1823

SLAC-119

MASTER

TAXL—A Simple Hierarchical Data Structure Manipulation System

Sheldon I. Becker

SLAC Report No. 119
June 1970

AEC Contract AT(04-3)-515

STANFORD LINEAR ACCELERATOR CENTER
Stanford University • Stanford, California

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

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

DISCLAIMER

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

SLAC-119
UC-32
(MISC.)

TAXL--A SIMPLE HIERARCHICAL DATA STRUCTURE MANIPULATION SYSTEM

SHELDON I. BECKER
STANFORD LINEAR ACCELERATOR CENTER
STANFORD UNIVERSITY
Stanford, California 94305

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

PREPARED FOR THE U.S. ATOMIC ENERGY COMMISSION UNDER CONTRACT NO. AT(04-3)-515

June 1970

Reproduced in the USA. Available from the Clearinghouse for Federal Scientific and Technical Information, Springfield, Virginia 22151.
Price: Full size copy $3.00; microfiche copy $ .65.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

THIS PAGE

WAS INTENTIONALLY

LEFT BLANK

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	THE NATURE AND ADDRESSING OF THE DATA STRUCTURE	6
	A. The Data Structure	6
	B. The Data Item	6
	C. The Range	8
	1. Method I	9
	2. Method II	11
	3. Method III	12
	4. Method IV	15
	D. Total Range Specification	17
	E. Subtree Context Specification	18
	1. Condition I	18
	2. Condition II	21
	3. Condition III	21
	F. Conclusion	23
III.	DESCRIPTION OF THE PRIMITIVES	24
	A. CREATE	26
	B. LABEL	28
	C. UNLABEL	29
	D. COUNT	31
	E. WRITE	32
	F. PUT	35
	G. COPY	37

Chapter		Page
	H. SEVER	41
	I. DELETE	44
	J. SAVE	48
	K. RESTORE	48
	L. Conclusion	49
IV.	THE TAXL/BASIC SYSTEM	50
	A. Syntax Analysis	52
	B. Command Classification	53
	C. Interface Between TAXL and BASIC	57
	D. Responses Following the Execution of Commands	63
	E. Conclusion	64
V.	AN IMPLEMENTATION AND ITS ANALYSIS	65
	A. Node and Dictionary Formats	68
	B. The Computation of Ranges	69
	C. Reducing Range Computation Time	73
	D. Memory Usage	76
	E. Access Time	80
	F. Operation Time	84
	G. System Measures	84
	H. Conclusion	89
VI.	FUTURE WORK AND SUMMARY	92
APPENDIX I		97
APPENDIX II		110

LIST OF TABLES

Page

1. The Primitives 25
2. Reserved Word List 27
3. Command Classification 54
4. Storage Requirements for a Hypothetical Data Base 81
5. System Measures 85

LIST OF FIGURES

Page

1a. Directed multirooted multibranching tree 7

1b. Directed acyclic graph 7

2. A Personnel file 10

3. Format of a node in the data base 66

4. Format of an atom used as a label in the data base. 67

5. A portion of a data base demonstrating the utility of below and above 75

CHAPTER I

INTRODUCTION

The work described here attacks two problems: the lack of agreement on the nature of certain aspects of nonnumeric computer processing, and the educational bottleneck resulting from the large numbers of people who know little about computers but who wish to see how they might be used in their work.

The first problem is a generic problem in the nature of nonnumeric processing. The essence of numeric calculations and operations has been known for some time, and the advent of high speed digital computers has solidified these concepts. Almost all general purpose digital computers have facilities for doing arithmetic, and depending on the size and cost of the machine, these facilities can be quite elaborate. This should not be surprising since the first uses to which digital computers were put were almost exclusively numerical calculations.

The notion that a general purpose digital computer can be a very general symbol manipulator began to grow from the early days of computing and is now an accepted notion throughout most of the computing community. As yet, however, there has been very little agreement on what constitutes general symbolic manipulation, i.e., nonnumeric calculation. This lack of agreement can be seen at the hardware level by the fact that there has been no unanimous introduction of pieces of hardware to do nonnumeric processing, as opposed, for example, to the existence of adders for numeric processing. On the software level, the great variety of "list processing" languages such as LISP (McCarthy [1962]), SNOBOL (Griswold [1968]), and L6 (Knowlton [1966]), and "associative" languages like LEAP (Feldman [1969]) and ASP (Lang [1968]) indicates that there are certainly divergent opinions on the nature of nonnumeric processing. One of the purposes

of this work, then, is to try to shed some light on a representation and subsequent manipulation of nonnumeric data.

The second problem with which this work deals is related to the fact that computers can be very useful tools in many areas, including both mathematical sciences and nonscientific fields such as history, government, sociology, law, etc.

The phenomenal growth of accessibility to computers and the number of people anxious to use computers have caused quite a bottleneck in the facilities for training these people. Introductory programming courses in universities and colleges are almost always overcrowded as students in the physical sciences and, more and more often now, the social sciences realize that computers might be able to help them in their own fields.

These immense numbers of people, who are eager and should learn how to use and how not to use computers in their own work, require that new methods of teaching and learning these skills be explored. The traditional university course, for example, lasting from at least several weeks to a quarter or semester is quickly becoming inadequate to serve the volume of people eager to acquire the knowledge of some programming language which might be useful to them. The length of time which is required for the computer novice to learn many of the computer languages and systems, with their increasing generality and complexity, is usually more than he and his instructor wish to spend.

Kemeny and Kurtz at Dartmouth have attempted to alleviate this problem by designing and implementing an interactive computer language and system called BASIC (Kemeny [1967]). The simplicity of the system and ease with which the language can be learned are evidenced by the fact that a very short formal lecture session is usually all that is necessary for the novice to begin writing programs

that are useful to him. The interactive nature of the system allows the novice to use the system at his leisure and to search for answers to questions which occur to him about the system by experimentation. The great utility of this approach to the training and teaching problem is attested to by the great number of BASIC systems which have been adopted by many computation centers and the wide support and use these systems are receiving from their users.

There are a number of problems for which the BASIC language and system is inadequate, but for these problems there are more general and more complex languages and systems which the novice can learn and use. However, for a great many common everyday problems, BASIC is entirely adequate, and the ease with which it can be so used bears strong evidence that BASIC's approach to the computer education bottleneck is a good one.

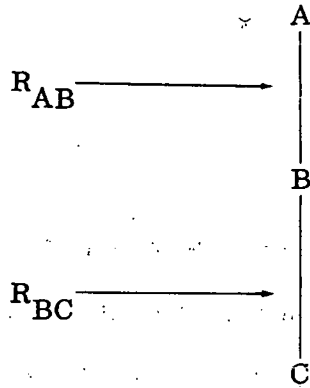
This work, then, is an attempt to combine partial solutions to these two problems: to shed further light on the nature of one aspect of nonnumeric processing, and to aid in reducing the educational bottleneck in that area.

The vehicle for this study is the design and test implementation of a nonnumeric data processing capability suitable for inclusion with a BASIC system. It is felt that such an addition would greatly enhance the already great appeal of BASIC to those nonscientific users who already view BASIC as a useful tool for the solutions of their numeric problems.

In trying to follow one of the rules for the development of BASIC, which was to find those few primitives which were not only basic and useful, but also of high pedagogic value, it was decided that the majority of current nonnumeric list-processing languages were too much data-structure oriented rather than problem-solving oriented. Hence, while these languages contain almost all the basic primitives for list processing, they are too difficult to use and of too little pedagogic value for those novice users for whom the nonnumeric capability is intended.

As different in format and applicability as the existing nonnumeric languages are, there is a common motive that runs through all of them. In one form or another, all of these languages emphasize the relationships between data as opposed to emphasizing the data themselves. In IPL (IPL [1961]) and LISP (McCarthy [1962]), for example, the sublist concept and associated mechanisms for creating, manipulating, and destroying such sublists deals with the relationships between not only atoms of data but also between other relationships. Much the same can be said for the pointer structuring capabilities in ALGOL W (Bauer [1969]) and in L6 (Knowlton [1966]), the basis for which was Wirth and Hoare's records and references (Wirth [1966]) and Ross's plex processing (Ross [1961]). It certainly appears that the ability to specify relationships which exist among data and to manipulate these relationships are at the heart of nonnumeric processing.

Having ascertained the centrality of the concept of relationship specification and manipulation to nonnumeric processing, we turn to finding those few primitives which are basic, useful, and of high pedagogic value. More explicitly, when considering nonnumeric processing and the relationship concept, the problem is to focus on some hopefully small subset of all possible relationships in order to simplify both the language and the concepts involved in teaching. To this end, I have chosen one type of relationship, the hierarchical relationship. Webster (Webster [1964]) defines "hierarchy" as "the arrangement of objects, elements, or values in a graduated series." Notice that the emphasis is on the arrangement of the objects rather than the objects themselves. The graduated nature of a hierarchy as defined, as well as the intuitive feelings of what constitutes a hierarchy, implies the true generality of this relationship: it exists or can easily be made to exist among data in a great many different kinds of data bases. In addition, two relationships can be different in their meaning but still be hierarchical.



In other words, R_{AB} and R_{BC} can both be hierarchical relationships but having different semantic content depending on the data A, B, and C. For example, A could describe a professor, B could describe his secretary, and C could describe her salary. The exact nature of the relationships is somewhat subjective and might be interpreted slightly differently by different users once the data A, B, and C are known; nonetheless, both relationships are hierarchical, and this fact is all that should be required for the user to specify, query, and manipulate the relationships and the data. More will be said about this and more examples will be given in later chapters.

Having limited the type of relationship, it remains to determine how to specify that this relationship exists or does not exist between data, how to query the data base in terms of the relationships which do or do not exist, and how to manipulate these relationships and so indirectly the data. The manner in which these operations should be specified should be simple and of high pedagogic and mnemonic value in order that the goals achieved in BASIC can be achieved here as well. Once these goals are met, the resulting system will be able to serve as both a data management system and an information retrieval system which is easy to use and easy to learn. The following chapters discuss and explain one way that this can be done.

CHAPTER II

THE NATURE AND ADDRESSING OF THE DATA STRUCTURE

A. The Data Structure

The data structure type first chosen to represent the hierarchical relationships discussed in the previous section was a multirooted multibranching tree with the arcs of the tree oriented away from the roots (see Fig. 1a). The nodes of the tree contain the data items and a directed path from node X to node Y in the tree indicates that node X is in a hierarchically superior relationship to node Y. Stated differently, node Y is within the hierarchical context of node X. If no directed path exists between node X and a node Z, then no hierarchical relationship exists between node X and node Z. Notice that node X and node Y do not have to be adjacent to one another, i.e., other nodes may exist along the path from node X to node Y.

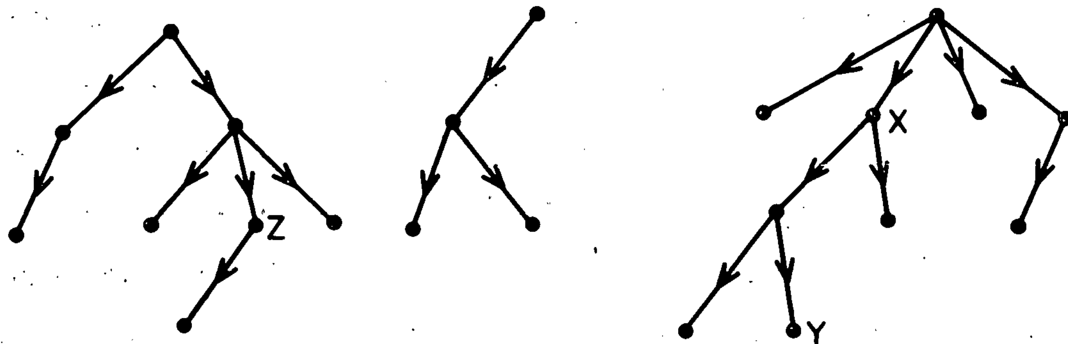
For reasons of generality, the multirooted multibranching tree data structure was extended slightly to directed acyclic graphs. This data structure can be conveniently visualized as a multirooted multibranching tree, some of whose branches might have grown together (see Fig. 1b). The nodes of the directed acyclic graph still contain the data items, and what was said previously about the existence or nonexistence of a hierarchical relationship between two nodes still holds.

B. The Data Item

As previously noted, each node of the graph contains a data item. A data item is any semantically meaningful label or set of labels the user chooses. Syntactically,

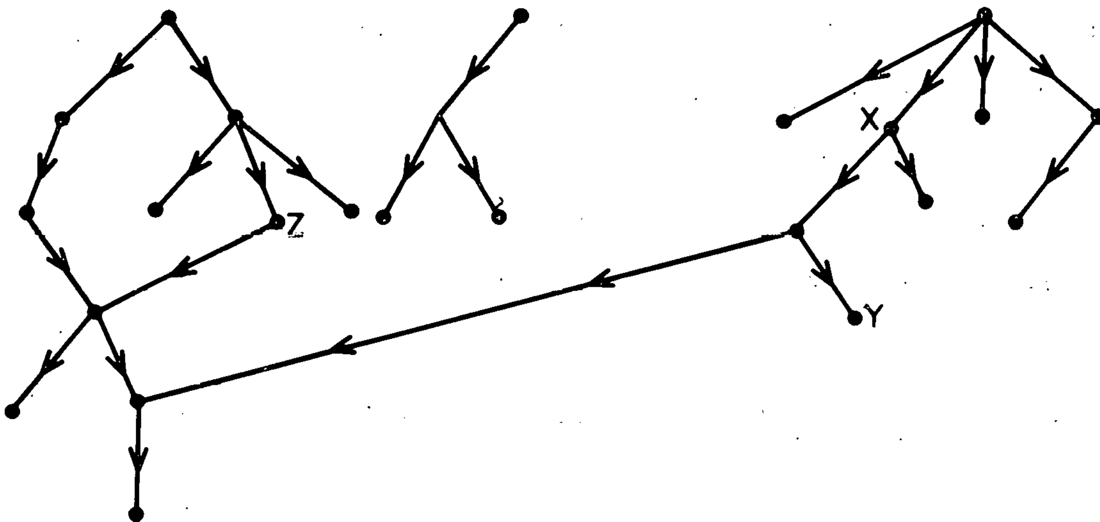
<label> : : = <identifier>|< number>

<data item> : : = <label>|< label> <data item>



Directed multirooted multibranching tree

(a)



Directed acyclic graph

(b)

1594A1

FIG. 1

where the syntactic classes <identifier> and <number> are as defined in the Algol 60 report (Revised Report 1963). Duplicate labels may occur within a data item.

In some complex retrieval systems, there is a syntactic distinction made between the semantically different concepts of a category of some kind and a particular instance of that category when referring to data items at nodes in a data structure. For example, "University" can be thought of as a category of which "Stanford" and "Texas" are instances. The system does not distinguish between labels which can denote categories and labels which can denote instances. The distinction between the concepts of category and instance is fairly easy to make for professionals in the computer field. However, for the potential user of this system, these concepts and their distinction may appear to be somewhat arbitrary and beside-the-point, complicating rather than simplifying the use of the system. For this reason, there is no syntactic distinction made between category and instance within data items in this system, and whatever semantic distinction exists between the labels which make up a data item can remain completely within the mind of the user.

Examples

Stanford	Lawyer Jones
Stanford University	Doctor Lawyer Jones
27	salary
age 27	salary 375.60 dollars per month

C. The Range

One of the basic concepts in this system is the manner whereby a subset of the set of all nodes in the graph, called a range, is referenced. Most of the

primitives operate on one or more subsets of nodes, or ranges, and each reference to a range is accomplished according to a common set of rules. There is a general principle, the Principle of Greater Specification, which applies when specifying a range. This principle states that when more information is given to specify a range, the cardinality of the range, i.e., the number of nodes referenced, cannot increase because of the added information; more usually, the cardinality decreases. Simply stated, the more carefully a set of nodes is described, the fewer nodes one is describing since only those nodes which satisfy all the descriptions are included in the range. As the different methods for referencing a range are discussed, it will be shown how the Principle of Greater Specification applies. It will also be seen that each method is a special case of following methods.

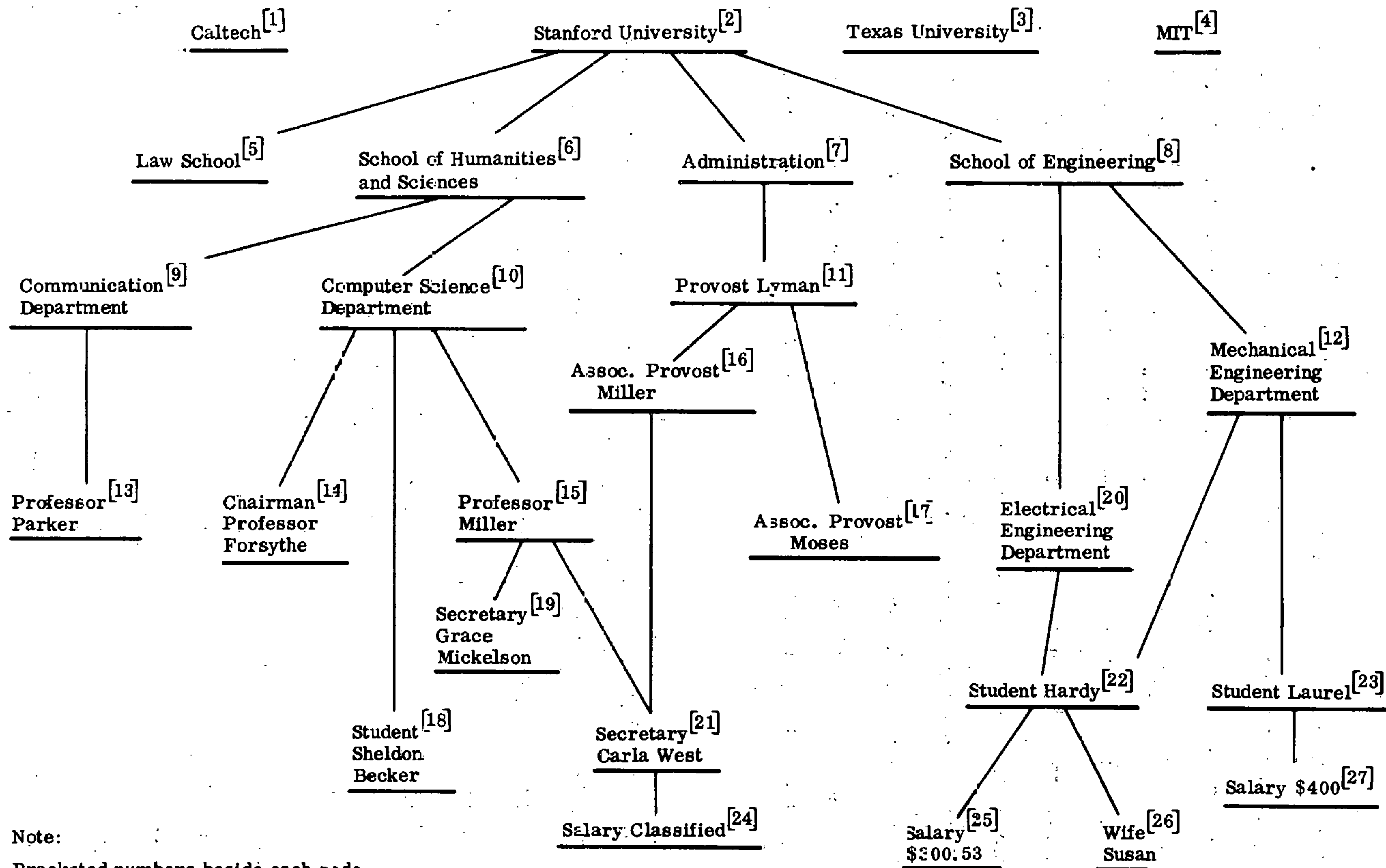
1. Method I

The simplest way of specifying a range is by evoking a label. The set of nodes which constitutes the range is then all those nodes in the graph which have the evoked label among the labels which make up the data item for that node.

Examples (see Fig. 2)

Evoked Label	Node numbers in range
Student	{18, 22, 23}
Provost	{11, 16, 17}
Miller	{15, 16}
University	{2, 3}
History	ϕ
Moses	{17}

The nodes in any range always form an unordered set.



Note:

Bracketed numbers beside each node are for identification purposes only and are not part of the data item at the node.

FIG. 2--A Personnel file.

2. Method II

Another way of specifying a range is by evoking one or more labels. The set of nodes which constitutes the range is then the intersection of all those sets of nodes (ranges) which would arise if each evoked label were evoked alone. The evoked labels can be in any order. If only one label is evoked, it is seen that this method of specifying a range is identical with Method I.

Examples (see Fig. 2)

Evoked Labels	Node numbers in range
Professor Miller	{15}
Provost Miller	{16}
Assoc. Provost Miller	{16}
Engineering Department	{12, 20}
Chairman Miller	∅
School of Humanities and Administration Sciences	∅
Miller Professor	{15}
Student Sheldon Becker	{18}
Student Sheldon	{18}
Student Becker	{18}
Sheldon Becker	{18}
Student	{18, 22, 23}
Sheldon	{18}
Becker	{18}

Notice that the Principle of Greater Specification holds here. The evoked label "Student" specifies a range consisting of three nodes (see Fig. 2); upon greater specification, "Student Hardy" for example, the range is reduced to one node.

There is an addition to Method II which can be used when the range which the user wishes to specify consists of a set of nodes which have precisely the set of labels the user evokes, and in precisely the same order. In the usual case, the range consists of the set of nodes which have at least the set of labels the user evokes, and in any order. The word "just" occurring before the evoked labels has pedagogic value in making it clear to the user that only those nodes are sought which contain precisely the evoked set of labels. "Precisely" or "exactly" might also be used.

Examples (see Fig. 2)

Evocation	Node numbers in range
just Student	∅
just Student Hardy	{22}
Student Hardy	{22}

3. Method III

A more complex and more powerful method of specifying a range is by hierarchical context. As was stated at the beginning of this chapter, since the data base is a directed acyclic graph, between any two nodes in the graph exactly one of the following two relationships holds:

(1) there is no directed path between the two nodes

(2) there are one or more directed paths between the two nodes.

Node Y is said to be within the hierarchical context of node X if one or more directed paths exist from node X to node Y. If no directed path exists between node X and node Y, then neither node is within the hierarchical context of the other node. Since the graph is acyclic, node X may never be within its own hierarchical context.

Let the symbol ">" indicate that the hierarchical relationship holds, i.e., $X > Y$ means that node Y is within the hierarchical context of node X. $X \ngtr Y$ means that node Y is not within the hierarchical context of node X.

<u>Examples</u> (see Fig. 2)

[8] $>$ [20]

[20] $\ngtr$ [8]

[8] $>$ [22]

[20] $>$ [22]

[20] $\ngtr$ [23]

[6] $>$ [19]

[7] $\ngtr$ [19]

[7] $\ngtr$ [7]

The hierarchical relation is nonreflexive, antisymmetric, and transitive.

The third method of range specification is accomplished by specifying two ranges in order according to Method II. The range thus specified consists of all those nodes in the first range which are within the hierarchical context of any node in the second range. More precisely, if R_1 and R_2 are the two ranges initially specified, then the range R within the hierarchical context of R_1 with respect to R_2 is defined by:

$$R = \left\{ y \middle| y \in R_1 \wedge \mathscr{F} x \in R_2 \ni x > y \right\}$$

In the preceding chapter, it was stressed that simplicity for the user be a primary goal. Reviewing the second method of specifying ranges, it can be seen that the data base is being addressed directly in terms of labels which the user has placed there (the ways in which this placement occurs will be described in the next chapter). Since these labels are purely the user's invention, they are semantically

meaningful to him. By allowing him to use these labels to address the data base, simplicity for him is thereby furthered.

Continuing in this spirit, the following manner of evoking ranges to be specified by Method III is suggested:

$$R_1 \text{ within } R_2$$

where R_1 and R_2 are evocations of the two ranges by Method II, in order. The word "within" is used as a delimiter, suggesting the hierarchical contextual relationship. Depending on how the user visualizes the data base, delimiters such as "in" or "under" might be used.

Examples (see Fig. 2)

Evocation	Node numbers in range
Student within Computer Science	{18}
Student within Department	{18, 22, 23}
Professor within Stanford	{13, 14, 15}
Professor within Humanities School	{13, 14, 15}
Professor within Communication	{13}
Professor within Administration	∅
Miller within Stanford	{15, 16}
Miller within Administration	{16}
Professor Miller within Administration	∅
Provost within Administration	{11, 16, 17}
Provost within Provost	{16, 17}

A natural and useful extension of specifying ranges by context is to specify a set of nodes not within a given context. More precisely, if R_1 and R_2 are the two ranges initially specified, then the range R not within the hierarchical context

of R_1 with respect to R_2 is defined by:

$$R = \left\{ y \mid y \in R_1 \wedge \forall x \in \bar{R}_2,\ x \not> y \right\}$$

The most natural extension for evoking ranges specified in this manner is to use "not within" as the delimiter between the evocation of ranges R_1 and R_2.

Examples (see Fig. 2)

Evocation	Node numbers in range
Student not within Computer Science	$\{22,\ 23\}$
Student not within Department	$\emptyset$
Professor not within Stanford	$\emptyset$
Miller not within Administration	$\{15\}$
Professor Miller not within Administration	$\{15\}$
Provost not within Administration	$\emptyset$
Provost not within Provost	$\{11\}$

4. Method IV

As Method II is a generalization of Method I, so Method IV is a generalization of Method III. In the preceding method, a first set of nodes is chosen by specifying a second set of nodes as context; the second set of nodes modifies the first set. Method IV allows a third context to be specified for the second set, a fourth context to be specified for the third set, etc.

In the general case, n ranges $R_1,\ R_2,\ \ldots,\ R_n$ in order are specified by Method II; $n \geq 2$. The ranges are associated in the following manner:

$$(R_1,\ (R_2,\ \ldots\ (R_{n-2},\ (R_{n-1},\ R_n))---)$$

Method III is first applied to the ordered pair of ranges R_{n-1} and R_n. The result of this application is a range, call it $R_{n-1,n}$. Method III is then applied

to the ordered pair of ranges R_{n-2} and $R_{n-1,n}$, resulting in a range $R_{n-2,n-1,n}$. Method III is continually reapplied to successive pairs of ranges until it is finally applied to the ordered pair of ranges R_1 and $R_{2,3,\ldots,n-1,n}$. The result of this final application is either null or a subset of the range specified by R_1, a range specified by successive hierarchical contexts. Notice that for n = 2, Method IV becomes Method III, and for the degenerate case of n = 1, Method II.

The natural extension for evoking ranges by successive hierarchical context is to evoke the n ranges by Method II, each evocation delimited by "within" or "not within":

$$R_1 \; d_{1,2} \; R_2 \; d_{2,3} \; R_3 \text{ ---- } d_{n-1,n} \; R_n$$

where R_i is the evocation of the ith range by Method II and $d_{i-1,i}$ is either "within" or "not within."

<u>Examples</u> (see Fig. 2)

Evocation	Node numbers in range
Secretary within Miller within Administration	{21}
Secretary within Miller within Computer Science	{19, 21}
Secretary within Miller not within Administration	{19, 21}
Secretary within Miller not within Computer Science	{21}
Student within Mechanical Engineering not within Humanities within Stanford	{22, 23}

Notice that the Principle of Greater Specification holds here. Also notice that the data base is still being addressed in terms of labels which are semantically meaningful to the user and in a manner which is very suggestive of the relationships which the user visualizes as holding between his data.

D. Total Range Specification

Since Method IV is a generalization of all the preceding methods, and the distinction between the methods will often not be needed, the combination of all the methods, i.e., Method IV, will henceforth be called the specification of a range by hierarchical context. Observe, however, that the contexts which are given are always hierarchically superior to the nodes which are being specified. In terms of visualization of the data base as drawn, for example, in Fig. 2, the contexts are always "above" the nodes which are being specified. Often, it is useful to be able to further qualify the nodes to be specified by looking at those nodes "below" the nodes which are being specified, i.e., those nodes which are hierarchically inferior to the nodes being specified.

To specify a range then, a set of nodes is first specified by hierarchical context. If it is not desired to further qualify the nodes so chosen by hierarchical context, then this set of nodes is the range. If it is desired to further qualify the nodes thus chosen by hierarchical context by checking for some condition or conditions which might exist in nodes hierarchically inferior to the nodes chosen by hierarchical context, this specification, to be described shortly and to be called specification by subtree context, is then given. It should be noted that specification by subtree context is specification by predicate, a well known method of naming sets. Specification by subtree context asks whether or not some condition holds within the subtrees of those nodes chosen by hierarchical context. Every node chosen by hierarchical context whose subtree meets the condition (or

conditions) named by the subtree context specification is retained in the range; all nodes which do not meet the condition (or conditions) named are discarded from the range.

E. Subtree Context Specification

Three conditions have been chosen to be used in the subtree context specification. One condition checks for the existence of a node (specified by Method IV) within the subtree. A second condition, which is a partial generalization of the first, checks for the existence of a given number of nodes (specified by Method IV) within the subtree. The third condition checks for the existence of a node, one of whose labels is numeric and the value of which is compared to a given number, within the subtree.

Within the subtree context specification, any of the three conditions can be evoked, or any combination of the three conditions separated by the logical connectives AND and ØR can be evoked. The unary logical operation NØT is built into the conditions and need not be explicitly provided.

The evocation of a range, therefore, consists of a specification by hierarchical context optionally followed by a specification by subtree context. If the latter is present, the hierarchical and subtree specifications are separated by the delimiter "wherever."

In BNF:

<range> ::= <hierarchical context specification>|
<hierarchical context specification>
wherever <subtree context specification>

1. Condition I

As stated previously, Condition I allows the user to check for the existence of a particular node, or hierarchical configuration of nodes, one of

which must be within the subtree of the node chosen by hierarchical context. Method IV is used to specify a set N of nodes. Every node which is specified by hierarchical context and which has within its proper subtree (i.e., hierarchically inferior to it) at least one node in N is retained in the range. Every node which is specified by hierarchical context but which does not have within its proper subtree at least one node in N is not considered to be in the range.

What is required for Condition I, then, is a specification of a temporary range N by Method IV, the nodes of which are then sought within the subtrees of the nodes specified by hierarchical context. If the latter set of nodes is called M, then the range is given by:

$$R = \left\{ x \middle| x \in M \wedge \mathscr{F} y \in N \ni x > y \right\}$$

In the evocation of a range using any of the three conditions (which all use Method IV to specify the set N), it is necessary to make clear to the user that a condition is sought, the result of which is essentially "yes" or "no", i.e., a predicate. To this end, the word "is" is inserted before the first occurrence of the word "within" in the Method IV specification used in the subtree context condition. Observe that since Method II is a degenerate form of Method IV not involving the use of the word "within," the above word insertion is not always able to be done. (See the third and last example below.)

Examples (see Fig. 2)

Evocation	Node numbers in range
Department within Humanities wherever Secretary Mickelson is within Professor Miller	{10}
Provost wherever Carla West is within Provost	{11, 16}
Provost wherever Carla West	{11, 16}

Assoc. Provost wherever Carla West is within Provost	{16}
Department within Stanford wherever Secretary is within Department	{10}
Department within Stanford wherever Secretary	{10}

Ranges including the negation of Condition I can also be specified. In this case, every node which is specified by hierarchical context and which does not have within its proper subtree at least one node in N is retained in the range. Every node which is specified by hierarchical context which has within its proper subtree at least one node in N is not considered to be in the range. More formally:

$$R = \{x | x \in M \wedge \forall y \in N, \quad x \not\succ y\}$$

The evocation of a range using this form of Condition I is accomplished by inserting the words "is not" before the first occurrence of the word "within" in the Method IV specification in the subtree context condition. The meaning of the word "not" following "is" essentially has the meaning "it is not the case that . . . " and should not be confused with the use of the word "not" first described in the discussion of Method III. For example, in the first example below,

N = {21} - subtree context

M = {9, 10} - hierarchical context

Examples (see Fig. 2)

Evocation	Node numbers in range
Department within Humanities wherever Secretary West is not within Professor within Department	{9}
Department within Stanford wherever Secretary is not within Department	{9, 12, 20}

2. Condition II

Condition II allows the user to check for the existence of a given number of nodes of a certain specification (by Method IV again) within the subtrees of those nodes specified by hierarchical context. What is required then is a temporary range N specified by Method IV, a relational operator P(e.g., =, ≠, >), and a number, q. Every node in M(i.e., those nodes specified by hierarchical context) which has within its subtree a number of nodes in N which stand in the given relation P to the given number q is retained in the range. Any node in M which does not have within its subtree a number of nodes in N which stands in the given relation to the given number is not considered to be in the range. More formally, if C is the set cardinality operator, P is the given relational operator, and q is the given number, then

$$R = \left\{ x \middle| x \in M \wedge \forall y \in N,\ x > y \wedge C(N)\ Pq \right\}$$

The evocation requires some word which denotes that a cardinality is being considered. The word "count" has been chosen both because of its inherent semantic content and because of its use in one of the primitives which is explained in the next chapter.

Examples (see Fig. 2)

Evocation	Node numbers in range
Department wherever count student ≤ 3	{9, 10, 12, 20}
Department wherever count student within Department = 2	{12}

3. Condition III

Condition III allows the user to check for the existence of a node, at least one of whose labels is numeric and the value of which stands in a given relation

P to a given number q. This condition can be used, for example, to check the values of ages or salaries in a personnel file.

As with the first two conditions, a temporary range N is specified by Method IV. Every node in M (i.e., those nodes specified by hierarchical context) which has within its subtree a node in N which has at least one label which is numeric and whose value stands in the given relation P to the given number q is retained in the range. If a node has more than one numeric label, only the first will be considered. Any node in M which does not have within its subtree a node in N which has at least one label which is numeric and whose value stands in the given relation P to the given number q is not considered to be in the range. More formally, if V is the value operator, i.e., a function whose argument is a numeric label and whose value is the value of the label, and if U is a predicate whose argument is a label and whose value is true if and only if the label is numeric, then the range R is defined by

$$R = \left\{ x \mid x \in M \wedge \mathscr{F} y \in N \ni (x \succ y \ \wedge \ \mathscr{F} \ell \subset y \ni (U(\ell) \wedge V(\ell)\, P q)) \right\}$$

Examples (see Fig. 2)

Evocation	Node numbers in range
Student within Stanford wherever Salary > 250	{22, 23}
Student within Stanford wherever Salary > 350.25	{23}
School wherever Salary within Professor > 1000	∅
School wherever Salary within Student > 300	{8}

F. Conclusion

It can now be seen that there are many ways to specify a range, and that the Principle of Greater Specification applies within each of the methods and conditions as well as over all of them. The user need only specify as little context as is required to choose those nodes toward which he wishes to draw attention. Greater specification can only reduce the number of nodes towards which he is drawing attention. In addition the data base is being addressed by labels which have semantic content to the user, in conjunction with English language words and forms which appear to have high semantic content with respect to the data base attention focusing which occurs.

Thus far, no attempt has been made to explain how ranges are used once they have been specified. The next chapter explains the use of the primitives which build and manipulate the data base. These primitives operate on ranges as specified by the rules explained in this chapter.

CHAPTER III

DESCRIPTION OF THE PRIMITIVES

The preceding chapter described methods whereby a range, i.e., a set of nodes in the data base, may be specified. Specifying a range simply focuses attention on a particular set of nodes; no nodes are added or deleted from the data base nor are any connections between nodes altered. In this chapter, a set of primitives for adding and deleting nodes and altering connections between nodes will be explained. Which nodes are affected by the primitives is determined by specifying as many ranges as each primitive requires.

It was seen in the preceding chapter that attention is focused on a set of nodes by citing labels which the user has specified and therefore have semantic content for him along with English words which are highly suggestive of the relationships which exist among the user's data. Continuing in this vein of making the language and system easy for the user to learn and use, it will be seen that each primitive is easily identifiable by an English keyword which is highly suggestive of the effect the primitive has on the data base. Table 1 lists the primitives by their keyword and gives the use of each.

The primitives are, of course, not absolutely primitive. There is a continuum of primitiveness, and a choice of what part of the continuum from which to choose any system's primitives must be made. The choice depends on the use to which the primitives will be put. If the primitives are too primitive, too many steps will be necessary to do any useful work. On the other hand, if the primitives are too general, the control over the structure being manipulated by the primitives will not be fine enough. These considerations have been taken into account when choosing the point along the continuum from which the primitives given in Table 1 were taken.

TABLE 1

THE PRIMITIVES

PRIMITIVE	USE
CREATE	Creates new nodes in the data base
LABEL	Adds labels to nodes in the data base
UNLABEL	Removes labels from nodes in the data base
WRITE	Writes part of the data base
COUNT	Counts nodes in the data base
PUT	Builds relationships in the data base
COPY	Copies nodes and relationships and builds relationships in the data base
SEVER	Destroys relationships in the data base
DELETE	Destroys nodes and relationships in the data base
SAVE	Saves part of the data base in secondary storage
RESTORE	Restores part of the data base from secondary storage

A. CREATE

The CREATE primitive adds a new node to the data base by giving the set of labels which the data item at the new node will contain. At least one label must be given, and the labels occur within the new data item in the order in which they are given. The syntax for the CREATE primitive is:

<create primitive> ::= CREATE <data items>

<data items> ::= <data item> | <data items> and <data item>

Notice that more than one data item may be created with one use of the CREATE primitive by separating the labels of the data items to be created by the word "and." The user may visualize the new nodes which are created as existing unattached as new roots of the graph in the data base. No connections are made or altered nor are any already existing nodes within the data base altered.

By this time the reader should observe that certain words are held in reserve status and recognized by the system as special delimiters. A complete list of these reserved words is given in Table 2. It is often the case that a user may want one or more of these words to occur as a label within one or more data items. Surrounding any word or set of words within a <data item> with quote marks causes the system not to treat any words within the quote marks as reserved words. In this way, any word may be included as a label within a data item.

Examples

CREATE Stanford University

CREATE MIT and Caltech

CREATE "A node containing all these words including the word create"

TABLE 2

RESERVED WORD LIST

ABOVE	LABEL
AND	NOT
AS	ØNLY
BEFORE	ØR
BELOW	PUT
CØPY	RESTØRE
CØUNT	SAVE
CREATE	SEVER
DELETE	TØ
FRØM	VALUE
INTØ	WHEREVER
IS	WITHIN
JUST	WRITE
	UNLABEL

See also Table 3 for the reserved words of BASIC.

All of the above words when not enclosed in quotes should be considered delimiters, including an "end of line" character which denotes the end of a command.

B. LABEL

The LABEL primitive allows the user to add new labels to already existing nodes. What is required is the specification of a set of nodes to which the labels will be added and the new set of labels. The set of nodes to which the labels will be added is given by specifying a range as described in the preceding chapter. The syntax for the LABEL primitive is:

<label primitive> ::= LABEL <range> as <data item>

The <data item> is the set of new labels. These new labels are added after the last label which existed at the node before the label primitive was evoked. The new set of labels is appended to every set of labels at all the nodes specified by the range.

If it is desired to insert a new set of labels at some point in the existing data item other than after the last label in the data item, the following alternative syntactic construction may be used:

<label primitive> ::= LABEL <range> as <data item>
before <label>

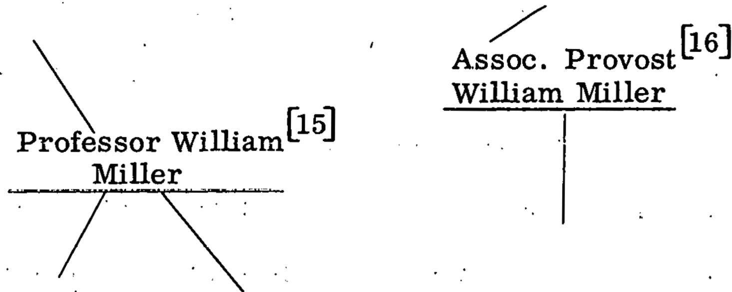
The new set of labels given by the <data item> will be inserted before the already occurring <label> in all the nodes specified by the <range> . If the already occurring <label> occurs more than once within some node in the <range> , the new set of labels given by the <data item> will be inserted before the first occurrence of the <label> . If the <label> does not already occur within some node in the <range> , the new set of labels is added at the end of the existing set of labels at that node, as in the previous construction.

Example (see Fig. 2)

Evocation:

LABEL Miller within Stanford as William before Miller

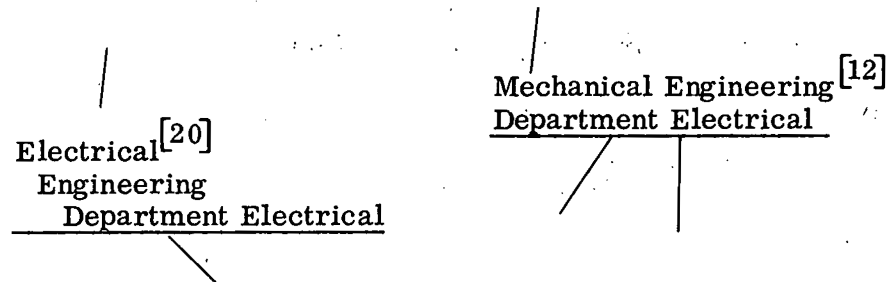
Effect:



Evocation:

LABEL Engineering Department as Electrical

Effect:



C. UNLABEL

The UNLABEL primitive allows the user to remove labels from existing nodes. As in the LABEL primitive, a set of nodes, specified by a range, along with the set of labels to be removed must be evoked. The syntax for this primitive is:

<unlabel primitive> ::= UNLABEL <range> as <data item>

All the nodes in the <range> are first identified, then all the labels in the <data item> are removed from each of these nodes. The labels in the <data item> are removed one by one and do not have to occur in the same order as they occur within the nodes in the <range> . Labels in the <data item> which do not already

exist at some node in the <range> cannot, of course, be removed. In addition, the last label cannot be removed from a node, thereby leaving a null data item. Observe that to completely relabel a node, the new labels should first be added using the LABEL primitive, then the old labels removed by using the UNLABEL primitive. If a label to be removed occurs more than once at some node in the <range> , only the first occurrence of that label will be removed. However, if a label to be removed occurs twice for example, within some node in the <range>, and that label occurs twice in the set of labels to be removed, then both occurrences will be removed.

Example (See Fig. 2)

Evocation:

UNLABEL Sheldon Becker as Student

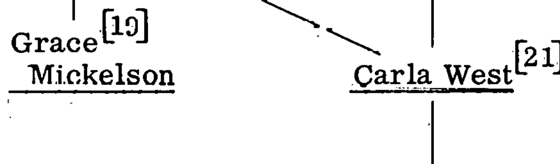
Effect:

Sheldon Becker[18]

Evocation:

UNLABEL Secretary within Computer Science as Secretary

Effect:



Evocation:

UNLABEL Electrical Engineering Department as Department Engineering

Effect:

Electrical[20]

Evocation:

UNLABEL Salary Classified as Salary Classified

Effect: Illegal

The preceding three primitives add new unattached nodes to the data base and add and delete labels from the data items at nodes specified by a range, respectively. The data base is altered by the use of these primitives at the data item level: no relationships between data are altered.

The following two primitives are useful for query purposes only. The data base is not altered in any way by the use of these primitives. Rather, an immediate response is typed out at the terminal.

D. COUNT

The COUNT primitive informs the user of the number of nodes in a given range, i.e., the cardinality of the range. The data base is not altered in any way by the use of this primitive.

The syntax for this primitive is:

<count primitive> : := COUNT <range>

Example (see Fig. 2)

Evocation:

COUNT Student within Stanford

Response: 3

Evocation:

COUNT Department within Stanford wherever Professor is within Department

Response: 2

Evocation:

COUNT Professor within Engineering School

Response: 0

E. WRITE

The WRITE primitive causes a part of the data base to be printed in an outline format with proper indentations to denote the various hierarchical levels. For each node, the labels in the data item at the node are printed in the order in which they occur within the data item.

The syntax of the first form of the WRITE primitive is:

<write primitive> ::= WRITE <range>

Every node in the <range>, with all the subtrees of each node properly indented, is printed. The subtrees at each level are printed in an arbitrary order. If there are no nodes in the specified range, then an indication of this fact is printed.

Example (see Fig. 2)

Evocation:

WRITE Department within Humanities

Response:

```
Communication Department
    Professor Parker
Computer Science Department
    Student Sheldon Becker
    Professor Miller
        Secretary Grace Mickelson
        Secretary Carla West
            Salary Classified
    Chairman Professor Forsythe
```

Evocation:

WRITE Professor within Electrical Engineering

Response:

Null range

If the data base has the form of a tree, then an outline, as demonstrated above, with no duplications within the outline, would always result from the use

of the WRITE primitive. However, since the data base has the form of a directed acyclic graph, unnecessary printing of duplicate subtrees could result in response to one evocation of the WRITE primitive. To alleviate this unnecessary printing, only the root node of any subtree which would be printed the second or subsequent time in response to a single evocation of the WRITE primitive, along with an indication that the entire subtree has already been output, will be printed.

Example (see Fig. 2)

Evocation:

```
WRITE Miller
```

Response:

```
Professor Miller
    Secretary Grace Mickelson
    Secretary Carla West
        Salary Classified
Assoc. Provost Miller
    Secretary Carla West <occurs above>
```

Evocation:

```
WRITE School of Engineering
```

Response:

```
School of Engineering
    Electrical Engineering Department
        Student Hardy
            Salary $300.53
            Wife Susan
    Mechanical Engineering Department
        Student Hardy <occurs above>
        Student Laurel
            Salary $400
```

Evocation:

WRITE Provost

Response:

```
Provost Lyman
    Assoc. Provost Moses
    Assoc. Provost Miller
        Secretary Carla West
            Salary Classified
Assoc. Provost Moses <occurs above>
Assoc. Provost Miller <occurs above>
```

It is sometimes desirable to print only the nodes in the range without their subtrees. The syntax of this second form of the WRITE primitive is:

<write primitive> ::= WRITE only <range>

<u>Example</u> (see Fig. 2)

Evocation:

WRITE only University

Response:

Texas University
Stanford University

Evocation:

WRITE only Student within Stanford

Response:

Student Laurel
Student Hardy
Student Sheldon Becker

Thus far, the primitives which have been introduced do not alter the relationships between data. The following four primitives build and destroy the hierarchical relationships between data.

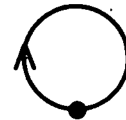
F. PUT

The PUT primitive builds hierarchical relationships between existing nodes. Two ranges are specified, and every node in the first range is made to be one level hierarchically inferior to every node in the second range, subject to two restrictions. Thus, if there are n nodes in the first range, and m nodes in the second range, then $n \times m$ hierarchical relationships are formed if none of the restrictions are violated. The restrictions are:

1) No more than 1 direct (i.e., one level) hierarchical relationship may exist between any two nodes. That is, the following situation may not occur:

However, the following is legal:

2) No node may be hierarchically inferior (or superior) to itself. That is, the following situations may not occur:



For every possible pair of nodes in the first and second ranges, respectively, a direct hierarchical relationship is built so long as none of the restrictions are violated. Under no conditions are any existing relationships altered in any way. The syntax for the PUT primitive is:

<put primitive> ::= PUT <range> $_1$ into <range> $_2$

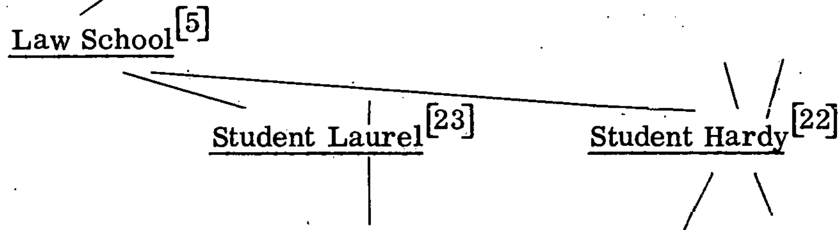
As suggested by the syntax, except when the restrictions would be violated, a direct (one level) hierarchical relationship is built from every node in <range> $_2$ to every node in <range> $_1$.

Example (see Fig. 2)

Evocation:

PUT Student within Engineering into Law School

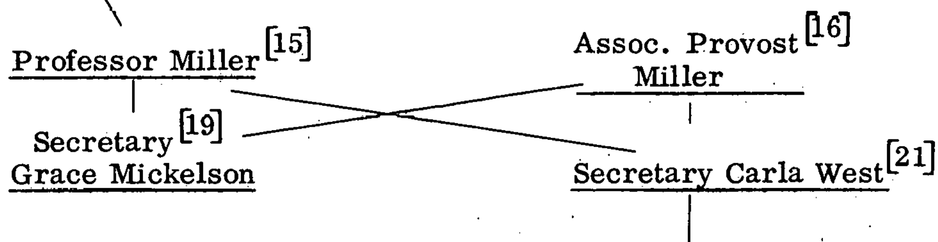
Effect:



Evocation:

PUT Secretary into Provost Miller

Effect:



A shorthand combination of the CREATE and PUT primitives is useful while building data bases. If <range>$_1$ is specified by Method II of the preceding chapter, i.e., without any hierarchical or subtree context, and if that range is null, then a node having the given set of labels will first be implicitly CREATE'd and a message output to the user that this creation has occurred. The PUT operation will then proceed as described.

Example (see Fig. 2)

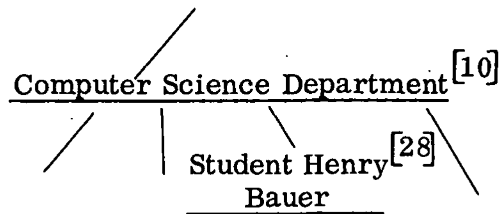
Evocation:

PUT Student Henry Bauer into Computer Science

Response:

Student Henry Bauer Created

Effect:



G. COPY

It is sometimes desirable to be able to copy part of the data base so that further processing may be done on the copy without disturbing the original. The COPY primitive gives the user this capability. This primitive has three syntactic forms, the first of which is:

<copy primitive> : := COPY <range>

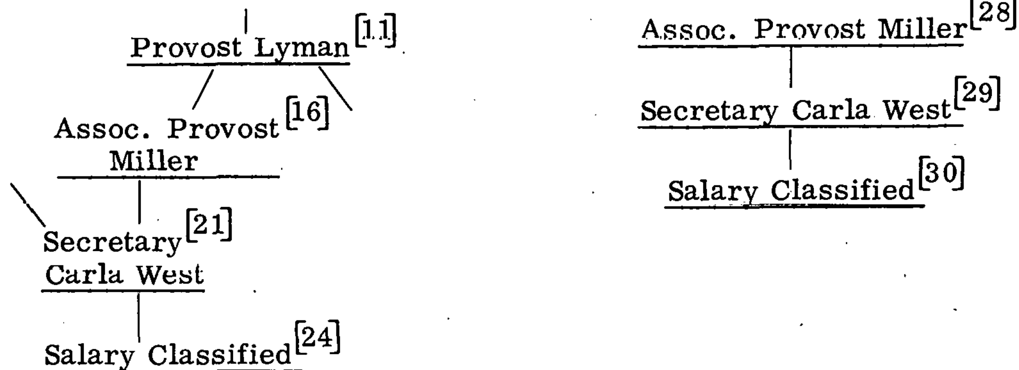
A copy of each node in the <range>, along with its entire subtree complete with all the relationships which exist there, is made. These copied nodes, with their subtrees, are left unattached as roots in the data base. The original nodes and their subtrees are not altered in any way.

Example (see Fig. 2)

Evocation:

COPY Provost Miller

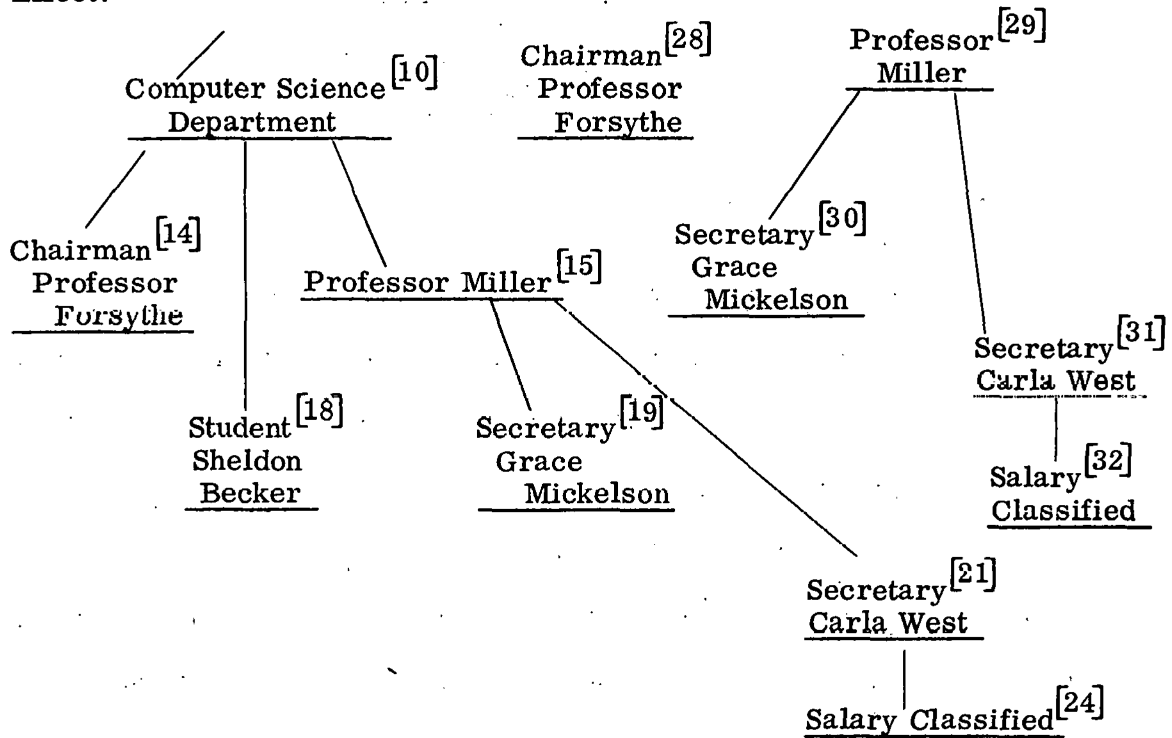
Effect:



Evocation:

COPY Professor within Computer Science within Stanford

Effect:



The second form of the COPY primitive allows the user to make a copy and PUT the copy somewhere into the hierarchy. More specifically, two <range>'s are specified. A copy of each node in the first range (along with its subtree and all connections intact) is made and PUT into the hierarchy for each node in the second <range> . That is, as many copies of the first range are made as there are nodes in the second range into which the copies are PUT. This second form of the COPY primitive is thus a shorthand combination of (possibly) several applications of the first form of the COPY primitive and PUT primitive operating on the copy.

The syntax is:

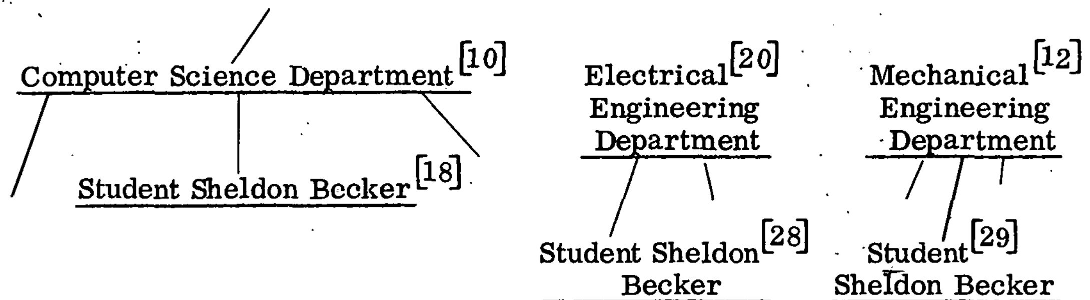
<copy primitive> ::= COPY <range> to <range>

Example (see Fig. 2)

Evocation:

COPY Student within Computer Science to Department within School of Engineering

Effect:



The third variation of the COPY primitive is, in reality, an addition which can be made to the first two forms. As suggested by the form of the WRITE primitive which allows the user to write out only the root nodes of certain subtrees by including the keyword "only" in the evocation of the primitive, only root nodes of specified subtrees can be copied, and in addition entered into the hierarchy if desired. The syntax of this variation of the first two forms of the COPY primitive is:

<copy primitive> ::= COPY only <range>

<copy primitive> ::= COPY only <range> to <range>

In the first case, only the nodes specified by the <range>, without their subtrees, are copied and the copies are left unattached as roots in the data base. In the second case, as many copies of the nodes specified by the first <range> (without their subtrees) as there are nodes in the second <range> are made and

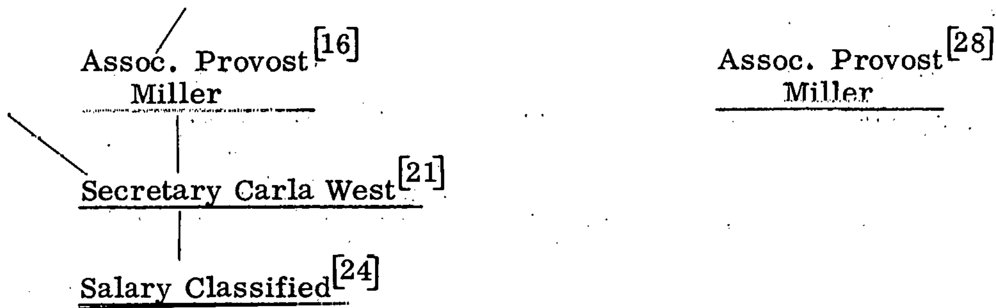
the copies are PUT into the nodes in the second <range>. In both cases there is no alteration of any kind made to the original nodes or their subtrees.

Example (see Fig. 2)

Evocation:

COPY only Provost Miller

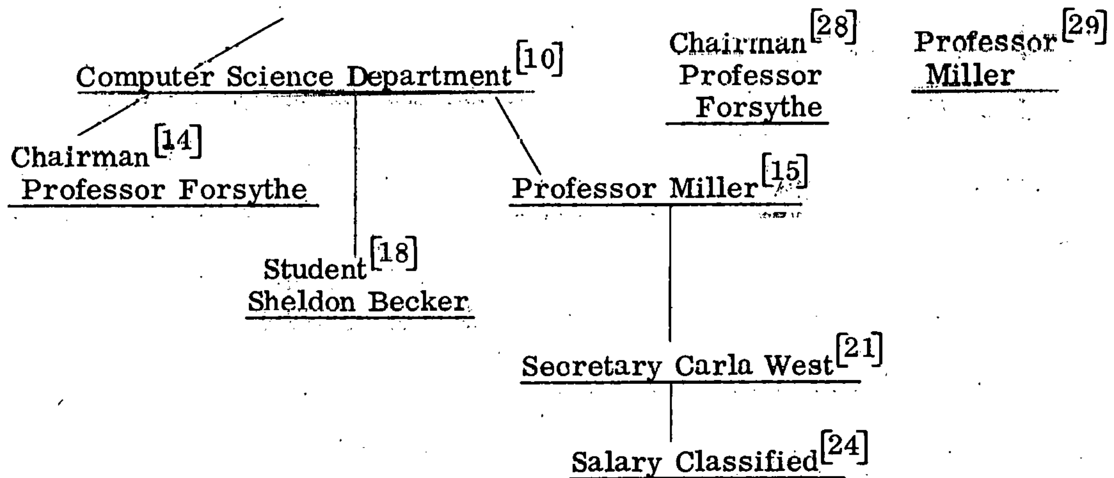
Effect:



Evocation:

COPY only Professor within Computer Science Department within Stanford

Effect:



Evocation:

COPY only Student within Computer Science to Department within School of Engineering

Effect: Same as a preceding example with "only" omitted, since all students within Computer Science have no subtrees.

The preceding two primitives, PUT and COPY, build new relationships between data and, in the latter case, implicitly create new data. The following two primitives, SEVER and DELETE, destroy relationships between data, and, in the latter case, destroys data as well.

H. SEVER

The SEVER primitive destroys relationships between data but never data itself. As with the DELETE primitive to follow, the SEVER primitive has two forms. In both forms, at least one range is specified. In the first form, all nodes in the specified range are made to have no nodes hierarchically superior to them; that is, those nodes are SEVER'ed from the tree and become roots of the tree. The subtrees of the nodes which are so cut off from above are left undisturbed unless some of the nodes within them are also being severed. The syntax of this form is:

<sever primitive> ::= SEVER <range>

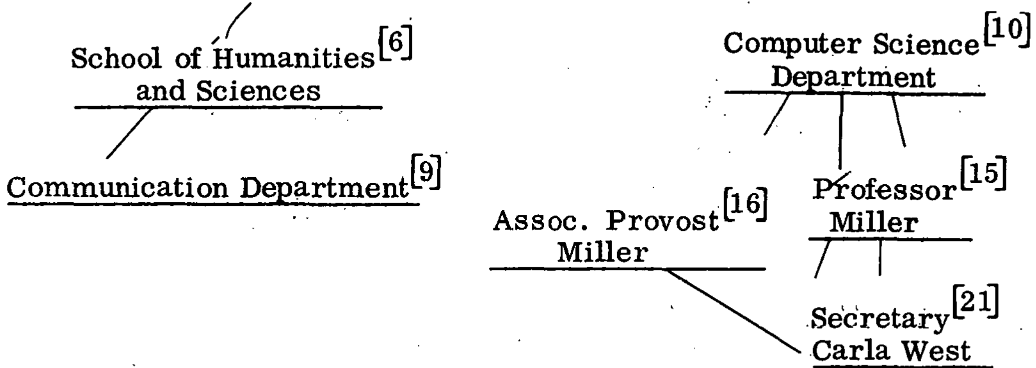
After this primitive has been evoked, all nodes in the specified <range> are roots of the tree, there being no nodes in the data base within which they exist.

Example (see Fig. 2)

Evocation:

SEVER Computer Science within Stanford

Effect:

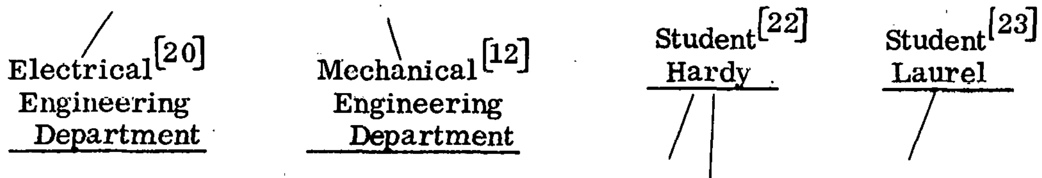


Observe that node [10] is now a root of the tree. Yet, a node in its subtree, i.e., node [21] remains attached to node [16]. Recall that no relationships within the subtree of a severed node are altered.

Evocation:

SEVER Student within Engineering Department within Stanford

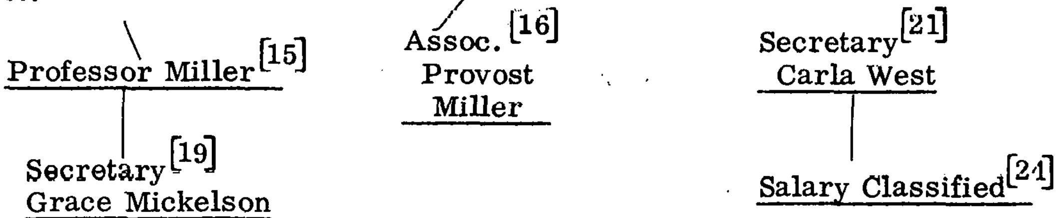
Effect:



Evocation:

SEVER Carla West

Effect:



In the first form of the SEVER primitive, all nodes in the range were completely severed from the tree; that is, all of the relationships which connected these nodes immediately from above the nodes were severed. The second form

of the SEVER primitive allows the user to selectively sever some of the relationships which connect the nodes immediately from above the nodes. The selection is accomplished by specifying a second range. Those connections immediately above a node in the first range are severed which causes that node to be within any node in the second range. Thus, nodes in the first range are severed from nodes in the second range. The syntax for this form is thus:

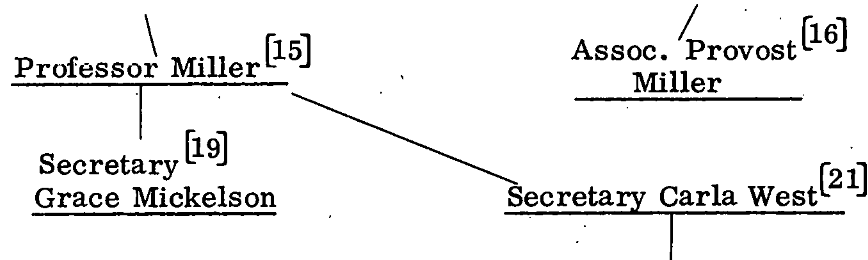
<sever primitive>::= SEVER <range> from <range>

Example (see Fig. 2)

Evocation:

SEVER Carla West from Assoc. Provost Miller

Effect:



Notice that only that relationship which causes node [21] to be within node [16] is broken.

Evocation:

SEVER Carla West from Stanford

Effect: Same as a previous example whose evocation was:

SEVER Carla West

since all connections immediately above node [21] cause node [21] to be within node [2].

Evocation:

SEVER Student within Computer Science from Electrical
Engineering Department

Effect: No effect since none of the connections above node [18] (the 1st range) causes node [18] to be within any node of the second range.

A unifying concept which may lessen any difficulty in understanding the distincition between the two forms of the SEVER primitive is the following. Think of the first form of the SEVER primitive as having a second range which specifies all the roots of the tree. The first form is thus a special shorthand version of the second form, since any node in the tree is either a root of the tree or is hierarchically inferior to some root of the tree.

I. DELETE

The DELETE primitive has two forms, precisely analogous to the SEVER primitive. The syntax of these forms is:

<delete primitive>::= DELETE <range>

<delete primitive>::= DELETE $\langle range \rangle_1$ from $\langle range \rangle_2$

As with the two forms of the SEVER primitive, the first form of the DELETE primitive is merely a shorthand version of the second form with an implicit range which specifies all the roots of the tree. Thus, in the action of this primitive, some node is being deleted from a set of nodes.

If a node x is in the first range and its position in the data base is such that x is also within the second range, then node x is a candidate for deletion. In order to understand the manner in which the DELETE primitive operates, several simplifying assumptions concerning the configuration of the data base near node x will first be made. As the operation of the DELETE primitive becomes clearer, these assumptions will be removed.

Consider first the simplifying assumption that node x together with its subtree is a complete unit, sharing its information with the rest of the data base only through, at most, node x (if x is a root, the information is not shared at all). More specifically, every node within the subtree of x is not within the subtree of any other nodes except other nodes within x's subtree or nodes hierarchically superior to x. Examples of such nodes x from Fig. 2 include [8], [2], [1], [22], and [21], but not [6] or [10] or [15] or [12] or [20].

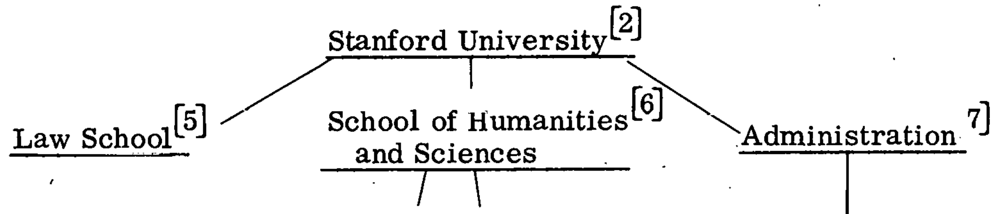
Now assume that <u>all</u> connections immediately superior to x cause x to be within the second range in an evocation of the DELETE primitive. The action of the DELETE primitive will then cause the node x and every node within the subtree of x to be erased from the data base.

<u>Example</u> (see Fig. 2)

Evocation:

DELETE School of Engineering from Stanford

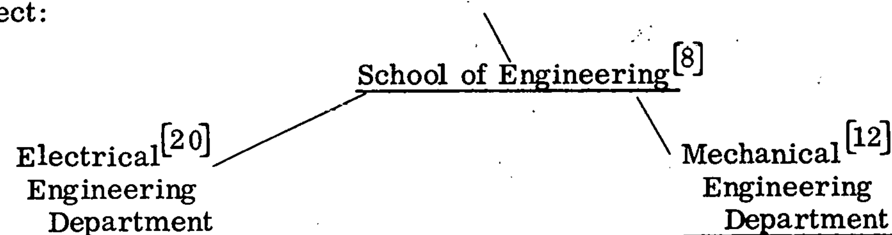
Effect:



Evocation:

DELETE Student within Engineering Department from School of Engineering

Effect:



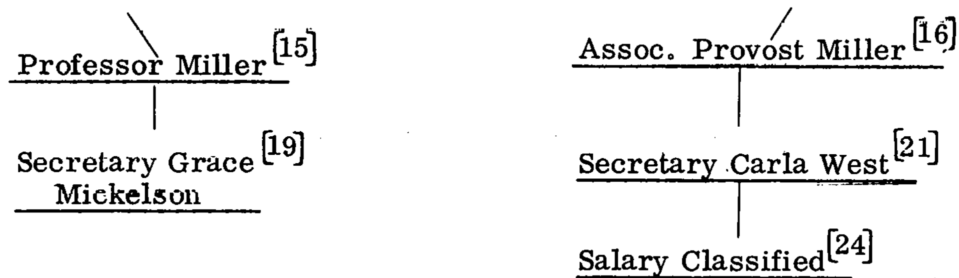
Now remove the most recent simplification and assume that not all connections immediately superior to x cause x to be within the second range in an evocation of the DELETE primitive. This means that x and its subtree contain information relevant to some other nodes in the data base besides the nodes from which x is to be deleted. It would thus be incorrect to erase x and its subtree from the data base; rather, only the connections between x and the nodes from which x is to be deleted should be erased. In this case, the DELETE primitive is seen to operate precisely as the SEVER primitive.

Example (see Fig. 2)

Evocation:

DELETE Carla West from Professor Miller

Effect:



Now remove the original simplification and assume that the subtree of x is not a complete unit and that information within x's subtree is shared with the rest of the data base through nodes other than x. More specifically, there exist nodes within the subtree of x which are within the subtree of nodes other than those within x's subtree or nodes hierarchically superior to x. Examples of such nodes x from Fig. 2 include [6], [10], [15], [12], and [20]. As before, it would be incorrect to erase those nodes (and their subtrees) within x's subtree which share common information with other parts of the data base, i.e., those nodes (and their subtrees) which are within the subtree of nodes other than those within x's subtree or nodes

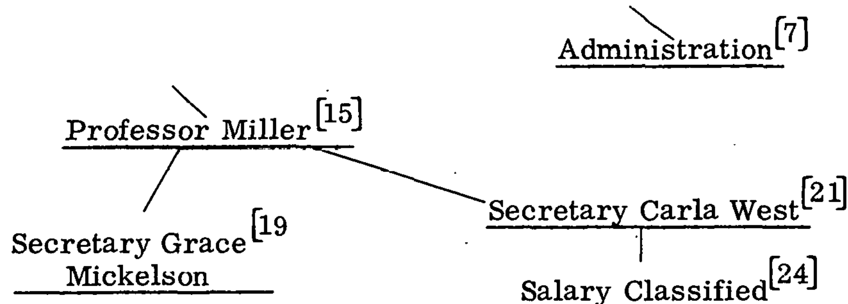
hierarchically superior to x. So, as before, only the connections between these common nodes and the rest of x's subtree is erased.

Example (see Fig. 2)

Evocation:

DELETE Provost Lyman from Administration within Stanford

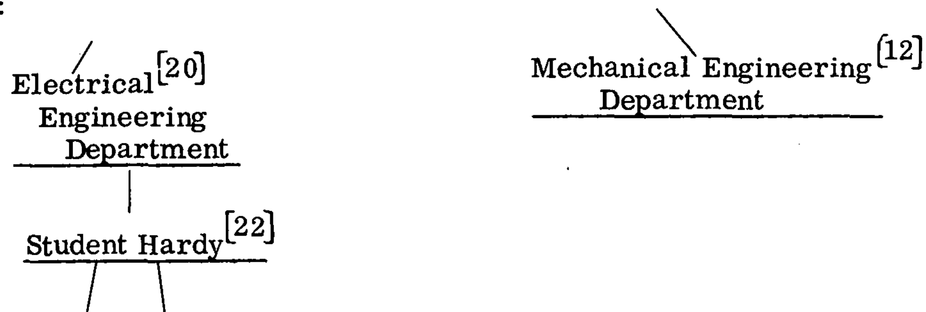
Effect:



Evocation:

DELETE Student from Mechanical Engineering

Effect:



The heart of any data base system is the ability of the user to create a data base, save it away somewhere (the user should not have to worry about where) and go away to do something else. At some later time, the user should be able to fetch his data base, query it or modify it, and save it away again for still further processing. As background to any implementation, therefore, there should be a file system. The following two primitives allow the user to interface with the file system in as easy and simple a manner as should be possible for him.

J. SAVE

The SAVE primitive allows the user to save parts of his data base in quantities of complete units as discussed in the preceding explanation of the DELETE primitive. The syntax for this primitive is:

<save primitive> : := SAVE <range>

The effect of an evocation of this primitive is to cause each node in the range with its entire subtree (nodes and connections) to be removed from the data base (just as with the DELETE primitive) but saved in such a way so that the structure which has been removed from the data base can be returned to the data base at some future time in precisely the same form in which it was saved.

The only restriction which applies to the use of this primitive is that the (sub) tree being saved must be a complete unit and cannot share its information with other parts of the data base except through its root node. Thus, each node and its subtree which is being saved must satisfy both simplifications mentioned in the explanation of the DELETE primitive. If part of the data base to be saved shares its information with other parts of the data base, the part to be saved must first be COPY'ed and then saved.

K. RESTORE

The RESTORE primitive restores nodes and their subtrees to the data base in precisely the form in which they were saved. This primitive has two forms, the second of which is a shorthand form for restoration and placement within the hierarchy. The syntax of the first form is:

<restore primitive> : := RESTORE <range>

The <range> in the use of this primitive must be a range specified by Method II of the preceding chapter. Thus, there may be no hierarchical context of any kind in the specification of this range. The set of saved nodes is searched and

the saved nodes having at least the set of labels as that specified in the <range> are removed (with their subtrees) from the saved area and restored to the working data base as separate trees with the root nodes of the saved subtrees becoming roots in the data base.

The second form of this primitive is:

<restore primitive> ::= RESTORE <range> to <range>

The first <range> must satisfy the same requirements as before and the same action occurs as before in the saved area. However, once the nodes are removed from the saved area, the action of this primitive is equivalent to an evocation of

PUT <range> within <range>

where the first <range> is the set of nodes just removed from the saved area, and the second <range> is the same as the second <range> in the evocation of the second form of the RESTORE primitive.

L. Conclusion

This chapter has described a set of primitives for creating, manipulating, querying, and destroying relationships and data within the data base. It should be observed that no arithmetic processing nor any programming structure is possible within the framework of primitives discussed thus far. The next chapter describes the design of a programming system in which TAXL and a language such as BASIC are incorporated to yield a system in which both numeric and nonnumeric data processing capabilities are available both independently and in a manner in which the numeric and nonnumeric data bases may interact.

CHAPTER IV

THE TAXL/BASIC SYSTEM

The range specification mechanism and its use in the evocation of the data base primitives presented thus far yields a system in which commands are interpreted as they enter the system and are executed immediately. In addition, no arithmetic capabilities have been introduced thus far. As was pointed out in the introduction, the BASIC programming language and system (Kemeny [1967]), designed by Kemeny and Kurtz at Dartmouth, provides an excellent easy-to-learn-and-use interactive computer system for arithmetic processing. Rather than design an arithmetic capability which would be included in TAXL, a design for merging a version of BASIC and TAXL will be given. The version of BASIC most nearly like that which will be discussed here is that version written at Hewlett-Packard, a system quite similar to the Dartmouth system. The reader is expected to be familiar with some BASIC system in the discussion which follows.

The degree of interaction available in this version of BASIC, as in most versions of BASIC, is different from the immediate interpretation and execution of the TAXL commands considered thus far. This difference should be well understood. The key to determining the degree of interaction of an on-line computer system is the specification of the data which is being manipulated.

Presumably, the data which the user would ultimately want to manipulate is the tree data base itself, along with the values of certain arithmetic variables and arrays. If the system recognizes commands to manipulate that collection of data, then that degree of interaction should be considered the strongest. Instead, if the system recognizes commands to manipulate commands which manipulate the

data base in question, then the degree of interaction is less strong. The Hewlett-Packard BASIC system is of this second kind. The data which is immediately manipulated by HP BASIC commands as they are entered via a teletype is a program buffer which contains commands which will manipulate the values of arithmetic variables and arrays when the program is executed. The commands found in the program buffer which, when executed, cause the values of variables and arrays to be manipulated and hereafter called BASIC commands, cannot be entered and executed directly as are the TAXL commands. It appears, then, that TAXL commands, as discussed thus far, are only executable directly as they are entered into the system, and that BASIC commands are only executable indirectly after they have been entered in a program buffer.

I feel that both degrees of interaction should be available for both TAXL and BASIC commands. That is, TAXL commands should be able to be put into a program buffer for later execution and certain BASIC commands should be able to be directly executable as they are entered into the system. In keeping with the spirit of this work, the distinction between which degree of interaction the user desires as he types commands into the system should be clear and straightforward.

It should be remembered that every TAXL command, as well as every BASIC command, begins with some English keyword which strongly suggests the action the execution of that command will have. Which degree of interaction the user desires for each command he enters into the system can be indicated by him by the presence or absence of an integer number preceding the English keyword which actually begins the command. Thus the following holds for both TAXL and BASIC commands. If an integer number is not present preceding the English keyword, the command will be interpreted and immediately executed. If an integer number is present preceding the English keyword, the command will not be immediately

executed but will be entered into a program buffer with a sequence number equal to the value of the integer number which preceded the command.

A. Syntax Analysis

In the latter case, the question of when the syntactic structure of the command is checked is open to debate. While this is basically a question of implementation, the answer will affect the learning behavior of the naive user. Since one of the design goals of the language is ease and speed of learning the language, the question should be considered here. Kemeny and Kurtz felt that the syntactic structure of the commands should be checked immediately to see if a syntactic error occurred in the command. This philosophy has been followed in HP BASIC, as in most BASIC implementations. The authors of some APL systems (Falkoff [1968]) follow a different philosophy. Their belief is that the command should be entered into the program buffer without its syntactic structure being checked, and not until the execution of the command is about to commence will the user be notified if a syntactic error has indeed occurred.

Psychological studies on learning and training behavior (Wolfle [1951]) indicate that immediate feedback speeds the learning process. Since one of the requirements of this system is that it be easily learned, immediate feedback of syntactic errors, wherever possible, seems preferable to delayed feedback. If the user is not informed of a syntactic error which occurs in a command as it is entered into the system, he may mistakenly feel that since the system has accepted the command, the command is correct. It is at precisely this moment, when the user's attention is more focused on the one command in question than at any other time in the program's formation, that the user should be informed if a syntactic error exists within the command. This concept is not too unlike programmed readers in which the reader must successfully answer a question before he can proceed. By the

time the user has given the command to begin execution of the commands in the program buffer, his attention will usually be more focused on the program as a whole and on its semantic structure rather than its syntactic correctness. To be informed of a syntactic error during the execution of the program would be more of a hindrance to clear thought than a help.

The conclusion of the preceding paragraph seems to be true only when the goal of the user is to learn the syntax of the language. Once this has been accomplished and the goal of the user is to write useful and logically complex programs, the facility of sketching out logical sections of program without having to be concerned with their syntactic correctness at that time seems to be important. Thus, the goal of the user should be a consideration in deciding whether the syntactic structure of a command is to be checked at command entry or at command execution. Since one of the goals of the current implementation is to facilitate learning of the language rather than writing large programs, the syntactic structure of a command is checked at command entry.

From an implementer's point of view, a translation from the command's external form to an internal format which is easier to execute and a syntactic check of the command can be accomplished at the same time. Rather than do the translation every time the command is encountered during program execution, it is more reasonable to do the translation once at command entry into the program buffer. Because a syntactic check may be performed during command translation with a minimum of extra effort, a syntactic check at command entry time is quite desirable.

B. Command Classification

A summary of BASIC and TAXL commands will now be given (see Table 3). Each command is placed in one of three categories which gives that command's

TABLE 3

COMMAND CLASSIFICATION

TAXL/BASIC Command	Classification	Use
CREATE	BOTH	Creates new nodes
LABEL	BOTH	Adds labels to nodes
UNLABEL	BOTH	Removes labels from nodes
WRITE	BOTH	Writes part of the data base
COUNT	BOTH	Counts nodes in the data base
PUT	BOTH	Builds relationships in the data base
COPY	BOTH	Copies nodes and relationships and builds relationships in the data base
SEVER	BOTH	Destroys relationships in the data base
DELETE	BOTH	Destroys nodes and relationships in the data base
SAVE	BOTH	Saves part of the data base in secondary storage
RESTORE	BOTH	Restores part of the data base from secondary storage
READ	BOTH	Reads numeric data from data block
DATA	BOTH	Enters numeric data into a data block
PRINT	BOTH	Types values of variables and arrays
LET	BOTH	Computes and assigns values to variables
DEF	BOTH	Defines an arithmetic function
DIM	BOTH	Declares dimensions of arrays
MAT---	BOTH	The 2-dimensional array instructions
GØ TØ	PROGRAM	Transfers control
IF	PROGRAM	Conditional transfer
FOR	PROGRAM	Sets up and operates a loop
NEXT	PROGRAM	Closes a loop
FOREACH	PROGRAM	Sets up and operates a loop to sequence through nodes in a range
GOSUB	PROGRAM	Transfers to a subroutine
RETURN	PROGRAM	Returns from a subroutine
STOP	PROGRAM	Stops a program
LIST	SBS	Lists commands in the program buffer
CLEAR	SBS	Removes commands from the program buffer
RUN	SBS	Initiates execution of commands in the program buffer

permitted degree of interaction. Some commands will be restricted to less than the highest degree of interaction. Commands classified by the sign PRØGRAM are only allowed to be entered into the program buffer for later execution, and so must always be preceded by an integer number when entered into the system. Commands which control program flow would make no sense if they were executed immediately upon entry to the system since they require a program to give them meaning. Other commands, the numeric data and data base manipulation commands, classified by the sign BOTH, may be executed immediately upon entry to the system or may be entered into the program buffer for later execution. Hence, these commands may have BOTH degrees of interaction with respect to the system.

A further set of commands will now be introduced which manipulate the program buffer. These are classified by the sign SBS (statement-by-statement) indicating that these commands must be entered into the system to be executed immediately and cannot be entered into the program buffer for later execution. Systems such as LISP 1.5 and most assembly languages allow such program manipulation commands to be programmable. However, this somewhat advanced concept is not essential for TAXL/BASIC.

Entry of a command into the program buffer is implicit and is indicated by preceding the command by an integer number. The command then has a sequence number equal to the value of the integer number. If a command is entered into the program buffer with a sequence number equal to the sequence number of a command already in the buffer, the new command replaces the old command.

LIST - Classification: SBS

The LIST command has three syntactic forms.

<list command> ::= LIST

<list command> ::= LIST <integer number>

<list command> ::= LIST <integer number> / <integer number>

The first form causes the listing of all commands in the program buffer, along with the sequence number of each, arranged in ascending numerical order. Commands may be entered into the program buffer in any order but will always be listed in ascending sequence order. The second form causes only the listing of the command in the program buffer having the given sequence number, if such a command exists within the buffer. The third form causes the listing, in ascending numerical sequence order, of all the commands having a sequence number whose value is equal to or greater than the first integer number given and is equal to or less than the second integer number given.

CLEAR - Classification: SBS

The CLEAR command removes commands from the program buffer, and has three syntactic forms, analogous to the LIST command:

<clear command> ::= CLEAR

<clear command> ::= CLEAR <integer number>

<clear command> ::= CLEAR <integer number> / <integer number>

The first form causes the erasing of all commands in the program buffer. The second form causes only the erasing of the command in the program buffer having the given sequence number, if such a command exists. The third form causes the erasing of all the commands in the program buffer having sequence numbers whose values lie between (and including) the given integer numbers.

RUN - Classification: SBS

The RUN command causes the program in the program buffer to begin execution. This command has two forms:

<run command> ::= RUN

<run command> ::= RUN <integer number>

The first form causes the execution of the program in the program buffer to begin with the command having the algebraically smallest sequence number. The second form causes the execution of the program in the program buffer to begin with the command having a sequence number equal to the value of the given integer number. If no such command exists, the user is notified.

Commands in the program buffer are normally executed in ascending numerical sequence unless this sequence is altered by the execution of a command having the classification PRØGRAM. The program stops executing either when a STOP command is executed, when control is transferred to a nonexisting command, or when the next command to be executed should be the command with the next highest sequence number and no such command exists. In any case, the user is notified where (by sequence number) the execution of the program is terminated.

C. Interface Between TAXL and BASIC

The commands of both TAXL and BASIC have now been presented. The control commands of BASIC have been adopted to properly organize program flow, and several commands for manipulating the program buffer have been given. Thus far, however, the only interface between TAXL and BASIC is at the program level. Commands from both languages may be evoked interchangeably for immediate execution and commands from both languages may occur in the program buffer. What is needed to make the system more useful is an interface at the data level.

The data for TAXL are the numeric and nonnumeric labels at nodes and the hierarchical relationships of the directed acyclic graph data base, while the data for BASIC are the numeric values of variables and array elements. Some BASIC systems include a limited string processing capability; however, such a capability varies so widely in the relatively few BASIC systems (e.g., Stanford [1968]) which possess one that this work will not concern itself with such a capability.

Recall now the kind of user for whom this system is intended. The main emphasis of his use of this system will be in handling nonnumeric data, the operations for which are available from the TAXL primitives and data base. The reasons for including BASIC are the presence of the programming control commands and the arithmetic processing capabilities which BASIC possesses. At the data level, BASIC operates only on numeric data. Thus, if an interface between BASIC and TAXL is to be made at the data level, it must be at the numeric data level. Recalling that labels at the nodes in TAXL's data base can be numeric, it becomes clearer that the data interface must exist at the numeric level, the only data type which the two systems have in common. The interface must thus concern itself with the convenient retrieval of numeric values from TAXL's numeric labels which can then be used in computation and assignment in BASIC commands, and also in the conversion and placing of the values of BASIC variables into TAXL's data base in the form of numeric labels. In addition, since the COUNT primitive in TAXL results in a number being output, the value thus obtained should also be able to be used in computation and assignment within BASIC.

The discussion of this data interface will now proceed in four steps: the extension of the use of one form of the subtree context specification for referencing values, a solution to the problem of whether an identifier which occurs in a

TAXL/BASIC command is a BASIC variable or a TAXL label, the introduction of a VALUE operator and an extension of the use of the COUNT primitive, and the introduction of a new sequencing statement pair, analogous to BASIC's FOR-NEXT sequencing pair, for sequencing through the nodes in a range.

Recall that in Condition III of the subtree context specification of ranges, the values of numeric labels were compared with given numbers. The values were retrieved by evoking a range and considering the values of any numeric labels which occurred in any of the data items of the nodes in that range. This same mechanism can now be used outside the subtree context specification of ranges, particularly in arithmetic expressions in LET statements of BASIC. Use of this mechanism in a BASIC construct, which retrieves more than one value, is not allowed and will be considered a semantic error. Admittedly, a construct which would allow the assignment or computation on a vector of values would be useful; however, this somewhat advanced concept would not add to the simplicity of the language. Since extensions and complications in other areas of the language would have to be made in order for this construct to have consistent application throughout the language, and since there will be alternate methods of performing the same computation, this construct is, therefore, not allowed.

Examples (see Fig. 2)

Evocation:

LET S = Salary within Laurel

Effect: The BASIC variable S is assigned the value 400

Evocation:

LET S = Salary within Stanford

Effect: Illegal, since the range yields more than one value.

Since there is no keyword which indicates that a value is being retrieved from the tree, confusion can arise over whether an identifier is a BASIC variable or a TAXL label.

For example, in the command

LET S = X

is X a BASIC variable whose value is to be assigned to S, or is X a TAXL label being used to reference a range consisting of one node with a numeric label, the value of which is to be assigned to S? This problem is solved by requiring that at any given moment, the set of BASIC variables and the set of active (not in the saved area) TAXL labels be disjoint. TAXL labels are created by evocations of the CREATE primitive (or by implicit creation in evocations of the PUT primitive), and are destroyed by evocations of the DELETE primitive. Once an identifier which was used as a TAXL label no longer occurs in the tree, it may be used as a BASIC variable. BASIC variables are created implicitly by their first occurrence on the left-hand side of LET statements; prior to this creation, their value is undefined and cannot be used. BASIC variables may be destroyed by their use on the left-hand side of a LET statement having an empty right-hand side. For example,

LET X =

destroys X as a BASIC variable and allows its subsequent use as a TAXL label.

With the mechanism described thus far, values may be retrieved from the TAXL data base and used in BASIC contexts. In order to allow the values of BASIC variables to be placed in the TAXL tree or removed from it, an operator which, when applied to a BASIC variable, returns its value is needed. This operation is automatic when a BASIC variable is used in any arithmetic context. However, in TAXL's LABEL or UNLABEL commands, for example, evoked

labels stand for themselves. In such a context, to force evaluation of the BASIC variable name to obtain its value which is then to be put into or removed from the label set of some node in the tree, the VALUE operator must be used.

Example (see Fig. 2)

Evocation:

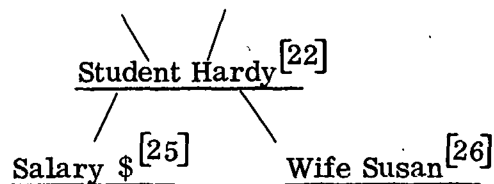
LET S = Salary within Student Hardy within Engineering

Effect: The BASIC variable S is assigned the value 300.53

Evocation:

UNLABEL salary within Student Hardy as value S

Effect:



Notice in the latter example, that a label S does not occur within any node in the specified range, and thus the VALUE operator must be used in order to remove the numeric label 300.53, the value of S, from the node.

The COUNT construct, which appears in the subtree context specification of ranges and is also a TAXL primitive, is extended only in the sense of where the construct can occur; it may now occur within any BASIC numeric expression and yields the number of nodes in the specified range.

Example (see Fig. 2)

Evocation:

LET N = COUNT Salary within Stanford

Effect: The BASIC variable N is given the value 3

FOREACH, NEXT - Classification: PROGRAM

The FOREACH and NEXT statements are loop control statements very similar to the FOR and NEXT statements of BASIC. Recall that the FOR statement causes a BASIC control variable to take on successive arithmetic values over a set of statements. Every reference to the control variable within the set of statements delimited by the FOR and NEXT statement has the value which is the current value of the control variable. The FOR statement gives the initial value, the final value, and the increment for the control variable. The execution of the NEXT statement causes the control variable to take on its next value and execution resumes following the FOR statement. When the control variable has taken on all of its prescribed values, execution resumes following the NEXT statement.

The syntax of the FOREACH statement is:

<foreach statement> ::= FOREACH <range>

Recall that every <range> must begin with a data item consisting of one or more labels. This data item then becomes the control data item which will take on successive values over the set of statements delimited by the FOREACH statement and its paired NEXT statement. The values which the control data item will take on are the nodes in the data base specified by the <range>. Every reference to the control data item within the set of statements within the FOREACH loop has the value which is the current value of the control data item, i.e., a node in the <range>. The execution of the NEXT statement causes the control data item to take on its next value and execution resumes following the FOREACH statement. When the control data item has taken on the value of all the nodes in the <range>, execution resumes following the NEXT statement. Because the nodes in a range are unordered, the control data item assumes its values in an arbitrary order.

The syntax of this version of the NEXT statement is:

<next statement> ::= NEXT <data item>

The rules for nesting of FOREACH statements follow the rules for nesting of FOR statements.

Examples (see Fig. 2)

Problem: Increase the salaries of all secretaries in the School of Humanities and Sciences by 10 percent.

Program:

```
FOREACH Salary within Secretary within Humanities School
LET S = Salary
UNLABEL Salary as value S
LET S = S + .1*S
LABEL Salary as value S
NEXT Salary
```

D. Responses Following the Execution of Commands

Everyone who has ever worked at a terminal using a system which has commands which are executed immediately upon entry to the system (having classification SBS in TAXL/BASIC) occasionally has the feeling that the command last entered might not have been executed at all or might have been executed incorrectly. This phenomenon occurs particularly among novice computer users, the intended users of TAXL/BASIC. Often, as seen by direct observation, quite a bit of output might be requested by the novice user to assure himself that the command in question was indeed executed correctly. It has also been observed that almost any short response by the system after the execution of any command in SBS mode informing the user that everything is "all right" and that the system "understood" and executed his command properly gives the user added confidence and almost completely obviates his need for the assurance output mentioned above.

Commands which inherently cause output at the terminal, PRINT, WRITE, and LIST, obviously need no assurance output. Commands being executed in PROGRAM mode, other than PRINT and WRITE, should have no assurance output because of the possible volume of such output and subsequent slowing of execution. Attention can thus be turned to commands being entered into the system for immediate execution.

If a syntax error occurs in the command, then proper notification of this fact is sufficient to convince the user that the system is paying attention to him. If there are no syntax errors, then execution of the command will commence, and if there are no semantic errors which occur while the command is being executed, then output as simple as

ØK

is enough to assure the user that everything is in fact okay. Semantic errors such as null ranges in TAXL commands, illegal tree structuring arising from an improper use of the PUT command, illegal label manipulation in the UNLABEL command, BASIC variables without values occurring in an arithmetic expression, illegal sequence numbers occurring in the RUN, LIST, or CLEAR commands, and others should be reported to the user as clearly as possible and the ØK message should be suppressed.

E. Conclusion

In this chapter, a design for the amalgamation of some variant of a BASIC system as suggested by Kemeny and Kurtz, and the TAXL language as described in preceding chapters, has been described and given the name TAXL/BASIC. The implementation of BASIC systems has been documented elsewhere. A subsequent chapter will describe a test implementation of TAXL/BASIC, with just enough of BASIC included to test the feasibility of such a system.

CHAPTER V

AN IMPLEMENTATION AND ITS ANALYSIS

This chapter describes a data structure and some algorithms used to implement a partial TAXL/BASIC system. Since implementations of BASIC systems have been described elsewhere (Braden [1968]), only the TAXL data structure and algorithms will be discussed here. The implementation is written in LISP 1.5 and is currently operating under the Stanford Campus Facility ØRVYL time-sharing monitor.

Because of the nature of interpreted LISP 1.5, the fact that TAXL/BASIC is written as an interpreter itself, and the time required to do the extensive page swapping which time-shared LISP requires, the current TAXL/BASIC system is too slow and too expensive for large scale operation. In addition, the central purpose of this work was to develop a user-system interface rather than a large operating system. Thus, the internal data structure and subsequent algorithms were not designed with speed and efficiency in mind. If a large scale implementation of TAXL/BASIC is attempted, it is suggested that the current implementation be studied to see what is required, and that at least the algorithms, if not the data structure itself, should be redesigned. A full implementation of the current data structure and algorithms, even if written in machine language, would probably fail to give adequate service in terms of response time and cost once the data structure exceeds the size which can be contained in primary storage.

In order to follow the listing of the interpreter in Appendix I, the reader must be familiar with LISP 1.5 (McCarthy [1962]), and with property list manipulation and list-structure alteration operations in particular. For those readers not so interested in the fine details, an outline flowchart of the interpreter is given in Appendix II. Initial entry to the interpreter is at A with the RUN FLAG reset.

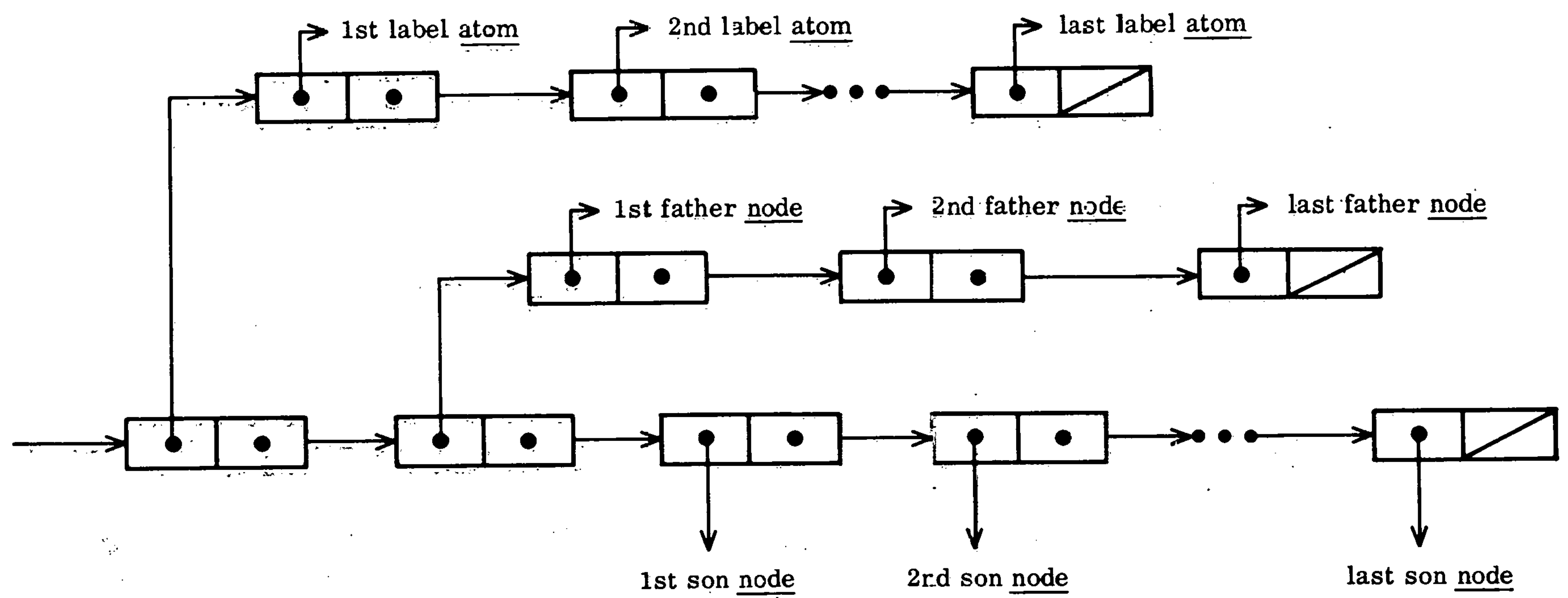


FIG. 3—Format of a node in the data base.

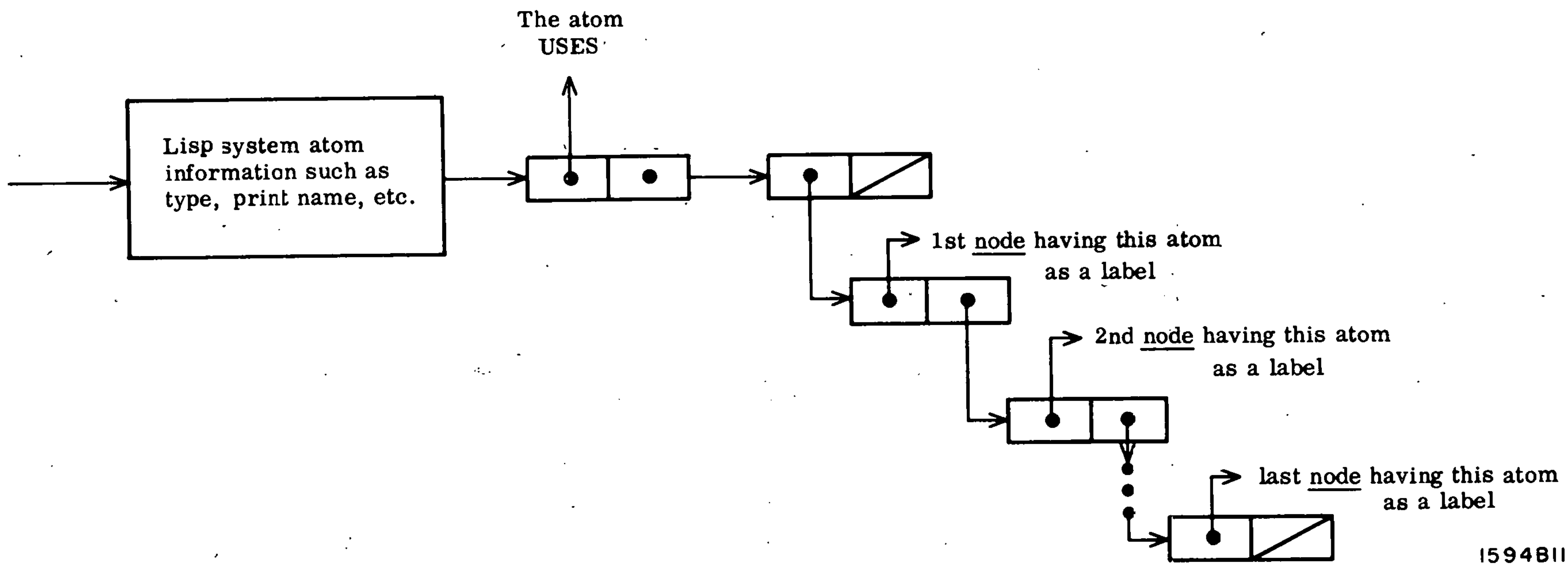


FIG. 4—Format of an atom used as a label in the data base.

The flowchart describes the logical flow of control and does not exactly parallel the programmed interpreter given in Appendix I. Most of the semantic error checking is absent from the flowchart, as are some of the various forms of some of the primitives.

A. Node and Dictionary Formats

The format for a node in the data base is given in Fig. 3. Thus, what is given at a node is a list of the labels that make up the data item at the node, a list of pointers to all immediately hierarchically superior nodes, and a list of pointers to all immediately hierarchically inferior nodes.

The format for the property list of an atom which is used as a label in the data base is given in Fig. 4. The atom USES indicates a following list of pointers to all nodes within which the atom in question is used as a label. The essential structure is that of a dictionary. For every label which occurs in the data base, there is an entry in the dictionary giving all uses of that label within the data base.

Most of the computation time required for the execution of a primitive is consumed in the computation of ranges. Hence, refinements in this computation or modifications in the data structure allowing such refinements will decrease the execution time significantly. Since the object of this study is not the design of such refinements, the algorithm presented for the computation of ranges was chosen for its programming simplicity. Once lists of the nodes in as many ranges as are required for the execution of a primitive are obtained, the execution of the primitive is fairly straightforward, as shown in the flowcharts in Appendix II. More will be said about computation time later in this chapter.

B. The Computation of Ranges

The computation of a range by Method II of the chapter discussing the evocation of ranges is essentially a set intersection operation. Suppose the range

Secretary Carla West

is to be computed. The property list of the atom label "Secretary" contains a list of pointers to all uses of this label, as do the property lists of the atom labels "Carla" and "West." The intersection of these lists is, by definition, a list of nodes which incorporates the range. A straightforward intersection of unordered sets as programmed in the current implementation given in Appendix I is the easiest to program but has a computation time on the order of the product of the number of elements in the sets. The computation time can be reduced to the order of the sum of the number of elements in the sets by ordering the sets according to any arbitrary but well defined ordering.

The computation of a range specified by hierarchical context is a more complicated operation. Given two lists of pointers to a set of nodes X and a set of nodes Y, it must be determined for which $x \epsilon X$,

X within Y

is true. Those x's for which the above is true are retained in the range; those x's for which it is not true are not retained in the range.

There are two principal ways of determining which nodes $x \epsilon X$ are within some node $y \epsilon Y$. One method is to start at each x, and by following the chain of father pointers beginning at node x, check each node encountered on the path from node x to the roots of the tree. If one of the nodes encountered is a y node, then the search can be terminated since it has been ascertained that x is within Y. If no y node is encountered on any path, then x is not within Y. Since all nodes on all paths must be checked, the case in which x is not within Y involves the maximum number of checks.

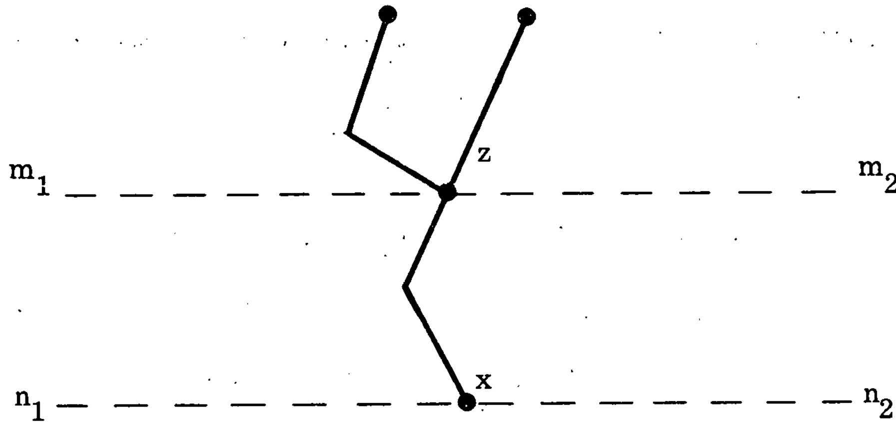
Another method for determining which x is within some set of nodes Y is to start at each node $y \epsilon Y$, in turn, and by following the chain of son pointers beginning at node y, check each node encountered on the path from y to the leaves of the tree. If one of the nodes encountered is an $x \epsilon X$, then this x is an x which is within Y. However, the search cannot be terminated since there may be an $x' \epsilon X$ which is hierarchically inferior to y, and hierarchically inferior to all nodes in Y only through the node x. Thus, if the search were terminated upon encountering x, it would never be ascertained that, in fact, x' is within Y.

Thus, in the general case, it is more advantageous to search from the x's upwards along the father chains than from the y's downward along the son chains. In addition, there will usually be more sons than fathers if the entire data base is considered, implying that to search downward would entail searching along many more paths than searching upwards.

In order to analyze quantitatively the implementation of the range finding mechanism, the maximum number of nodes accessed in order to determine which x's are within Y will be used as a measure of the computation required. As mentioned previously, the case in which x is not within Y involves the maximum number of node accesses since every node on every path from each x upward to the roots of the data base must be accessed.

Assume first that the data base has the form of a true tree rather than an acyclic directed graph. Effectively, this means that each node has, at most, one father. Thus, assigning level 0 to each root node and defining the level of a node to be numerically one greater than the level of its father, each node has precisely one well defined level. With this formulation, n node accesses are required to traverse the path from a node x at level n upward to a root.

The assumption that the data base has the form of a tree will now be removed. Thus, there may exist a node z at level m, $m < n$ which has two fathers, where z is the first encountered node above x for which this is true.



Indeed, node z may be at level m_1 with respect to one path from a root and at level m_2 with respect to the other path. Thus, node x may be at two levels at once, depending on which path through node z is being considered.

Let n_1 be the level of node x with respect to the path which makes node z at level m_1, and let n_2 be the level of node x with respect to the path which makes node z at level m_2. Thus,

$$n_1 - m_1 = n_2 - m_2$$

and $n_1 - m_1$ is the number of node accesses to traverse from node x upward to node z. From node z, $m_1 + m_2$ node accesses are required to traverse both paths from node z upward to a root.

Total: $(n_1 - m_1) + m_1 + m_2 = n_1 + m_2 = n_2 + m_1$

Now generalize the preceding case and assume that node z has p fathers, $p \geq 1$. Thus, node z may be at as many as p levels $m_1, m_2, \ldots, m_p$. Therefore, node x may be at as many as p levels $n_1, n_2, \ldots, n_p$, where node x is at level n_i with respect to the path which causes node z to be a level m_i, $1 \leq i \leq p$.

As before,

$$n_1 - m_1 = n_2 - m_2 = \ldots = n_p - m_p$$

and $n_i - m_i$ is the number of node accesses to traverse from node x upward to node z. From node z, $m_1 + m_2 + \ldots + m_p$ node accesses are required to traverse the p paths from node z upwards to the roots.

Total:

$$(n_i - m_i) + m_1 + m_2 + \ldots + m_p = n_i + m_1 + m_2 + \ldots + m_{i-1} + m_{i+1} + \ldots + m_p$$

where n_i is the number of node accesses required to traverse from node x upward to node z and thence to a root by the ith father path from node z, and

$$m_1 + m_2 + \ldots + m_{i-1} + m_{i+1} + \ldots + m_p$$

is the number of node accesses required to traverse from node z upward to the roots by the p-1 remaining father paths from node z.

The above total may be rewritten as

$$(n_i - m_i) + m_1 + m_2 + \ldots + m_p \qquad 1 \leq i \leq p$$

where $n_i - m_i$ is the number of node accesses required to traverse from node x upward to node z, and

$$m_1 + m_2 + \ldots + m_p$$

is the number of node accesses required to traverse from node z upward to the roots along the p father paths from node z.

None, all, or some of these p paths from node z might themselves split further at levels closer to the roots. If the jth path, $1 \leq j \leq p$, so splits, then m_j is not the true number of node accesses from node z along this path to a root, but must be computed by the above treatment, recursively.

In the worst case, the superstructure from node x upward toward the data base roots forms a tree. Assuming that the average upward branching factor is

b, $b > 1$, and there are L levels from node x up to the roots (counting the roots as level zero), then the maximum number of nodes to traverse is the number of nodes in this tree. Including node x, the number of nodes is given by

$$\frac{b^L - 1}{b - 1} \quad .$$

The specification of ranges by subtree context proceeds in much the same manner, with the nodes specified by hierarchical context used where the roots of the tree were used in the previous discussion. In one variant, the number of nodes obtained are counted and compared to the result of some numeric computation. In another variant, the labels in the nodes so obtained are individually checked to see if they are numeric, and if so, their value is compared to the result of some numeric computation. Nodes specified by hierarchical context having subtrees obeying the required conditions are included in the range, as described in a preceding chapter.

C. Reducing Range Computation Time

The method which has been considered in analyzing the range finding mechanism consists of a traverse upward to the roots of the data base. In this method, the search is terminated when a root of the data base is encountered. In addition, a downward search along the son chain toward the leaves of the data base, as described previously, might be more efficient in certain particular cases. In such situations, the user should be able to take advantage of his particular data structuring to reduce the amount of computation required to determine a range. The following mechanism allows the user to specify whether an upward or downward search is to be made, and at the same time, to specify a terminating condition for an upward search other than the occurrence of a root or a terminating condition for

a downward search other than a leaf. The mechanism can only be used in conjunction with specifying a range by hierarchical context or subtree context requiring that searching be done.

The first form allows the user to denote that the usual upward search is to be performed and to specify a terminating condition other than the occurrence of a root. The terminating search condition is a set of nodes T specified by Method II. Thus, when searching for a range

x within y below T

the search upward is terminated successfully by an occurrence of a y node and terminated unsuccessfully by an occurrence of a root <u>or a member of the terminating set T</u>. To imply to the user that the search is carried on only below (and including) the nodes which are in the set T, the delimiter "below" is used to separate the end of the range specification by Method IV and the specification of the set T.

A portion of a data base shown in Fig. 5 demonstrates how this feature can be used to advantage. The range evoked by

Professor within Reading Committee within Student Y

is identical to the range evoked by

Professor within Reading Committee within Student Y below Department.

In the former case, however, it is not determined that Professors W and X are not included in the range until all the paths from Professors W and X upward to the roots of the data base have been traversed. In the later case, the search is terminated upon encountering the node Computer Science Department. This early termination can save quite a bit of computation, particularly if the portion of the data base shown in Fig. 5 occurs many levels down from the roots.

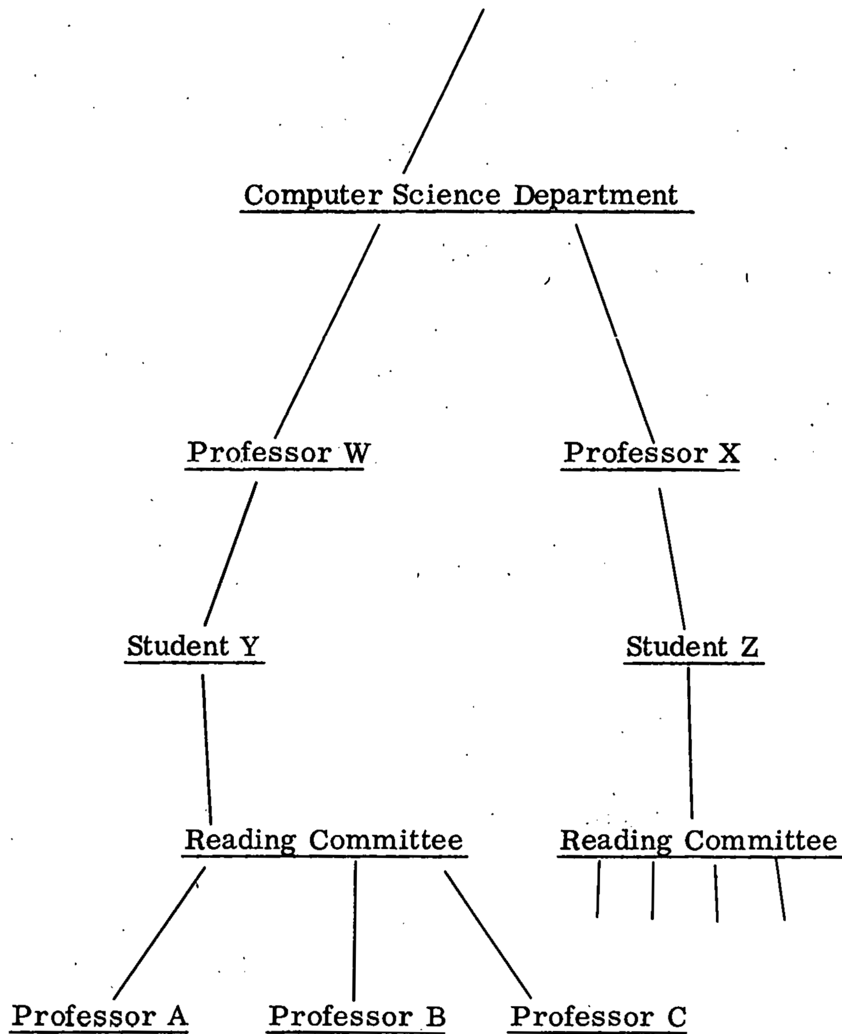


FIG. 5--A portion of a data base demonstrating the utility of below and above

The second form allows the user to denote that a downward search is to be performed and to specify a terminating condition other than the occurrence of a leaf. As suggested by the previous construction, a terminating set T of nodes, specified by Method II is separated from the end of the range specification by Method IV by the delimiter "above," or equivalently so as not to introduce another reserved word, the delimiter pair "not below."

Example (see Fig. 5)

Professor within Computer Science not below Student

An analysis of the implementation will now be made. The amount of memory required for a data base implemented in this way will be considered, as well as the factors that affect the time required to access the data base.

D. Memory Usage

In order to gain some sort of perspective on the amount of memory required to contain a complete data base, a formulation of data base requirements will be made. A fairly representative data base configuration will then be described and the amount of memory required to represent this hypothetical data base will be computed. Throughout the formulation and computation, Figs. 3 and 4 should be consulted.

Assume that throughout the data base there is an average branching factor $s (s > 1)$. That is, on an average, each node has s sons. Let L be the number of levels of the tree, numbering the level of the root nodes as zero. Then the number of nodes in the tree is given by

$$N = \frac{s^{L+1} - 1}{s-1}$$

If we let

m_i = the number of labels at node i

f_i = the number of fathers at node i

s_i = the number of sons of node i,

then the amount of storage required to represent node i in this implementation (see Fig. 3) is given by

Labels:	$8 + 8\,m_i$
Fathers:	$8 + 8\,f_i$
Sons:	$8\,s_i$
TOTAL:	$8(2 + m_i + f_i + s_i)$

Note that 8 bytes are required to store a pair of pointers. Thus, for the nodal structure of the data base, the total amount of storage is given by

$$8 \sum_{i=1}^{N} (2 + m_i + f_i + s_i)$$

Now consider the possible dictionary structures. Each label which occurs anywhere within the data base has a dictionary entry. As seen in Fig. 4, there is a list of pointers associated with this entry to every node in the data base in which that label occurs. Thus, there are as many pointers out of the dictionary as there are (not necessarily distinct) labels at nodes in the data base, given in this formulation by

$$\sum_{i=1}^{N} m_i$$

Since eight bytes are required to store a pair of pointers in the current implementation, the amount of storage required for the pointers is given by

$$8 \sum_{i=1}^{N} m_i$$

The only other significant contribution to memory utilization arises from the storage of the labels themselves. Since each label occurs only once in the dictionary, independently of its usage within the nodal structure, the amount of storage required for these labels depends upon the number of <u>distinct</u> labels. At a minimum, these can be only one distinct label which occurs as the only label at every node in the data base. Theoretically, there is no maximum number of distinct labels since the number of labels occurring at any node is not limited. However, since we have assumed that there are m_i labels at node i, then a maximum will be achieved by further assuming that all the labels across the data base are different. Thus, the maximum total number of distinct labels is given by

$$\sum_{i=1}^{N} m_i$$

Assuming an average of q characters per label, the amount of storage required for the labels themselves is given by

$$q \sum_{i=1}^{N} m_i$$

Thus, there are three constitutents of memory usage:

Nodal structure:	$8 \sum_{i=1}^{N} (2 + m_i + f_i + s_i)$
Dictionary pointers:	$8 \sum_{i=1}^{N} m_i$
Label storage:	$q \sum_{i=1}^{N} m_i$
TOTAL:	$8 \sum_{i=1}^{N} (2 + m_i + f_i + s_i) + (q + 8) \sum_{i=1}^{N} m_i$

A hypothetical data base will now be described. The parameters of this data base are not completely random but are based on a small sample of data bases built by students learning to use TAXL (see the conclusion of this chapter).

Assume that the data base has an average depth of seven levels and that each node has an average of four sons, i.e.,

$$L = 6 \text{ and } s = s_i = 4$$

and, therefore, the number of nodes in the data base is

$$N = \frac{4^7 - 1}{4 - 1} = 5461$$

Now assume that each node has an average of three labels and two fathers, i.e.,

$$m_i = 3, \; f_i = 2, \text{ and } s_i = 4$$

Thus, the storage requirements for an average node are given by

Labels:	8 + 8(3) = 32 bytes
Fathers:	8 + 8(2) = 24 bytes
Sons:	8(4) = 32 bytes
TOTAL:	88 bytes/node

Hence, to represent the nodal structure of the data base requires

$$88 \text{ bytes/node} \cdot 5461 \text{ nodes} = 480,568 \text{ bytes}$$

The amount of storage required for the dictionary pointers is

$$8 \sum_{i=1}^{5461} 3 = 131,064 \text{ bytes}$$

As indicated in the formulation, the maximum total number of distinct labels could be calculated as

$$\sum_{i=1}^{5461} 3 = 16,383 \text{ labels}$$

However, a more realistic estimate can be made by the following assumptions.

It will be assumed that at each level there is a common label which serves as an attribute, and that at every node at that level, there are two other labels, the set of which are disjoint across the level and the entire data base. These two other labels at each node serve as a value for the common attribute. Thus, since there are four sons for each node, at level i, there are

$$2 \times 4^i + 1$$

distinct labels, counting the root level as level zero. Therefore, over seven levels, there are

$$\sum_{i=0}^{6} (2 \times 4^i + 1) = 10,926$$

distinct labels and hence

$$6 \times 10,926 = 65,556 \text{ bytes}$$

required to store all the labels in the data base. It is assumed that these are an average of six characters per label.

Table 4 summarizes the storage requirements of this hypothetical data base and gives the percentage of storage required for each data base component.

E. Access Time

In the discussion of the time required to compute the range x within y, the particular configuration of the data base and the manner in which both the nodes, named by x and the nodes named by y, are distributed throughout the configuration are the most important factors to consider. Because the particular distribution of x's and y's are such an important consideration, to hypothesize a particular data configuration and then analyze this particular configuration as before would not accurately enough characterize access time in general. However, there are several important observations which can be made.

TABLE 4

STORAGE REQUIREMENTS FOR A HYPOTHETICAL DATA BASE

	Bytes	Percentage of Total
Node Structure	480,568	71.0
Dictionary Pointers	131,064	19.4
Labels	65,556	9.6
TOTAL	677,188	100.0

3 labels per node

2 fathers per node

4 sons per node

6 characters per label

7 levels

As described previously, in the computation of the range

X within Y

the maximum time required occurs when there are no nodes $x \epsilon X$ within any nodes $y \epsilon Y$. In this case, a search has to be performed beginning at each node x along the chain of father pointers to the roots of the data base. At each node z encountered along a path toward the roots, an identity test must be made to see if z is identical with any of the nodes $y \epsilon Y$. If we take the total number of comparisons for identity as a measure of access time, then it can be seen that the number of nodes $y \epsilon Y$ times the total number of nodes along the path(s) from a particular $x' \epsilon X$ to the data base roots characterizes the maximum access time required to determine if x' belongs in the range. Since this computation must be performed for each $x \epsilon X$, the maximum number of comparisons required to determine the range

X within Y

is given by

$$\sum_{x \epsilon X} C(Y)\, P(x) = C(Y) \sum_{x \epsilon X} P(x)$$

where C is the cardinality operator and P(x) is the total number of nodes encountered along all paths from a node x along the chain of father pointers to the roots of the data base (see Section B of this chapter).

There are two other factors, mentioned briefly earlier in this chapter, which contribute to access time. Both of these factors arise in the computation of the sets X and Y.

As described in Chapter II, the sets X and Y are specified by Method II. Thus, in order to specify the set X, n labels $x_1, x_2, \ldots, x_n$ are specified and the set X is comprised of all those nodes in the data base which have at least the n labels mentioned above. A parallel argument can be made for the set Y. The contribution

to access time thus arises from:

1) finding each of the n labels in the dictionary
2) forming the intersection of the n sets of pointers, the ith set being associated with the label x_i and pointing to all those nodes in the data base which contain x_i as a label.

In the current implementation, the operation of finding each of the n labels $x_1, x_2, \ldots, x_n$ is performed automatically by the LISP system. This should be accomplished by a hash addressing scheme.

The USE-lists associated with each dictionary entry are unordered in the current implementation, and hence each of the n-1 intersections which must be performed requires a number of operations proportional to the product of the number of pointers which occur in the sets to be intersected.

If the hashing function used to find the appropriate entries in the dictionary is a good one, the time required to find the n labels which constitute the set X is proportional to n. Assuming that m labels constitute the set Y, the time required to look up the n + m labels is given by k(n+m), where k is some constant dependent on the hashing function.

As explained previously the time required to perform the n-1 intersections which define the set X and the m-1 intersections which define the set Y is given by

$$(n-1)K_x^2 + (m-1)K_y^2$$

where K_x is the average number of pointers associated with each of the n labels which constitute the set X, and K_y is the average number of pointers associated with each of the m labels which constitute the set Y.

Thus the total access time is given by

$$k(n+m) + (n-1)K_x^2 + (m-1)K_y^2 + C(Y)\sum_{x\epsilon X} P(x).$$

F. Operation Time

Once all the ranges required for a given command have been computed, the time required to complete the operation called for by the given command generally depends only on the cardinality of the range(s) upon which the command will operate. For all those commands which require only one range, the time required to complete the operation is proportional to the number of nodes in that range. For those commands which require two ranges, the time required is proportional to the product of the number of nodes in each range. It should be noted that the operation time for all those commands which require additional tree searching for their operation, i.e., WRITE and DELETE, are influenced by the subtree structure below the nodes in the computed range(s). Also, the operation of the SAVE and RESTORE commands depends on the structure and extent of the secondary storage dictionary.

G. System Measures

In Table 5 measures of significant TAXL system functions in the current implementation are summarized. The formulae given in the table show the nature of the dependence of the system's functions upon the parameters involved. Proportionality factors are not given. Except for the memory utilization, all of the systems functions give a measure of access time in terms of the cardinalities, denoted by the operator C, of certain sets which are involved in the particular system function. Such sets include sets of nodes, sets of labels, sets of fathers at a node, and sets of sons at a node.

The measures for memory utilization and the time to compute X within Y have been derived previously in this chapter. Since the USE-lists are unordered in the current implementation, the time required to perform the intersection of two such lists is proportional to the product of their cardinalities.

TABLE 5

SYSTEM MEASURES

Memory Utilization	$8\sum_{i=1}^{N}(2 + m_i + f_i + s_i) + (q+8)\sum_{i=1}^{N} m_i$
Time to perform intersection of USE-lists W and Z	$C(W) \cdot C(Z)$
Time to compute X within Y (the sets X and Y are already defined)	$C(Y) \sum_{x \in X} P(x)$
Time to add a label to a node in the data base	CONSTANT
Time to remove a label l from a node q in the data base	$m_q + l_u$
Time to add a node q to the data base	C(new fathers (q))
Time to remove a node q from the data base	C(sons (fathers (q))) + C(fathers (sons (q)))

N: number of nodes in the data base

m_i: number of labels at node i

f_i: number of fathers of node i

s_i: number of sons of node i

q: average number of characters per label

C: cardinality operator

P(x): total number of ancestors of node x

l_u: number of uses of the label l in the data base

In order to add a label to a node in the data base the following operations must be performed:

1) Add a node pointer to the USE-list associated with the label.

2) Add a dictionary pointer to the label list at the node.

Since the USE-list is unordered, the node pointer may be added to the front of the USE-list, an operation not depending on the cardinalities of any sets. Since the usual option for adding a label to a node requires that the label be added at the end of the label list, the label list at the node must be searched to find its end. However, if the label list is stored in reverse order, the new label can be added to the front of the list, an operation not depending on the cardinality of the label list. Thus, the time required to add a label to a node is a constant.

In order to remove a label l from a node q in the data base, the following operations must be performed:

1) Remove the node pointer from the USE-list associated with the label.

2) Remove the dictionary pointer from the label list at the node.

Since the USE-list must be searched in order to remove the node pointer, time proportional to l_u, the number of uses of the label l in the data base (i.e., the cardinality of the USE-list), is required. In addition, since the label list at the node must be searched for the dictionary pointer, time proportional to m_q, the number of labels at node q is also required. Thus, in order to remove a label from a node in the data base, time proportional to $m_q + l_u$ is required.

In order to add a node q to the data base, the following operations are required:

1) Add the new father pointers to the father list of q.

2) Add a pointer to q to the son list of each new father of q.

Since the father and son lists at nodes are unordered, the new father pointers may be added to the front of the father list of q, requiring time proportional to

the number of new fathers of q. Thus, the total time required is proportional to C(new fathers (q)).

In order to remove a node q from the data base, the following operations must be performed:

1) Remove the son pointers to q from each of the fathers of q.

2) Remove the father pointers to q from each of the sons of q.

The father and son lists must, therefore, be searched for the pointers to be removed, and this operation must be performed for each father and son of q. Thus, the time required to remove the son pointers to q from the fathers of q is proportional to

C(sons (fathers (q)))

In a similar manner, the time required to remove the father pointers to q from each of the sons of q is proportional to

C(fathers (sons (q)))

Thus, the total time required to remove a node q from the data base is proportional to the sum of the two cardinalities given above.

A useful refinement to make in the implementation is to keep the USE-list ordered. The system function measurement which would be improved by this refinement would be the time required to perform the intersection of two USE-lists. Since the lists would be ordered, the time required would be proportional to the <u>sum</u> of the cardinalities of the lists rather than the product. However, the time required to add a label to the data base would increase since the node pointer could no longer be added to the front of the USE-list but would have to be added at its appropriate place in the ordered list. Thus, the time required to add a label l to the data base would depend on l_u, the cardinality of the USE-list of l. It is felt that this refinement would be a useful addition to the implementation,

since USE-lists are intersected for almost every range definition. The operation occurs much more frequently than the addition of a label to the data base.

A better refinement would be to keep the USE-lists not only ordered, but ordered in the form of a balanced tree (Knuth [1970]). The advantages of storing a USE-list of cardinality n in this fashion arise from the fact that the time required to insert a new element, to delete an old element, and to find the smallest element in such a tree, each requires time proportional to $\log_2 n$. (The algorithms for performing these functions will appear in The Art of Computer Programming, Volume 3, by D. E. Knuth).

Thus, labels may be added and deleted from the data base with a logarithmic dependence on l_u rather than a linear dependence. The additional storage required for structuring the USE-lists in this fashion affects only the multiplicative constant in the formula for the dictionary pointer storage, the formula for which would now be

$$24 \sum_{i=1}^{N} m_i$$

since one father and two sons pointers would be required for each entry.

Additional refinements, and possibly the best refinements in the representation and algorithms for addressing and manipulating the nodal structure, might well be in the direction of hash, or scatter storage techniques (Morris [1968]). In the LEAP system (Feldman [1969]), a hash addressing scheme based on a hash of two elements of an object-attribute-value triple provides a convenient and useful method for the retrieval of information concerning the user-defined relationships among a universe of items.

Hash coding is the simulation of an associative memory, and since TAXL is an associative semantic processor, it is felt that research into new methods

of using hash coding techniques might well uncover more efficient ways of implementing a system such as TAXL.

H. Conclusion

A test implementation and analysis of a limited TAXL/BASIC system has been described. Several comments can be made concerning the analysis.

The factor of eight which appears in the formula for total memory usage arises from the fact that in the version of LISP 1.5, in which the TAXL interpreter is written, eight bytes are required to store one LISP element (a pair of pointers). This factor can be reduced by writing TAXL in some other list processing language system (Hansen [1969]) or by using special data structures designed for TAXL in particular, and embedded in some assembly language system.

Otherwise, it can be seen that the total memory usage depends linearly on the number of nodes in the data base, as well as on the number of labels at those nodes and on the interconnections between those nodes. Upon considering the formula for access time, it can be seen that the access time depends linearly on each of the cardinalities of the sets X and Y and on the depth and number of paths to the roots of each $x \epsilon X$.

Operation time for various of the primitives could be decreased by imposing an order on the lists of fathers and sons at the nodes in the data base. The ordering of the labels is defined by the user, and hence an internal ordering could not be imposed on them without complicating the algorithms which manipulate the labels. An ordering imposed on the fathers and sons would allow a faster retrieval of specific fathers or sons. However, the time required to insert new fathers and sons into an ordered list would consume more time than if the list were not ordered.

On the basis of classroom utilization, the feasibility of such a system, as described in this work, from the user's point of view with respect to the goals discussed in the introduction has been ascertained as affirmative. On two separate occasions, lectures were given to the type of potential user of TAXL/BASIC as described in the introduction to this work. One such group was composed of students enrolled in a graduate course in communications. Their only previous computer experience was a limited introduction to terminal processing via a BASIC system. The other group was composed of summer school students and teachers enrolled in an introductory course for computing in the humanities and social sciences. Their only previous computer experience was a four-week exposure to Algol W with no terminal processing. Even though the two groups were at different levels in their educational experiences and their limited computer experiences were of a different nature, their ability to grasp and learn how to use the TAXL/BASIC system was fairly uniform. After only two hours of classroom lecture and ten minutes of terminal usage instruction (which included log-on, log-off, and other non-TAXL/BASIC procedures), almost all the students, working in groups of two or three, were able to use the system with a fairly high degree of assurance in at least an experimental mode to answer most of their remaining questions.

The students were asked to build and manipulate data bases which would be of interest to them in their work. Political cross affiliations between members of the United States Senate and House of Representatives, a bartender's guide of ingredients for different drinks, and an inventory of an army supply depot were some of the examples for which the students found TAXL/BASIC useful and interesting.

From this preliminary survey of the utility of TAXL/BASIC with respect to its intended goals, it appears that the system meets its intended requirements. Experimentation with more economical, more complete systems able to handle larger data bases is required before more complete results can be obtained.

CHAPTER VI

FUTURE WORK AND SUMMARY

Throughout the course of this work, several topics have arisen which tend to complement the present state of the work as described in this paper.

In order to better test how easily the system can be learned and used by computer novices, a well written user's manual could be compiled with its prospective audience well in mind. Graduated exercises on which the student could work while using TAXL/BASIC at a terminal could be provided.

The interface between TAXL and BASIC could be made more complete. By defining a good string manipulation facility for BASIC, this string manipulation facility could be interfaced with TAXL's label structure. As mentioned previously entire 1-dimensional arrays of numeric values could be retrieved from a range, each of whose nodes contain numeric labels. In addition, the LABEL and UNLABEL commands could be extended so that with one command evocation, the set of numeric labels, whose values reside in a 1-dimensional BASIC array, could be added to or removed from a range of nodes in the TAXL data base. Finally, a good external encoding for the data base could be designed and the READ and DATA statements of BASIC could be extended to allow the reading of portions of the TAXL data base.

Currently, the TAXL addressing structure is semantic and associative. The range specification mechanism could be expanded by allowing syntactic addressing. Thus, constructs such as SON OF ... and FATHER OF ... and compounds of these would be allowed.

A facility for labeling arcs, which might stand for attributes whose values could be found as the labels at the nodes which terminate the arcs, could be introduced. This might allow a more concise and more easily manipulable data

representation in those cases in which there are many attribute-value pairs describing a hierarchically superior node. Multiple arcs between two nodes could also be introduced.

Currently, intersections and unions of ranges may be specified by the successive application of several of the primitives operating on the ranges, the intersections or unions of which are being sought. An explicit facility for specifying intersections and unions of ranges would be useful in those contexts where they are required frequently.

By allowing a dynamic macro facility, the user could define his own primitives as successive applications of the TAXL primitives or other user defined primitives. The macro could have the form of a tree which the user can create in the data base. The root node of this tree could contain the keyword which would cause the tree to be scanned and evaluated when a command beginning with the same keyword is evoked. The macro tree would be required to have a certain form so that in scanning the tree in some predetermined order, the system could fill in the templates occuring in the macro tree with the range specifications occurring in the calling command and initiate execution of the commands found in the tree in a proper order.

The suggestions made in the concluding sections of the previous chapter concerning memory utilization, access time, and operation time could be carefully worked out to improve the speed and efficiency of the TAXL system.

In summary, this paper has presented an easy to learn and use data management and manipulation system for computer novices.

Chapter I outlined the need for such systems and suggested the uses to which they could be put.

Chapter II discussed the format of the data base and the mechanism for addressing such a data base in semantic and associative terms.

Chapter III introduced eleven primitives for constructing, destroying, and otherwise manipulating and querying the data base. The primitives are designed to operate on portions of the data base addressed by the range mechanism discussed in the second chapter. In this way, the addressing mechanism and the operational primitives are clearly separated.

In Chapter IV, the design of a system in which TAXL and a numeric processing system possessing logical programming capabilities was introduced. Additional primitives for managing the programming structure were introduced.

Chapter V discussed a current implementation of TAXL and gave measures of memory utilization and access time in terms of natural parameters of the system.

The current chapter outlined possible future work and summarized this paper.

REFERENCES

Bauer, H., Becker, S., Graham, S., Satterthwaite, E., ALGOL W Language Description, Computer Science Department Report CS110, Stanford University, September 1969.

Braden, H., Wulf, W., "The Implementation of a BASIC System in a Multiprogramming Environment," CACM, Vol. II. No. 10, October 1968; p. 688.

Falkoff, A., Iverson, K.E., The APL/360 Terminal System, ACM Symposium on Interactive Systems for Experimental Applied Mathematics, (Academic Press, New York, 1968); p. 22.

Feldman, J., Rovner, P., "An Algol-Based Associative Language," CACM, Vol. 12, No. 8, August 1969; p. 439.

Griswold, R. et al., The SNOBOL 4 Programming Language, (Prentice Hall, Inc., Englewood, New Jersey, 1968).

Hansen, W. J., The Impact of Storage Management on Plex Processing Language Implementation, Computer Science Department Report CS113, Stanford University, July 1969.

Information Processing Language - V. Manual, ed. by Allen Newell, Rand Corporation. (Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1961).

Kemeny, J., Kurtz, T., BASIC Programming, (John Wiley and Sons, New York, 1967).

Knowlton, K., "A Programmers Description of L^6," CACM, Vol. 4, No. 6, June 1961; p. 616.

Knuth, D. E., The Art of Computer Programming, Vol. 3, (Addison-Wesley Publishing Company).

Knuth, D. E., Optimum Binary Search Trees, Computer Science Department Report CS149, Stanford University, January 1970.

Lang, C., Gray, J., "A Ring Implemented Associative Structure Package," CACM, Vol. 11, No. 8, August 1968; p. 550.

McCarthy, J. et al., LISP 1.5 Programmers Manual, (The MIT Press, Cambridge, Massachusetts, 1962).

Morris, R., "Scatter Storage Techniques," CACM, Vol. 11, No. 1, January 1968; p. 38.

Ross, D. T., "A Generalized Technique for Symbol Manipulation and Numerical Calculation," CACM, Vol. 4, No. 3, March 1961; p. 147.

Stanford/Basic User's Manual, Stanford Computation Center Campus Facility, Stanford University, Stanford, California, December 1968.

Webster's Third New International Dictionary, (G and C Merriam Company, Springfield, Massachusetts, 1964).

Wirth, N., Hoare, C.A.R., "A Contribution to the Development of ALGOL," CACM, Vol. 9, No. 6, June 1966; p. 413-431.

APPENDIX I

LISTING OF THE INTERPRETER

```
VERBOS(NIL)
CSET (OPSP ( (+ . 3) (- . 3) (* . 4) (/ . 4) (NEG . 4) (** . 5)
                                                       (SUB . 6)  ))
CSET (STAK NIL)
CSET (OUTP NIL)
CSET (AXP NIL)
CSET (CHK NIL)
CSET (RELS (EQ NEQ GR GE LS LE))
CSET (RUN NIL)
CSET (SYNTAX NIL)
CSET (SBS T)
CSET (SUB SUB)
CSET (LINE NIL)
CSET (LINES NIL)
CSET (THISLINE NIL)
CSET (UNIQ NIL)
CSET (ISNF NIL)
CSET (NLRG $$$NULL RANGE$)
CSET (QUOT $$$"$)
CSET (DLIM ($$$CARRIAGE RETURN$ NOT IN WHEREVER IS INTO WITHIN
          UNDER TO FROM AS BY AND BEFORE EQ NEQ  GR GE LS LE))
CSET (ONLF NIL)
CSET (DELT NIL)
CSET (IMPRDLIM $$$IMPROPER DELIMITER... $)
CSET (IAE $$$ILLEGAL ARITHMETIC EXPRESSION$)
CSET (AVCS $$$ARITHMETIC VARIABLE CANNOT HAVE A SUBSCRIPT... $)
CSET (AIMS $$$ARRAY IDENTIFIER MUST HAVE A SUBSCRIPT... $)
CSET (LNMI $$$LINE NUMBER MUST BE INTEGER$)
CSET (NCTN (NOT IN))
CSET (PLTF NIL)
CSET (ELSL NIL)
CSET (OK OK)
CSET (NXTW NIL)
CSET (INTO (INTO WITHIN UNDER TO))
CSET (CFLG NIL)
CSET (KFLG NIL)
CSET (WHEREVER WHEREVER)
CSET (APVAL APVAL)
CSET (PVAL PVAL)
CSET (USES USES)
CSET (XTUP NIL)
CSET (NUMS NIL)
CSET (EMRK $$$CARRIAGE RETURN$)
CSET (BYAS (BY AS))
DEFINE ((
  (MAIN (LAMBDA NIL (PROG (X)
 M1 (COND (CHK (PROG2 (PROG2 (PRIN1 SBS) (PRIN1 SYNTAX))(PRINT RUN))))
    (COND (RUN (COND ((CDR THISLINE) (PROG2
                        (CSETQ THISLINE (CDR THISLINE))
                        (CSETQ LINE (CDAR THISLINE))))
                     (T (STPP NIL))))
          (T (TREAD 0)))
 M2 (COUNT 50000)  (CSETQ NXTW (TREED))  (CSETQ KFLG NIL)
    (SETQ X (FTCH))
    (COND ((NUMBERP X) (PROG NIL
       (COND ((NOT(FIXP X))(MESS (QUOTE $$$LINE NUMBER MUST BE INTEGER$)
                     BLANK NIL)))
       (COND ((EQ NXTW EMRK) (PROG2 (LINESRCH X NIL)
                                    (BEGIN (QUOTE MAIN) NIL))))
       (CSETQ SYNTAX T)  (CSETQ SBS NIL)
```

```
      (CSETQ LINE (CONS X NIL)) (SETQ X (FTCH))     )))
    (COND
         ((EQ X (QUOTE DELETE)) (DELE T))
         ((EQ X (QUOTE SEVER)) (DELE NIL))
         ((EQ X (QUOTE COPY)) (PUCP NIL))
         ((EQ X (QUOTE PUT)) (PUCP T))
         ((EQ X (QUOTE WRITE)) (WRIT))
         ((EQ X (QUOTE LABEL)) (MAKE T))
         ((EQ X (QUOTE UNLABEL)) (MAKE NIL))
         ((EQ X (QUOTE COUNT)) (COUT))
         ((EQ X (QUOTE CREATE)) (CRET))
         ((EQ X (QUOTE LET)) (LETT))
         ((AND (NOT SBS) (EQ X (QUOTE GO))) (PROG2 (GOTO)
                  (COND (RUN (GO M2)) (T NIL))))
         ((AND SBS (EQ X (QUOTE LIST))) (LIS))
         ((AND SBS (EQ X (QUOTE CLEAR)))(CSETQ KFLG (CSETQ LINES NIL)))
         ((AND SBS (EQ X (QUOTE RUN))) (PROG2 (RNN) (GO M2)))
         ((EQ X (QUOTE STOP)) (COND (RUN (STPP T))))
         ((EQ X (QUOTE SHOWU)) (SHOWU (TREAD NIL)))
         ((EQ X (QUOTE TRACE)) (TRACE (TREAD NIL)))
         ((EQ X (QUOTE UNTRACE)) (UNTRACE (TREAD NIL)))
         ((EQ X (QUOTE PRINT)) (PRYN))
         ((EQ X (QUOTE XTOP)) (PRINT (CDR (QUOTE XTOP))))
         (T (MESS (QUOTE $$$ILLEGAL COMMAND... $) X NIL)))
    (COND (SYNTAX (PROG2 (PROG2 (CSETQ SYNTAX NIL) (CSETQ SBS T))
                      (LINESRCH (CAR LINE) T))))
    (COND ((AND KFLG SBS) (PRINT CK) ))
    (GC M1>
DEFINE ((
  (ITM (LAMBDA NIL (PROG (Y L LT NEWF)
     (CSETQ UNIQ NIL)
     (COND ((MEMB NXTW DLIM) (MESSCP))
           ((EQ NXTW (QUOTE UNIQUELY)) (CSETQ UNIQ (FTCH))))
  I1 (SETQ L (COND ((ATOM (SETQ LT (FTCH))) (APPEND L (CONS LT NIL)))
                   (T (APPEND L LT))))
     (COND ((MEMB NXTW DLIM) (SETQ LT L)) (T (GO I1)))
     (COND (SYNTAX (RETURN NIL)))
   I (SETQ NEWF (OR NEWF Y))
     (SETQ Y (NINUS (CAR LT)))
    (COND ((MEMB (CAR LT) (CDR LT)) (SETQ L (EFFACE (CAR LT) L)))
         ((SETQ LT (CDR LT)) (GO I)))
     (SETQ L (CONS L NIL))
     (COND ((AND PUTF (OR NEWF Y) (NOT CFLG)) (CREATE L NIL))
           (T (RETURN L)))
     (CSETQ XTOP (CONS L XTOP))
    (COND (RUN (RETURN L)))
     (MESSN L NIL)  (PRINT (QUOTE $$$ CREATED$))
     (RETURN L>
DEFINE ((
  (NINUS (LAMBDA (X)
    (COND ((OR CFLG (GET X USES)) NIL)
         ((AND PUTF (NOT (GET X USES))) T)
         (T (MESS (QUOTE $$$NOT IN USE... $) X T>
DEFINE ((
  (ITMS (LAMBDA NIL (PROG (RES)
  I (SETQ RES (CONS (ITM) RES))
    (COND ((EQ NXTW (QUOTE AND)) (PROG2 (FTCH) (GO I))))
    (RETURN RES>
DEFINE ((
  (FTCH (LAMBDA NIL (PROG (TEMP)
    (SETQ TEMP NXTW)
```

```
   (CSETQ NXTW (CCND ((CCND (RUN (NULL LINE)) (T (EOL))) EMRK)(T
      (PROG (A B D)
         (COND ( (NOT (ATOM (SETQ A (TREED))))
                        (CCND (AXP (RETURN A)) (T
                    (MESS (QUOTE $$$LISTS NOT ALLOWED$) BLANK NIL)) ))
                ((AND (NUMBERP A) (NOT (FIXP A)))
                    (COND ((SETQ D (MEMBER A NUMS)) (RETURN D))
                          (T (PROG2 (CSETQ NUMS (CONS A NUMS))
                                    (RETURN A))))))
                ((NOT (EQ (CAR (SETQ B (EXPLODE A))) QUOT))
                      (RETURN A))
                ( (SETQ B (CDR B)) (GO L5)))
 L2      (COND ( (CCND (RUN (NULL LINE)) (T (EOL)))
                        (MESS (QUOTE $$$MISSING QUOTE$) BLANK NIL))
                ((AND (NUMBERP (SETQ A (TREED))) (NOT (FIXP A)))
                    (PROG2 (COND ((SETQ B (MEMBER A NUMS))
                                        (SETQ D (CONS B D)))
                                 (T (PROG2 (CSETQ NUMS (CONS A NUMS))
                                        (SETQ D (CONS A D)))))
                           (GO L2)))
               ( (ATCM A ) (SETQ B (EXPLODE A)))
                (AXP (RETURN A))
                (T (MESS (QUOTE $$$LISTS NOT ALLOWED$) BLANK NIL)))
         (COND ( (EQ (CAR B) QUOT) (RETURN D)))
 L4      (COND ((EQ (CAR B) QUOT) (RETURN(APPEND1 D (MKATOM)))))
         (RLIT (CAR B))
         (COND( (SETQ B (CDR B)) (GO L4)))
         (SETQ D (APPEND1 D (MKATOM)))
         (GO L2)
 L5      (RLIT (CAR B))
         (COND ( (NULL (SETQ B (CDR B)))(SETQ D (APPEND1 D (MKATCM))))
                ((EQ (CAR B) QUOT)(RETURN(APPEND1 D (MKATOM))))
                ( T  (GO L5)))
         (GO L2) ))))
   (CCND (SYNTAX (COND ((OR AXP (ATCM TEMP)) (APPEND1 LINE TEMP))
                       (T (CSETQ LINE (APPEND LINE
                                 (APPEND1 (CONS QUOT TEMP) QUOT)))))))
   (RETURN TEMP>
DEFINE ((
  (TREED (LAMBDA NIL (PROG (X)
     (RETURN (COND (RUN (PROG2 (PRCG2 (SETQ X (CAR LINE))
                                (CSETQ LINE (CDR LINE))) X))
           (T (TREAD NIL>
DEFINE ((
  (RANGE (LAMBDA (FLG) (PROG (RNGE)
    (COND ((NULL FLG)
              (CONC ((EQ NXTW WHEREVER) (SETQ RNGE (WHEREPRT XTOP)))
                    (T (PROG2 (SETQ RNGE (INPT))
                              (CCND ((EQ NXTW WHEREVER)
                                       (SETQ RNGE (WHEREPRT RNGE)))
                                    (T NIL)))))))
          ((NUMBERP FLG) (SETQ RNGE (INPT))))
    (RETURN  RNGE>
DEFINE ((
  (WRIT (LAMBDA NIL (PROG (X)
   (CSETQ CNLF NIL)
   (CCND ((EQ NXTW (QUOTE CNLY)) (PROG2 (CSETQ CNLF T) (FTCH))))
   (CCNC ((EQ NXTW EMRK) (SETQ X XTOP))
         ((NULL (SETQ X (RANGE NIL))) (MESS NLRG BLANK T)))
   (CCND((NOT(EQ NXTW EMRK))  (MESS IMPRDLIM NXTW NIL)))
   (CCND (SYNTAX (RETURN NIL)))
```

```
   (CSETQ ELSL NIL)
   (CCND ((NULL X)  (RETURN (PRINT (QUOTE $$$NOTHING TO WRITE$)))))
 W (CCND (X (LEVEL (CCNS (CAR X) NIL) 2))
         (T (RETURN NIL)))
   (SETQ X (CDR X))  (GO W>
DEFINE ((
  (LEVEL (LAMBDA (Q SP) (PROG (X)
   A  (COND ((NULL Q) (RETURN NIL)))
      (TTAB SP)       (SETQ X (CAAR Q))
    B (PRIN1 (CAR X))
      (CCND ((SETQ X (CDR X)) (PROG2 (PRIN1 BLANK) (GO B))))
      (CGND ((MEMB (CAR Q) ELSL) (PROG2  (PRINT (QUOTE
               $$$   <OCCURS ABOVE>$))
               (PROG2 (SETQ Q (CDR Q))) (GO A)) )))
      (TERPRI)     (CSETQ ELSL (CCNS (CAR Q) ELSL))
      (COND (ONLF (RETURN NIL)))
      (LEVEL (CDDR (CAR Q)) (PLUS SP 3))
      (SETQ Q (CDR Q))
      (GO A>
DEFINE ((
  (CREATE (LAMBDA (X FAT) (PROG (Y Z)
    (SETQ Z (GENSYM1 (QUOTE FATH)))
    (CSET Z (CUND (FAT (CONS FAT NIL)) (T NIL)))
    (RPLACD X (CUNS Z NIL))
    (SETQ Y (CAR X))
 C1 (CEFLIST (LIST (LIST (CAR Y) (CCNS X (GET (CAR Y) USES)))) USES)
    (CUND ((SETQ Y (CDR Y)) (GO C1)))
    (RETURN X>
DEFINE ((
  (NCMN (LAMBDA (X) (PROG (Y RES USE1 USE2)
     (SETQ Y (SETQ X (CAR X)))
     (SETQ RES (GET (CAR X) USES))
  N1 (COND ((SETQ X (CDR X)) (SETQ USE2 NIL))
           (ONIC (GO N3)))
           (T (RETURN RES)))
     (SETQ USE1 (GET (CAR X) USES))
  N2 (CCND ((MEMB (CAR USE1) RES)(SETQ USE2 (CONS (CAR USE1) USE2))))
     (CCND ((SETQ USE1 (CDR USE1)) (GO N2)))
     (SETQ RES USE2)     (GO N1)
  N3 (CCND (RES (SETQ USE2 (CAAR RES)))
           (T (RETURN USE1))))
  N4 (CCND ((MEMB (CAR USE2) Y) NIL)
           (T (GC N5)))
     (CCND ((SETQ USE2 (CDR USE2)) (GC N4)))
     (SETQ USE1 (CCNS (CAR RES) USE1))
  N5 (SETQ RES (CCR RES))     (GC N3>
DEFINE ((
  (PUCP (LAMBDA (FLG)  (PROG (RNG1 RNG2)
    (CCND ((EQ NXTW EMRK) (MESSCP)))
    (CSETQ PUTF FLG)      (CSETQ CNLF NIL)
    (CCND ((EQ NXTW (QUOTE ONLY))
       (CCND (PUTF (MESS IMPRDLIM NXTW NIL))
             (T (PROG2 (FTCH) (CSETQ CNLF T))))))
    (SETQ RNG1 (RANGES NIL))
    (CCND ((MEMB NXTW INTO) (FTCH))
          ((AND (NOT PUTF) (EQ NXTW EMRK)) (PROG2
             (SETQ RNG2 (CONS XTOP NIL)) (GO P1)))
          (T (PROG2 (MESSOP)  (RETURN NIL))))
    (SETQ RNG2 (RANGES NIL))
 P1 (CCND ((NOT (EQ NXTW EMRK)) (MESS IMPRDLIM NXTW NIL)))
    (COND (SYNTAX (PROG2 (CSETQ PUTF NIL) (RETURN NIL))))
```

```
    (COIT (QUOTE PUT1) RNG1 RNG2)  (CSETQ PUTF NIL>
DEFINE ((
  (MEMB (LAMBDA (X L) (PROG NIL
  M (COND ((NULL L) (RETURN NIL))
          ((EQ X (CAR L)) (RETURN X)))
    (SETQ L (CDR L))  (GO M>
DEFINE ((
  (MEMBER (LAMBDA (X L) (PROG NIL
  M (COND ((NULL L) (RETURN NIL))
          ((EQUAL X (CAR L)) (RETURN (CAR L))))
    (SETQ L (CDR L))  (GO M>
DEFINE ((
  (PUT1 (LAMBDA (TBPL WHRE) (PROG (TBCM Q CARQ)
     (SETQ TBCM TBPL)
  P2 (COND ((NULL TBCM) (RETURN WHRE)))
     (SETQ TBPL (CAR TBCM))      (SETQ Q WHRE)
  P1 (SETQ CARQ (CAR Q))
     (COND ((NOT PUTF) (PROG2 (CSETQ ELSL NIL)
                              (SETQ TBPL (COPY TBPL NIL))))
           ((EQ TBPL CARQ) (PROG2 (COND (SBS(PROG2 (PRIN1 (QUOTE
     $$$NODE CANNOT BE PLACED IN ITSELF... $))(MESSN CARQ T))))(GO Q2)))
           ((ISIN CARQ TBPL)(PROG2(COND (SBS(PROG.(PROG2(MESSN CARQ NIL)
        (PRIN1 (QUOTE $$$ IS ALREADY IN $))) (MESSN TBPL T)))) (GO Q2)))
           ((MEMB CARQ (CAR (GET (CADR TBPL) APVAL)))
                          (PROG2 (COND (SBS (PROG2 (PROG2  (MESSN TBPL NIL)
        (PRIN1 (QUOTE $$$ IS ALREADY IN $))) (MESSN CARQ T)))) (GO Q2)))
           (T (CSETQ XTOP (EFFACE TBPL XTOP))))
     (CSETQ KFLG T)
     (COND ((EQ Q XTOP) (PROG2 (NCONC XTOP (CONS TBPL NIL)) (GO Q1)))
           (T (PROG2
              (NCONC CARQ  (CONS TBPL NIL))
              (CSET(CADR TBPL)(CONS CARQ(CAR(GET(CADR TBPL) APVAL)))))))
  Q2 (COND ((SETQ Q (CDR Q)) (GO P1)) (T NIL))
  Q1 (SETQ TBCM (CDR TBCM))      (GO P2>
DEFINE ((
  (EFFACE (LAMBDA (X L) (PROG (Y HL)
    (COND ((NULL L) (RETURN NIL))
          ((EQ X (CAR L)) (RETURN (CDR L))))
    (SETQ HL L)
  E (SETQ Y L)      (SETQ L (CDR L))
    (COND ((NULL L) (RETURN HL))
          ((EQ X (CAR L)) (PROG2 (RPLACD Y (CDR L)) (RETURN HL)))
          (T (GO E>
DEFINE ((
  (COPY (LAMBDA (X FAT) (PROG (Y Z)
    (SETQ Y (CONS (COPT (CAR X)) NIL))
    (CSETQ ELSL (CONS (CONS X Y) ELSL))
    (SETQ FAT (CREATE Y FAT))   (COND (ONLF (RETURN Y))),
    (SETQ X (CDDR X))
  C (COND ((NULL X) (RETURN Y)))
    (NCONC Y (COND ((SETQ Z (ASOC (CAR X) ELSL)) (CONS Z NIL))
                   (T (CONS (COPY (CAR X) FAT) NIL))))
    (SETQ X (CDR X))   (GO C>
DEFINE ((
  (ASOC (LAMBDA (X Y) (PROG NIL
   A (COND ((EQ (CAAR Y) X) (RETURN (CDAR Y)))
           ((SETQ Y (CDR Y)) (GO A)))   (RETURN NIL>
DEFINE ((
  (COPT (LAMBDA (X) (PROG (Y)
   C (COND ((NULL X) (RETURN  Y)))
     (SETQ Y (NCONC Y (CONS (CAR X) NIL)))
```

```
      (SETQ X (CDR X))    (GO C>
DEFINE ((
  (MESS (LAMBDA (X Y Z) (PROG NIL
     (COND ((OR (AND Z (NOT SBS)) (AND RUN (NOT Z))) (RETURN NIL)))
     (PRIN1 X)    (PRINT Y)
     (RESET)                 (BEGIN (QUOTE MAIN) NIL>
DEFINE ((
  (MESSOP (LAMBDA NIL (PROG NIL
     (PRINT (QUOTE $$$MISSING OPERAND$))
     (RESET)              (BEGIN (QUOTE MAIN) NIL>
DEFINE ((
  (MESC (LAMBDA (X Y) (PROG NIL
    (COND (SBS (MESS X Y T)))
    (PRIN1 X)    (PRIN1 Y)   (PRIN1 (QUOTE $$$  LINE $))
    (PRINT (CAAR THISLINE))
    (RESET)     (BEGIN (QUOTE MAIN) NIL>
DEFINE ((
  (RESET (LAMBDA NIL (PROG NIL
    (CSETQ CFLG NIL)  (CSETQ SYNTAX NIL) (CSETQ RUN NIL)   (CSETQ SBS T)
    (CSETQ AXP NIL)   (CSETQ PUTE NIL>
DEFINE ((
 (SHOWU (LAMBDA (X) (PROG (Y)
  A (COND ((NULL X) (RETURN NIL)))
    (PRINT (CAR X))
    (PRINT (SETQ Y (COND ((AND (NUMBERP (CAR X)) (NOT (FIXP (CAR X))))
                              (GET (MEMBER (CAR X) NUMS) USES))
                         (T (GET (CAR X) USES))))))
    (COND (Y (PROG (Z)
                 (SETQ Z Y)
               P (PRINT (CAR (GET (CADR (CAR Z)) APVAL)))
                 (COND ((MEMB (CAR Z)(CDR Z)) (MESS (QUOTE DUPLICATES)
                          (CAR Z))))
                 (COND ((NULL (SETQ Z (CDR Z))) (RETURN NIL)))
                 (GO P)) ))
    (SETQ X (CDR X)) (GO A>
DEFINE ((
  (ISIN (LAMBDA (UI B) (PROG (X)
    (COND ((NULL (SETQ X (CAR(GET (CADR UI ) APVAL)))) (RETURN NIL)))
  I (COND ((OR (EQ (CAR X) B) (ISIN (CAR X) B)) (RETURN UI))
          ((SETQ X (CDR X)) (GO I)))
    (RETURN NIL>
DEFINE ((
  (DELE (LAMBDA (X) (PROG (RNGE)
    (COND ((EQ EMRK NXTW) (MESSOP)))
    (CSETQ DELT X)
    (COND ((EQ NXTW (QUOTE ONE)) (PROG2 (FTCH) (SETQ RNGE T))))
    (SETQ X (RANGES NIL))
    (COND (RNGE (PROG NIL
           (SETQ RNGE X)
         Z (RPLACD (CAR RNGE) NIL)
           (COND ((SETQ RNGE (CDR RNGE)) (GO Z)))
           (RETURN NIL) )))
    (COND ((EQ NXTW (QUOTE FROM))(PROG2 (FTCH)(SETQ RNGE (RANGES NIL))))
          (T (SETQ RNGE (CONS XTOP NIL))))
    (COND ((NOT (EQ NXTW EMRK)) (MESS IMPROLIM NXTW NIL)))
    (COND (SYNTAX (RETURN NIL)))
    (DOIT (QUOTE DEL1) X RNGE>
DEFINE ((
  (DEL1 (LAMBDA (X RNG) (PROG (TEM1 FL TEMR)
 D1 (COND((SETQ FL(SETQ TEMR(CAR(GET (CADR(SETQ TEM1 (CAR X)))APVAL))))
           (COND ((PROG (FLG XRNG)     DS2  (SETQ XRNG RNG)
```

```
         DS1 (COND((OR(EQ(CAR TEMR)(CAR XRNG))(ISIN (CAR TEMR)(CAR XRNG)))
                (PROG2 (PROG2 (EFFACE TEM1 (CAR TEMR)) (SETQ FLG T))
                      (SETQ FL (EFFACE (CAR TEMR) FL))   )   )
                 ((SETQ XRNG (CDR XRNG)) (GO DS1)))
              (COND ((SETQ TEMR (CDR TEMR)) (GO DS2))
                    (T (CSET (CADR TEM1 ) FL)))
              (RETURN FLG))  NIL)
                   (T (PROG2 (COND (SBS (PROG2 (MESSN (CAR X) NIL)
                     (PRINT(QUOTE $$$ NOT IN RANGE TO SEVER/DELETE$)))))
                           (GO D2)))))
            ((NOT (EQ RNG XTOP))(PROG2 (COND (SBS(PROG2 (MESSN (CAR X) NIL)
                    (PRINT (QUOTE $$$ HAS NO FATHERS$))))) (GO D2))))
     (COND (FL NIL)
           (DELT (PROG2 (COND ((AND (EQ XTOP RNG) (EQ TEM1 (CAR XTOP)))
                  (CSETQ XTOP (SETQ RNG (CDR XTOP))))
                           (T (CSETQ XTOP (EFFACE TEM1 XTOP))))
                      (DISC TEM1)))
           ((NULL (MEMB TEM1 XTOP))  (CSETQ XTOP (CONS TEM1 XTOP))))
     (CSETQ KFLG T)
  D2 (COND ((SETQ X (CDR X)) (GO D1)) (T NIL))
     (RETURN T>
DEFINE ((
  (DISC (LAMBDA (X) (PROG (Y LBLS)
     (SETQ LBLS (CAR X))        (SETQ Y (CDDR X))
  D2  (DEFLIST (LIST (LIST (CAR LBLS)
                (EFFACE X (GET (CAR LBLS) USES)))) USES)
     (COND ((SETQ LBLS (CDR LBLS)) (GO D2)))
  D1  (COND ((NULL Y) (RETURN NIL))
            ((GREATERP (NMEM (CAR(GET (CADR(CAR Y))APVAL))) 1)
                (PROG2 (DEL1 Y (CONS X NIL))(RETURN NIL))))
     (DISC (CAR Y))
     (SETQ Y (CDR Y))
     (GO D1>
DEFINE ((
  (MAKE (LAMBDA (X) (PROG (CIDS RNGS RNGE EIDS IDS ID Z Y)
     (COND ((EQ NXTW EMRK) (MESSOP)))
     (CSETQ ELSL NIL)
     (SETQ RNGS (RANGES NIL))      (SETQ IDS (CAR ELSL))
     (COND ((MEMB NXTW BYAS) (FTCH))
           ((AND (NULL X) (EQ EMRK NXTW))
              (COND (SYNTAX (RETURN NIL))  (T (GO UN5))) )
           (T (MESSOP)))
     (COND ((EQ NXTW EMRK) (MESSOP)))
     (CSETQ CFLG T)
     (SETQ IDS (CAR (ITM)))   (CSETQ CFLG NIL)
     (COND (SYNTAX (RETURN NIL)))
     (COND (X (GO LB)))
     (COND((NOT(EQ NXTW EMRK)) (MESS IMPROLIM NXTW NIL)))
 UN5 (COND ((NULL (SETQ RNGE (CAR RNGS)))(PROG2 (COND ((NOT RUN)
                 (PRINT NLRG)))  (GO UN4))))
 UN3 (SETQ Y (COPT (CAR (SETQ X (CAR RNGE)))))
     (SETQ Z IDS)
 UN1 (COND ((MEMB (CAR Z) (CAR X)) (SETQ Y (EFFACE (CAR Z) Y)))
           (T (PROG2(COND (SBS  (PROG2 (PROG2
                 (PRIN1 (CAR Z)) (PRIN1 (QUOTE $$$ DOES NOT LABEL $)) )
                 (MESSN X T)))) (GO UN2))))
     (COND ((SETQ Z (CDR Z)) (GO UN1))
           ((NULL Y) (PROG2 (COND (SBS (PROG2
              (PRIN1 (QUOTE $$$CANNOT REMOVE ALL LABELS FROM $))
              (MESSN X T)))) (GO UN2))))
     (RPLACA X Y)    (SETQ Z IDS)
```

```
 UN6 (DEFLIST (LIST (LIST (CAR Z) (EFFACE X (GET (CAR Z) USES))))USES)
     (CCND ((SETQ Z (CDR Z)) (GC UN6)))
     (CSETQ KFLG T)
 UN2 (CCND ((SETQ RNGE (CDR RNGE)) (GO UN3))(T NIL))
 UN4 (COND ((SETQ RNGS (CDR RNGS)) (GO UN5)))
     (RETURN NIL)
  LE (SETQ ID NIL)      (SETQ CIDS (COPT IDS))
     (COND ((EQ NXTW (QUOTE BEFCRE)) (PROG2 (FTCH) (SETQ ID NXTW))))
     (CCNC ((EQ ID EMRK) (MESSOP))
           (T (FTCH)))
     (COND((NOT(EQ NXTW EMRK)) (MESS IMPROLIM NXTW NIL)))
 LB6 (COND ((NULL (SETQ RNGE (CAR RNGS))) (PROG2 (COND ((NOT RUN)
                      (PRINT NLRG))) (GO LB4))))
 LB5 (SETQ Y (CAAR RNGE))   (SETQ IDS (SETQ Z (COPT CIDS)))
 LB1 (COND ((MEMB (CAR Z) Y)  (PROG2 (COND (SBS (PROG2 (PROG2
               (PRIN1 (CAR Z)) (PRIN1 (QUOTE $$$ ALREADY LABELS $))   )
               (MESSN (CAR RNGE) T) ))) (GO LB2)))
           ((SETQ Z (CDR Z)) (GO LB1)))
     (SETQ Z CIDS)
     (CCND ((NULL IC)     (SETQ EIDS Y))
           (T (SETQ EIDS IDS)))
 LB9 (COND ((NULL (CDR EIDS)) NIL)
           (T (PROG2 (SETQ EIDS (CDR EIDS)) (GO LB9))))
     (CCNC ((NULL IC)     (PRCG2 (RPLACD EIDS  IDS ) (GO LB8))))
     (COND ((EQ ID (CAR Y)) (PRCG2 (PROG2  (RPLACD EIDS Y)
               (RPLACA (CAR RNGE)  IDS ))  (GO LB8))))
     (SETQ X (CDR Y))
 LB3 (COND ((NULL X) (PROG NIL
               (CGND ((NOT SBS) (RETURN NIL)))
               (PRIN1 ID) (PRIN1 (QUCTE $$$ DOES NOT LABEL $))
               (MESSN (CAR RNGE) NIL)  (PRIN1 (QUOTE $$$ -- $))
               (MESSN (CONS IDS NIL) NIL)
               (PRINT (QUOTE $$$ ADDED AT END$))
               (RPLACD Y  IDS )  ))
           ((EQ ID (CAR X))(PROG2(RPLACD Y IDS )(RPLACD EIDS X)))
           (T (PRCG2 (PROG2 (SETQ Y X)  (SETQ X (CDR X))) (GO LB3))))
 LB8 (DEFLIST(LIST(LIST(CAR Z) (CCNS(CAR RNGE) (GET(CAR Z) USES))))USES)
     (CCNC ((SETQ Z (CDR Z)) (GC LB8)))
     (CSETQ KFLG T)
 LB2 (COND ((SETQ RNGE (CDR RNGE)) (GO LB5)) (T NIL))
 LB4 (CCND ((SETQ RNGS (CDR RNGS)) (GO LB6)))
     (RETURN NIL>
DEFINE ((
  (CCUT (LAMBDA NIL (PROG (X)
    (SETQ X (RANGE NIL))
    (CCNC((NOT(EQ NXTW EMRK)) (MESS IMPROLIM NXTW NIL)))
    (CCND (SYNTAX (RETURN NIL)))
    (PRINT (NMEM X>
DEFINE ((
  (NMEM (LAMBDA (X) (PROG (K)
    (SETQ K 0)
  N (CCND ((NULL X) (RETURN K)))
    (SETQ K (ADD1 K))     (SETQ X (CDR X))  (GO N>
DEFINE ((
  (CRET (LAMBDA NIL (PROG (X)
    (CSETQ CFLG T)
    (COND ((EQ (CAAR (SETQ X (ITMS))) EMRK) (MESSOP))
          ((NCT (EQ NXTW EMRK))(MESS IMPROLIM NXTW NIL)))
    (CCND (SYNTAX (RETURN NIL)))
  C (COND (X (CREATE (CAR X) NIL)) (T (GO C1)))
    (CSETQ XTOP (CCNS (CAR X) XTOP))  (SETQ X (CDR X))
```

```
    (CSETQ KFLG T)  (GU C)
  C1 (CSETQ CFLG NIL)  (RETURN NIL>
DEFINE ((
   (WHEREPRT (LAMBDA (RNGE) (PROG (X W)
       (FTCH)    (CSETQ ISNF T)
       (COND (SYNTAX NIL) ((NULL RNGE) (RETURN NIL)))
       (COND ((EQ NXTW (QUOTE COUNT)) (PROG2 (FTCH)
                   (SETQ W (QUOTE KEEPCNT))))
             ((EQ NXTW (QUOTE VALUE)) (PROG2 (FTCH)
                   (SETQ W (QUOTE KEEPVAL))))
             (T (SETQ W (QUOTE KEEP))))
       (SETQ X (RANGE 0))
       (RETURN (W X RNGE>
DEFINE ((
   (KEEP (LAMBDA (TSTNODES NODES) (PROG (X H)
       (COND (SYNTAX (RETURN NIL)))
       (COND ((NULL TSTNODES) (RETURN (COND (ISNF NIL) (T NODES)))))
   K1  (SETQ H TSTNODES)
       (COND ((NULL NODES) (RETURN X)))
   K2  (COND ((NULL H) (COND (ISNF NIL)(T (SETQ X (CONS (CAR NODES) X)))))
             ((OR (EQ (CAR H) (CAR NODES)) (ISIN (CAR H)(CAR NODES)))
                (COND (ISNF (SETQ X (CONS (CAR NODES) X))) (T NIL)))
             (T (PROG2 (SETQ H (CDR H)) (GO K2))))
       (SETQ NODES (CDR NODES)) (GO K1>
DEFINE ((
   (KEEPCNT (LAMBDA (TSTNODES NODES) (PROG (X REL NUM CNT H)
       (SETQ REL (FTCH)) (SETQ NUM (FTCH))
       (COND((NOT(NUMBERP NUM))(MESS(QUOTE $$$NOT A NUMBER... $) NUM NIL))
             ((NOT (MEMB REL RELS)) (MESS (QUOTE
                $$$NOT A RELATIONAL OPERATOR... $) REL NIL)))
       (COND (SYNTAX (RETURN NIL)))
       (COND ((NULL TSTNODES) (RETURN NIL)))
   K1  (SETQ H TSTNODES)
       (COND ((NULL NODES) (RETURN X)))
       (SETQ CNT 0)
   K2  (COND ((NULL H) (GO K3))
             ((ISIN (CAR H) (CAR NODES)) (SETQ CNT (ADD1 CNT))))
       (SETQ H (CDR H))     (GO K2)
   K3  (COND   ((NRELOPM CNT REL NUM) (SETQ X (CONS (CAR NODES) X))))
       (SETQ NODES (CDR NODES))      (GO K1>
DEFINE ((
   (KEEPVAL (LAMBDA (TSTNODES NODES) (PROG (X Y REL NUM H)
       (COND((NOT(NUMBERP NUM))(MESS(QUOTE $$$NOT A NUMBER... $) NUM NIL))
             ((NOT (MEMB REL RELS)) (MESS (QUOTE
                $$$NOT A RELATIONAL OPERATOR... $) REL NIL)))
       (COND (SYNTAX (RETURN NIL)))
     (SETQ REL (FTCH))    (SETQ NUM (FTCH))
     (COND ((NULL TSTNODES) (RETURN NIL)))
  K1 (SETQ H TSTNODES)
     (COND ((NULL NODES) (RETURN X)))
  K2 (COND ((NULL H) NIL)
           ((AND (SETQ Y (FNUM (CAAR H))) (OR (EQ (CAR H) (CAR NODES))
                   (ISIN (CAR H) (CAR NODES)))    (NRELOPM Y REL NUM))
              (SETQ X (CONS (CAR NODES) X)))
           (T (PROG2 (SETQ H (CDR H)) (GO K2))))
     (SETQ NODES (CDR NODES))   (GO K1>
DEFINE ((
   (NRELOPM (LAMBDA (N RELP M)  (PROG NIL
     (RETURN
       (COND ((EQ RELP (QUOTE EQ)) (EQUAL N M))
             ((EQ RELP (QUOTE NEQ)) (NOT (EQUAL N M)))
```

```
             ((EQ RELP (QUOTE GR)) (GREATERP N M))
             ((EQ RELP (QUOTE GE)) (OR (GREATERP N M) (EQUAL N M)))
             ((EQ RELP (QUOTE LS)) (LESSP N M))
             ((EQ RELP (QUOTE LE)) (OR (LESSP N M) (EQUAL N M>
DEFINE ((
  (MESSN (LAMBDA (X FLG) (PROG NIL
    (COND ((NOT SBS) (RETURN NIL)))
    (SETQ X (CAR X))
  A (PRIN1 (CAR X))
    (COND ((SETQ X (CDR X)) (PROG2 (PRIN1 BLANK) (GO A))))
    (COND (FLG (TERPRI>
DEFINE ((
  (INPT (LAMBDA NIL (PROG (INPL NOTF X XH Y YH)
    (CSETQ ELSL (ITM))
    (COND (SYNTAX (GO S1))
          ((NULL (SETQ YH (NCMN ELSL))) (RETURN NIL)))
    (SETQ INPL (CONS YH NIL))
 S1 (COND ((EQ NXTW (QUOTE IS)) (PROG2 (FTCH) (CSETQ ISNF
             (COND ((EQ NXTW (QUOTE NOT)) (PROG2 (FTCH) NIL))(T T))))))
 Z3 (COND ((MEMB NXTW NOTN) NIL)  (SYNTAX (RETURN NIL))  (T (GO Z0)))
    (COND ((EQ NXTW (QUOTE NOT))
             (PROG2 (SETQ INPL (CONS NIL INPL)) (FTCH)))
          (T (SETQ INPL (CONS T INPL))))
    (COND ((EQ NXTW (QUOTE IN)) (FTCH))
          (T (MESS IMPRDLIM NXTW NIL)))
    (SETQ X (ITM))
    (COND (SYNTAX (GO Z3))
          ((NULL (SETQ YH (NCMN X))) (RETURN NIL)))
    (SETQ INPL (CONS YH INPL))
    (GO Z3)
 Z0 (SETQ YH (SETQ Y (CAR INPL)))  (SETQ INPL (CDR INPL))
 Z4 (COND ((NULL INPL) (RETURN YH)))
    (CSETQ NOTF (CAR INPL))  (SETQ X (CADR INPL))  (SETQ XH NIL)
    (SETQ INPL (CDDR INPL))
 Z1 (COND ((ISIN (CAR X) (CAR Y)) (PROG2
             (COND (NOTF (SETQ XH (CONS (CAR X) XH))) (T NIL))
             (GO Z2))))
    (COND ((SETQ Y (CDR Y)) (GO Z1)))
    (COND (NOTF NIL) (T (SETQ XH (CONS (CAR X) XH))))
 Z2 (SETQ Y YH)
    (COND ((SETQ X (CDR X)) (GO Z1)))
    (COND ((SETQ Y (SETQ YH XH))  (GO Z4)))
    (RETURN NIL>
DEFINE ((
  (RANGES (LAMBDA (FLG) (PROG (X)
    R (SETQ X (CONS (RANGE FLG) X))
      (COND ((EQ NXTW (QUOTE AND)) (PROG2 (FTCH) (GO R))))
      (RETURN X>
DEFINE ((
  (DOIT (LAMBDA (F XH YH) (PROG (X)
    (SETQ X XH)
 D3 (COND ((NULL (CAR YH)) (PROG2 (PRINT NLRG) (GO D4))))
 D1 (COND ((NULL (CAR X)) (PROG2 (PRINT NLRG) (GO D2))))
    (F (CAR X) (CAR YH))
 D2 (COND ((SETQ X (CDR X)) (GO D1)))
    (SETQ X XH)
 D4 (COND ((SETQ YH (CDR YH)) (GO D3)))
    (RETURN NIL>
DEFINE ((
  (FNUM (LAMBDA (X) (PROG NIL
   N (COND ((NUMBERP (CAR X)) (RETURN (CAR X)))
```

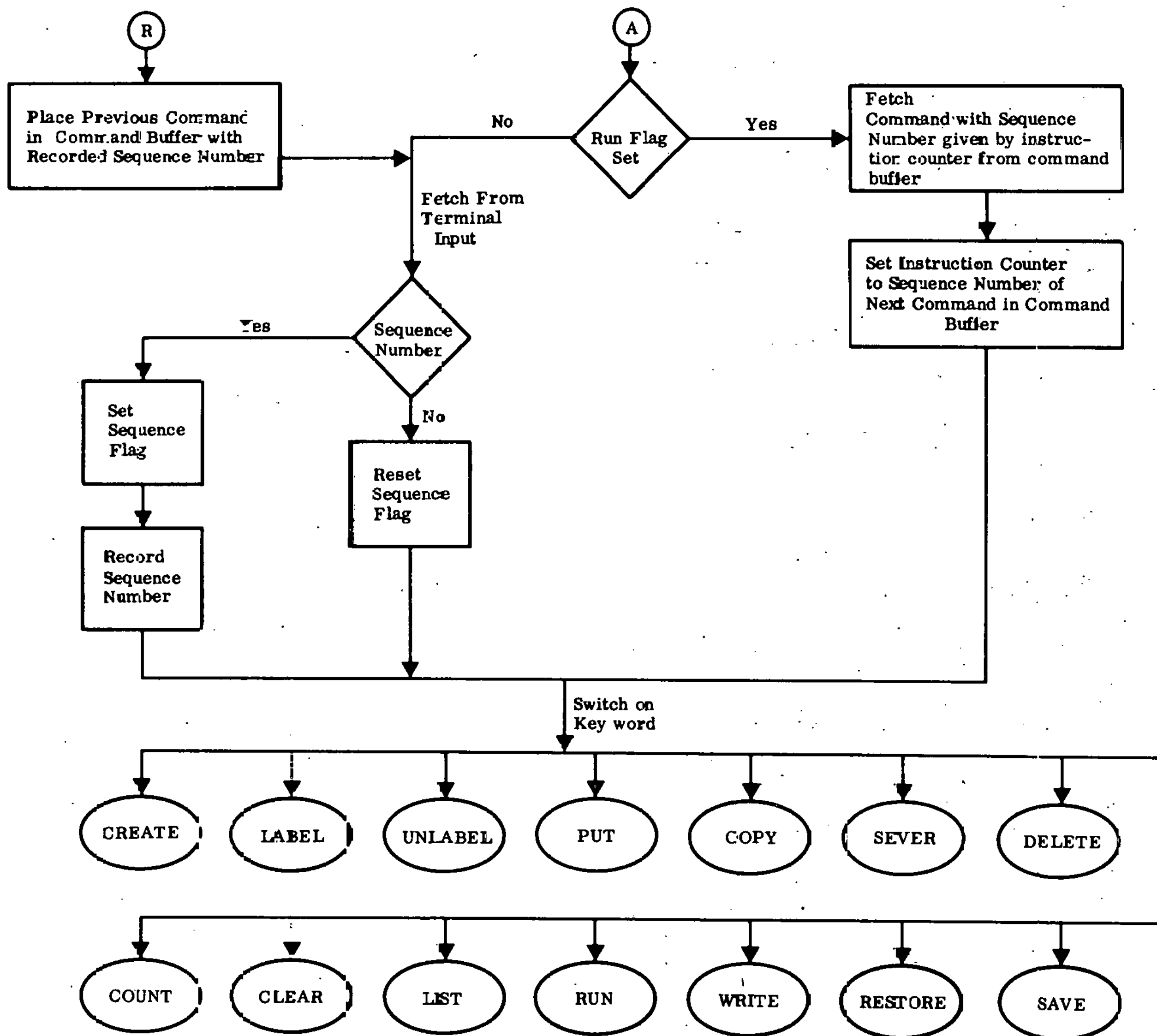
```
          ((SETQ X (CDR X)) (GO N)))
    (RETURN NIL>
DEFINE ((
  (LIS (LAMBDA NIL (PROG (Y X)
    (COND ((EQ NXTW EMRK)
       (COND ((SETQ X LINES) (RETURN (PROG NIL
                               L (PRINT (CAR X))
                                 (COND ((SETQ X (CDR X)) (GO L))))))
            (T (MESS (QUOTE $$$NOTHING TO LIST$) BLANK T)))))
    (COND ((NOT (AND (NUMBERP (SETQ X (FTCH))) (FIXP X)))
             (MESS (QUOTE $$$LINE NUMBER MUST BE INTEGER$) BLANK NIL))
          (T
            (COND ((NOT (AND (SETQ Y (LINESRCH X 0)) (EQ (CAAR Y) X)))
              (MESS (QUOTE $$$NOTHING TO LIST$) BLANK T))
                (T (RETURN (PRINT (CAR Y)))))))      >
DEFINE ((
    (LINESRCH (LAMBDA (L FLG)  (PROG (Y Z)
       (COND ((OR (NULL LINES) (GREATERP (CAAR LINES) L))
         (RETURN (COND ((OR (NULL FLG) (NUMBERP FLG)) NIL)
                      (T (CSETQ LINES (CONS LINE LINES))))))
            ((EQ (CAAR LINES) L)
         (RETURN (COND ((NULL FLG) (CSETQ LINES (CDR LINES)))
                      ((NUMBERP FLG) LINES)
                      (T (RPLACA LINES LINE))))))
       (SETQ Z LINES)      (SETQ Y (CDR LINES))
    M (COND ((OR (NULL Y) (GREATERP (CAAR Y) L))
         (RETURN (COND ((OR (NULL FLG) (NUMBERP FLG)) NIL)
                      (T (RPLACD Z (CONS LINE Y))))))
            ((EQ (CAAR Y ) L)
         (RETURN (COND ((NULL FLG) (RPLACD Z (CDR Y)))
                      ((NUMBERP FLG) Y)
                      (T (RPLACA Y LINE))))))
       (SETQ Z Y)     (SETQ Y (CDR Y))      (GO M>
DEFINE ((
  (RNN (LAMBDA NIL (PROG (X Y)
    (COND ((EQ NXTW EMRK) (SETQ Y LINES))
          ((NOT (AND (NUMBERP (SETQ X (FTCH))) (FIXP X)))
            (MESS LNMI BLANK NIL))
          ((NOT (AND (SETQ Y (LINESRCH X 0))  (EQ (CAAR Y) X)))
            (MESS (QUOTE $$$NO SUCH LINE NUMBER$) BLANK NIL)))
    (CSETQ THISLINE Y)   (CSETQ LINE (CDAR THISLINE))
    (CSETQ SBS NIL)     (CSETQ RUN T>
DEFINE ((
  (STPP (LAMBDA (X) (PROG NIL
    (PRINT BLANK)       (PRIN1 (QUOTE STOP))
    (PRIN1 (COND (X (QUOTE $$$ AT LINE $))(T (QUOTE $$$ AFTER LINE $))))
    (PRINT (CAAR THISLINE))
    (CSETQ THISLINE NIL)    (CSETQ SBS T)    (CSETQ RUN NIL)
    (BEGIN (QUOTE MAIN) NIL>
DEFINE ((
  (LETT (LAMBDA NIL (PROG (W Z X Y)
    (COND ((NOT (ATOM NXTW)) (MESS
              (QUOTE $$$ILLEGAL ALGEBRAIC VARIABLE$)
              BLANK NIL)))
    (CSETQ AXP T)  (SETQ X (FTCH))
    (SETQ W (GET X PVAL))
    (COND ((NOT (ATOM NXTW)) (PROG NIL
      (COND ((NUMBERP W) (MESC AVCS X)))
      (SETQ X (CONS X NIL))
      (CSETQ OUTP (CONS NIL NIL))     (PPOP (FTCH))
      (CSETQ OUTP (CDR OUTP))
```

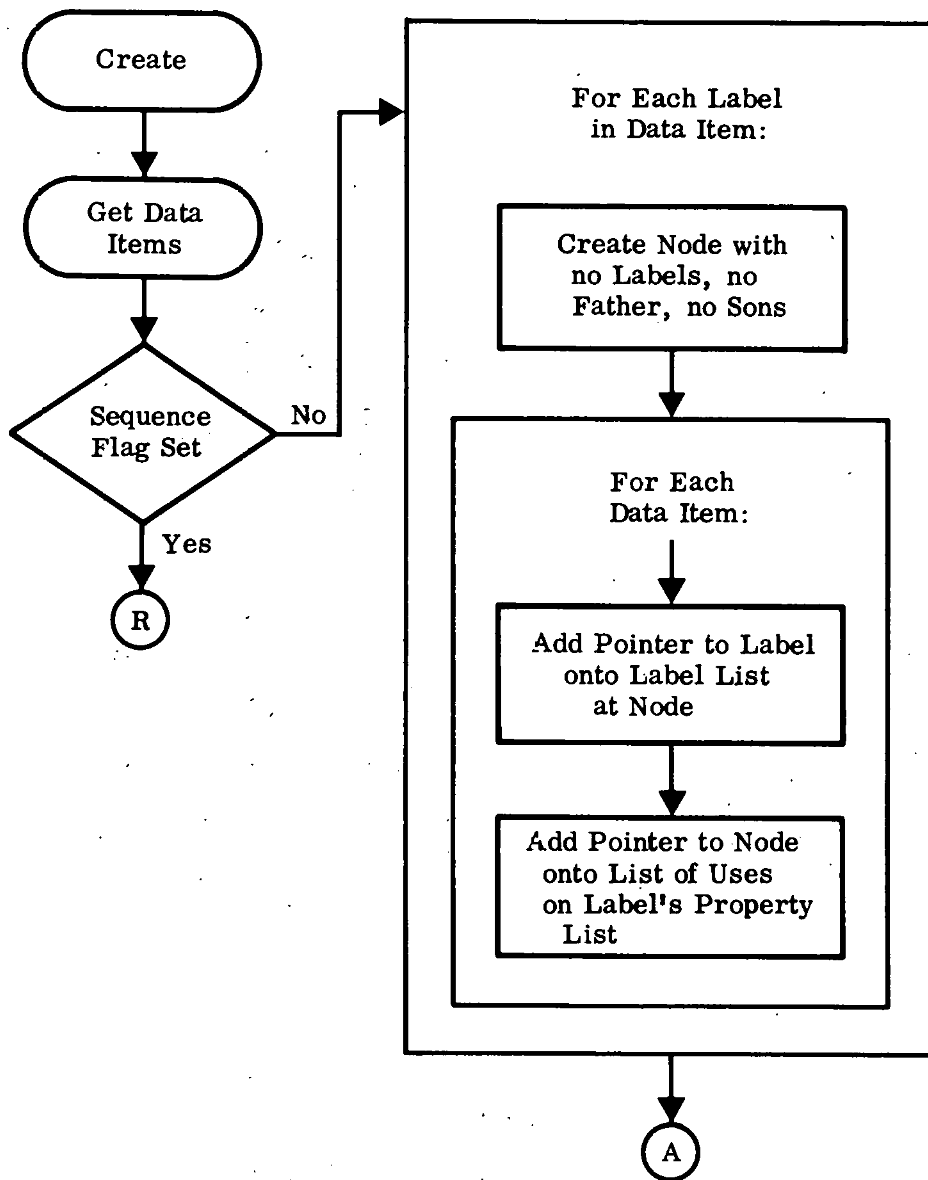
```
      (COND (SYNTAX (RETURN CUTP)))
      (SETQ Z (EVLL OUTP))   ))
          ((OR (NULL W) (NUMBERP W)) NIL)
          (T (MESC AIMS X)))
    (CONC ((EQ NXTW EQSIGN) (FTCH))
          (T (MESS IMPRDLIM NXTW NIL)))
    (COND ((EQ NXTW EMRK) (MESSCP)))
    (SETQ Y (CONS NIL NIL))
  D (CCND ((EQ NXTW EMRK) NIL)
          (T (PROG2 (APPEND1 Y (FTCH)) (GO D))))
    (CSETQ AXP NIL)    (CSETQ OUTP (CONS NIL NIL))
    (PPOP (CDR Y))  (CSETQ CUTP (CDR OUTP))
    (COND (SYNTAX (RETURN OUTP)))
    (SETQ Y (EVLL OUTP))
    (LCND ((ATOM X) (CUND ((CNUM Y) (DEFLIST (LIST (LIST X Y)) PVAL))
                          (T NIL)))
          ((SETQ W (SASS Z W)) (RPLACD W Y))
          (T (DEFLIST (LIST (LIST (CAR X) (CONS (CONS Z Y)
                          (GET (CAR X) PVAL)))) PVAL)))
    (CSETQ KFLG T)      (RETURN Y>
DEFINE ((
  (AEXP (LAMBDA (INP) (PROG (FLG INC X)
      (CCND ((EQ (CAR INP) PLUSS) (SETQ INP (CDR INP)))
            ((EQ (CAR INP) DASH) (PROG2
               (CSETQ STAK (CONS (CONS (QUOTE NEG) 4) STAK))
               (SETQ INP (CDR INP)))))
    A (COND ((NULL INP) (COND (FLG (RETURN CUTP))
                          (T (MESS IAE BLANK NIL))))
            ((ATOM INP) (SETQ INP (CCNS INP NIL))))
      (SETQ X (CAR INP))
      (COND ((ATOM X)
         (COND ((SETQ INC (SASSOC X OPSP NIL)) (PROG NIL
                (COND ((NULL FLG) (MESS IAE BLANK NIL)))
               C (COND ((LESSP (CDAR STAK) (CDR INC))
                         (CSETQ STAK (CONS INC STAK)))
                   (T (PROG2 (PROG2 (CCND ((NULL (CAAR STAK))
                                          (RETURN (SETQ FLG NIL)))
                                     (T (APPEND1 OUTP (CAAR STAK))))
                             (CSETQ STAK (CDR STAK)))(GO C))))))
               (T (PROG2 (PRCG2 (CCND (FLG (MESS IAE BLANK NIL)))
                        (APPEND1 CUTP X))
                        (COND ((AND (CDR INP) (NOT (ATOM (CADR INP)))
                                    (NOT (NUMBERP X)))
                          (PROG2 (PPOP (CAR (SETQ INP (CDR INP))))
                                 (APPEND1 OUTP SUB)))) ))))
            (T (PPOP X)))
      (SETQ FLG (NOT FLG))      (SETQ INP (CDR INP))    (GO A>
DEFINE ((
  (PPCP (LAMBDA (X) (PROG NIL
    (CSETQ STAK (CONS (CONS LPAR 0) STAK))     (AEXP X)
  P (CCND ((EQ (CAAR STAK) LPAR) (RETURN (CSETQ STAK (CDR STAK)))))
    (APPEND1 OUTP (CAAR STAK))    (CSETQ STAK (CDR STAK))    (GO P>
DEFINE ((
  (CNUM (LAMBDA (X)
    (COND ((NUMBERP X) X)
          (T (MESC AIMS BLANK>
DEFINE ((
  (CARY (LAMBDA (X)
    (CCND ((NOT (ATCM X)) X) (T (MESC AVCS BLANK>
DEFINE ((
  (EVLL (LAMBDA (X) (PROG (Y VAL)
```

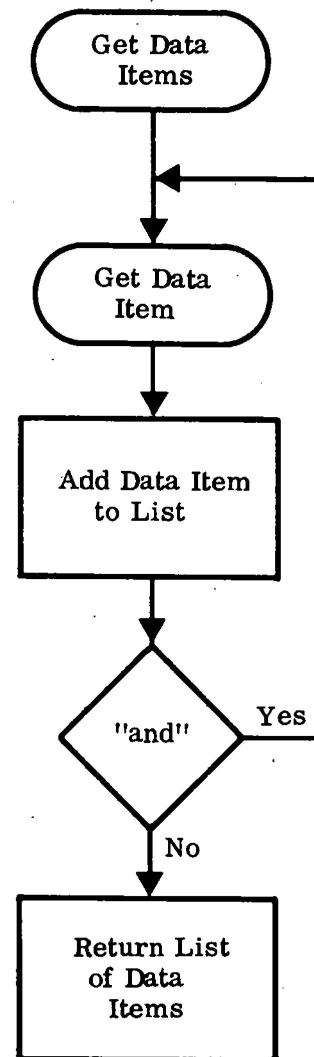
```
     (CSETQ STAK NIL)
   E (COND ((SASSOC (CAR X) OPSP NIL) (PROG NIL
                  (SETQ Y (CAR X))
                  (SETQ VAL
          (COND ((EQ Y PLUSS) (PLUS (CNUM (CADR STAK)) (CNUM (CAR STAK))))
                  ((EQ Y DASH) (DIFFERENCE(CNUM(CADR STAK))(CNUM(CAR STAK))))
                  ((EQ Y STAR) (TIMES (CNUM(CADR STAK)) (CNUM (CAR STAK))))
                  ((EQ Y SLASH) (DIVIDE (CNUM(CADR STAK)) (CNUM (CAR STAK))))
                  ((EQ Y (QUOTE **)) (COND ((MINUSP (CNUM (CADR STAK)))
                  (MESC (QUOTE $$$BASE IN EXPONENTIATION CANNOT BE NEGATIVE$)
                        BLANK)) (T (EXPT (CADR STAK) (CNUM (CAR STAK))))))
                  ((EQ Y SUB) (COND ((SETQ VAL (SASS
                       (CNUM (CAR STAK)) (CARY (CADR STAK)))) (CDR VAL))
                                   (T (MESC (QUOTE
                      $$$ARRAY VARIABLE HAS NO VALUE$) BLANK))))
                  (T (MINUS (CNUM (CAR STAK))))))
                  (COND ((EQ Y (QUOTE NEG)) (CSETQ STAK (CDR STAK)))
                        (T (CSETQ STAK (CDDR STAK))))
                  (CSETQ STAK (CONS VAL STAK))   ))
              ((NUMBERP (CAR X)) (CSETQ STAK (CONS (CAR X) STAK)))
              ((SETQ Y (GET (CAR X) PVAL)) (CSETQ STAK (CONS Y STAK)))
            (T(MESC(QUOTE $$$ARITHMETIC OR ARRAY VARIABLE HAS NO VALUE... $)
                        (CAR X) )))
     (COND ((SETQ X (CDR X)) (GO E)))
     (RETURN (CAR STAK>
DEFINE((
  (GOTO (LAMBDA NIL (PROG (Y X)
     (COND ((EQ NXTW (QUOTE TO)) (FTCH))
           (T (MESS IMPROLIM NXTW NIL)))
     (COND ((NOT (AND (NUMBERP (SETQ X (FTCH))) (FIXP X)))
                      (MESS LNMI BLANK NIL))
           (SYNTAX (RETURN NIL))
           ((NOT (AND (SETQ Y (LINESRCH X O)) (EQ (CAAR Y) X)))
                      (MESC (QUOTE $$$NO SUCH LINE NUMBER... $) X)))
     (CSETQ THISLINE Y)  (CSETQ LINE (CDAR THISLINE>
DEFINE ((
  (SASS (LAMBDA (X Y) (PROG NIL
  S (COND ((NULL Y) (RETURN NIL))
          ((EQUAL (CAAR Y) X) (RETURN (CAR Y))))
     (SETQ Y (CDR Y))  (GO S>
DEFINE ((
  (GETVAL (LAMBDA (Y) (PROG NIL
     (CSETQ OUTP (CONS NIL NIL))      (PPOP Y)
     (CSETQ OUTP (CDR OUTP))
     (COND (SYNTAX (RETURN OUTP)))
     (RETURN (EVLL OUTP>
DEFINE ((
  (PRYN (LAMBDA NIL (PROG (X)
     (COND ((EQ NXTW EMRK) (MESSOP)))
   P (CSETQ AXP T)  (SETQ X (FTCH))     (CSETQ AXP NIL)
     (COND ((NOT (ATOM NXTW)) (SETQ X (CONS X (CONS (FTCH) NIL)))))
     (COND (SYNTAX NIL) (T (PRINT (GETVAL X))))
     (COND ((EQ NXTW EMRK) (RETURN NIL)) (T (GO P>
MAIN NIL
```

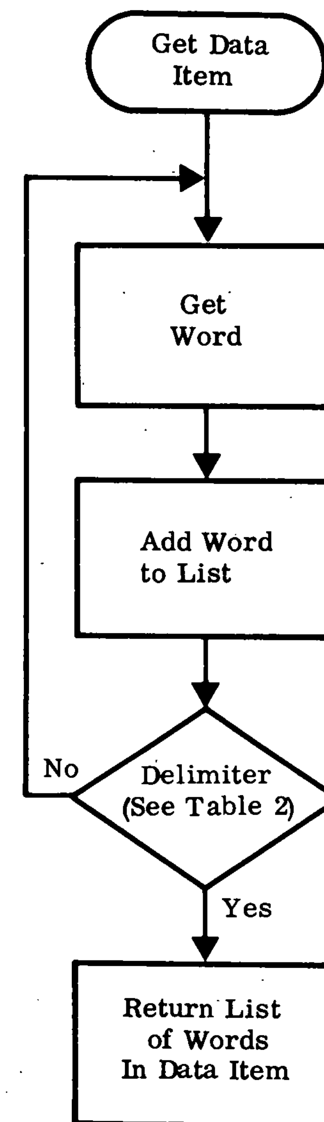
APPENDIX II

LOGICAL FLOWCHART OF THE TAXL INTERPRETER

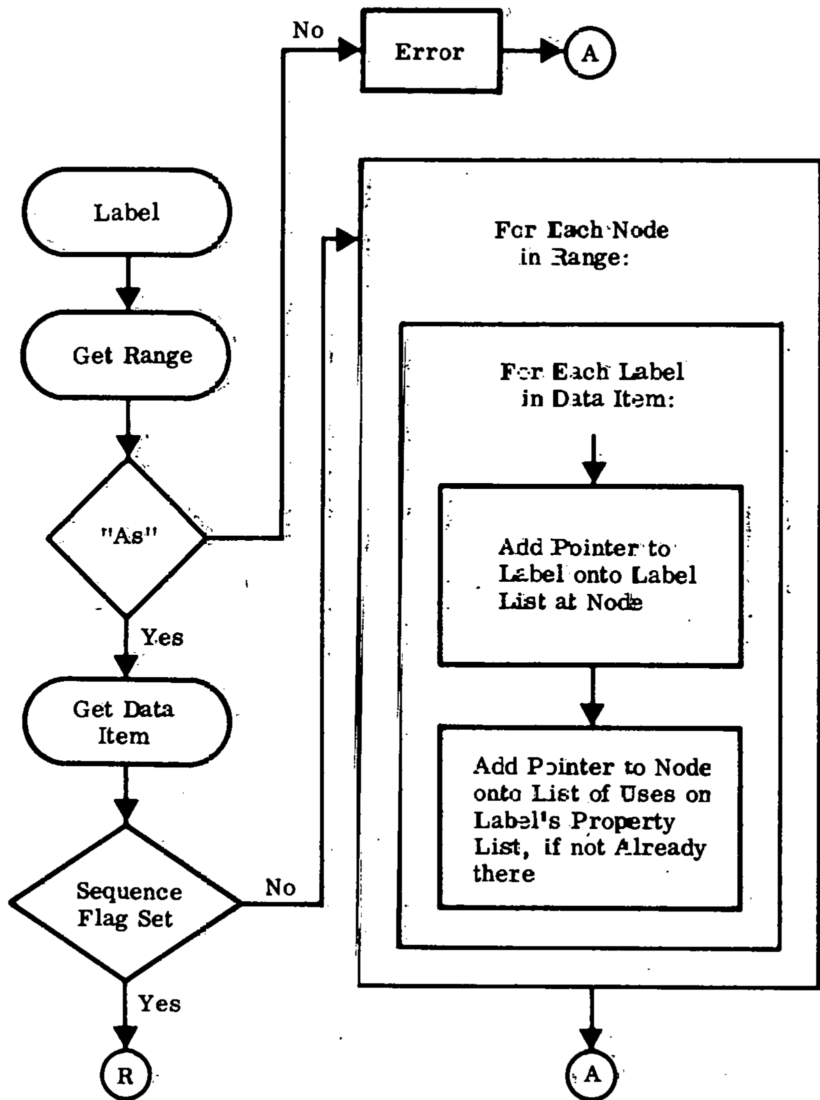


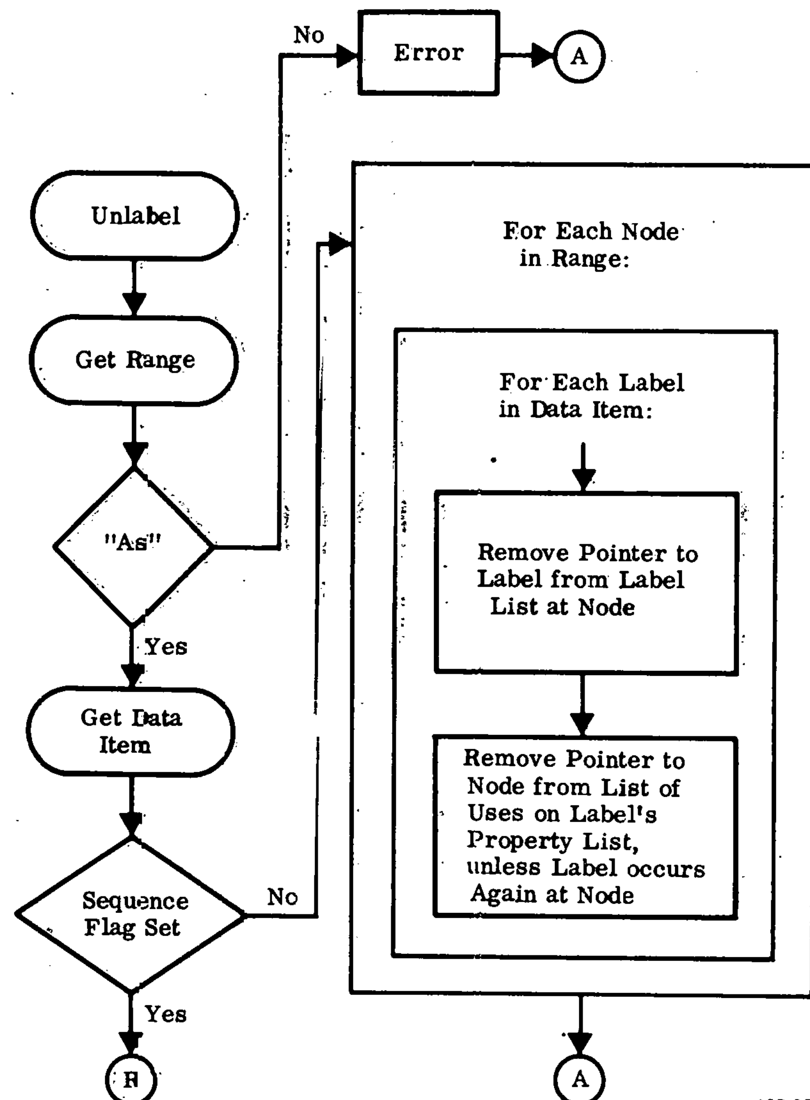


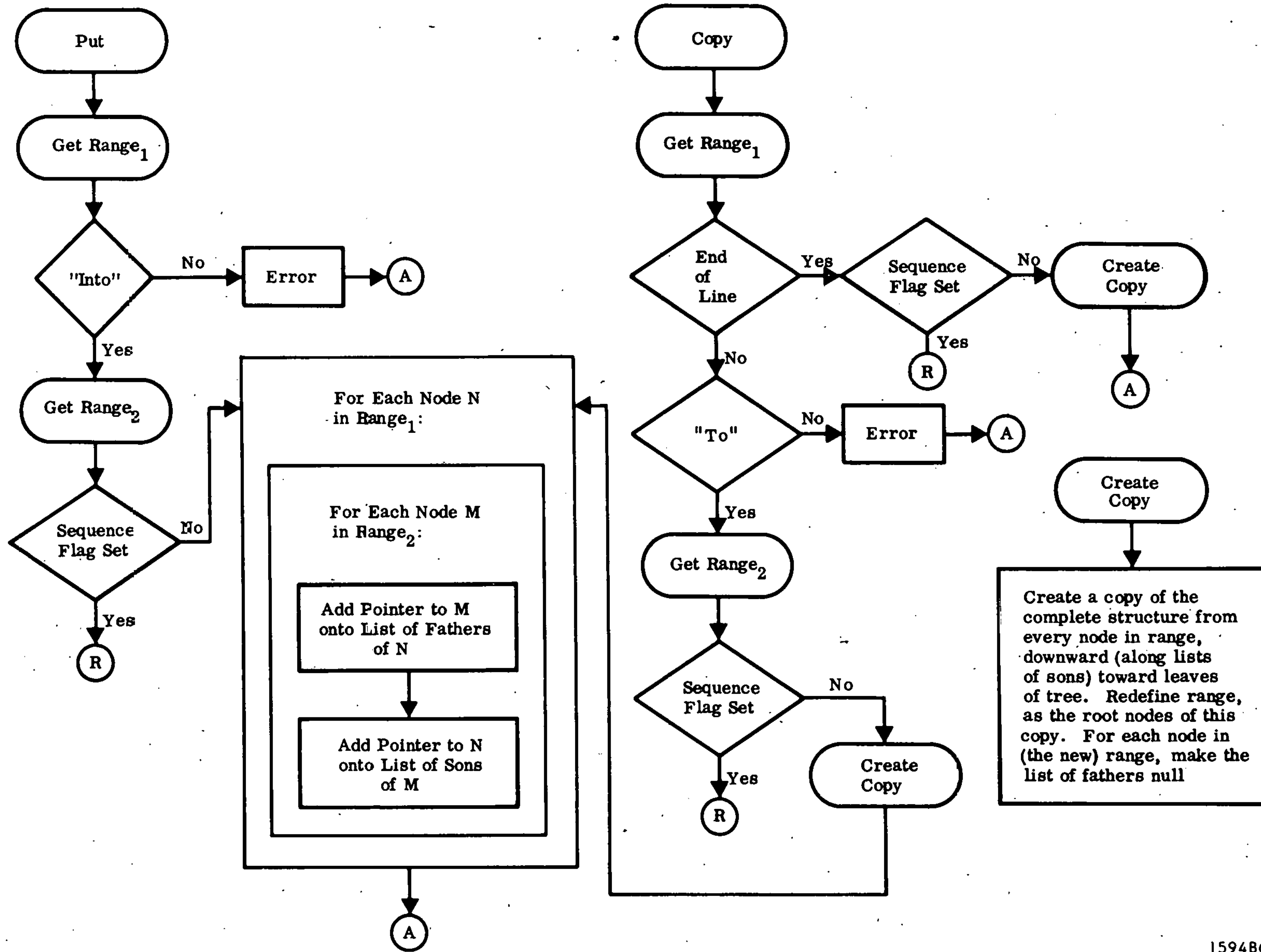


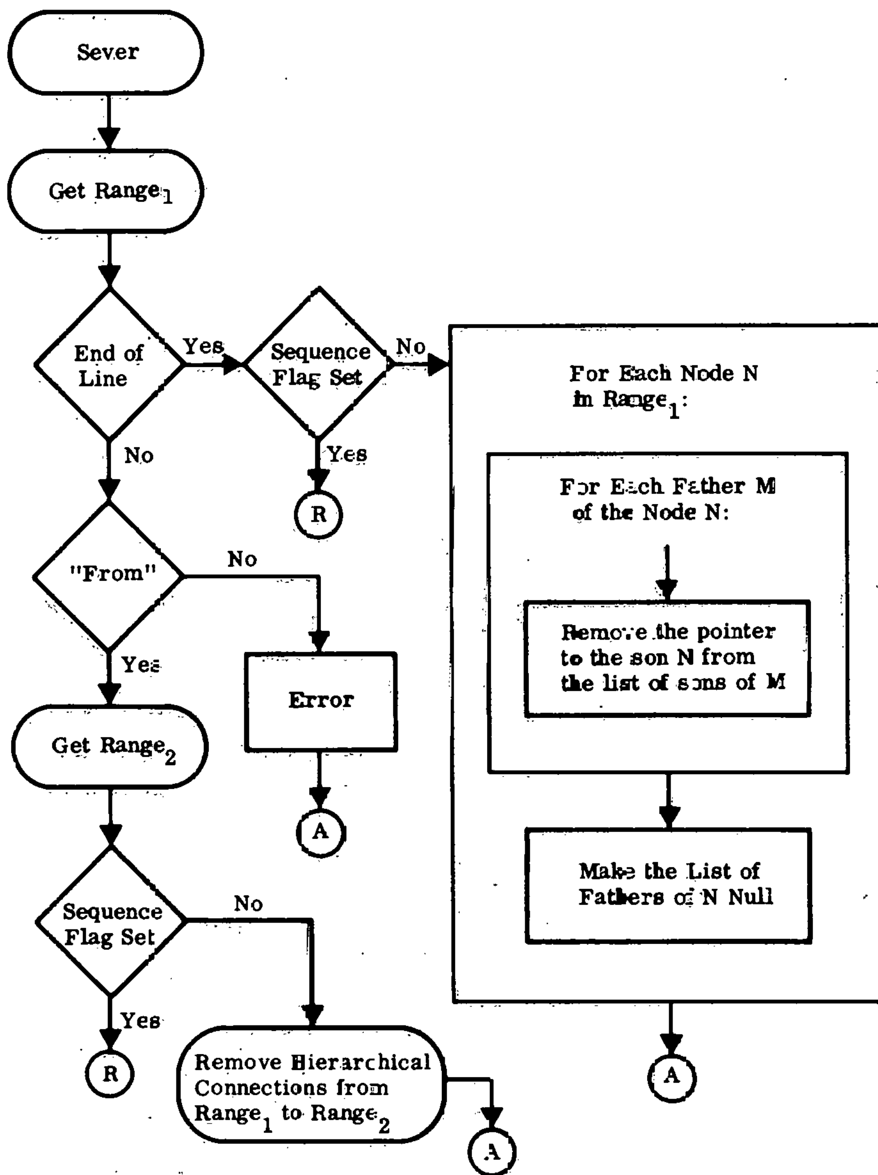


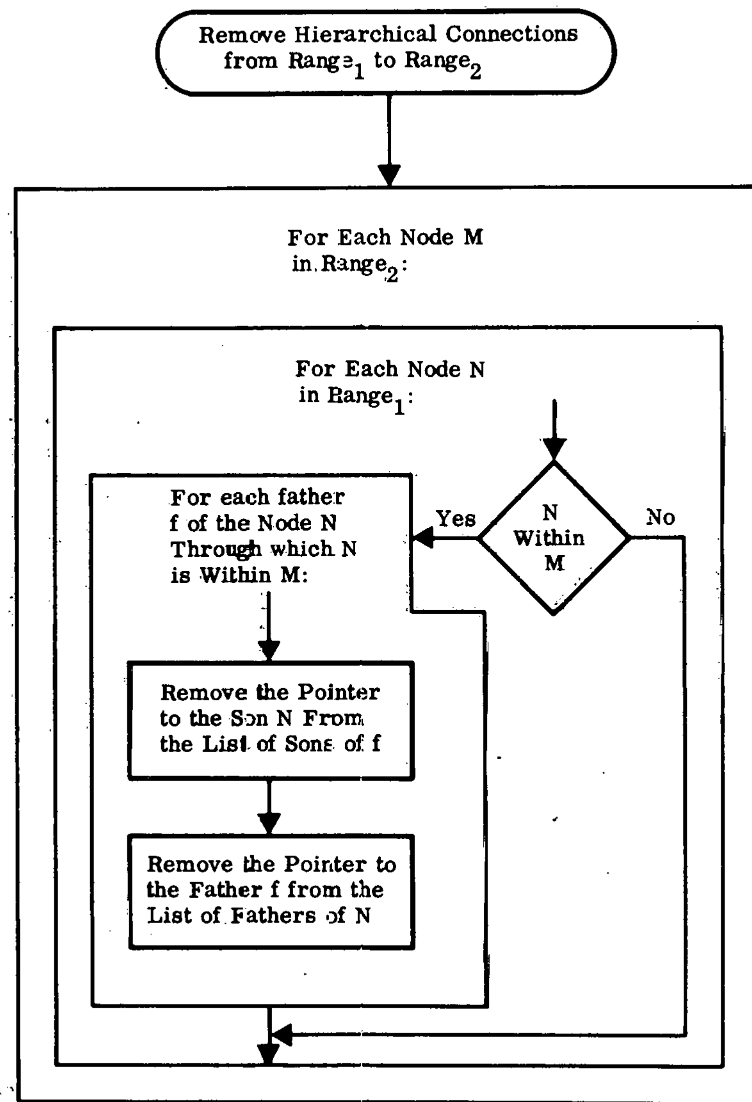
1594B4


No
Error
A
Label
Get Range
"As"
Yes
Get Data Item
Sequence Flag Set
No
Yes
R
For Each Node in Range:
For Each Label in Data Item:
Add Pointer to Label onto Label List at Node
Add Pointer to Node onto List of Uses on Label's Property List, if not Already there
A

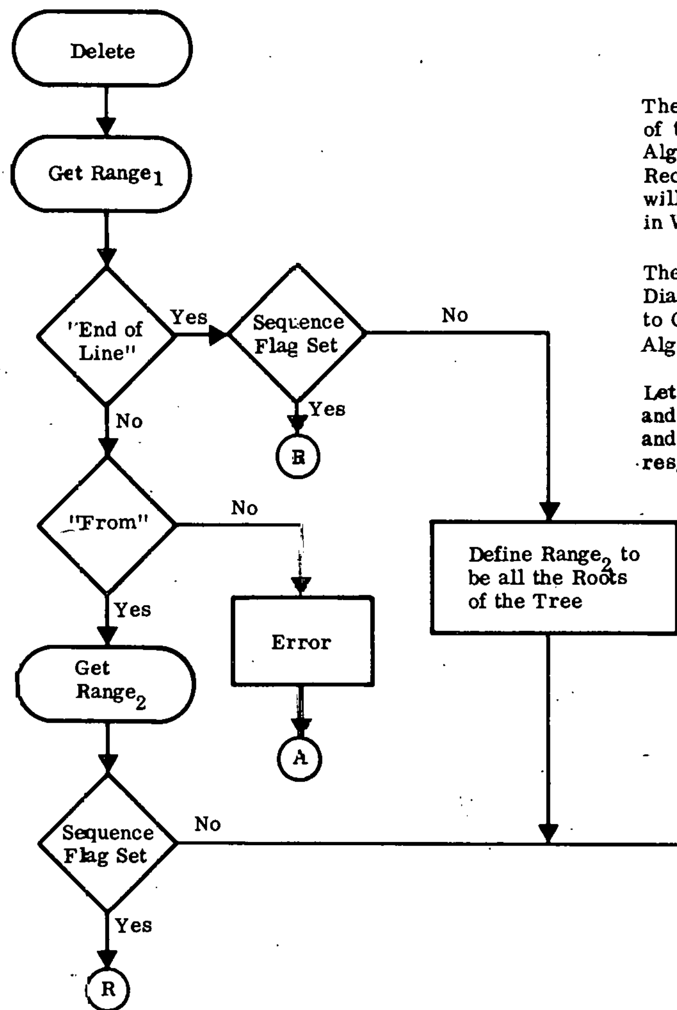

No
Error
A
Unlabel
Get Range
"As"
Yes
Get Data Item
Sequence Flag Set
No
Yes
R
For Each Node in Range:
For Each Label in Data Item:
Remove Pointer to Label from Label List at Node
Remove Pointer to Node from List of Uses on Label's Property List, unless Label occurs Again at Node
A
159485


Put
Get Range1
"Into"
No
Error
A
Yes
Get Range2
Sequence Flag Set
No
Yes
R
For Each Node N in Range1:
For Each Node M in Range2:
Add Pointer to M onto List of Fathers of N
Add Pointer to N onto List of Sons of M
A
Copy
Get Range1
End of Line
Yes
Sequence Flag Set
No
Create Copy
Yes
R
A
No
"To"
No
Error
A
Yes
Get Range2
Sequence Flag Set
No
Create Copy
Yes
R
Create Copy
Create a copy of the complete structure from every node in range, downward (along lists of sons) toward leaves of tree. Redefine range, as the root nodes of this copy. For each node in (the new) range, make the list of fathers null
159486





159487



The Remainder of this Algorithm is Recursive and will be Stated in Words.

The accompanying Diagram will Help to Clarify the Algorithm

Let nodes n and f be $Range_1$ and $Range_2$ respectively

1. Remove Hierarchical Connections from $Range_1$ to $Range_2$

2. If n now has at least one father remaining, then quit (A nonclosed subtree has been encountered)

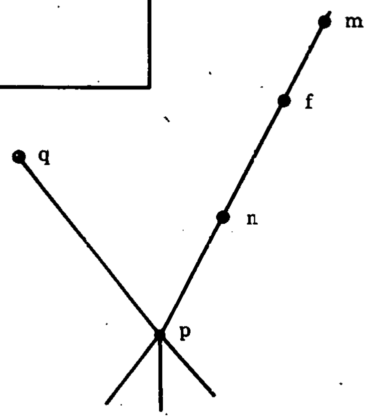
   Otherwise,

3. Remove all labels from n (See flowchart for unlabel)

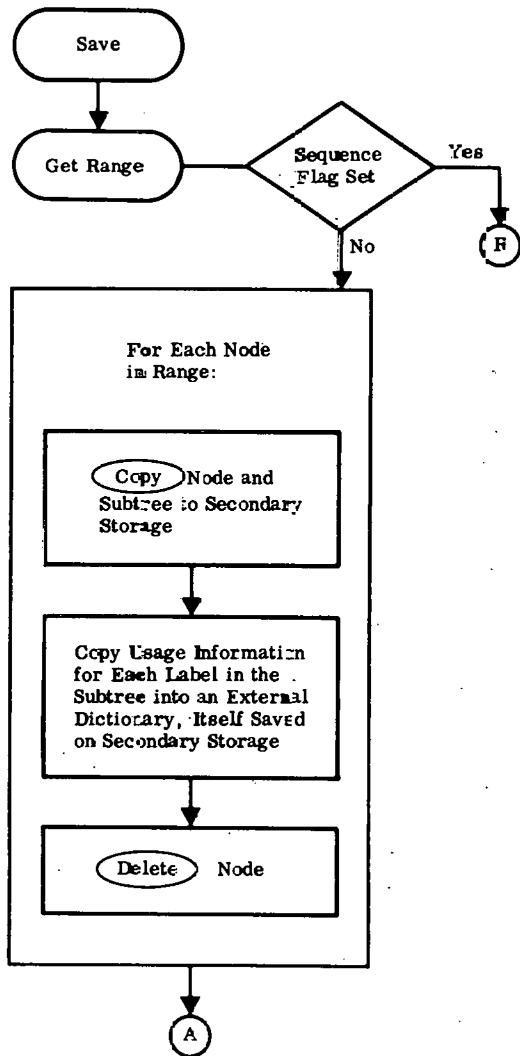
4. Check each of the sons of n (i.e., all nodes p)

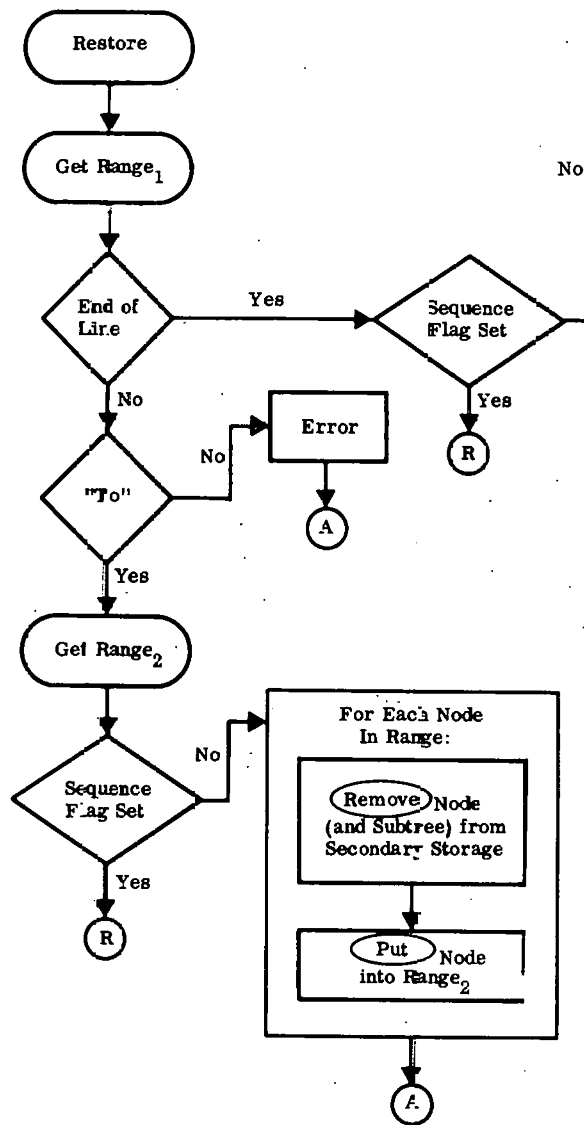
   If p has only one father (i.e., n), then, recursively, go to Step 3 with argument p. (The algorithm remains in this recursive loop as long as a nonclosed subtree is not encountered.

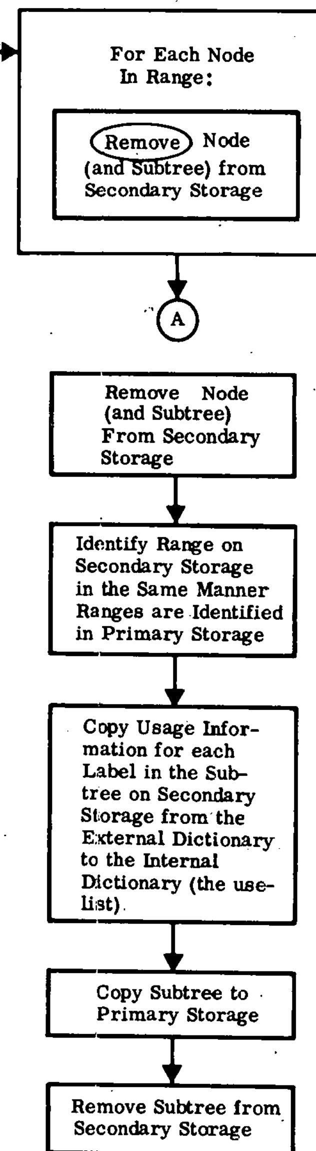
   If p has more than one father (i.e., there is a node q in addition to n), then recursively, go to Step 1 removing connections from p to n.



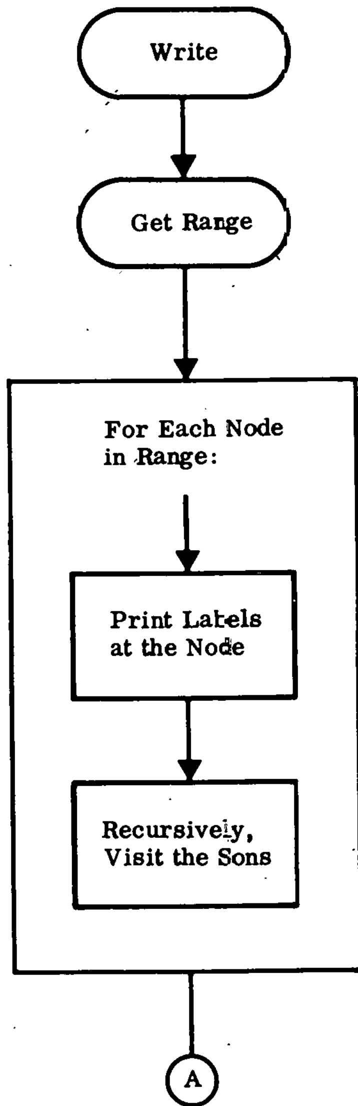
159488

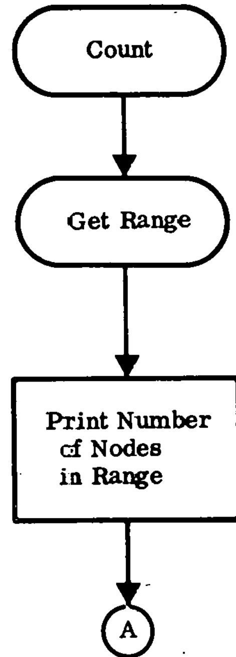


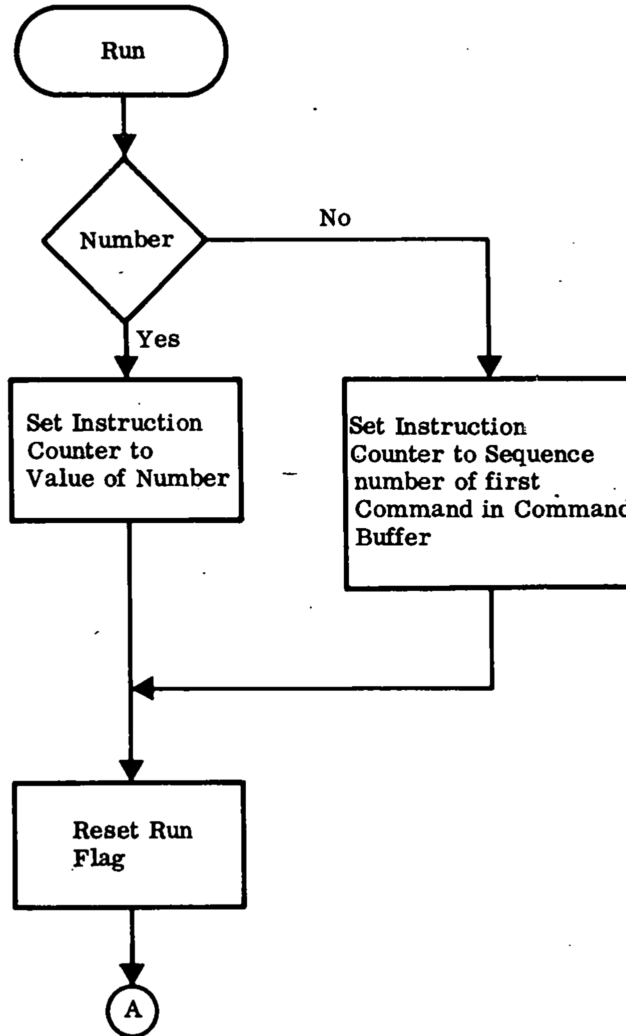


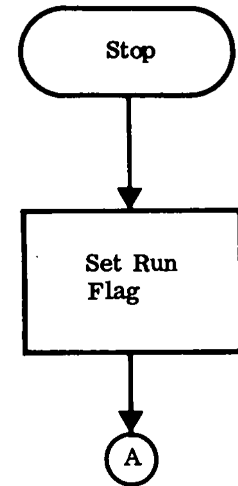


159489









159410