
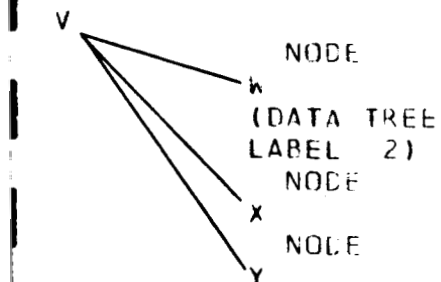
END TREE

figure 5 (cont.)

Manipulative Routines:

Given an arbitrary tree structure the following routines perform various operations on trees; in particular several find the names of nodes with specified properties.

FUNCTION PREC (A) - gives as its function value the name of the precedent of node A. If A is a root PREC is set to zero.

SUBROUTINE TWIGS (A, N, ARRAY) - finds all twigs (nodes without antecedants) in the subtree with root A. N denotes the number of twigs found and the n names are put into ARRAY (which must have been sufficiently dimensioned by the calling program).

SUBROUTINE DEPTH (A, L, N, ARRAY) - finds all nodes of depth L below node A, returns the number found as N, and returns their names in ARRAY. Note that N may be zero.

SUBROUTINE REMOVE (A, ARG) - removes the argument ARG from the list of data stored at node A. If ARG is repeated in the list every instance is deleted.

SUBROUTINE DETACH (A) - detaches node A from its precedent. A is now the root node of a separate tree.

FUNCTION COPY (A) - makes a separate tree that is an exact copy (both with respect to structure and data stored at nodes) of the subtree with root A. It delivers as functional value the name of the root of the new tree.

SUBROUTINE IRALST (A) - destroys the entire tree with root A, and returns its cells to the pool of available cells.

The following examples show some of the uses of these subroutines:

1) Suppose we are at an arbitrary node X in a tree and wish to find its root. We could use the following coding:

```
100     Y = PREC (X)
        IF (Y) 101, 102, 101
101     X = Y
        GO TO 100
102     ROOT = X
          .
          .
          .
```

or 2) Suppose we wish to find all "brothers" of a given node
X, (i. e., all nodes with the same precedent node). We could
use the following coding:

```
        DIMENSION ARRAY (30)
          .
          .
          .
        CALL DEPTH (PREC (X), I, N, ARRAY)
          .
          .
          .
```

Note that node X will itself be included in the list of brothers.

Tree Equality and Isomorphism:

In matrix theory two matrices are equal if their corresponding
elements are equal. Implicit in this definition is the assumption
that the two matrices were of the same dimension.

With rooted trees we have several concepts of "equality".
We may ask if the tree structures are completely identical,
regarding both structure and data stored at the individual
nodes; or we may merely ask if the structures are in some sense
equivalent, disregarding their data contents.

Even the concept equivalence between structures depends
on our intended usage. Consider the following three trees:
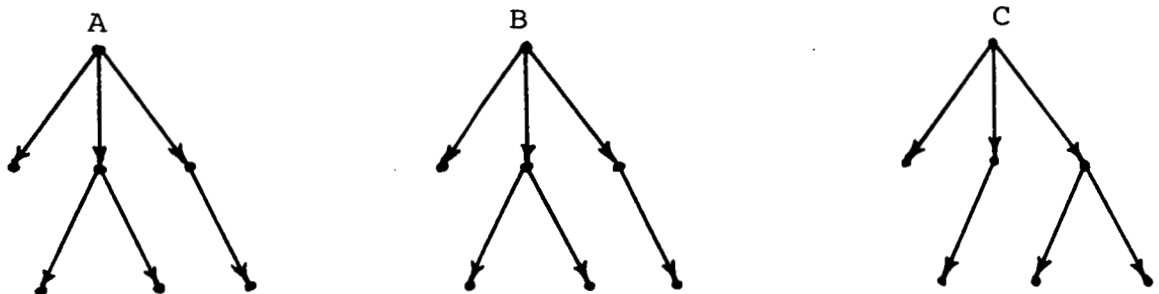

figure 6

Clearly A and B are equivalent, but is A equivalent to C? All
we have done is to interchange the sequence by which two of the
branches descend from the precedent node. In some problems this
difference might be significant, in others not. Therefore the
following terminology will be used: A is isomorphic to C, and
A is sequence isomorphic to B. More formally we have:

Definition: Two trees T_1 and T_2 are called isomorphic if there
exists a 1-1 mapping φ: $T_1 \rightarrow T_2$ such that if node a is the precedent
of node b in T_1, then $\varphi(a)$ is the precedent of $\varphi(b)$ in T_2.

Definition: Two trees T_1 and T_2 are called sequence isomorphic
if they are isomorphic, and if the mapping φ also preserves the
sequence with which the nodes descend when represented as a
list structure.

Definition: Two trees T_1 and T_2 are called equal (sequence
equal) if they are isomorphic (sequence isomorphic) and the
data associated with corresponding nodes is identical.

Using this set of definitions we have the two following
functions for determining tree equivalence.

FUNCTION TISOM (A, B, ISW) - compares the two trees A and B
    for isomorphism if ISW = 0 (or for sequence isomorphism
    if ISW = 1). It returns zero as the function value if
    they are isomorphic, non-zero if not.

FUNCTION TEQUAL (A, B, ISW) - compares the two trees A and B
    for equality if ISW = 0, (or for sequence equality if
    ISW = 1). It returns zero as the function value if they
   are equal, non-zero otherwise

V. GRAPHIC REPRESENTATION OF TREES

It is often useful to be able to graphically represent a
data tree with pencil and paper. The following conventions will
insure that the pencil sketch faithfully mirrors the internal
structure in the computer.

1) Let the root be at the top of the sketch, and its antecedents
    on levels below.

2)   Every time a new node (or tree) is attached to a given node, draw this new structure to the right of all current antecedents from the given node.

3)   The TWIG and DEPTH routines will return from left to right.

4)   All searches or descents in the structure will take the left most path at any possible branch.

For example: given
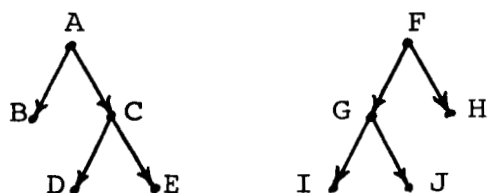
CALL ATTACH (A, F) would yield:
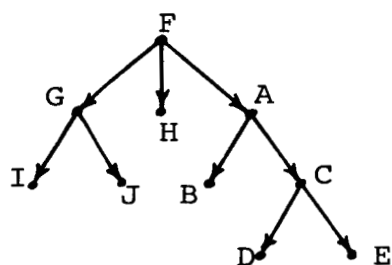
and the two statements

                    CALL DEPTH (F, 2, N, ARRAY)
                    CALL TWIGS (F, N, ARRAY)
will yield respectively
                    I, J, B, C
and

                    I, J, H, B, D, E.

VI.   INPUT AND OUTPUT

     With the exception of the PRTREE routine, no provision is made for the input or output of tree structures. If the user wishes to read or write an individual tree, he will in general want to format his data and its structure in a way that is meaningful to him. He can then use a combination of TREETRAN and standard FORTRAN input-output routines.

VII. EXAMPLE OF A PROBLEM USING TREE STRUCTURED DATA

The syntax of language may be regarded in a very natural way as a tree structure, where a part of speech (say a prepositional phrase) is composed of distinct parts (preposition, and object of the prepositional phrase, together with adjective modifiers, if any), and this part of speech is a part of a still higher level structure. The root node of such a tree may be labeled "the sentence", while the twigs consist of the specific words that make up the sentence.

For example, the sentence "the hungry dog eagerly ate the meat in the white dish", might be represented by the tree in figure 7.

A program to generate syntactically correct sentences was written in TREETRAN using this idea. Suppose we were to ask the computer to write a declarative sentence. It would begin with a tree called "sentence" which syntactically must have branches to nodes called "subject part" and "verb part" Now the program examines the twigs of this tree and discovers that the twig "subject part" is not a word; it must be further defined. Here the program must pause. The syntax permits various types of subjects. All the program can do is present the various syntactically permissible structures and figuratively ask "which one do you want?"

This is the semantic question "what meaning is this sentence intended to convey". It might be answered by having the computer communicate directly with the user and let him type in the correct choice; or might be answered by calling a decision routine which examines a set of data about which the sentence purports to be a meaningful statement. In the current program the computer merely generates a random number which then answers the question, although often in a semantically meaningless way.

Perhaps we decide that the "subject part" should consist of a "singular noun" preceded by a "definite article" and adjective modifier"; this tree structure is then attached to a larger one at the node "subject part". On the next cycle the program will again examine all twigs of the sentence tree; it will encounter

Sentence

subject part — verb Part

definite article → THE

adjective modifier → HUNGRY

noun → DOG

verb phrase

object of verb

adverb modifier → EAGERLY

verb → ATE

definite article → THE

noun → MEAT

prepositional modifier

preposition → IN

object of preposition

definite article → THE

adjective modifier → WHITE

noun → DISH
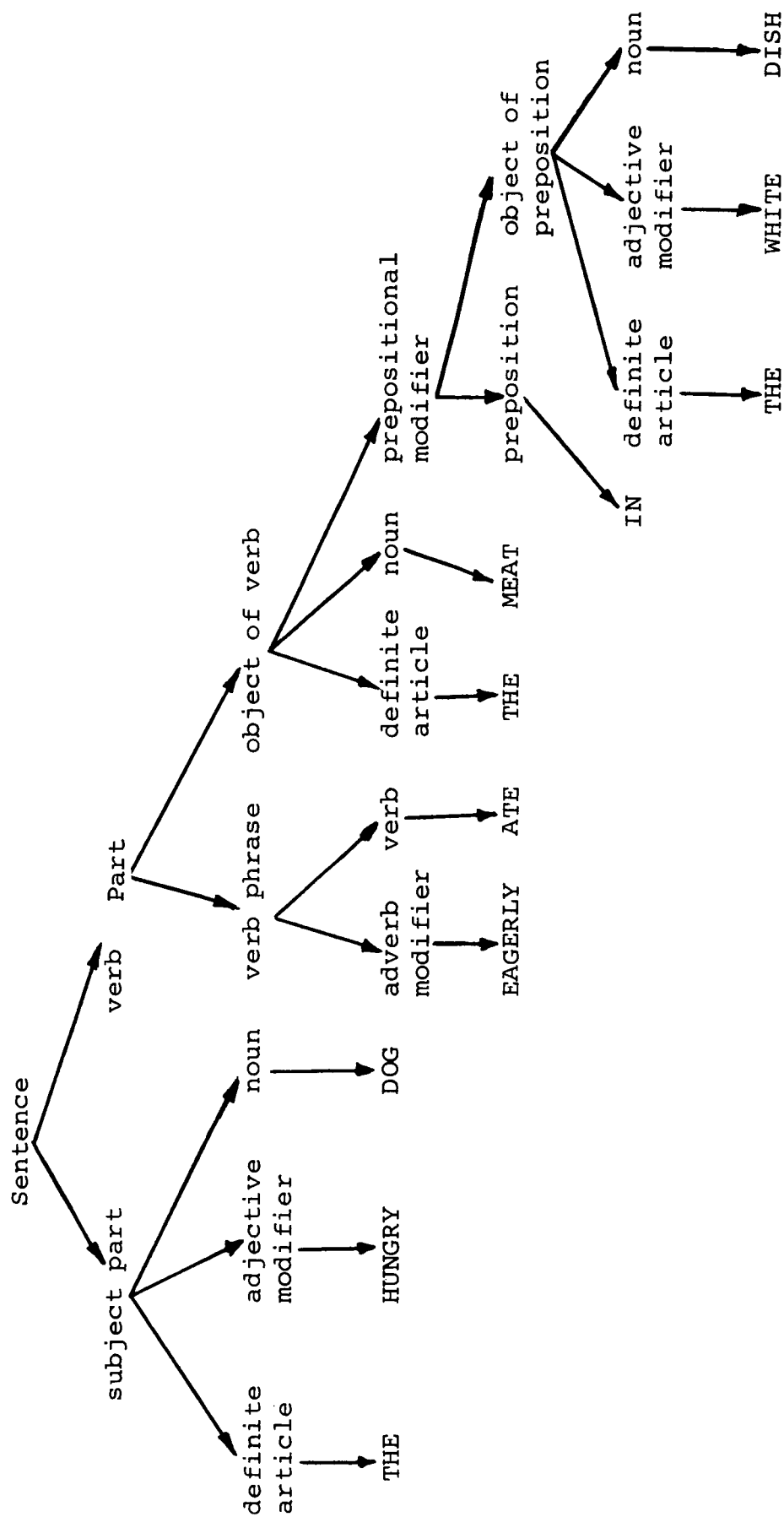
figure 7

FLOW CHART FOR SENTENCE GENERATION
FROM SYNTAX TABLES
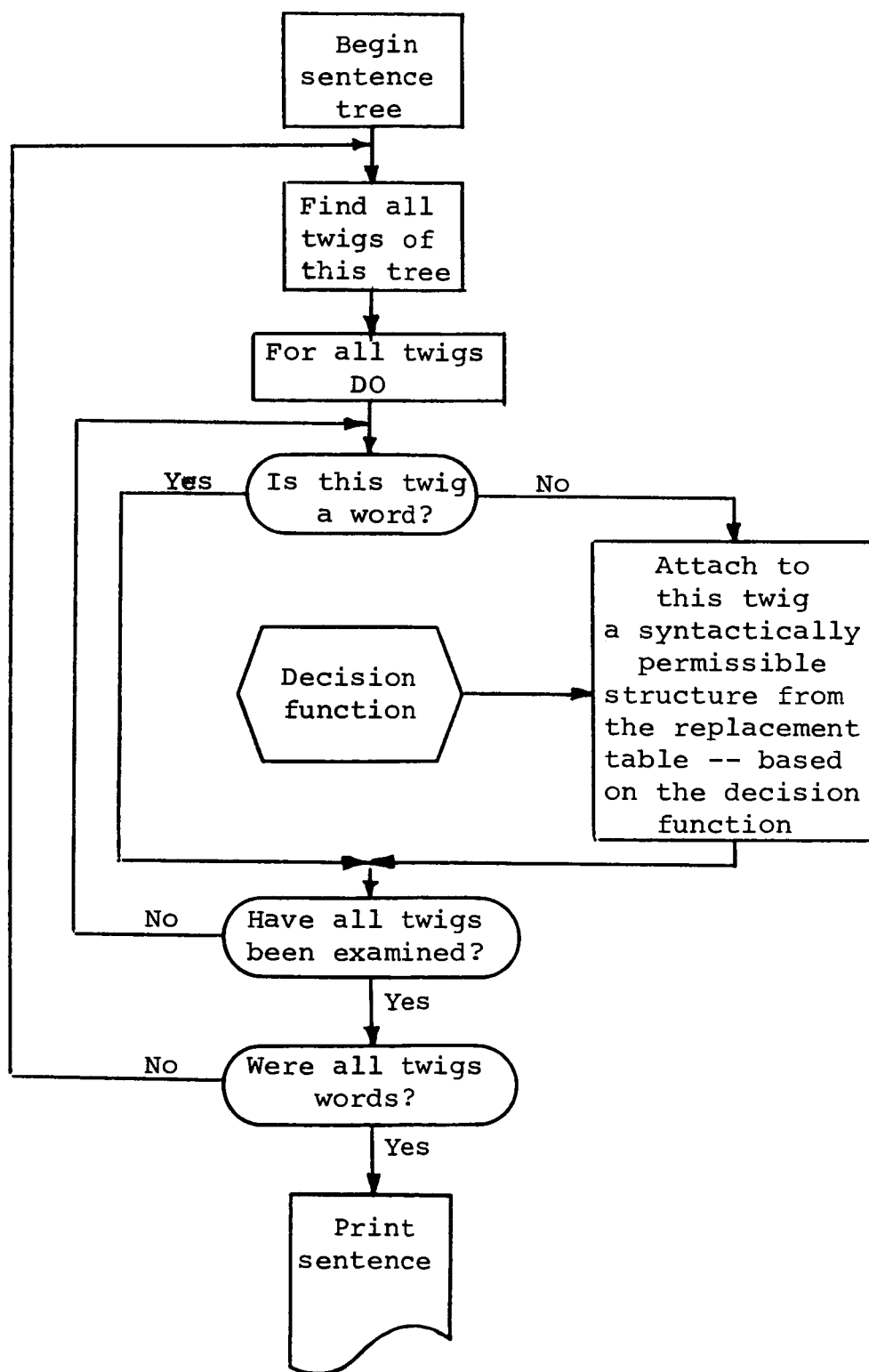
figure 8

```
C
C       GENERATE N SENTENCES
C
        DO 1000 M=1,NSENT
        SENT=COPY(PARTS(1,2))
C
C       FIND ALL TWIGS OF THE SENTENCE TREE
C
  500 CALL TWIGS (SENT,NTWIGS,TWIG)
C
C       ARE ANY OF THE TWIGS STRUCTURAL PARTS OF THE SENCENCE (IE. NOT
C       WORDS OR PUNCTUATION).  IF SO ATTACH MORE STRUCTURE TO THESE
C       TWIGS
C
        ISW=0
        DO 503 I=1,NTWIGS
C
C       CHECK WHETHER EACH TWIG IS IN THE REPLACEMENT TABLE
C       IF NOT IT IS A WORD.
C
        PART=DATA(TWIG(I),FLAG)
        DO 501 J=1,NRT
        IF(PART.EQ.TABLE(J,1)) GO TO 502
  501 CONTINUE
        GO TO 503
C
C       YES IT IS.  GENERATE A RANDOM INDEX .GE. 3 AND USE IT TO
C       PICK A TREE FROM THE REPLACEMENT TABLE9
C
  502 ISW=1
        CHOICE=TABLE(J,2)
        INDEX=INT(RNG(1)*CHOICE+3.0)
C
C       COPY THE CHOSEN TREE AND ATTACH TO THIS TWIG.
C
        X=COPY(TABLE(J,INDEX))
        CALL ATTACH (X,TWIG(I))
  503 CONTINUE
C
C       WERE ALL TWIGS WORDS
C
        IF(ISW) 500,504,500
C
C       YES,  WE ARE DONE, PRINT THE SENTENCE
  504 DO 505 K=1,NTWIGS
        TWIG(K)=DATA(TWIG(K),FLAG)
  505 CONTINUE
        CALL PRTREE (SENT,2)
        WRITE(6,5) (TWIG(K),K=1,NTWIGS)
        CALL IRALST(SENT)
```

figure 9

the twig "definite article" and again say "these are the syntactically permissible definite articles in my vocabulary, which one do you want?" And so on.

Figure 8 shows the flow chart for the syntactic sentence generator, and figure 9 shows the coding of the generation section. Sentences of varying complexity were generated and printed, together with their corresponding syntax trees, every .02 minutes. Some typical generated sentences include: "THE LEAN CATS PUSHED THE WHITE MICE", THIS BIG AND HUNGRY DOG LIKED THIS BLACK MAN." or "ONE WOMAN, THAT RARELY PUSHES THIS SLOW MOUSE, LIKES THE BIG CATS".

VIII. PROGRAMMING LIMITATIONS AND CONSIDERATIONS

In the tree isomorphism and equality routines there exists a limitation which is unlikely to be encountered, specifically no single node of the tree may have more than 20 antecedents. In addition the tree equality routine only compares the first piece of data (through DATA) at each node.

PRTREE will unambiguously display only trees of depth 13 or less.

In addition the programmer must be aware of some of the implications in creating trees of trees. When a tree is stored as data at the node of a higher level tree only its name is stored as datum. Therefore:

1) No upward linkage is provided. Thus while it is is possible to descend from higher levels, it is impossible to ascend in the structure (unless the programmer himself provides the upward link such as putting the name of the higher level node in the data list of the root node of the lower tree, or into auxillary storage).

2) The tree isomorphism and equality routines will check on one level only; and in the case of equality corresponding nodes must have exactly the same tree (including name of root) stored as data.

3) Care must be taken in creating recursive loops; eg. having

ROOT NODE . . . OF TREE STRUCTURE

```
S
  NODE
  DEC
       NODE
       SSUB
            NODE
            SSUBP
                 NODE
                 ADJPSN
                      NODE
                      SDART
                           NODE
                           THIS
                      NODE
                      ADJMOD
                           NODE
                           CONADJ
                                NODE
                                ADJ
                                     NODE
                                     BIG
                                NODE
                                CONJ
                                     NODE
                                     AND
                                NODE
                                ADJ
                                     NODE
                                     HUNGRY
                      NODE
                      SNOUN
                           NODE
                           DOG
            NODE
            SVERBP
                 NODE
                 STVP
                      NODE
                      STVBP
                           NODE
                           STVERB
                                NODE
                                TVPST
                                     NODE
                                     LIKED
                      NODE
                      OBJP
                           NODE
                           ADJPSN
                                NODE
                                SDART
                                     NODE
                                     THIS
                                NODE
                                ADJMOD
                                     NODE
                                     ADJ
                                          NODE
                                          BLACK
                                NODE
                                SNOUN
                                     NODE
                                     MAN
            NODE
```
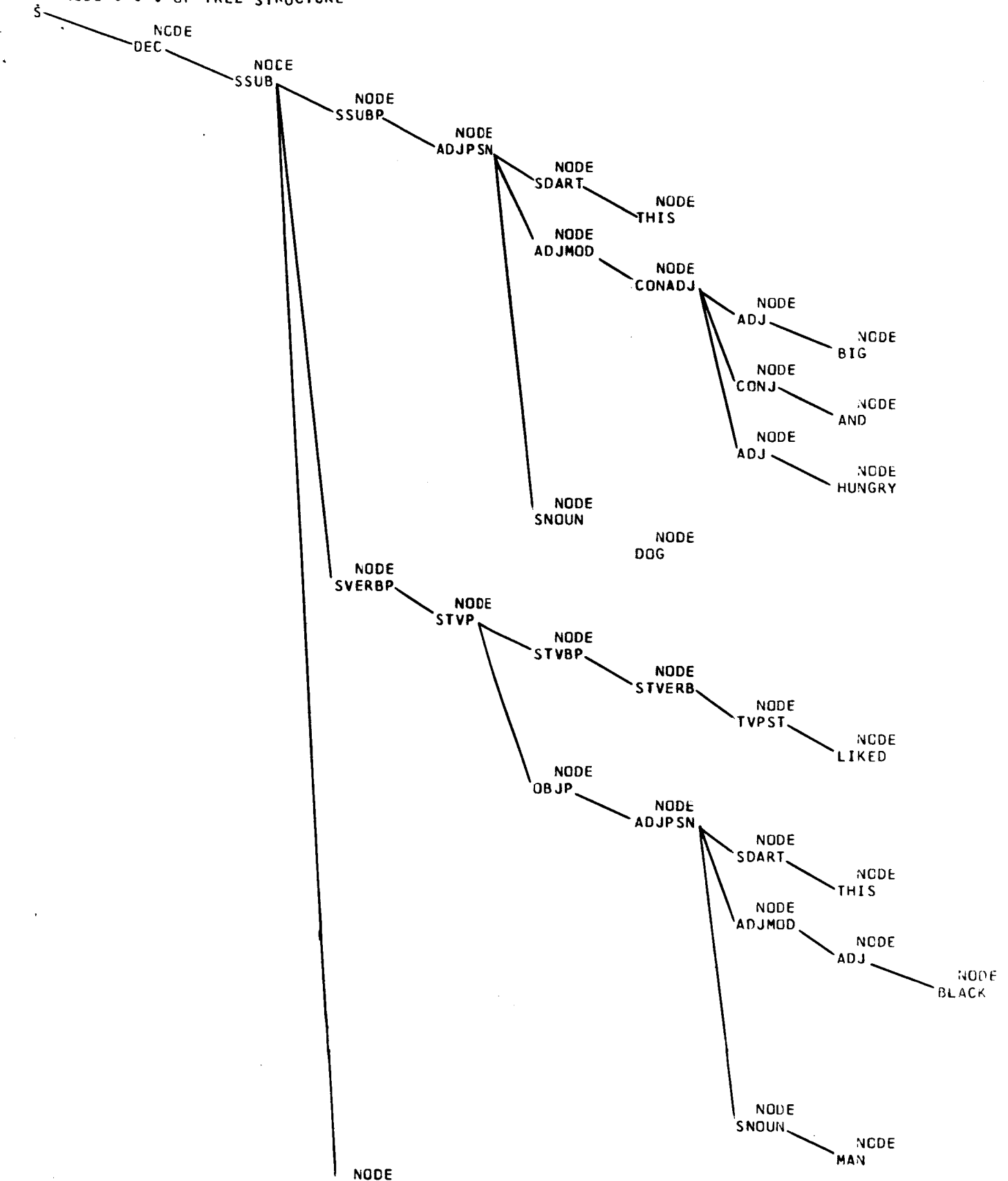
END TREE

THIS BIG AND HUNGRY DOG LIKED THIS BLACK MAN .

one of the descendant nodes of a tree   point to itself (root
of the tree of which it is a part) as data.   This is permissible
but dangerous.

    4) A single tree structure may be provided as data at sev-
eral nodes of a higher level tree.   This permits multiple access
to the root of a given tree, but requires care in the case of
erasure.   A lower level data tree should not be erased unless
all references to it have been deleted.

APPENDIX A

TREETRAN Subroutines

Calling sequence	Function
ADD (A, DATUM)	Adds the word of datum to any data already stored at node A.
DEPTH (A, L, N, ARRAY)	Delivers the names of all nodes in the tree of depth L below the root A, as the first n words of ARRAY. N indicates total number of nodes found at this depth (and may be zero). ARRAY must be dimensioned by the user.
DETACH (A)	Node A (and corresponding tree) is detached from any structure it may be in.
DLIST (A, N, ARRAY)	Delivers the entire list of data stored at node A as the first n words of ARRAY. N indicates total words of data found at A. ARRAY must be sufficiently dimensioned in the calling program.
INITAS (SPACE, n)	A SLIP routine which initializes the list structure. SPACE is an array which must have previously been dimensioned SPACE (n).
IRALST (A)	Destroys the entire tree with root A, and returns the cells to working memory.
PRTREE (A, I)	Prints off line the tree with root A. Variable spacing represents the depth of each node below A. The contents of each node is printed under the following formats:

Calling sequence	Function
	$I = 1$ Integer (I8) $= 2$ Alphabetic (A6) $= 3$ Floating point (E11.4) $= 4$ Octal (O12) All data at the node is printed.
REMOVE (A, DATUM)	Removes the specified word of datum from the list of data at node A.
TWIGS (A, N, ARRAY)	Searches the entire tree structure determined by the root A and puts the name of each twig (node without descendents) into the first n words of ARRAY. N denotes total number of twigs found. ARRAY must be sufficiently dimensioned by calling program.

Table of TREETRAN functions

	Execution	Delivers as function value
ATTACH (A, B)	Attaches the tree structure with antecedent of node B. If A is not a root, it is detached from its current tree structure, a diagonistic is printed, and then A is attached to B.	Name of precedent node (i.e., B)
COPY (A)	Makes an exact copy of the tree with root node A, including structure and stored data.	Name of the root of the copy tree.
CREATE (N, ARG1, ARG2)	Creates a node and stores the n(=0, 1, or 2) words of data (ARG1, ARG2) at the node.	Name of the created node.
DATA (A,FLAG) IDATA (A,FLAG)	Delivers a single piece of datum that was stored at node A. If several words of data are stored, it returns the first. The two entries DATA and IDATA merely facilitate the FORTRAN fixed-floating point convention.	Single word of datum. If none is found, a very large positive value is returned, and FLAG is set to TRUE."
NAMTST (DATUM)	Checks whether the datum is the name of a subtree.	0 if DATUM names a subtree. -1 otherwise.

A-3

PREC (A)

Finds the precedent node of node A.

Name of precedent node; or zero if A is a root.

TISOM (A, B, ISW)

Compares trees A and B for a structural isomorphism where:

ISW = 0, isomorphism can be any mapping that preserves descendence relation

ISW = 1, the mapping must also preserve sequence of descendent nodes. (i.e., structures must be identical as lists)

0 if trees are isomorphic 1.0 if trees are not isomorphic.

TEQUAL (A, B, ISW)

Compares trees A and B for isomorphism as above, but with additional constraint that paired nodes also contain identical data.

0 if equal 1.0 if unequal

APPENDIX B

The tree structures of TREETRAN are internally repre-
sented as list structures. To process these list struc-
tures, the subroutine package uses the symmetric List Pro-
cessing (SLIP) language developed by Dr. J. Weizenbaum of
MIT. The definition of the SLIP language may be found
in his article in the Comm. of the ACM, September 1963.
A more expository description of SLIP is being written
as a CSC Technical Report.

Each node in a tree is stored as a list containing
pointers to each of the descendant nodes (sublists), words
of data associated with the node, and a pointer to the
precedent node. To facilitate some of the routines, the
data associated with a node is always found at the bottom
of its list.

The SLIP list structure corresponding to the tree in
figure 3 may be visualized as follows:

{ Word of memory sym-
bolically represented
in FORTRAN by TREE
(an alias)
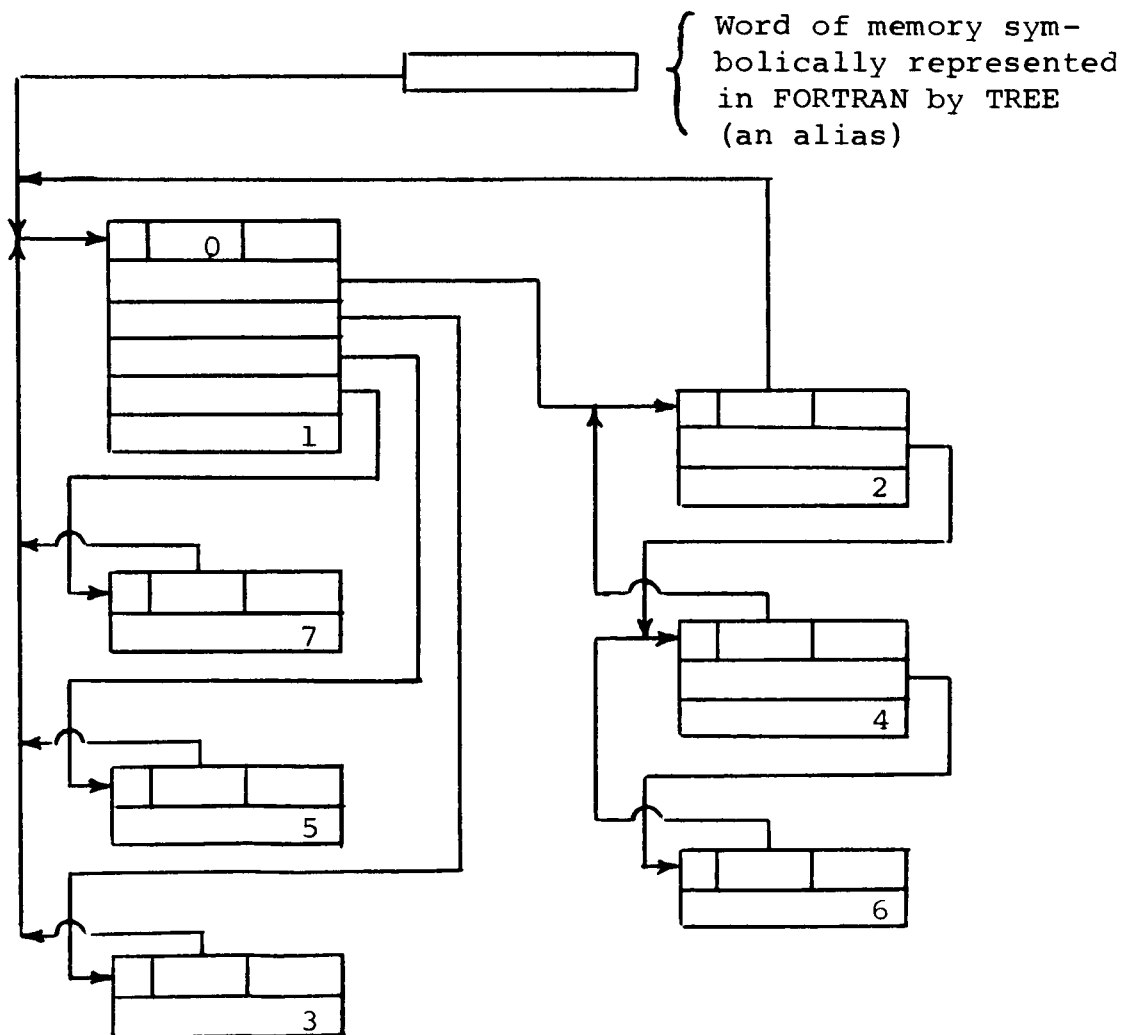
In this diagram each box symbolizes a SLIP cell, which inturn consists of two consecutive words of memory. The string of cells comprising a single list need not be consecutively stored (although this is suggested by the diagram above), but are in general spread throughout that portion of memory reserved for list structures by INITAS. In a SLIP cell the first word contains the linkage information stringing the cells of a single list together; this information is used solely by the SLIP routines and has been omitted in the diagram. The second word of each cell contains various forms of information (pointers, data, etc.) and are manipulated by the TREETRAN subroutine.

We need be concerned with only three types of SLIP cells, each of which is uniquely identified in the SLIP system. First, the header cell is the first cell in any list. It defines the list and contains certain bookkeeping information about the list. The TREETRAN structure uses the left link of the 2nd word of the header in a non-standard way, as the pointer to the precedent node. The only restriction this imposes is that one may not use the attribute lists of SLIP in conjunction with the TREETRAN package.

The second type of SLIP call is a name cell which "points" to (or names) a sublist. We use the cells in the standard way to point to descendant nodes (sublists). The third SLIP cell is a datum cell, which merely contains any word of datum which the user puts there.

A word of caution is in order regarding the SLIP convention of naming (or pointing to) lists. In SLIP a cell that points to or names a list has a unique format in which the machine address of the list being pointed to (actually its header) is repeated in two fields of the word. In addition, if the cell is a name cell in a list it is given an identifying tag of 1 (versus 0 for a datum cell). However, when one constructs a tree of trees and stores the name of a tree as datum at a node, the system stores a regular name cell, but without the identifying tag, as a datum cell.

Now the NAMTST routine determines whether a cell names a sublist by checking only for the repeated address construction.* Thus we have two ways of determining whether a cell points to a sublist; by its identifying tag and by NAMTST. SLIP subroutines use both of them. Thus, while there is no confusion in the present package, care must be taken that modification of the system by the addition of more SLIP routines does not destroy the distinction between pointers that function as name cells (pointing to descendant nodes of a given tree) and pointers that function as datum cells (pointing to lower level tree structures being treated as data structures).

In the TREETRAN subroutines frequent reference is made to the SLIP subroutines LRDROV, ADVSNR, and ADVLER. These employ a different kind of SLIP list. Briefly if a routine must descend into a list structure to perform some function, it must keep track of where it is in the structure and how it got there. One of the easiest methods of keeping track of this information is to create a second auxiliary list. SLIP provides such a system and calls these auxiliary working lists, READER lists. LRDROV creates these temporary working lists and advance functions such as ADVSNR and ADVLER use them. For a thorough description of READER lists and their function the user is refered to Weizenbaum's article in the Communications or the CSC Technical Report.

The following SLIP routines are included in the TREETRAN package.

ID	DELETE
LNKL	INITAS
LNKR	IRALST
CONT	LCNTR

---

* It also verifies that the word being pointed to is really a header cell and the repeated address construction is not just a freakish coincidence.

MADOV	LIST
SETDIR	LISTMT
SETIND	LOCT
STRDIR	LOFRDR
STRIND	LPNTR
ADVLER	LRDROV
ADVLNR	LVLRV1
ADVLWR	MTLIST
ADVLR	NAMSTS
ADVSER	NUCELL
ADVSNR	NXTLFT
ADVSWR	NXTRGT
ADVSR	RCELL
	REED

LISTINGS OF TREETRAN ROUTINES

1. Routines that create and structure trees; store and retrieve
   data stored at nodes:

   a. CREATE
   b. ATTACH
   c. DETACH
   d. DATA
   e. DLIST
   f. ADD
   g. REMOVE
   h. COPY

2. Output routines

   a. PRTREE

3. Manipulative routines that deliver the names of nodes with
   specific properties

   a. DEPTH
   b. TWIGS
   c. PREC

4. Equality and isomorphism routines

   a. TEQUAL
   b. TISOM
   c. TRACE
   d. ORDER
   e. PERM

```
$IBFTC CREATX
      INTEGER FUNCTION CREATE (N,ARG1,ARG2)
C     THIS FUNCTION CREATES A NODE (ACTUALLY A LIST) WITH
C     N WORDS OF ASSOCIATED DATA, IE. ARG1 AND ARG2. N MAY EQUAL
C     0,1, OR 2.  THERE NEED ONLY BE AS MANY DATA ARGUMENTS AS
C     INDICATED BY N.
      L=LIST(9)
      IF (N.EQ.0) GO TO 1
      CALL ADD (L,ARG1)
      IF (N.EQ.1) GO TO 1
      CALL ADD (L,ARG2)
    1 CREATE=L
      RETURN
      END




$IBFTC ATTACX
      REAL FUNCTION ATTACH (A,B)
C     THIS SUBROUTINE ATTACHES NODE A (AND ANY SUCCEEDING BRAINCHES)
C     IN THE TREE FOLLOWING NODE B.  IF A ALREADY HAS A PRECEEDING
C     NODE, A FLAG IS PRINTED, A IS DETACHED FROM ITS CURRENT
C     PRECEDANT AND B IS MADE ITS PRECEDANT
      INTEGER A
      LINKUP=LNKL(CONT(A+1))
      IF(LINKUP.EQ.0) GO TO 2
      CALL DETACH (A)
    1 FORMAT (12H BEFORE NODE,O13,1X,23HWAS INSERTED AFTER NODE,O13,/4X,
     125HIT WAS DETACHED FROM NODE,O13)
      WRITE (6,1) A,B,LINKUP
    2 LR=LRDROV(B)
      DATUM=ADVLER(LR,FLAG)
      M=NXTLFT(A,LPNTR(LR))
      CALL SETIND (-1,LNKL(B),-1,A+1)
      CALL RCELL (LR)
      ATTACH=B
      RETURN
      END
```

```
$IBFTC DETACX
      SUBROUTINE DETACH (A)
C     THIS SUBROUTINE DETACHS NODE A (AND ANY SUCCEEDING BRANCHES)
C     FROM THE TREE AND CREATES A NEW TREE WITH ROOT A.
      INTEGER R,A,X
      LINKUP=LNKL(CONT(A+1))
C
C     DOES THIS NODE HAVE A PRECEDENT
C
      IF(LINKUP.EQ.0) GO TO 3
C
C     YES, FIND IT.  THEN FIND THE CELL IN THE PRECEDENT LIST POINTING
C     TO NODE A
C
      CALL SETDIR (0,LINKUP,LINKUP,LINKUP)
      R=LRDROV(LINKUP)
    1 FLAG=ADVLR(R,1,1)
      IF(FLAG)5,2,5
    2 IF(LNKR(CONT(LNKL(CONT(R))+1)).NE.LNKR(A)) GO TO 1
C
C     REMOVE NAME OF NODE A FROM PRECEDENT LIST, SET PRECEDENT POINTER
C     AT A TO ZERO
C
      X=LNKL(CONT(R))
      LR=LNKR(CONT(X))
      LL=LNKL(CONT(X))
      CALL SETIND (-1,-1,LR,LL)
      CALL SETIND (I1,LL,I1,LR)
      CALL RCELL (X)
      CALL SETIND (-1,0,-1,A+1)
      CALL RCELL (R)
    3 RETURN
    4 FORMAT (25H NO NAME POINTING TO NODE,O13,1X,31HWAS FOUND IN ITS PR
     1ECEDANT NODE)
  104 FORMAT (1H0,10X,6HDETACH,8X,7HLNKR(A),8X,1HA,12X,6HLINKUP,/21X,
     1          O12,4X,O12,4X,O12)
  105 FORMAT (12X,11HPOINTING AT,O14)
    5 WRITE (6,4) A
      STOP
      END




$IBFTC DLISTX
      SUBROUTINE DLIST (A,N,ARRAY)
C     THIS SUBROUTINE RETRIEVES ALL DATA STORED AT NODE A AND STORES
C     IT IN THE FIRST N WORDS OF ARRAY
C
      DIMENSION ARRAY(1)
      N=0
      LR=LRDROV(A)
  100 X=ADVLER(LR,FLAG)
      IF(FLAG)102,101,102
  101 N=N+1
      ARRAY(N)=X
      GO TO 100
  102 CALL RCELL (LR)
      RETURN
      END
```

```
$IBMAP DATAX    20
        ENTRY    IDATA
        ENTRY    DATA
*    THIS FUNCTION - DATA (A,FLAG) - RETIREVES A
*    SINGLE PIECE OF DATA ASSOCIATED WITH NODE
*    A.  IT HAS TWO ENTRIES TO FACILITATE
*    FIXED-FLOATING PT. CONVENTIONS OF FORTRAN.
*    IF NO VALUE IS FOUND IT RETURNS A
*    LARGE VALUE AND SETS FLAG TO TRUE.
DATA    CLA      =1                      REMEMBER WHICH
        STO      SWTCH                   ENTRY WAS USED
        TRA      *+2
IDATA   STZ      SWTCH
        SXA      EXIT,4
        CLA*     3,4                     GET AND STORE A
        STO      A
        CALL     LRDROV(A)               MAKE A READER
        STO      R
        CALL     ADVLER(R,FLAG)          GET FIRST ELEMENT
        STO      DATUM
        CALL     RCELL(R)                RETURN READER TO LAVS
        CLA      FLAG
        TNZ      EMPTY
        CLA      FALSE          SET FLAG TO FALSE
        STO*     4,4
        CLA      DATUM          RETURN WITH DATA
        TRA      EXIT                 IN ACC.
EMPTY   CLA      TRUE           SET FLAG TO TRUE
        STO*     4,4
        CLA      SWTCH                   NO DATA AT THIS
        TZE      FXD                     NODE
        CLA      LARGE
        TRA      EXIT
FXD     CLA      BIG
EXIT    AXT      **,4
        TRA      1,4
SWTCH   PZE
R       PZE
FLAG    PZE
DATUM   PZE
A       PZE
LARGE   DEC      9999999999.
BIG     OCT      7777777777
FALSE   PZE
TRUE    OCT      777777777777
        END
```

```
$IBFTC ADDX
      SUBROUTINE ADD (A,DATUM)
C     THIS SUBROUTINE ADDS THE DATUM TO THE SET OF DATA ALREADY
C     STORED AT NODE A
C
      IL=NUCELL(Z)
    . LL=LNKL(CONT(A))
      CALL SETIND (-1,-1,IL,LL)
      CALL SETIND (-1,IL,-1,A)
      CALL SETIND (0,LL,A,IL)
      CALL STRIND (DATUM,IL+1)
      RETURN
      END




$IBFTC REMOVX
      SUBROUTINE REMOVE (A,DATUM)
C     THIS SUBROUTINE REMOVES THE DATUM FROM THE LIST OF DATA STORED
C     AT NODE A.
C
      LR=LRDROV(A)
  100 X=ADVLER(LR,FLAG)
      IF(FLAG)102,101,102
  101 IF(X.NE.DATUM) GO TO 100
      CALL DELETE (LPNTR(LR))
      GO TO 100
  102 CALL RCELL (LR)
      RETURN
      END




$IBFTC COPYX
      FUNCTION COPY (A)
C     THIS FUNCTION CREATES A TREE THAT IS AN EXACT COPY (BOTH
C     STRUCTURALLY AND IN DATA AT NODES) AS THE TREE WITH ROOT A.
C     IT RETURNS THE NAME OF THE ROOT OF THE COPY AS FUNCTION VALUE
C
      DIMENSION EQUIV(30,2)
      REAL LOFRDR
C
C     COPY THE ROOT NODE
C
      K=1
      EQUIV(K,2)=CREATE (0)
      EQUIV(K,1)=A
      LRTEMP=LRDROV(A)
  100 X=ADVLER(LRTEMP,FLAG)
      IF(FLAG)102,101,102
  101 CALL ADD (EQUIV(K,2),X)
      GO TO 100
  102 CALL RCELL (LRTEMP)
C
C     CREATE READER TO DESCEND TREE A, AND START DESCEND
C
      LR=LRDROV(A)
  150 Y=ADVSNR(LR,FLAG)
```

```
      IF(FLAG)250,160,250
C
C     WE HAVE ENCOUNTERED A NEW NODE , COPY IT AND LINK IT UP
C
  160 K=K+1
      EQUIV(K,2)=CREATE(0)
      EQUIV(K,1)=Y
      LRTEMP=LRDROV(Y)
  200 X=ADVLER(LRTEMP,FLAG)
      IF(FLAG)202,201,202
  201 CALL ADD (EQUIV(K,2),X)
      GO TO 200
  202 CALL RCELL (LRTEMP)
C
C     WHAT NODE ARE AT IN THE ORIGINAL TREE. FIND EQUIVILENT IN
C     COPY TREE AND LINKUP.
C
      SEARCH=LOFRDR(LR)
      DO 205 I=1,K
      IF(SEARCH.EQ.EQUIV(I,1)) GO TO 206
  205 CONTINUE
      GO TO 300
  206 CALL ATTACH (EQUIV(K,2),EQUIV(I,2))
      GO TO 150
C
C     ALL NODES HAVE BEEN FOUND, WE ARE BACK TO A
C
  250 CALL RCELL (LR)
      COPY=EQUIV(1,2)
      RETURN
C
C     ERROR ROUTINE
C
    1 FORMAT (1H0,33HPRECEDENT NODE NOT IN EQUIV TABLE)
  300 WRITE (6,1)
      STOP
      END
```

```
$IBFTC PRTREX
      SUBROUTINE PRTREE (A,J)
C     THIS SUBROUTINE PRINTS OUT ENTIRE SUBTREE THAT HAS A AS ITS ROOT.
C     J CONTROLS FORMAT AS FOLLOWS
C             J=1   INTEGER (I8)
C              =2   ALPHABETIC (A6)
C              =3   FLOATING PT. (E11.4)
C              =4   OCTAL (O12)
      DIMENSION FMT1(2),FMT2(2),FMT3(2),FMT4(2),FMT5(3),FMT6(3),VSP(13)
      DIMENSION FMT7(4),FMT8(4),DTREE(30)
      EQUIVALENCE (DATUM,IDATUM)
      DATA (FMT1(I),I=1,2)/6H(03X, ,6HI8)    /
      DATA (FMT2(I),I=1,2)/6H(03X, ,6HA6)    /
      DATA (FMT3(I),I=1,2)/6H(03X, ,6HE11.4)/
      DATA (FMT4(I),I=1,2)/6H(03X, ,6HO12)   /
      DATA (FMT5(I),I=1,3)/6H(03X, ,6H6H  NO,6HDE)    /
      DATA (FMT6(I),I=1,3)/6H(03X, ,6H7HNO D,6HATA)   /
      DATA (FMT8(I),I=1,4)/6H(03X, ,6H6HLABE,6HL  ,I2,,6H1H))   /
      DATA (FMT7(I),I=1,4)/6H(03X, ,6H10H(DA,6HTA TRE,6HE)      /
      DATA (VSP(I),I=1,5)/6H(03X, ,6H(13X, ,6H(23X, ,6H(33X, ,6H(43X, /
      DATA (VSP(I),I=6,10)/6H(53X, ,6H(63X, ,6H(73X, ,6H(83X, ,6H(93X, /
      DATA (VSP(I),I=11,13)/6H(103X,,6H(113X,,6H(123X,/
    1 FORMAT (34H1ROOT NODE . . . OF TREE STRUCTURE)
    2 FORMAT (9H END TREE)
    3 FORMAT (37H1ROOT NODE . . . OF DATA TREE, LABEL ,I2)
C
C     MAKE READER FOR TREE, INITIALIZE, AND PRINT ROOT NODE
C
      WRITE (6,1)
      LR=LRDROV(A)
      NDPT=0
      NDT=0
   50 LEVEL=0
      ISW=0
      ASSIGN 202 TO L
      GO TO 300
C
C     WE HAVE PRINTED ALL THE DATA AT THIS NODE, ADVANCE STRUCTURALLY
C     TO THE NEXT NODE
C
  100 Y=ADVSNR(LR,FLAG)
      IF(FLAG)103,101,103
C
C     ARE WE GOING UP OR DOWN THE TREE.
C
  101 IF(LEVEL - LCNTR(LR))106,200,102
  102 LEVEL=LEVEL-1
      ASSIGN 101 TO L
      GO TO 300
C
C     DONE WITH THIS NODE
C
  103 IF(LEVEL - LCNTR(LR))106,105,104
  104 LEVEL=LEVEL-1
      ASSIGN 103 TO L
      GO TO 300
  105 WRITE (6,2)
      CALL RCELL (LR)
C
C     HAVE ALL DATA TREES BEEN PRINTED, IF NOT PRINT NEXT IN LIST
```

```
C
  106 IF(NDPT.GE.NDT) RETURN
      NDPT=NDPT+1
      WRITE (6,3) NDPT
      LR=LRDROV(DTREE(NDPT))
      GO TO 50
C
C     PRINT HEADING AND DATA AT THIS NODE
C
  200 LEVEL=LEVEL+1
      ASSIGN 201 TO L
      GO TO 300
  201 WRITE (6,FMT5)
      ISW=0
      IF(LISTMT(Y).EQ.0) GO TO 209
      DATUM=ADVSWR(LR,FLAG)
      IF(ID(CONT(LPNTR(LR))).NE.1) GO TO 203
  202 DATUM=ADVLER(LR,FLAG)
      IF(FLAG)208,203,208
  203 ISW=1
C
C     IS THIS DATUM A DATA TREE
C
      IF(NAMTST(DATUM))2036,2031,2036
C
C     YES IT IS. IS IT A DATA  TREE ALREADY FOUND,IF NOT PUT IN LIST
C
 2031 IF(DATUM.EQ.A) GO TO 410
      IF(NDT.EQ.0) GO TO 2033
      DO 2032 K=1,NDT
      IF(DATUM.EQ.DTREE(K)) GO TO 2034
 2032 CONTINUE
 2033 IF (NDT.GE.30) GO TO 400
      NDT=NDT+1
      DTREE(NDT)=DATUM
      LABDT=NDT
      GO TO 2035
 2034 LABDT=K
 2035 WRITE (6,FMT7)
      WRITE (6,FMT8) LABDT
      GO TO 202
 2036 GO TO (204,205,206,207),J
  204 WRITE (6,FMT1) IDATUM
      GO TO 202
  205 WRITE (6,FMT2) DATUM
      GO TO 202
  206 WRITE (6,FMT3) DATUM
      GO TO 202
  207 WRITE (6,FMT4) DATUM
      GO TO 202
  208 IF(ISW)210,209,210
  209 WRITE (6,FMT6)
  210 GO TO 100
C
C     SET UP VARIABLE SPACED FORMATS
C
  300 IF(LEVEL.GE.13) GO TO 301
      FMT1(1)=VSP(LEVEL+1)
```

```
      FMT2(1)=VSP(LEVEL+1)
      FMT3(1)=VSP(LEVEL+1)
      FMT4(1)=VSP(LEVEL+1)
      FMT5(1)=VSP(LEVEL+1)
      FMT6(1)=VSP(LEVEL+1)
      FMT7(1)=VSP(LEVEL+1)
      FMT8(1)=VSP(LEVEL+1)
  301 GO TO L,(101,103,201,202)
C
C     ERROR ROUTINE, FOUND TOO MANY DATA TREES
C
  400 PRINT 401,
      LABDT=99
      GO TO 2035
  401 FORMAT (5X,48HMORE THAN 30 DATA TREES FOUND. LABEL 99 ASSIGNED)
C
C     RECURSIVE REFERENCE TO ROOT
C
  410 PRINT 411,
      LABDT=98
      GO TO 2035
  411 FORMAT (5X,46HRECURSIVE REFERENCE TO ROOT. LABEL 98 ASSIGNED)
      END
```

```
$IBFTC DEPTHX
      SUBROUTINE DEPTH (A,L,NNF,NAMES)
C     THIS SUBROUTINE FINDS ALL NODES THAT ARE OF DEPTH L BELOW
C     NODE A. NNF IS THE NUMBER OF NODES FOUND AT THIS DEPTH
C     AND THE NAME OF EACH FOUND NODE IS STORED IN THE ARRAY
C     NAMES WHICH MUST HAVE BEEN SUFFICIENTLY DIMENSIONED
C     BY THE CALLING PROGRAM
      REAL NAMES(1)
    1 FORMAT (4X,3H101,4X,O12,3X,O12,3X,F3.0,I5)
    2 FORMAT (4X,3H103,4X,O12,3X,O12,3X,F3.0,I5)
    3 FORMAT (1H1,9HSTATEMENT,5X,4HNAME,10X,6HREADER,5X,4HFLAG,6H NODES)
C
C     CREATE READER AND CHECK L
C
      LR=LRDROV(A)
      NNODES=0
      IF(L.EQ.1) GO TO 200
C
C     DESCEND TREE TO DEPTH L-1
C
  101 X=ADVSNR(LR,FLAG)
      IF(FLAG)110,102,110
  102 IF(LCNTR(LR).LT.L-1) GO TO 101
      GO TO 104
C
C     WE ARE AT DEPTH L-1, ALL NAMES AT THIS NODE ARE AT DEPTH L
C
  103 X=ADVLNR(LR,FLAG)
      IF(FLAG)105,104,105
  104 NNODES=NNODES+1
      NAMES(NNODES)=X
      GO TO 103
C
C     ASCEND ONE LEVEL AND CONTINUE SEARCHING
C
  105 LD=LVLRV1(LR)
      X=ADVLWR(LR,FLAG)
      IF(FLAG)107,106,107
  106 IF(NAMTST(X))101,102,101
  107 IF(LCNTR(LR))110,110,105
C
C     ENTIRE SUBTREE HAS BEEN SEARCHED
C
  110 NNF=NNODES
      CALL RCELL(LR)
      RETURN
C
C     IF L=1, ONLY A LINEAR SEARCH OF NODE A IS NECESSARY
C
  200 X=ADVLNR(LR,FLAG)
      IF(FLAG)110,201,110
  201 NNODES=NNODES+1
      NAMES(NNODES)=X
      GO TO 200
      END
```

```
$IBFTC TWIGSX
      SUBROUTINE TWIGS (A,N,ARRAY)
C     THIS SUBROUTINE FINDS ALL TWIGS THAT CAN BE REACHED FROM VERTEX
C     A. N IS THE TOTAL NUMBER FOUND. THE NAMES OF THE TWIGS ARE PUT
C     INTO ARRAY (WHICH MUST BE SUFFICENTLY DIMENSIONED BY THE USER)
      DIMENSION ARRAY(1)
C     CREATE READER AND BEGIN DESCENT
C
      LR=LRDROV(A)
      N=0
      CAND=A
      GO TO 101
  100 CAND=ADVSNR(LR,FLAG)
      IF(FLAG)104,101,104
C
C     IS THE CANDIDATE A TWIG.
C
  101 LRTEMP=LRDROV(CAND)
      X=ADVLNR(LRTEMP,FLAG)
      CALL RCELL(LRTEMP)
      IF(FLAG)102,103,102
C
C     YES, PUT ITS NAME INTO ARRAY
C
  102 N=N+1
      ARRAY(N)=CAND
      GO TO 100
C
C     NO, KEEP SEARCHING
C
  103 GO TO 100
C
C     ALL DONE
C
  104 CALL RCELL (LR)
      RETURN
      END




$IBFTC NPRECX
      INTEGER FUNCTION PREC(A)
C     THIS FUNCTION DELIVERS AS FUNCTIONAL VALUE THE NAME OF THE
C     NODE PRECEEDING NODE A.  IF A IS THE ROOT OF THE TREE
C     NPREC IS SET TO ZERO
      INTEGER A
      LINKUP=LNKL(CONT(A+1))
      IF(LINKUP.EQ.0) GO TO 1
      CALL SETDIR (0,LINKUP,LINKUP,LINKUP)
    1 PREC=LINKUP
      RETURN
      END
```

```
$IBFTC TEQUAX
      FUNCTION TEQUAL (A,B,ISW)
C     THIS FUNCTION COMPARES TREES A AND B. IT RETURNS FUNCTIONAL VALUE
C     ZERO IF THE NODES CONTAIN IDENTICAL DATA AND ARE STRUCTURALLY
C     THE SAME (IN THE FOLLOWING SENSE)
C         ISW=0    ISMORPHIC - THERE EXISTS SOME MATCHING BETWEEN NODES
C         ISW=1    SEQ. ISOMORPHIC - THE TREES DECEND IN THE SAME SEQ.
      LOGICAL FLAG
      I=0
      J=ISW
      CALL TRACE (A,B,I,J,FLAG)
      IF(FLAG) GO TO 100
      TEQUAL=1.0
      RETURN
  100 TEQUAL=0.0
      RETURN
      END




$IBFTC TISOMX
      FUNCTION TISOM (A,B,ISW)
C     THIS FUNCTION STRUCTURALLY COMPARES THE TWO TREES A AND B.
C     IT RETURNS ZERO AS FUNCTIONAL VALUE IF THE TREES ARE ISMORPHIC
C     (IN THE FOLLOWING SENSE)
C         ISW=0    ISOMORPHIC - THERE EXISTS SOME MAPPING OF A ONTO B
C         ISW=1    SEQ. ISOMORPHIC - THE MAPPING PRESERVES NOT ONLY
C                  THE DESCENDENCE RELATION, BUT ALSO THE SEQUENCE WITH
C                  WHICH THE STRUCTURE DESCENDS FROM A GIVEN NODE
C     (NOTE, THERE MAY EXIST SEVERAL ISOMORPHIC MAPPINGS. THE ROUTINE
C     DOES NOT EXHIBIT THE ONE ACTUALLY FOUND)
C
      LOGICAL FLAG
      I=1
      J=ISW
      CALL TRACE (A,B,I,J,FLAG)
      IF(FLAG) GO TO 100
      TISOM=1.0
      RETURN
  100 TISOM=0.0
      RETURN
      END
```

Brief Description of TRACE Subroutine

In order to determine whether an isomorphism exists between two different trees every possible pair of corresponding nodes in the two trees must be checked to see if they preserve the isomorphism property. A preliminary screen which considers a pair of nodes as a possible pair only if they have the same number of antecedants reduces the problem to that of checking a large number of possible pairings instead of an astronomical number.

TRACE begins by pairing the root nodes, then enumerating all possible pairings of their antecedents. The routine then considers each of these pairs in turn as the roots of their respective subtrees and enumerates all possible pairs of their antecedents. This procedure continues until either

1. the possible pairs are all twigs in which case, an isomorphic mapping between nodes along this branch of the trees has been verified. (Remaining branches must be similarily verified.)

or

2. no possible pairing exists in which case another possible pairing at a higher level must be tried. (All possible pairs must be similarily rejected to establish that no isomorphism exists.)

In order that TRACE can keep track of its position in the two tree structures, as well as the possible pairs that remain to be checked, a system of cascading lists has been used.

Data structure of cascading lists
used by TRACE

There are two different types of lists, called X and Y lists which are linked together to form the cascading lists.

Assume that TRACE has tentatively paired two nodes α and α' in trees A and A' respectively. Then on the basis of permutations of the two sets of antecedents $\alpha_1 \alpha_2 \ldots \alpha_n$ and $\alpha'_1, \alpha'_2 \ldots \alpha'_n$ it enumerates several sets of possible pairing of

these antecedents. For each possible pairing it must specify
that node α_1 is paired with node α_4', node α_3 with α_2', etc.
The contents of a Y-list is just such a specification of a single
possible pairing. The X-list contains pointers to each of these
Y-lists. Thus the X-list enumerates all possible pairings of
the antecedent nodes, given the assumption that nodes α and α'
are paired. Now α and α' must have been paired in some higher
level Y-list hence we let the X-list point back to this Y-list,
and similarly we let the set of Y-lists point back to the X-list
which enumerates them. In this manner can both ascend and de-
scend the tree structures while checking them for isomorphism.

This linked chain of lists cascades in the sense that as
TRACE descends the structure it creates X and Y lists, and as
it ascends it erases them. At any given time TRACE must be
using the bottom cell of a specified X or Y-list. This con-
vention defines a number of implicit linkages.

GROSS FLOW CHART OF TRACE

```
                              ┌──────────────────┐
                              │ Take bottom pair │   (300)
                              │ of current Y-list│
                              └──────────────────┘
                                       │
                              ┌──────────────────┐
                              │ Find antecedents │   (302)
                              │   of this pair   │
                              └──────────────────┘
                                       │
            No        ╱──────────────────────────╲      Yes
     ┌────────────────┤  Are the nodes in         ├──────────────┐
     │  (400)         ╲  this pair, twigs?        ╱               │  (350)
     ▼                 ╲──────────────────────────╱               ▼
┌───────────────────┐                                    ┌──────────────┐
│  Make possible    │                                    │ Pop bottom   │
│  pairing (Y) lists│                                    │ of current   │
│  on basis of the  │                                    │   Y-list     │
│number of antecedents│                                  └──────────────┘
└───────────────────┘                                            │
  (3032)    │                                         ┌───────────┘
    ╱───────────────╲    No                           ▼              No
   ╱ Can we make a   ├────────────┐         ╱──────────────────╲ ───────┐
   ╲ non-empty list? ╱            │         ╲  Is Y-list empty? ╱ (351)  │
    ╲───────────────╱   (500)     │          ╲──────────────────╱        │
        │ Yes (414)   ┌────────────────┐              │ Yes              │
        ▼             │  Back up to    │      ╱──────────────────╲  Yes  │
┌──────────────┐      │  X-list and    │      ╲  Is 2nd cell     ├───────│
│  Make new    │      │  pop bottom.   │      ╲  of Y-list =0?   ╱       │
│ X and Y-lists│      │  Erase Y-list  │       ╲──────────────────╱      │
└──────────────┘      └────────────────┘              │ No  (352)        │
                              │                ┌──────────────────┐      │
            No    ╱──────────────╲             │   Back up to     │      │
     ┌───────────┤  Is X-list     │            │    X-list.       │      │
     │           ╲  empty?        ╱            │  Erase Y-list.   │      │
     ▼            ╲──────────────╱             │   Back up to     │      │
┌──────────┐           │ Yes                   │   new Y-list     │      │
│Get a new │ (501)     ▼                        │and pop bottom.  │      │
│ Y-list   │      ┌──────────────┐             │  Erase X-list.   │      │
└──────────┘      │  Back up to  │             └──────────────────┘      │
                  │   Y-list     │                      │                │
                  │ Erase X-list │                      │                │
                  └──────────────┘                      │                │
                         │                              │                │
            ╱───────────────╲                           │                │
            ╲  Is 2nd cell   ├──────┐                   │                │
            ╲  of Y-list =0? ╱  No  │                   │                │
             ╲───────────────╱      │                   │                │
                    │ Yes     (601) │                   │                │
Trees are not      ╱────╲           │             ╱────╲   (600)         │
equal or isomor. ─ ─►│Return│       │             │Return│ ◄─────────────┘
                     │ ≠0   │       │             │ =0   │
                      ╲────╱                        ╲────╱
                                                       ▲
                                                Trees are equal
                                                 or isomorphic
```

Schematic structure of X and Y-Lists at one point in
the comparison of trees a and a'

a

b   c

d   e   f   g

h   i   j

a'

b'   c'

d'   e'   f'   g'

h'   i'   j'   k'

--- denotes implicit
          pointer
0 denotes backwards
          pointer
1 denotes forwards
          pointer

Y-List

0		0
1	a	a'

X-List

0
1
1

Y-Lists

0		
1	b	c'
1	c	b'

0		
1	b	b'
1	c	c'

X-List

0
1

Y-List

0		
1	f	f'
1	g	g'
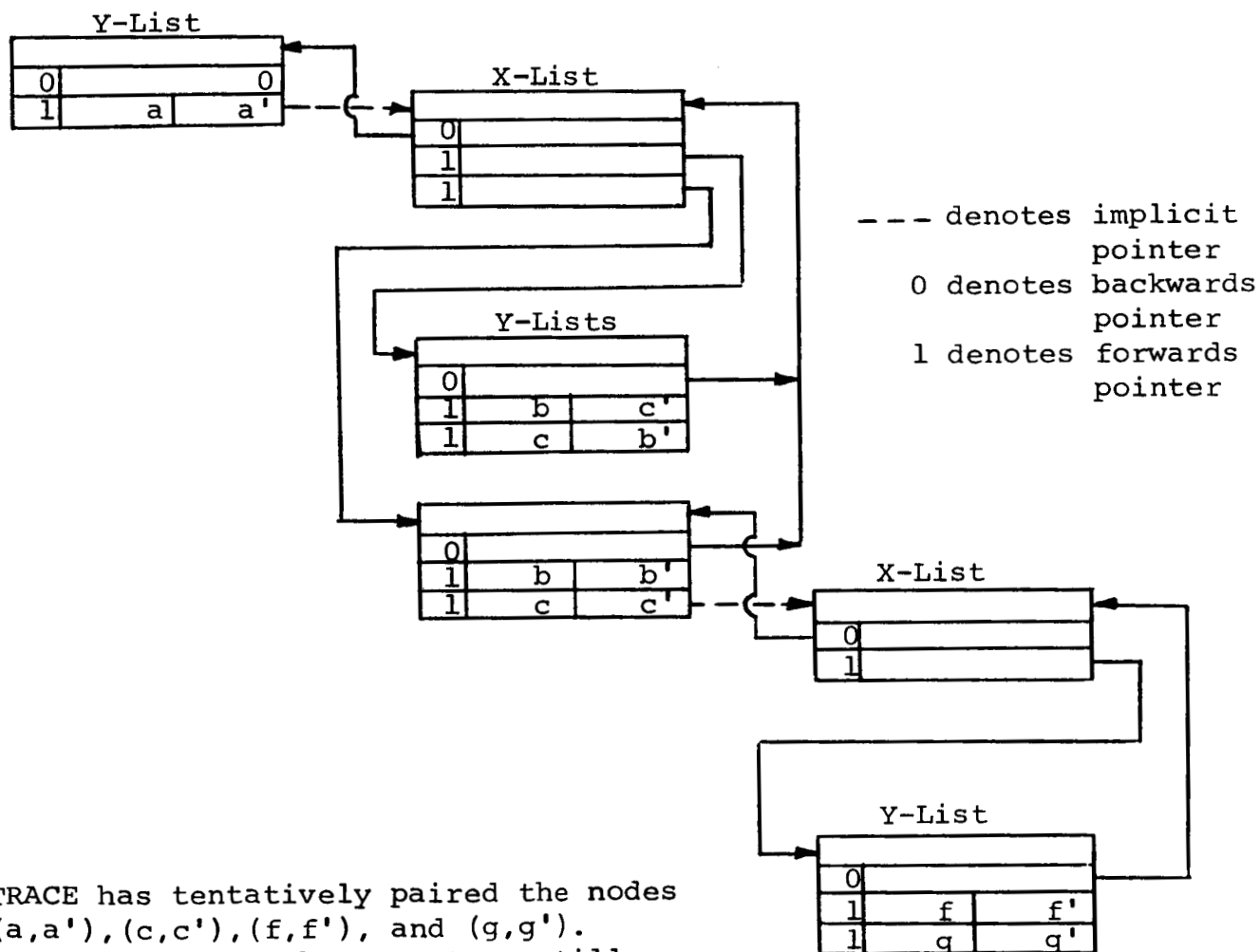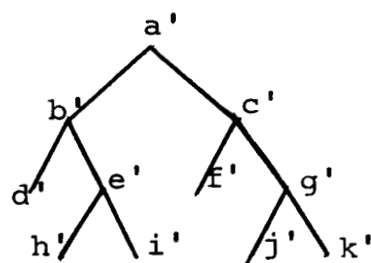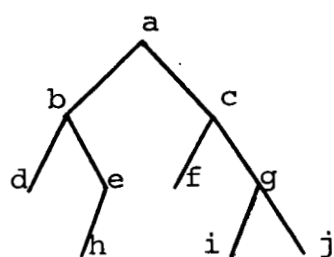
TRACE has tentatively paired the nodes
(a,a'),(c,c'),(f,f'), and (g,g').
It must descend the structure still
further to verify that the pairings (f,f')
and (g,g') are acceptable in that they are/or lead to twigs.
It must then backup and try to verify the branch defined by
(b,b') in the second Y-list. When this fails, it will back up
to the first x-list, find the next Y-list (containing (b,c')
and (c,b')) and try these pairs.

```
$IBFTC TRACEX
      SUBROUTINE TRACE (A,B,SW,SW2,FLAG)
C     THIS SUBROUTINE TRACES THE TREES A AND B TO SEE IF THEY ARE
C     EQUAL (SW = 0) OR ISOMORPHIC (SW = 1).  IF THEY ARE EQUAL OR
C     ISOMORPHIC  FLAG IS SET TO TRUE, IF NOT FALSE
C
      INTEGER X,Y,CELL,LIST1(20,2),LIST2(20,2),KOPY(20),FIRST,P(10),
     1         INDEX(20),DUMPL(20)
      LOGICAL FLAG,LFLAG
C
C     CREATE LISTS X AND Y, BEGIN PAIRING WITH ROOT NODES
C
  200 Y=LIST(9)
      ICELL=NXTLFT (0,Y)
      CALL SETDIR (1,A,B,TEMP)
      ICELL=NXTLFT (TEMP,Y)
      GO TO 300
C
C     FIND DESCENDENTS OF CURRENT POSSIBLE PAIRS
C
C     GET BOTTOM CELL OF CURRENT Y LIST
C
  300 CELL=LNKL(CONT(Y))
      IPAIR=LNKL(CONT(CELL+1))
      CALL SETDIR (0,IPAIR,IPAIR,IPAIR)
      JPAIR=LNKR(CONT(CELL+1))
      CALL SETDIR (0,JPAIR,JPAIR,JPAIR)
      IF(SW)302,301,302
  301 IF(IDATA(JPAIR).NE.IDATA(IPAIR)) GO TO 500
  302 CALL DEPTH(IPAIR,1,N1,LIST1(1,1))
      CALL DEPTH(JPAIR,1,N2,LIST2(1,1))
C
C     ARE THEY BOTH TWIGS
C
      IF(N1.EQ.0) GO TO 350
C
C     CAN WE MAKE A POSSIBLE PAIRING OF THESE TWO LIST OF NODES
C     BASED ON THEIR DESCENDENTS
C
      DO 303 K=1,N1
      CALL DEPTH(LIST1(K,1),1,LIST1(K,2),DUMPL)
      CALL DEPTH(LIST2(K,1),1,LIST2(K,2),DUMPL)
  303 CONTINUE
C                                     .
C     ORDER BOTH,LISTS ON DESCENDING NUMBER OF DESCENDENTS
C
      IF(SW2)3032,3031,3032
 3031 CALL ORDER (LIST1,N1)
      CALL ORDER (LIST2,N1)
 3032 DO 304 L=1,N1
      IF(LIST1(L,2).NE.LIST2(L,2)) GO TO 500
  304 CONTINUE
      GO TO 400
C     YES, MAKE LISTS OF ALL POSSIBLE PAIRINGS OF THE DECENDENTS OF
C     THESE TWO NODES.
C     MAKE A NEW X LIST AND MAKE IT POINT BACK TO CURRENT Y LIST
  400 X=LIST(9)
      CALL SETDIR (0,Y,Y,TEMP)
      ITEMP=NXTLFT (TEMP,X)
C
```

```
C     PARTITION THE LIST OF DESCENDENT NODES INTO SETS WHICH THEMSELVES
C     HAVE EQUAL NUMBERS OF DESCENDENTS
C
      KPART=0
      IF(SW2)406,4001,406
 4001 IF(N2.EQ.1) GO TO 4021
      DO 402 L=2,N2
      IF(LIST2(L,2).EQ.LIST2(L-1,2)) GO TO 402
      KPART=KPART+1
      P(KPART)=L-1
  402 CONTINUE
 4021 KPART=KPART+1
      P(KPART)=N2
C
C     WE MUST TRY ALL POSSIBLE PAIRINGS BASED ON PERMUTATIONS WITHIN
C     THESE SETS.
C
  406 DO 407 L=1,N2
      KOPY(L)=LIST2(L,1)
  407 CONTINUE
      IF(SW2)414,4071,414
 4071 L=1
      MAXL=1
      FLAG=.TRUE.
      IF(P(1).GT.1) GO TO 408
      IF(KPART.EQ.1) GO TO 4073
      DO 4072 LK=2,KPART
      IF((P(LK)-P(LK-1)).GT.1) GO TO 408
 4072 CONTINUE
 4073 LFLAG=.FALSE.
      GO TO 409
  408 LFLAG=.TRUE.
C
C     PERMUTE THE INDICES OF PARTITION SET L
C
  409 IF(L.EQ.1) GO TO 411
  410 N=P(L)-P(L-1)
      FIRST=P(L-1)+1
      GO TO 4111
  411 N=P(1)
      FIRST=1
 4111 IF(N.GT.1) GO TO 412
      IF(LFLAG) GO TO 417
      LFLAG=.TRUE.
  412 CALL PERM (N,INDEX(FIRST),FLAG)
C
C     PERMUTE THE NAMES INTHE COPY LIST AS INDICATED
C
      LAST=FIRST+N-1
      DO 413 K=FIRST,LAST
      KC=INDEX(K)+FIRST-1
      KOPY(K)=LIST2(KC,1)
  413 CONTINUE
C
C     MAKE A Y-LIST (POSSIBLE PAIRING) FROM THE COPY LIST. INSERT A
C     POINTER TO IT IN LIST X, AND HAVE IT POINT BACK TO THE X LIST.
C
  414 Y=LIST(9)
```

```
          CALL SETDIR (0,X,X,TEMP)
          ICELL=NXTLFT (TEMP,Y)
          DO 415 K=1,N2
          CALL SETDIR (1,LIST1(K,1),KOPY(K),TEMP)
          ICELL=NXTLFT (TEMP,Y)
      415 CONTINUE
          CALL SETDIR (1,Y,Y,TEMP)
          ICELL=NXTLFT (TEMP,X)
C
C         ARE THERE MORE PERMUTATIONS TO BE FOUND
C
          IF(SW2)300,4151,300
     4151 IF(FLAG) GO TO 417
          IF(L.EQ.1) GO TO 411
      416 L=L-1
          FLAG=.TRUE.
          GO TO 409
      417 IF(L.EQ.KPART) GO TO 300
          L=L+1
          IF(L.GT.MAXL) GO TO 418
          FLAG=.FALSE.
          GO TO 410
      418 MAXL=L
          GO TO 410
C
C         WE HAVE REACHED A PAIR OF TWIGS SUCCESSFULLY.  DELETE THE BOTTOM
C         CELL OF THIS Y-LIST AND KEEP TRYING. IF THIS Y-LIST IS EMPTY
C         BACK UP ONE LEVEL AND CONTINUE
C
      350 NBCELL=LNKL(CONT(CELL))
          TRASH=DELETE (CELL)
          CNBC= CONT(NBCELL+1)
      351 IF(ID(CNBC).EQ.1) GO TO 300
          IF(CNBC)352,600,352
C
C         ERASE THE X-LIST, ITS Y-LIST,  AND BACK UP TO NEXT Y-LIST
C
      352 X=LNKR(CNBC)
          CALL SETDIR (0,X,X,X)
          CALL IRALST(Y)
          Y=LNKR(CONT(LNKR(CONT(X))+1))
          CALL SETDIR (0,Y,Y,Y)
          CALL IRALST(X)
          TRASH=DELETE (LNKL(CONT(Y)))
          CNBC=CONT(LNKL(CONT(Y))+1)
          GO TO 351
C
C         WE ARE UNABLE TO FIND A MATCH.   BACK UPTO THE X-LIST AND
C         TRY THE NEXT POSSIBLE PAIRING. DELETE THIS  Y-LIST
C
      500 X=LNKR(CONT(LNKR(CONT(Y))+1))
          IF(X)5001,601,5001
     5001 CALL SETDIR (0,X,X,X)
          CALL IRALST(Y)
          TRASH=DELETE (LNKL(CONT(X)))
          NBC=LNKL(CONT(X))
          CNBC=CONT(NBC+1)
          IF(ID(CNBC).EQ.1) GO TO 501
```

```
C
C       THIS X-LIST IS EMPTY, BACK UP FURTHER
C
        Y=LNKR(CNBC)
        CALL SETDIR (0,Y,Y,Y)
        CALL IRALST (X)
        GO TO 500
C
C       BEGIN WORK ON A NEW Y-LIST
C
   501  Y=LNKR(CNBC)
        CALL SETDIR (0,Y,Y,Y)
        GO TO 300
C
C       ALL DONE, SUCCESSFUL
C
   600  FLAG=.TRUE.
        CALL IRALST (Y)
        RETURN
C
C       FAILED
C
   601  FLAG=.FALSE.
        CALL IRALST (Y)
        RETURN
        END




$IBFTC ORDERX
        SUBROUTINE ORDER (ARRAY,N)
C       THIS SUBROUTINE ORDERS AN ARRAY(20,2) OF WHICH THE FIRST N
C       ROWS ARE NON-EMPTY, IN DESCENDING ORDER OF THE SECOND COLUMN
        INTEGER ARRAY(20,2),TEMP
        NM1=N-1
        DO 102 I=1,NM1
        IF(ARRAY(I,2).GE.ARRAY(I+1,2)) GO TO 102
        DO 101 J=1,I
        K=I+1-J
        L=I+2-J
        DO 100 J1=1,2
        TEMP=ARRAY(K,J1)
        ARRAY(K,J1)=ARRAY(L,J1)
        ARRAY(L,J1)=TEMP
   100  CONTINUE
        IF (J.EQ.I) GO TO 102
        IF(ARRAY(K,2).LE.ARRAY(K-1,2)) GO TO 102
   101  CONTINUE
   102  CONTINUE
        RETURN
        END
```

```
$IBFTC PERMX
      SUBROUTINE PERM (N,INT,FLAG)
C     THIS SUBROUTINE IS BASED ON ALGORITHM 102 (ACM. JUNE 1962). IT
C     PERMUTES THE INTEGERS 1 THRU N.  FOR THE FIRST CALL, FLAG MUST
C     BE SET TRUE, PERM THEN SIMPLY RETURNS THE N INTEGERS IN ASCENDING
C     ORDER IN THE ARRAY INT.  SUCCESSIVE CALLS WILL RETURN INT WITH
C     A NEW PERMUTATION (IN LEXICOGRAPHIC ORDER). FLAG IS RESET TO
C     TRUE WHEN ALL PERMUTATIONS HAVE BEEN OBTAINED.
C
C     JOHN PFALTZ, 1 JULY 1965
C
      INTEGER INT(N),Q(20)
      LOGICAL FLAG,FLAG2
C
      IF(FLAG) GO TO 113
      IF(FLAG2) GO TO 110
      FLAG2=.TRUE.
      NM1=N-1
      DO 101 J=2,NM1
      I=N-J
      IF(INT(I).LT.INT(I+1)) GO TO 102
  101 CONTINUE
      FLAG=.TRUE.
      RETURN
  102 DO 103 K=1,N
      Q(K)=0
  103 CONTINUE
      DO 104 K=I,N
      J=INT(K)
      Q(J)=J
  104 CONTINUE
      INTP1=INT(I)+1
      DO 105 K=INTP1,N
      IF(Q(K).NE.0) GO TO 106
  105 CONTINUE
  106 INT(I)=K
      Q(K)=0
      DO 108 K=1,N
      IF(Q(K).EQ.0) GO TO 107
      I=I+1
      INT(I)=Q(K)
      GO TO 108
  107 IF(I.GE.N) GO TO 109
  108 CONTINUE
  109 RETURN
C
C     ALTERNATE BYPASS
C
  110 ITEMP=INT(N)
      INT(N)=INT(N-1)
      INT(N-1)=ITEMP
      IF(INT(N).GT.1) GO TO 111
      NM1=N-1
      DO 112 I=1,NM1
      IF(INT(I).LT.INT(I+1)) GO TO 111
  112 CONTINUE
      FLAG=.TRUE.
      RETURN
  111 FLAG2=.FALSE.
      RETURN
```

```
C
C      INITIALIZATION
C
113 DO 114 I=1,N
    INT(I)=I
114 CONTINUE
    IF(N.EQ.1) GO TO 115
    FLAG2=.TRUE.
    FLAG=.FALSE.
115 RETURN
    END
```

LISTING OF SLIP ROUTINES USED
IN TREETRAN

1. Primative functions locally coded in MAP for IBM 7090/94

   a. ID
   b. LNKL
   c. LNKR
   d. CONT
   e. MADOV
   f. SETDIR
   g. SETIND
   h. STRDIR
   i. STRIND

2. FORTRAN IV SLIP routines substantially as given by Dr. Weizenbaum in Comm. of ACM, Sept. 1963

```
$·IBMAP IDX       5                                                    SLIP
        ENTRY   ID                                                     SLIP
*    THIS PRIMATIVE FUNCTION - ID(CELL) - PRESENTS AS AN INTEGER THE ID SLIP
*    PORTION OF CELL.                                                  SLIP
ID      CLA     =0                                                     SLIP
        LDQ*    3,4             GET CELL                               SLIP
        LLS     2               SHIFT ID PORTION INTO ACC              SLIP
        TRA     1,4                                                    SLIP
        END                                                            SLIP


$IBMAP LNKLX      5                                                    SLIP
        ENTRY   LNKL                                                   SLIP
*    THIS PRIMATIVE FUNCTION - LNKL(CELL) - PRESENTS AS AN INTEGER THE SLIP
*    MACHINE ADDRESS CONTAINED IN THE LEFT LINK FIELD OF CELL.         SLIP
LNKL    CLA*    3,4             GET CELL                               SLIP
        ANA     MASK            MASK OUT ID AND LNKR                   SLIP
        ARS     18              SHIFT INTO ADDRESS                     SLIP
        TRA     1,4                                                    SLIP
MASK    OCT     077777000000                                          SLIP
        END                                                            SLIP


$IBMAP LNKRX      3                                                    SLIP
        ENTRY   LNKR                                                   SLIP
*    THIS PRIMATIVE FUNCTION - LNKR(CELL) - PRESENTS AS AN INTEGER THE SLIP
*    MACHINE ADDRESS CENTAINED IN THE RIGHT LINK FIELD OF CELL.        SLIP
LNKR    CLA*    3,4             GET CELL                               SLIP
        ANA     MASK            MASK OUT ID AND LNKL                   SLIP
        TRA     1,4                                                    SLIP
MASK    OCT     000000077777                                          SLIP
        END                                                            SLIP


$IBMAP CONTX      5                                                    SLIP
        ENTRY   CONT                                                   SLIP
        ENTRY   INHALT                                                 SLIP
*    THESE TWO PRIMATIVE FUNCTIONS - CONT(A), AND INHALT(A) - DELIVER  SLIP
*    AS FUNCTIONAL VALUES THE CONTENTS OF THE WORD WHOSE  MACHINE ADDRESSSLIP
*    IS THE INTEGER STORED IN A.   USE CONT IF THE CONTENST IS USED IN A SLIP
*    FLOATING POINT EXPRESSION,  INHALT IF USED IN A FIXED PT. EXPRESS. SLIP
CONT    TRA     *+1                                                    SLIP
INHALT  CLA*    3,4             GET ADDRESS STORED IN A                SLIP
        STA     *+1                                                    SLIP
        CLA     **              GET DATA                               SLIP
        TRA     1,4                                                    SLIP
        END                                                            SLIP
```

```
$IBMAP MADOVX    2                                            SLIP
       ENTRY    MADOV                                         SLIP
·*    THIS PRIMATIVE FUNCTION - MADOV(A) - PRESENTS AS AN INTEGER    SLIP
*     FUNCTIONAL VALUE THE MACHINE ADDRESS OF A.                SLIP
MADOV  CLA      3,4                 GET THE LOCATION OF A       SLIP
       TRA      1,4                                            SLIP
       END                                                    SLIP



$IBMAP SETDX    12                                            SLIP
       ENTRY    SETDIR                                         SLIP
*     THIS PRIMATIVE - SETDIR(I,L,R,CELL) - STORES DIRECTLY    I  IN THE ID SLIP
*     FIELD, L IN THE LEFT LINK FIELD, AND R IN THE RIGHT LINK FIELD OF  SLIP
*     CELL.   IF ANY OF I, L, OR ( ARE SET TO -1, THEN THE CORRESPONDING  SLIP
*     FIELD IS LEFT UNCHANGED.                                 SLIP
SETDIR CLA*     3,4                 GET I                      SLIP
       TMI      *+3                 IS IT NEGATIVE             SLIP
       ALS      33                     NO, SHIFT INTO PREFIX   SLIP
       STP*     6,4                     AND STORE              SLIP
       CLA*     4,4                 GET L                      SLIP
       TMI      *+3                 IS IT NEGATIVE             SLIP
       ALS      18                     NO, SHIFT INTO DECREMENT SLIP
       STD*     6,4                     AND STORE              SLIP
       CLA*     5,4                 GET R                      SLIP
       TMI      *+2                 IS IT NEGATIVE             SLIP
       STA*     6,4                     NO, STORE IN ADDRESS   SLIP
       TRA      1,4                                            SLIP
       END                                                    SLIP



$IBMAP SETIX    18                                            SLIP
       ENTRY    SETIND                                         SLIP
*     THIS PRIMATIVE - SETIND(I,L,R,A) - FUNCTIONS EXACTLY AS SETDIR,  SLIP
*     EXCEPT THAT IT STORES I, L, AND R IN THE CELL WHOSE ADDRESS IS   SLIP
*     FOUND IN A.  (IE. INDIRECTLY)                            SLIP
SETIND CLA*     6,4                 GET ADDRESS OF CELL        SLIP
       STA      *+6                                            SLIP
       STA      *+9                                            SLIP
       STA      *+11                                           SLIP
       CLA*     3,4                 GET I                      SLIP
       TMI      *+3                 IS IT NEGATIVE             SLIP
       ALS      33                     NO, SHIFT INTO PREFIX   SLIP
       STP      **                      AND STORE              SLIP
       CLA*     4,4                 GET L                      SLIP
       TMI      *+3                 IS IT NEGATIVE             SLIP
       ALS      18                     NO, SHIFT INTO DECREMENT SLIP
       STD      **                      AND STORE              SLIP
       CLA*     5,4                 GET R                      SLIP
       TMI      *+2                 IS IT NEGATIVE             SLIP
       STA      **                      NO, STORE IN ADDRESS   SLIP
       TRA      1,4                                            SLIP
       END                                                    SLIP
```

```
$IBMAP STRDX    3                                                    SLIP
       ENTRY   STRDIR                                                SLIP
*    THIS PRIMATIVE FUNCTION - STRDIR(DATUM,CELL) - STORES THE DATUM  SLIP
*    DIRECTLY IN CELL.   THE DATUM CAN BE EITHER FIXED OR FLOATING POINT SLIP
*    THE DATUM IS ALSO RETAINED AS THE FUNCTION VALUE, HENCE THIS     SLIP
*    PRIMATIVE CAN BE NESTED.                                         SLIP
STRDIR CLA*    3,4              GET DATUM                             SLIP
       STO*    4,4              STORE IT                              SLIP
       TRA     1,4                                                   SLIP
       END                                                           SLIP



$IBMAP STRIX    5                                                    SLIP
       ENTRY   STRIND                                                SLIP
*    THIS PRIMATIVE FUNCTION - STRIND(DATUM,A) - IS THE SAME AS STRDIR, SLIP
*    EXCEPT THAT IS STORES THE DATUM IN THE CELL WHOSE MACHINE ADDRESS  SLIP
*    IS CONTAINED IN A.  (IE. INDIRECTLY)                            SLIP
STRIND CLA*    4,4              GET ADDRESS OF CELL                  SLIP
       STA     *+2                                                   SLIP
       CLA*    3,4              GET DATUM                            SLIP
       STO     **               STORE IT                            SLIP
       TRA     1,4                                                   SLIP
       END                                                           SLIP
```

```
$IBFTC ADLERX
      .FUNCTION ADVLER(LR,A)
C         ADVLER(LR,A)    = ADVANCE LINEARLY RIGHT TO NEXT ELEMENT
C·        THIS FUNCTION ADVANCES TO THE RIGHT (DOWN) THROUGH LIST WHICH IS
C         SPECIFIED BY READER 'LR' UNTIL ELEMENT CELL OR HEADER CELL IS DETECTEL
C         FLAG -'A' - WILL BE SET TO ZERO IF TERMINATION POINT IS ELEMENT
C         AND VALUE OF CELL, I.E. THE DATUM, WILL BE THE VALUE OF THIS
C         FUNCTION.
C         FLAG 'A' WILL BE SET TO NON-ZERO IF HEADER CELL CAUSED
C         TERMINATION.  VALUE OF FUNCTION WILL BE SET TO ZERO
C
      A = ADVLR(LR,0,0)
      IF(A)1,2,1
    2 ADVLER = REED(LR)
    1 RETURN
      END




$IBFTC ADLNRX
      FUNCTION ADVLNR(LR,A)
C         ADVLNR(LR,A)    = ADVANCE LINEARLY RIGHT TO NEXT NAME
C         THIS FUNCTION ADVANCES TO THE RIGHT (DOWN) THROUGH LIST WHICH IS
C         SPECIFIED BY READER 'LR' UNTIL NAME    CELL OR HEADER CELL IS DETECTEL
C         FLAG -'A' - WILL BE SET TO ZERO IF TERMINATION POINT IS NAME
C         AND VALUE OF CELL, I.E. THE DATUM, WILL BE THE VALUE OF THIS
C         FUNCTION.
C         FLAG 'A' WILL BE SET TO NON-ZERO IF HEADER CELL CAUSED
C         TERMINATION.  VALUE OF FUNCTION WILL BE SET TO ZERO
C
      A = ADVLR(LR,1,1)
      IF(A)1,2,1
    2 ADVLNR = REED(LR)
    1 RETURN
      END




$IBFTC ADLWRX
      FUNCTION ADVLWR(LR,A)
C         ADVLWR(LR,A)    = ADVANCE LINEARLY RIGHT TO NEXT WORD
C         THIS FUNCTION ADVANCES TO THE RIGHT (DOWN) THROUGH LIST WHICH IS
C         SPECIFIED BY READER 'LR' UNTIL WORD    CELL OR HEADER CELL IS DETECTEL
C         FLAG -'A' - WILL BE SET TO ZERO IF TERMINATION POINT IS WORD
C         AND VALUE OF CELL, I.E. THE DATUM, WILL BE THE VALUE OF THIS
C         FUNCTION.
C         FLAG 'A' WILL BE SET TO NON-ZERO IF HEADER CELL CAUSED
C         TERMINATION.  VALUE OF FUNCTION WILL BE SET TO ZERO
C
      A = ADVLR(LR,1,0)
      IF(A)1,2,1
    2 ADVLWR = REED(LR)
    1 RETURN
      END
```

```
$IBFTC ADVLRX
      FUNCTION ADVLR(LR,J,K)
C
C     THIS FUNCTION - ADVLR(L,J,K) - IS THE BOOKKEEPING GUTS OF ALL THE
C     LINEAR LEFT ADVANCES.  L IS THE ALIAS OF THE READER FOR THE LIST
C     BEING SEARCHED9  J AND K INDICATE THE TYPE OF ADVANCE, NAME,
C     ELEMENT, OR WORD.  SINCE A LINEAR ADVANCE NEVER DESCENDS
C     SUBLISTS, THE READER IS NEVER EXTENDED, MERELY ITS POINTER
C     CHANGED.  IF A CELL WITH THE DESIRED CHARACTERISTICS IS FOUND,
C     THE FUNCTIONAL VALUE IS ZERO,  IF NOT, MINUS ONE.
C
      CLR = CONT(LR)
    5 LK = LNKR(CONT(LNKL(CLR)))
      CAND = CONT(LK)
      CALL SETDIR(-1,LK,-1,CLR)
       IF (ID(CAND)-2)1,2,1
    1    IF (ID(CAND)-J)3,4,3
    3    IF (ID(CAND)-K)5,4,5
    4 ADVLR = 0.
      GOTO 6
    2 ADVLR = -1.0
    6 CALL STRIND(CLR,LR)
      RETURN
      END




$IBFTC ADSERX
      FUNCTION ADVSER(LR,A)
C
C        ADVSER(LR,A) =   ADVANCE STRUCTUALLY ELEMENT RIGHT
C      THIS FUNCTION ADVANCES TO THE NEXT TO THE RIGHT (DOWN) CELL WHICH IS A
C        ELEMENT CELL.
C      IF THIS CELL IS A ELEMENT THE FLAG IS SET TO ZERO AND THE DATUM
C        OF THIS CELL IS DELIVERED AS ITS FUNCTIONAL VALUE.
C      IF THIS CELL IS A HEADER CELL THE FLAG IS SET TO MINUS ONE.
C        THIS INDICATES AN END OF A LIST(ANY LEVEL LIST).
C      THIS ADVANCE FOLLOWS THROUGH ALL LEVELS OF SUBLISTS FOR LIST INDICATED.
C
C
      A = ADVSR(LR,0,0)
      IF(A)1,2,1
    2 ADVSER = REED(LR)
    1 RETURN
      END
```

```
$IBFTC ADSNRX
      FUNCTION ADVSNR(LR,A)
C
C         ADVSNR(LR,A) =  ADVANCE STRUCTUALLY NAME     RIGHT
C      THIS FUNCTION ADVANCES TO THE NEXT TO THE RIGHT (DOWN) CELL WHICH IS A
C         NAME   CELL.
C      IF THIS CELL IS A NAME     THE FLAG IS SET TO ZERO AND THE DATUM
C         OF THIS CELL IS DELIVERED AS ITS FUNCTIONAL VALUE.
C      IF THIS CELL IS A HEADER CELL THE FLAG IS SET TO MINUS ONE.
C         THIS INDICATES AN END OF A LIST(ANY LEVEL LIST).
C      THIS ADVANCE FOLLOWS THROUGH ALL LEVELS OF SUBLISTS FOR LIST INDICATED.
C
      A = ADVSR(LR,1,1)
      IF(A)1,2,1
    2 ADVSNR = REED(LR)
    1 RETURN
      END




$IBFTC ADSWRX
      FUNCTION ADVSWR(LR,A)
C      THIS FUNCTION - ADVSWR(LR,A) - ADVANCES ONE CELL TO THE RIGHT
C      (DOWN) IN THE LIST WHOSE READER HAS ALIAS LR.  IF THE ADVANCE IS
C      POSSIBLE,  FLAG A IS SET TO ZERO AND THE DATUM OF THAT CELL IS
C      DELIVERED AS FUNCTIONAL VALUE.  IF NOT (IE. WE ARE AT THE BOTTOM
C      OF THE STRUCTURE), FLAG A IS SET TO MINUS ONE9
C
      A = ADVSR(LR,1,0)
      IF(A)1,2,1
    2 ADVSWR = REED(LR)
    1 RETURN
      END




$IBFTC ADVSRX
      FUNCTION ADVSR(L,J,K)
C      THIS SUBROUTINE - ADVSR(L,J,K) - IS THE BOOKKEEPING GUTS OF ALL
C      THE STRUCTURAL RIGHT ADVANCES.  L IS THE ALIAS OF THE READER FOR
C      THE LIST.  J AND K INDICATE WHETHER WE ARE  INVOLVED IN A NAME,
C      ELEMENT, OR WORD ADVANCE.  IT EXTENDS THE READER CHAIN AS
C      NECESSARY IN SEARCHING FOR THE DESIRED CELL.   IF A CELL WITH THE
C      DESIRED CHARACTERISTICS IS FOUND, THE FUNCTIONAL VALUE IS ZERO.
C      IF NOT, MINUS ONE.  (NOTE. ADVSR WILL SEARCH THE ENTIRE LIST
C      STRUCTURE TO THE RIGHT OF THE STARTING POINT TO FIND AN ACCEPTABLE
C      CELL.)
C
         R = CONT(L)
       LCP = LNKL(R)
         CAND = CONT(LNKL(R))
         IF (ID(CAND)-1)1,6,1
    1 LCP = LNKR(CAND)
      CALL SETDIR(-1,LCP,-1,R)
      CAND = CONT(LCP)
        IF (ID(CAND)-2)3,4,3
    3      IF(ID(CAND)-J)7,8,7
    7      IF(ID(CAND)-K)5,8,5
```

```
   5     IF(ID(CAND)-1)1,6,1
   6 M=NUCELL(Z)
     CALL STRIND(R,M)
     CALL STRIND(CONT(L+1),M+1)
     CALL SETIND(-1,INHALT(LCP+1),LCNTR(L)+1,L+1)
     CALL SETDIR(-1,-1,M,R)
     CAND = CONT(INHALT(LNKL(R)+1))
     GOTO 1
   4 IF (LCNTR(L))9,10,9
  10 ADVSR = -1.0
     GOTO 12
   9 LK = LNKR(R)
     R = CONT(LK)
     CALL STRIND(CONT(LK+1),L+1)
     CAND = CONT(LNKL(R))
     CALL RCELL(LK)
     GOTO 1
   8 ADVSR = 0.0
  12 CALL STRIND(R,L)
     RETURN
     END




$IBFTC DELETX
     FUNCTION DELETE(K)
C
C         DELETE THE CELL WHOSE ADDRESS IS FOUND IN -K-.
C             IF THE CELL IS A HEADER. A DIAGNOSTIC IS PRINTED,PROGRAM CONTINUED
C             AND VALUE OF FUNCTION IS SET TO ZERO.
C             THE VALUE OF THE FUNCTION IS THE CONTENTS OF THE DELETED CELL.
C
         IF(ID(CONT(K))-2)1,2,1
   2     WRITE (6,901)
         DELETE = 0.
     RETURN
 901     FORMAT(1H4,97HAN ATTEMPT HAS BEEN MADE TO DELETE A HEADER - ZERO
     1 HAS BEEN DELIVERED AND THE PROGRAM CONTINUED/1X,31HSLIP MESSAGE
     I- ROUTINE DELETE   )
   1     DELETE = CONT(K+1)
         LL = LNKL(CONT(K))
         LR = LNKR(CONT(K))
     CALL RCELL(K)
     CALL SETIND(-1,-1,LR,LL)
     CALL SETIND(-1,LL,-1,LR)
     RETURN
     END
```

```
$IBFTC INITX
      SUBROUTINE INITAS(M,N)
C
C     THIS SUBROUTINE INITIALIZES THE STRUCTURE OF THE DIMENSIONED
C     ARRAY SPACE INTO A LIST OF AVAILABE SPACE (LAVS).
C
      COMMON /SLIPC/AVSL
      DIMENSION M(N)
C
C     THIS PART SETS UP THE LIST OF AVAILABLE SPACE (LAVS)
C
      DO 2 I=1,N
    2    M(I) = 0
         K = N-2
      DO 3 I=1,K,2
    3 CALL SETDIR(-1,-1,MADOV(M(I+2)),M(I))
      CALL SETDIR(0,MADOV(M(N-1)),MADOV(M(1)),AVSL)
      RETURN
      END




$IBFTC IRALX
      FUNCTION IRALST(P)
C
C     THIS FUNCTION RETURNS A NAMED LIST TO LAVS UNLESS ITS CELLS ARE
C     A SUBLIST OF ANOTHER LIST.
C
C     THIS VERSION OF IRALST IS NON-STANDARD IN THAT IT DOES NOT CHECK
C     TO SEE IF THE HEADER REFERS TO A DESCRIPTION LIST.
C
      L=LOCT(P)                                                    SLIP
      CALL SETIND(-1,-1,LCNTR(L)-1,L+1)
         IRALST = LCNTR(L)
      IF(IRALST)2,2,1
    2 CALL MTLIST(P)
    4 CALL RCELL(L)
    1 RETURN
      END




$IBFTC LCNTRX
      FUNCTION LCNTR(K)
C
C     THIS FUNCTION - LCNTR(R) - GIVES AS FUNCTIONAL VALUE THE LEVEL
C     COUNTER OF READER R.
C
      LCNTR = LNKR(CONT(K+1))
      RETURN
      END
```

```
$IBFTC LISTX
      FUNCTION LIST(K)
.C
C     THIS FUNCTION CREATES A NAMED LIST (EMPTY) WHICH CANNOT BE
C     UNINTENTIONALLY ERASED.
C
C
          LIST = NUCELL(Z)
      CALL SETDIR(0,LIST,LIST,LIST)
      CALL SETIND(2,LIST,LIST,LIST)
          IF (K-9)2,1,2
    2 CALL SETIND(-1,-1,1,LIST+1)
          K = LIST
    1 RETURN
      END
```

```
$IBFTC LSTMTX                                                          SLIP
      FUNCTION LISTMT(P)
C
C     THIS TEST FUNCTION - LISTMT(P) - CHECKS TO SEE IF THE LIST NAMED  SLIP
C     BY P IS AN EMPTY LIST.   IF SO, THE FUNCTION VALUE IS ZERO,       SLIP
C     OTHERWISE IT IS MINUE ONE.                                        SLIP
C                                                                       SLIP
      L = LOCT(P)
      IF(INHALT(L)-INHALT(LNKR(CONT(L))))3,4,3                          SLIP
    4 LISTMT = 0
      RETURN
    3 LISTMT = -1
      RETURN
      END
```

```
$IBFTC LOCTX                                                           SLIP
      FUNCTION LOCT(K)
C
C     THIS TEXT FUNCTION - LOCT(K) - IS USED TO VERIFY THAT THE CONTENTSSLIP
C     OF CELL K NAMES A LIST , WHERE IT IS REQUIRED BY THE CALLING      SLIP
C     ROUTINE.  IF K DOES NOT NAME A LIST AN ERROR MESSAGE IS PRINTED   SLIP
C     AND EXECUTION STOPPED,   IF IT IS O.K. THE FUNCTION VALUE IS      SLIP
C     MERELY THE ARGUMENT.                                             SLIP
C                                                                       SLIP
      IF(NAMTST(K))1,2,1
    2    LOCT = K
      RETURN
    1    PRINT 901
      WRITE (6,901)
      CALL FXEM(500)
         STOP
  901 FORMAT(1H4,113HA LIST WAS REQUIRED AS AN OPERAND BUT WAS NOT FOUND
     1 THE PROGRAM WAS REGRETFULLY TERMINATED BY SLIP ROUTINE  LOCT. //)
      END
```

```
$IBFTC LOFRDX
      FUNCTION LOFRDR(K)
.C
C     THIS FUNCTION - LOFRDR(R) - GIVES AS FUNCTIONAL VALUE THE MACHINE
C     ADDRESS OF THE HEADER OF THE LIST FOR WHICH THIS IS A READER.
C     (IT ALSO APPEARS TO MAKE THE LIST AN EMPTY LIST.)
C
      L = LNKL(CONT(K+1))
      CALL SETDIR(0,L,L,L)
      LOFRDR = L
      RETURN
      END




$IBFTC LPNTRX
      FUNCTION LPNTR(K)
C
C     THIS FUNCTION - LPNTR(R) - GIVES AS FUNCTIONAL VALUE THE MACHINE
C     ADDRESS OF THE CELL THE READER R IS CURRENTLY POINTING AT.
C
      LPNTR = LNKL(CONT(K))
      RETURN
      END




$IBFTC LRDOVX
      FUNCTION LRDROV(P)
C
C     THIS FUNCTION - LRDROV(P) - ASSIGNS A READER FOR THE LIST WITH
C     ALIAS P.  (IE.  A CELL IS TAKEN FROM LAVS, PUT IN THE FORM OF A
C     READER,  AND MADE TO POINT AT THE HEADER OF P.)    2TS FUNCTIONAL
C     VALUE IS THE MACHINE ADDRESS OF THIS CELL.
C
       LRDROV = NUCELL(Z)
      CALL SETIND(3,LOCT(P),0,LRDROV)                                   SLIP
      CALL SETIND(0,P,0,LRDROV+1)
      RETURN
      END
```

```
$IBFTC LVLR1X
      FUNCTION LVLRV1(K)
.C
C     THIS FUNCTION CAUSES THE READER TO ASCEND ONE LEVEL IN ITS STACK
C
      LVLRV1 = K
      IF (CONT(LVLRV1+1))2,3,2
3 RETURN
2     L = LNKR(CONT(LVLRV1))
      CALL    STRIND(CONT(L),LVLRV1)
      CALL    STRIND(CONT(L+1),LVLRV1+1)
      CALL    RCELL(L)
      RETURN
      END




$IBFTC MTLISX
      FUNCTION MTLIST(P)
C
C     THIS FUNCTION RETURNS A LIST TO LAVS.  THE BOTTOM CELL OF LAVS
C     IS MADE TO POINT TO THE TOP OF THE LIST, AND THE BOTTOM OF THE
C     LIST BECOMES THE BOTTOM OF LAVS.
      COMMON /SLIPC/AVSL
      M=LOCT(P)                                                    SLIP
      IF (LISTMT(P))3,4,3
3     LR = LNKR(CONT(M))
      LL = LNKL(CONT(M))
      CALL SETIND(-1,M,M,M)
      CALL SETIND(-1,-1,LR,LNKL(AVSL))
      CALL SETDIR(-1,LL,-1,AVSL)
      CALL SETIND(-1,-1,0,LNKL(AVSL))
4     MTLIST = M
      RETURN
      END




$IBFTC NAMETX                                                     SLIP
      FUNCTION NAMTST(K)
C
C     THIS TEST FUNCTION - NAMTST(K) - CHECKS THE CONTENTS OF CELL K.  SLIP
C     IF IT IS THE NAME OF A LIST,  THE FUNCTION VALUE IS ZERO,  OTHER- SLIP
C     WISE IT IS MINUS ONE.                                            SLIP
C                                                                      SLIP
      IF (LNKL(K)-LNKR(K))1,4,1                                        SLIP
4 IF (ID(CONT(K))-2)1,2,1
2 IF(CONT(LNKR(CONT(LNKL(CONT(K)))))-CONT(K))1,3,1
3     NAMTST = 0
      RETURN
1     NAMTST = -1
      RETURN
      END
```

```
$IBFTC NUCELX
       FUNCTION NUCELL(X)
C
C      THIS FUNCTION GETS A NEW CELL FROM LAVS.  IT CHECKS THE ID OF
C      THE CELL BEFORE DELIVERY, IF IT IS 1 (CELL REFERS TO A SUBLIST)
C      THE READER OF THAT SUBLIST IS DECREMENTED BY 1.
C
       COMMON /SLIPC/AVSL
          M = LNKR(AVSL)
          IF (M)1,2,1
     2 PRINT 901
       WRITE (6,901)
       STOP
     1 IF (ID(CONT(M))-1)3,4,3
C
C      THIS CELL IS A NAME CELL, LOWER REFERENCE COUNTER OF LIST NAMED
C      AND ERASE IF POSSIBLE.
C
     4 CALL IRALST (CONT(M+1))
     3 CALL SETDIR (-1,-1,LNKR(CONT(M)),AVSL)
       CALL STRIND (0,M)
       CALL STRIND (0,M+1)
       NUCELL=M
       RETURN
   901 FORMAT (1H1,6X,81HLIST OF AVAILABLE SPACE EXHAUSTED - PROGRAM TERM
      1INATED BY SLIP ROUTINE NUCELL  ///)
       END




$IBFTC NXTLFX
       FUNCTION NXTLFT(M,A)
C
C      THIS FUNCTION -NXTLFT(M,A) - STORES THE DATUM M IN THE CELL TO
C      THE LEFT (ABOVE) OF THE CELL SPECIFIED BY THE LEFT LINK OF A.
C      THIS NEW CELL IS TAKEN FROM LAVS, AND ITS MACHINE ADDRESS IS THE
C      FUNCTION VALUE.
C
          IL = NUCELL(Z)
       NXTLFT = IL
         LL = LNKL(CONT(A))
       CALL SETIND(-1,-1,IL,LL)
       CALL SETIND(-1,IL,-1,A)
       CALL SETIND(0,LL,A,IL)
          IF(NAMTST(M))1,2,1
     2 CALL SETIND(1,-1,-1,IL)
       CALL SETIND(-1,-1,LCNTR(M)+1,M+1)
     1 CALL STRIND(M,IL+1)
       RETURN
       END
```

```
$IBFTC NXTRTX
      FUNCTION NXTRGT(M,A)
C
C     THIS FUNCTION - NXTRGT(M,A) - STORES THE DATUM M IN THE CELL TO
C     THE RIGHT (BELOW) THE CELL SPECIFIED BY THE RIGHT LINK OF A.  THIS
C     NEW CELL IS TAKEN FROM LAVS, AND ITS MACHINE ADDRESS IS THE
C     FUNCTION VALUE.
C
          IR = NUCELL(Z)
      NXTRGT = IR
        LR = LNKR(CONT(A))
      CALL SETIND(-1,IR,-1,LR)
      CALL SETIND(-1,-1,IR,A)
      CALL SETIND (0,A,LR,IR)
          IF (NAMTST(M))1,2,1
    2 CALL SETIND(1,-1,-1,IR)
      CALL SETIND(-1,-1,LCNTR(M)+1,M+1)
    1 CALL STRIND(M,IR+1)
      RETURN
      END




$IBFTC RCELLX
      SUBROUTINE RCELL(CELL)
C
C     THIS SUBROUTINE RETURNS A CELL TO LAVS.
C
      COMMON /SLIPC/AVSL
      CALL SETIND(-1,-1,CELL,LNKL(AVSL))
      CALL SETDIR(-1,CELL,-1,AVSL)
      CALL SETIND(-1,-1,0,CELL)
      RETURN
      END




$IBFTC REEDX
      FUNCTION REED(K)
C
C     THIS FUNCTION - REED(K) - HAS AS ARGUMENT K, THE ALIAS OF A READER
C     IT LOOKS AS THE CELL TO WHICH THE READER IS POINTING AND DELIVERS
C     ITS DATUM AS FUNCTIONAL VALUE.
C
      REED = CONT(LNKL(CONT(K))+1)
      RETURN
      END
```

References

Avondo-Bodino, G. Economic Applications of the Theory of
    Graphs, 1962, Gordon and Breach, New York

Berge, C. The Theory of Graphs and its Applications, 1962,
    John Wiley & Sons, London

Weizenbaum, J. "Symetric List Processor" Communications
    of the ACM, September 1963