THE GRAPH-PROCESSING LANGUAGE GROPE

Technical Report NL-22

Jonathan Slocum

August, 1974

NATURAL LANGUAGE RESEARCH FOR CAI

Sponsored by

THE NATIONAL SCIENCE FOUNDATION

Grant GJ 509X

This report constituted the author's M.A. Thesis in Computer Sciences at The University of Texas at Austin.

Department of Computer Sciences

The University of Texas at Austin

THE GRAPH-PROCESSING LANGUAGE GROPE

by

JONATHAN SLOCUM, B.A.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the degree of

MASTER OF ARTS

THE UNIVERSITY OF TEXAS AT AUSTIN

Table of Contents

PREFACE

Another programming language?

A little over four years ago the author and another researcher, Daniel P. Friedman, initiated the design of a FORTRAN-based package of routines for efficient processing of general-purpose directed graph data structures. The developers of GROPE 1, as the "language" came to be called, felt some justification in at least delaying the full design of a programming language complete with its own syntax. The first consideration was purely historical accident: GROPE was initially conceived and developed as a graph-processing extension to SLIP, which is itself a list-processing extension to FORTRAN; thus the maintenance of FORTRAN as a host language, even after the inevitable divorce from SLIP, was natural. Second, there was the problem of portability: as the data structures and associated operations evolved into what was obviously becoming a very powerful tool, it was felt that any installation should be able to implement and use this tool with minimum aggravation. FORTRAN is, and seems likely to remain for some time, the most widely spread and frequently used scientific programming language in the world today. GROPE was therefore carefully designed so that any machine-dependent properties existed solely in the few "primitives" which could be hand-coded in (FORTRAN-callable) assembly language by the local implementer. Third, though in some sense related to the second reason, is the consideration that there are many utility packages in existence (statistical, graphic display, plotting, I/O, communication, etc.) which are written in FORTRAN, or are at least

FORTRAN callable, and we had no desire whatsoever to isolate the prospective user from these carefully-developed programs. A fourth consideration, which has obviously been overruled in light of this thesis, was that we had no desire to add yet another member to the already populous field of programming languages: it was felt that the FORTRAN version served well enough, since the manipulation of graph structures, rather than the invention of a programming language, was the goal of the project. The fifth and final consideration was that we wished to devote our efforts to the proper development of a graph-processing tool, rather than divide them in some confusing manner between that and some proper syntax for a new language.

After well over three years of intensive development which resulted in two implementations [1] and a doctoral thesis [8], we reached the point where we could declare the project "complete". But now we had to face the growing and persuasive criticism that FORTRAN as a host language was a limiting factor in using the tool. There are features lacking in the control structure of FORTRAN which constrained programmers from "natural" implementation of their ideas; GROPE was imbedded in an artificial intelligence research environment where LISP was the accepted tool. Although all the programmers knew FORTRAN, they complained of its various weaknesses -- lack of recursion, for example. On the other hand, they were not unimpressed with the phenomenal processing speeds obtained from GROPE programs, nor with the re-discovered wealth

of operating system support software, etc., that had been unavailable in LISP.

In effect, the development of the language advanced in this thesis has been goaded by popular demand. The effort has been to provide the programmer the best of both worlds: the system support features and processing speed available in FORTRAN, and also the flexibility and natural usefulness which should be in every programming language. The author studied other programming languages, looking for syntactic features, control structures, etc., that seem to make a language "natural and useful". Some of the ideas in GROPE 2 (the complete-with-syntax version of GROPE 1) are therefore admittedly borrowed from other languages, sometimes with special adaptations, yet there is still a sizeable measure of originality in the design. The author also instigated many conversations with several individuals concerned with programming language theory and design -- especially the major adviser. The opinions and reflections of these people constituted a strong influence on the final design, and their ideas have been deeply appreciated during the course of the last two years. There is likely not a single facet of this design which has not been "tested" on at least one of these individuals, if not actually proposed by one of them, and needless to say this language would not be what it is without their help.

It would not be possible to present the Data language and Programming language (GROPE 2, as it is called) in a meaningful way without describing the data structures and operations (GROPE 1) which are subsumed; however, the research and development of the latter does not consitute material actually being presented for the thesis. With Dan Friedman's encouragement, I have included, in the introduction and first three chapters, some material drawn from his doctoral thesis -- as good an introduction to the GROPE theory of graph-processing as any. Its presence is for the purpose of conveying the meaning of graph processing in GROPE, so that the structure and operations of the thesis language(s) might become meaningful. I am grateful for his permission to include this material.

Finally, the language has been carefully designed to be compatible with FORTRAN, although to my mind it lacks almost any characteristic which would lead the user to realize this, if left to his own devices. It is certainly hoped that the user will find GROPE 2 as easy to learn, and as natural to use, as the author intended to render it.

INTRODUCTION

The purpose of this document is to introduce the reader to GROPE 2, a modular programming language with hierarchical control structures and recursion. The GROPE 2 translator (written in GROPE) generates GROPE-FORTRAN code and thus serves as a very sophisticated FORTRAN preprocessor. With this tool the user may perform extensive list- and graph-processing in a FORTRAN-compatible environment. The assumption is made that the reader knows FORTRAN and has some knowledge of list- and graph-processing techniques -- in particular LISP. Following an overview of graph processing and programming languages, this thesis will address in chapters: (1) the abstract data structures provided by GROPE, with conventions for drawing them; (2) list and graph processing by searching linear structures (lists, and structures much like lists); (3) some important generalizations, including capabilities and operations which make GROPE truly unique and exciting; (4) the Data Language, which offers a method for the linear description of the primary data structures -- much like the S-expressions of LISP; (5-6) the Programming Language, with complete syntax, English descriptions, and numerous diagrams, tables and flow-charts to illustrate its capabilities; and (7) some concluding remarks with notations about some programs currently running in GROPE 1. Appendix A provides a concise description of every available GROPE procedure -- the arguments, operational effect, and returned values. Appendix B is a guide to the special GLOBAL variables: these provide the programmer considerable control

over the free-field I/O operations. The user may specify two active input files, two active output files, an echo-print file, an error message file, and a file for trace messages, and effect other controls.

Graph data structures

Graphs are important data representations in many fields: bonding structures in chemistry, Feynman diagrams in physics, sociograms in sociology, circuit diagrams in electrical engineering, and flow networks in operations research are all instances of graph structures. The determinations of maximal network flow, the shortest path between two nodes, a Hamiltonian path, or optimal line balance are all usually formulated in terms of graph processing algorithms. There are graph algorithms for the well-known "Traveling Salesman Problem", for finding the maximal spanning tree, and for information retrieval and natural language processing. Graph algorithms have been applied to the "Four Color Problem", the solution of the "Knight's tour", and the determination of transitive closures.

Graph representation methods

"Incidence arrays" are a well-known means of representing graphs using very primitive data structures: a graph is represented by a square matrix (two-dimensional array) A, having one row and one column for each node. An arc from node i to node j with label (or value) v is denoted by the storage of v in array location A[i,j]. The main problem with this representation is its lack

of flexibility: it simply cannot represent complex data structures -- such as nodes or arcs which are graphs. Associating additional values with nodes and arcs (which, incidentally, must almost always be of the same scalar type) or allowing parallel arcs (between the same two nodes) requires much additional storage or leads to ad hoc solutions. Another significant problem with this representation is the relative inability to do dynamic processing: it is difficult to allow a graph to grow through the addition of nodes, since few programming languages allow an array to grow by adding rows and columns; it is even more difficult to allow the number of graphs to increase dynamically, since arrays are almost always required to be declared at compile-time. Another problem is the deletion of arbitrary nodes (as opposed to arcs) in such a representation.

An example of simulating graph structures through less-primitive structures involves the use of property lists (attribute-value pairs). Typically in LISP [21], nodes are represented by "atoms", and their associated property lists indicate arcs to other nodes; in languages such as SLIP [38] and IPL-V [23], a similar method represents nodes as lists, and their associated "description lists" indicate the arcs. This property-list representation forces graph algorithms to be inefficient in terms of time due to the necessity for searching these property lists for each arc access. In addition, this representation makes arc traversal in both directions difficult -- a property required in many graph algorithms (such as finding a critical

path in a PERT network).

When simulating graphs by using extensible data structures, the user defines (at compile time, again) blocks of core as nodes -- records in COBOL [37], based variables in PL/I [19], or plexes in AED [30], L^6 [18], and PASCAL [41]. Arcs are represented by pointers from one block to another. The specified fields within a node store the information associated with that node and with the arcs leaving that node. The problem here is that the burden of defining accessing primitives and higher-level operations is on the programmer. In addition, the programmer is responsible for storage management and I/O. Using the programmer-defined data type feature of SNOBOL 4 [11], the user is not responsible for storage management or some of the basic accessing primitives, but the high-level operations and I/O problems remain. The plex representation also tends to constrain the dynamic properties of (the number of) arcs in some of these systems.

In short, user-supplied software packages for graph processing tend to be restrictive and error-prone; even so, much effort is expended in implementing such a system before the programmer can seriously consider the algorithm which he is actually trying to program. And when he tries to apply his costly package to the solution of new problems, he typically finds that at least a partial redesign is necessary, due to the lack of generality in the original system. The existence of a powerful, efficient, general-purpose graph-processing language should solve all these problems.

We feel that GROPE is such a language.

The general class of graph processing problems for which GROPE was designed is characterized by two aspects: they deal with graph structures which are interrelated in complex ways and contain symbolic as well as numeric data, and whose solutions require graph structures to grow, shrink, and be modified both dynamically and irregularly. These problems are precisely those for which the simplistic simulations described above are most inadequate. There were three major design criteria for the development of GROPE: flexibility, mutability, and efficiency.

There should be a means for representing a variety of forms of data. There should be labelled graphs, nodes, and arcs; multiple arcs between the same two nodes should be allowed. There should be a provision for the natural representation of hierarchical graphs (nodes with values that are graphs) and other complex graph relationships. There should be supporting structures such as lists and sets for "temporary" deposit of information during graph processing. There should be special mechanisms for searching and processing graphs. True flexibility demands most if not all of these features.

There must be (preferably high-level) operations for dynamic modification of graphs -- operations that create, destroy, and change features (such as labels and values) of graphs, nodes and arcs. Programming convenience is greatly enhanced if the programmer

does not have to "push pointers" in order to accomplish this task.

Finally, these operations and the associated storage management must be handled efficiently; this requirement is dictated by the combinatorial nature of many algorithms for graph processing. For programmer convenience and safety, the storage management must include automatic bookkeeping for the dynamic allocation and recovery of storage -- like the free space list and garbage collector of LISP.

Graph-processing languages

Since directed graphs are often used for informal description and analysis of data structures, and since being able to program directly in terms of the structures which are natural to an applications area is a well-known advantage, it is surprising that directed graphs have not been accepted as a "primitive" data structure in any major programming language. There are, however, some minor languages which have included directed graphs. HINT [12], GRASPE [6], GEA [4], and LINKNET [3] are the dominant examples of this philosophy.

HINT and GRASPE, each associated with an already-developed programming language, were designed for symbolic structure manipulations. HINT is compiled into IPL-V. GRASPE is a library of LISP functions. GEA and LINKNET were designed to perform numerical data analysis within a complex but relatively static data structure, and each is associated with an algebraic language: GEA being

a syntactic extension of ALGOL [22] which is pre-processed into ALGOL, and LINKNET being a library of FORTRAN functions. We shall compare and contrast these four languages with GROPE in light of the design criteria discussed in the preceding section.

In terms of representational flexibility, GEA and LINKNET deal only with numeric scalars as values of nodes and arcs, whereas HINT, GRASPE and GROPE provide for symbolic node and arc values; only these three allow hierarchical structures. Only GROPE allows more than one type of node and one type of arc. HINT, GRASPE, and GROPE have list processing as a support feature.

In terms of dynamic operations, only HINT, GRASPE and GROPE allow creation and destruction of graphs. LINKNET does not even provide for the dynamic creation and destruction of nodes and arcs, but rather the programmer must produce code to effect these capabilities.

In terms of efficiency, GRASPE and HINT are tied to their respective hosts for the representation of graphs -- via property lists. Efficiency here is poor due to the implied requirement for property list searches for every arc access. GEA uses lists to represent graphs; although little can be said about the efficiency of GEA since details of the preprocessor are unavailable, is may be presumed that arc accesses in this case also require list searches. LINKNET and GROPE use plex structures for graph representation, which likely aids the cause of efficiency;

however, programmer efficiency in LINKNET is reduced since no high-level graph-processing primitives are available -- only primitives to change the contents of a field in a plex. The GROPE operations are indeed high-level, and very efficient. GRASPE, GEA, and GROPE have garbage collectors. HINT uses the storage manager of IPL-V, and LINKNET has no storage management.

What may be regarded as statements of deficiencies regarding the above languages must be qualified: such remarks pertain to our design criteria; each language would appear to be a useful model for the class of problems with which it is concerned, although in many cases their efficiency is poor. Our judgements are concerned with the qualities we feel should be inherent in any graph-processor intended for wide use, rather than for special purposes.

GROPE

GROPE 1 is a successfully-implemented graph processing extension to FORTRAN; in the sense that it is a library of functions, it parallels SLIP. GROPE not only provides high-level graph processing primitives, but also provides a number of other data structures, including a complete list-processing package which enhances and supports graph processing. There are a number of major new ideas embodied in the GROPE data structures and operations; GROPE provides a set of building blocks (atoms, arcs, nodes, graphs, lists and sets) and operations for building arbitrarily

complex graphs. In addition, there are a number of primitives that perform unusual operations (such as moving a node from one graph to another). Although the programmer can create quite complex structures, experience has shown that their manipulation remains straightforward. Support operations are equally important to the efficient development of graph algorithms: GROPE allows list, set, and array processing; there is an extensive I/O facility, and a garbage collector. For the user with a large application in mind, there is a built-in (but optional) software "associative" virtual memory (hopefully more efficient than paged virtual memory for GROPE applications), embodying a scheme for the permanent maintenance of a large data base. Throughout the design and implementation of GROPE there has been a fanatical concern with efficiency and an almost equally-serious endeavor to maintain generality.

The GROPE 2 programming language design has emerged over a period of time during which the author studied perhaps twenty "major" programming languages, looking for "natural" features which tend to render a language easy to learn, easy to use, and largely self-documenting. Ease of implementation was <u>not</u> considered where programmer convenience was involved. However, many features which could have been included (say, pointer variables) were not, for the simple reason that the language was being designed to simplify GROPE programming -- rather than to be the "ultimate" programming language.

GROPE 2 has been most strongly influenced by BLISS [43] Some remnants of FORTRAN may be observed, notably the availability of formatted I/O and labelled COMMON blocks, but this is intended to allow communiction with FORTRAN routines. There are five variable types: ALPHAnumeric, GROPE, INTEGER, LOGICAL(Boolean) and REAL. Identifiers may have any one of five scopes: EXTERNAL (procedures), GLOBAL, LOCAL, OWN, and (labelled) COMMON. There is no block structure per se, but the control structures (statements) are highly structured and of quite sufficient power to allow and encourage GOTO-less programming. (There is, however, a somewhat weak GOTO.) As in LISP, any readable GROPE data structure may be "quoted" in the programming language; in addition, there are other syntactic constructs which resemble the above (without the quotes) that key the automatic creation of GROPE structures, with and without a prior search. (There is an automatic "pattern-matching" search facility.)

GROPE 2 does not follow the lead of LISP and BLISS in allowing statements to have values -- making them, in effect, expressions. This was considered, but abandoned because it might encourage programmers to construct indeciperable code sequences. (In this respect LISP has a unique notational advantage; on the other hand, no one has accused LISP of being self-documenting -- readable to the novice.) The assignment operation, however, is an expression rather than a statement: this satisfies much of the demand for statements with values while retaining readability. Also, any

expression may appear in a context normally (in other languages) requiring a statement.

With this brief overview of some of the salient features of GROPE, we proceed with the development of GROPE 1 in chapters 1-3, followed by the Data language in chapter 4, then the Programming language in chapters 5-6. The Programming language in particular is developed top-down; it is hoped that by having a "global" view of the language while learning it, the reader will find the task easier. It is expected that the reader has some notion of what an identifier is, what a procedure call is, etc., so that this organization will not be too disruptive by mentioning constructs yet undefined.

CHAPTER I

The Data Structures

In this chapter the GROPE data structures are introduced. These structures are viewed as abstract entities; in order to aid the user in visualizing the structures and operations upon them, a graphic orthodoxy is presented which encompasses the primary structures and the important relationships among them. The operations introduced are those necessary for the creation and accessing of the structures. Figures and charts are provided in explication of the major concepts involved.

Atoms

Atoms are the "data constants" of GROPE; for the most part, they are like LISP atoms -- they are words or numbers. (For simplicity, arrays and procedures as atoms will be temporarily ignored.) "Words" are alphameric character strings: GROPE-2, HALLEY, ZEBRA, and This-Is-An-Atom are examples of alphameric atoms. Any integer or real number which is legal in FORTRAN may also be an atom: 258, 2.71828, and 6.660E18 are examples of numeric atoms. Unlike LISP's numeric atoms, those of GROPE enjoy every privilege allotted to alphameric atoms; also, the alphameric atoms of GROPE may be arbitrarily long -- there is no limit on the number of characters such an atom may contain other than that imposed by machine memory.

Lists

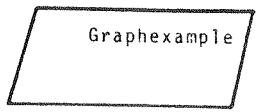
The list is a basic GROPE data structure; as in LISP, a list is represented as a left parenthesis, (, followed by the elements of the list in their appropriate (user-controlled) order, followed by a matching right parenthesis,) . A newly-created list is empty, so it is represented by nothing more than a matched pair of parentheses: () . The user may add elements to a list by stacking them (putting them "in front", or to the left, of every other item in the list), or by queuing them (putting them "behind" all other items in the list). Unlike LISP, these operations are equally (in-) expensive. Using another mechanism to be discussed later, the user may also insert an item anywhere in a list, or substitute one item for any item in a list, or delete any item from a list. Some examples of lists are: (A 2 s), (This is a LIST) , and (W (X Y) Z) . Note that one element of the last example is itself a list -- commonly called a sublist. The last list contains three items (the sublist counts as one, even though it contains two items); the first list also contains three items. Thus we speak of the length of a list as being the number of elements it contains -- there is a GROPE function LENGTH which if called with a list as an argument will return the number of items in the list. In the examples above, A, 2, s, This, is, a, LIST, W, X, Y, and Z are atoms. An empty list has length zero.

Sets

A third basic data structure, itself much like a list, is the set; rather than being delimited by parentheses pairs, however, sets are delimited by matched pairs of braces: { and }. As in lists, the elements of sets are said to be ordered. Sets usually contain atoms, though there is no reason why they cannot contain lists or other sets (not necessarily subsets in the mathematical sense), or indeed any other GROPE structures. Thus lists and sets both may contain any structure as an element. While the user may order the elements of lists in any way he chooses, the order of the elements in sets is entirely controlled by GROPE -- allowing efficient operation of the member, union, intersection, and difference procedures. And while the user may include any given item any number of times in the same list, each item in a set will be represented only once. It is still true, however, that any given item may appear in any number of different sets -- and of course in any number of lists. There may be an arbitrary number of sets and lists.

Graphs

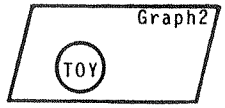
Graphs are of course a major GROPE structure; we visualize them as being planes, and draw them as parallelograms:



Every graph must have a label (which is written in the upper right-hand corner); atoms serve as useful labels. The graph above may not seem very exciting because there is nothing on it -- it is empty.

Nodes

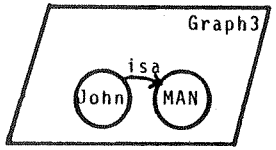
Nodes are the structures that go on graphs and begin to make them worthwhile. We draw nodes as circles within the bounds of some graph; nodes, too, have labels (inscribed):



Every node must be on some graph; there may be any number of nodes on any one graph, and any number of graphs. Every node must have a label -- atoms serve as good labels here, too. Every graph "knows" what nodes are on it, and every node similarly "knows" which graph it is on.

Arcs

Any two nodes may be connected with a labelled, directed structure known as an arc; we draw an arc as an arrow pointing from one node to another, with its (necessary) label written in close proximity:



The arc pictured above, labelled with the atom isa, is directed from the node labelled John to the node labelled MAN. Arcs generally represent relationships between nodes, and their labels serve to name the particular relationship involved. There may be any number of arcs leaving any particular node, and any number arriving at any node; each arc "knows" the node from which it emanates, and the one to which it points. Each node "knows" every arc that leaves it, and every one that terminates at it.

Operations

An atom is created by one of the GROPE operations; the user specifies a number, or character string, etc., which will be the print-image of the atom (that which appears when the atom is "printed"); this bit-string may be retrieved (for arithmetic interpretation, for example) via the function IMAGE. When the user creates a list or set, it is empty -- it has no elements. When a graph is created, the user must specify a label (any atom he desires) for that graph; the GROPE function which creates a node requires a label (for the node) and a graph (on which the node is to reside). The function which creates an arc requires three arguments: the node from which the arc is to emanate, a label for the arc, and the node at which the arc terminates. Given a graph, node, or arc, the function LABEL will retrieve its label; given any node, the function GRAPH will retrieve the graph on which that node resides; given any arc, the function FRNODE will retrieve the node from

which that arc originates, and the function TONODE will retrieve the node to which that arc points. There is another attribute available to the user: each of the items mentioned above (atom, list, set, graph, node, arc) may have a value associated with it. The function VALUE retrieves the value of its argument structure. We draw the value relationship by means of a dotted arrow (not an arc) from the structure with a value to the structure which is the value. Table 1.1 illustrates the successive operations, and their arguments, that construct the graph data structure in Figure 1.1; Table 1.2 illustrates the arguments and values returned by various retrieval functions. The reader is urged to study these examples until he is quite certain "how" these operations have the claimed effects.

Operation	Arguments			Result
create graph	EUROPE			g
create node	LONDON	g		x
create node	PARIS	g		y
create node	ROME	g		z
create arc	z	SOUTH	x	a
create arc	x	NORTH	y	b
create arc	y	NORTH	z	c
create arc	z	SOUTH	y	d
change value	b	CROSSED		
change value	y	VISITED		

Table 1.1 Creation of a Graph Data Structure

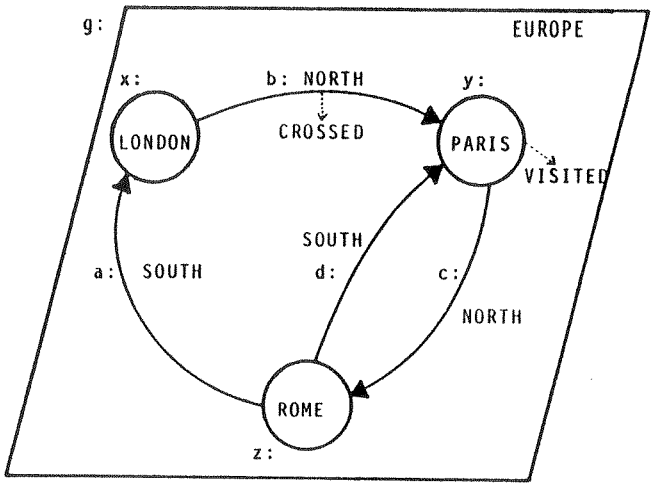


Figure 1.1 A Graph Data Structure

	g	x	y	z	a	b	c	d
graph		g	g	g				
frnode					z	x	y	z
tonode					x	y	z	y
label	EUROPE	LONDON	PARIS	ROME	SOUTH	NORTH	NORTH	SOUTH
value			VISI-TED			CROS-SED		

Table 1.2 Retrieval from a Graph Data Structure

System Sets

There is another group of GROPE structures which the programmer will come to find useful -- the system sets. A system set is an ordered collection of elementary structures, each of which satisfies some predetermined property. It is like a user set in that no item is represented more than once in the set (there is no redundancy); however, except for the atomset (which is hashed), the order of the items may be controlled by the user. There are seven types of these sets: each may contain only one specific type of structure. Many graph algorithms will need to search these sets as part of their operational requirement.

The atomset

There is one universal atomset in GROPE; only atoms appear in the atomset. The order of this particular system set is controlled by GROPE, since items are entered by means of a hash algorithm; but the user may search the set, and add and delete atoms at will. Every atom in this set is aid to be "related", hence this is called a "relate set", or relset.

The graphset

There is one universal graphset; only graphs appear in the graphset. The programmer may control order within the set, search it, and add and delete graphs at will. Every graph in the set is also said to be related, therefore this too is a relset.

The nodeset

Each graph has its own nodeset -- the set of nodes on that graph. (This is how a graph "knows" its nodes, as mentioned earlier.) The order is user-controlled, and the set may be searched and modified by the user. A node must "know" it is on a graph before it will allow itself to appear in that graph's nodeset. (Recall that the GRAPH of a node is available.) A node in any nodeset is "related", therefore every nodeset is also a relset.

The rseto

Each node has its own rseto -- the set of arcs pointing out from that node. (This is how a node "knows" its outgoing arcs.) Every arc in the set must have the same FRNODE, and the frnode must be the node for which this set is the rseto. Every arc in the set is "related"; this set type is the fourth and final relset. The user may control the order of the arcs in the set, and may search and modify the set.

The rseti

Every node also has its rseti -- the set of incoming arcs. The TONODE of every arc in the rseti of some node is that node. The arcs in the rseti are said to be "attached"; therefore this type of set is an "attach set", or attset. Its ordering is also user-controlled, and he may search it to modify it as desired.

The nset

Every GROPE structure which is the label of any node has an nset -- the set of nodes with that label. No node with a different label may be in the nset. The user controls the order and as usual has modification priviledges. The nodes in this set are also called "attached", so this is another attset.

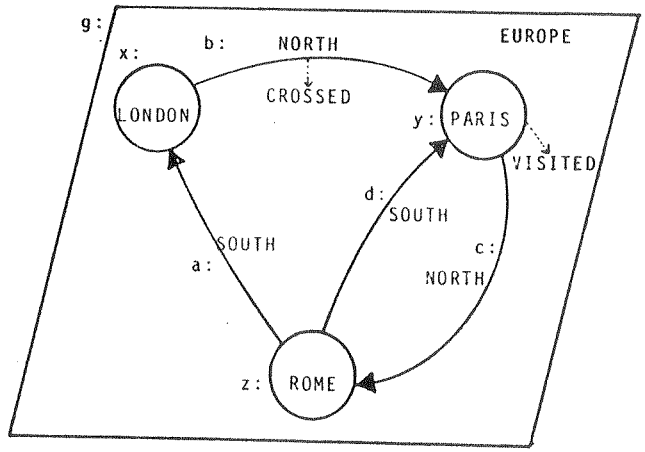
The gset

The last of the system sets is the gset; every graph label has its own gset -- the set of graphs which share that label. The graphs in the gset are "attached"; this rounds out the class of sets called attset. The usual ordering, searching, and modifying facilities are available.

Operations

Lists, sets, and the system sets comprise the class of structures known as linear structures. Strict ordering of elements is inherent in linear structures: in the case of lists, order is established only by the user; in the case of sets, order is established by GROPE only; the system sets are slightly different. When the user causes the creation of a new atom, graph, node or arc, GROPE will automatically add it to the appropriate relset, and except for atoms (for which there is no attset) will add it to the appropriate attset. Thus the user is relieved of the burden of constantly finding these sets. However, when the new item is

added, it will be either stacked or queued into the set(s) -- according to the position of a stack-queue mode switch QSMODE (which may be changed as often as desired by the user). In this way, the programmer "controls" the order of items in the system sets -- except, of course, for the atomset. Later, mechanisms will be discussed by which the programmer may further control the ordering within these six system sets. Figure 1.2 shows some example graph structures, and Table 1.3 illustrates the system sets which are implied by the drawings. Given a graph, the GROPE function NODESET will return its nodeset; similarly, RSETI and RSETO are defined over nodes, and NSET and GSET are defined over atoms. ATOMSET and GRAPHSET are "global" variables which are bound to the appropriate two sets.



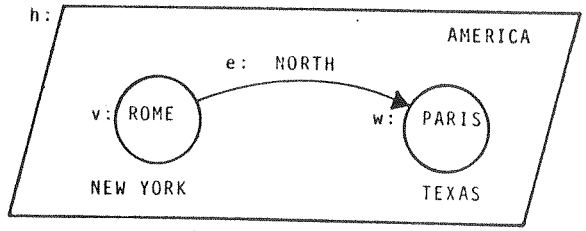


Figure 1.2 Structures which emphasize the nset and graphset

	LONDON	PARIS	ROME	g	h	v	w	x	y	z
graphset	(g,h)									
nset		(x)	(y,w)	(z,v)						
nodeset				(x,y,z)	(v,w)					
rseto						(e)	()	(b)	(c)	(a,d)
rseti						()	(e)	(a)	(b,d)	(c)

Table 1.3 System-set Retrievals for the Structure in Figure 1.2

The objects

There are just two more GROPE data structures -- one of which is not actually structured, since it consumes no core space of its own. This one is the "pseudo integer". There are many occasions for using integers as values of other structures; sometimes it is desirable to employ lots of different integers, and if this is the case then it would be nice if they required as little space of their own as possible -- especially when there is no need for them to have any structural capabilities (such as values). GROPE provides a large set of integers which "look" like GROPE atoms with integer IMAGEs. These pseudo integers are not atoms; they may not serve as labels. The only piece of information they embody is their IMAGE, and they require no data space of their own. But they make very nice values. Their allowable range is restricted and implementation-dependent, but in general it will be on the order of ±64,000. The function PSEUDO will take a small (absolute value) integer as its argument and return a pseudo integer whose IMAGE is that integer; this pseudo integer may be used as the programmer desires -- within the prescribed limits.

The last data structure to be defined is the reader. Actually, readers are among the more important structures in GROPE, but they are also among the more complicated. Since Chapter II is entirely concerned with searching linear structures with readers, little will appear here. Essentially, readers are the only search mechanism

in GROPE. They may be considered somewhat like pointers (they are not), since we will draw them that way: they are pictured as short, stubby arrows pointing at the place where they are "currently stationed". There are high-level operations for moving them about in a controlled manner.

Atoms, lists, sets, graphs, nodes, arcs, pseudo integers and readers make up the class of structures known as objects. The sole reason for this classification is that there is another attribute of graphs, nodes and arcs: they each may have an object. The function OBJECT, given a graph, node or arc, will return the object which the programmer earlier associated with that graph, node or arc. The association is established, and changed as desired, by the user. Initially, no arc, node or graph has an object, so, like VALUE, OBJECT will return the "false value" (zero) if no object has been associated with that structure. We draw the structure-object relationship by means of a dashed (not dotted) arrow from a structure to its object.

In summary

At atom is a "data constant" with a printable image which the user specifies. There may be any number of atoms; each may appear in the atomset, and may have a value. Each may have an nset and a gset.

A list is an ordered sequence of GROPE structures; there may be any number of lists, each of which may contain any number of

elements. Each may have a value (which is not considered to be in the list). The user controls which and in what order items will appear in the list.

A set is an ordered sequence of GROPE structures; there may be any number of sets, and any number of distinct elements in any set. Each set may have a value (which is not considered to be in the set). The user controls which items appear in the set (though none may appear more than once), and GROPE controls the ordering of those items within the set.

A graph is a labelled structure on which a (possibly empty) collection of nodes may reside; only these nodes may appear in its nodeset. The graph may appear at most once in the graphset. It may have an object and a value, and appear in the gset of its label.

A node is a labelled structure that resides on a graph. It may be linked to other nodes (as well as to itself) by directed arcs. It has a (possibly empty) set of those arcs outgoing called its rseto, and a (possibly empty) set of those arcs incoming called its rseti. It may appear in the nset of its label, and in the nodeset of its graph. It may have an object and a value.

An arc is a directed, labelled structure that links two nodes. It may appear in the rseto of its frnode (the node where it originates), and in the rseti of its tonode (the node where it terminates). It may have an object and a value.

There is one atomset -- the ordered sequence of atoms.

There is one graphset -- the ordered sequence of graphs.

The nodeset of a graph is the ordered sequence of nodes on that graph.

The rseto of a node is the ordered sequence of arcs emanating from that node.

The rseti of a node is the ordered sequence of arcs pointing to that node.

The nset of an atom is the ordered sequence of nodes (on any graphs), each with that atom as its label.

The gset of an atom is the ordered sequence of graphs, each with that atom as its label.

A relate set is the atomset, the graphset, a nodeset, or an rseto.

An attach set is an rseti, an nset, or a gset.

A system set is a relset or an attset.

A linear structure is a list, set, or system set.

An object is an atom, list, set, graph, node, arc, pseudo integer, or reader.

A value is an object or system set.

Atoms, graphs, nodes and arcs may be related.

Graphs, nodes and arcs may be attached.

Table 1.4 tabulates all the data structures and their attributes.

	is attset	is relset	is syset	is linear	is text	is object	is label	has value	has object	has label	may relate	may attach
pseudo int						x						
reader						x						
atom						x	x	may			x	
arc						x	x	may	may	must	x	x
node					x	x	x	may	may	must	x	x
graph					x	x	x	may	may	must	x	x
list				x	x	x	x	may				
set				x	x	x	x	may				
atomset		x	x	x	x							
graphset		x	x	x	x							
nodeset		x	x	x	x							
rseto		x	x	x	x							
rseti	x		x	x	x							
nset	x		x	x	x							
gset	x		x	x	x							

Table 1.4 The GROPE data structures and their attributes

CHAPTER II

Processing Linear Structures

The most frequently employed method for graph processing is accessing sequential elements of one or more of the system sets -- such as the nodeset of a graph, or the rseto of a node. The reader is the chief search and access mechanism provided by GROPE. The reader, along with its associated manipulative operations, allows the programmer to process a linear structure step-by-step. The GROPE function READER(LS) will create and return as its value a new reader for the text (linear structure) LS. Given any reader R, the function TO(R) will traverse the reader out -- advance it to the next item (to the right) in the linear structure it is reading -- and return as its functional value the element arrived at. The first TO of a reader which is yet unmoved will produce the first element in the linear structure; the second traversal will produce the second, etc. Thus Figure 2.1 illustrates an algorithm for finding the nth component of any linear structure. Normally, one searches a linear structure in this manner only once for any particular process; therefore there must be a way of ascertaining when the structure has been processed once. There are two ways. One is to set up a loop which increments a variable by one for each access, and terminates when the value of that variable (initially set to one) equals the LENGTH of the linear structure; Figure 2.2 illustrates this possibility. There is also a predicate

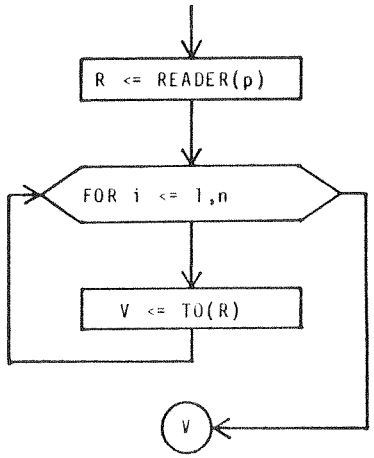


Figure 2.1 The n^{th} Component of p

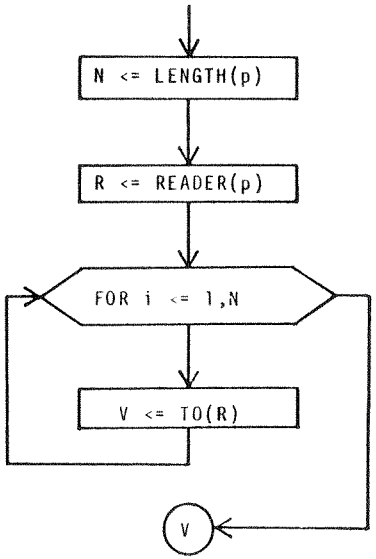


Figure 2.2 The Last Component of p

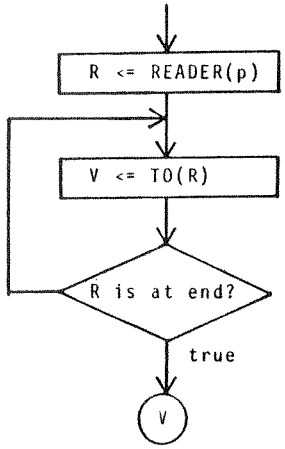


Figure 2.3 The Last Component of p

which, given any reader, determines whether that reader is at the end of the linear structure it is reading; Figure 2.3 illustrates this possibility. Obviously, after the reader advances to a new item, and before it advances to the next item, any algorithm may operate. This is the usual processing method in GROPE.

In GROPE, all linear structures are symmetric -- that is, one may search them in either direction; one may even move forwards for a while, then backup, then proceed again, etc. Moreover, these structures are "circularly linked", which means that, from the last item, one may move a reader (with TO) directly around to the first item; similarly, from the first item one may move a reader (with TI) directly around to the last item. This traverse in facility opens a new dimension in processing linear structures. The programmer is in no way bound to perform a strictly linear, sequential search. Using TO and TI, the user may move a reader forwards and backwards through a linear structure as long as necessary to satisfy any processing requirements.

The operations TI and TO are only two of the many reader operations available in GROPE. For instance, it is easily conceivable that one might wish to process an entire list -- including sublists -- in a single pass. There are three additional operations which provide for this possibility: DESCEND, ASCEND, and a predicate which answers whether a reader is "deep" inside a structure.
At any time the programmer may command a reader to descend to another structure: DESCEND(R,LS) descends the reader R to the linear structure LS. While in this descended state, the reader may be traversed over the structure LS just as described before; the reader may even descend to another structure (there is no limit on depth). When a sub-structure has been processed completely, the programmer may cause the reader to ascend back to the structure from which it most recently descended. Since a reader does not "lose its place" when it descends, one may continue processing from the point where he descended, perhaps descending to another sub-structure. In this way, entire lists may be searched in depth; the same is true, of course, for any linear structure (such asa set). In general, such searches are slightly complicated by the fact that the programmer may not know when he is processing the "main" structure; it is illegal to try to ascend out of the top-level structure. Therefore there is the predicate to ascertain whether the reader is deep. Usually, when the reader is at the end, and is deep answers false, then the list (or whatever) has been searched completely. Figure 2.4 shows the algorithm for a complete search of a list, with the dotted line indicating where any processing of a non-list element would take place.

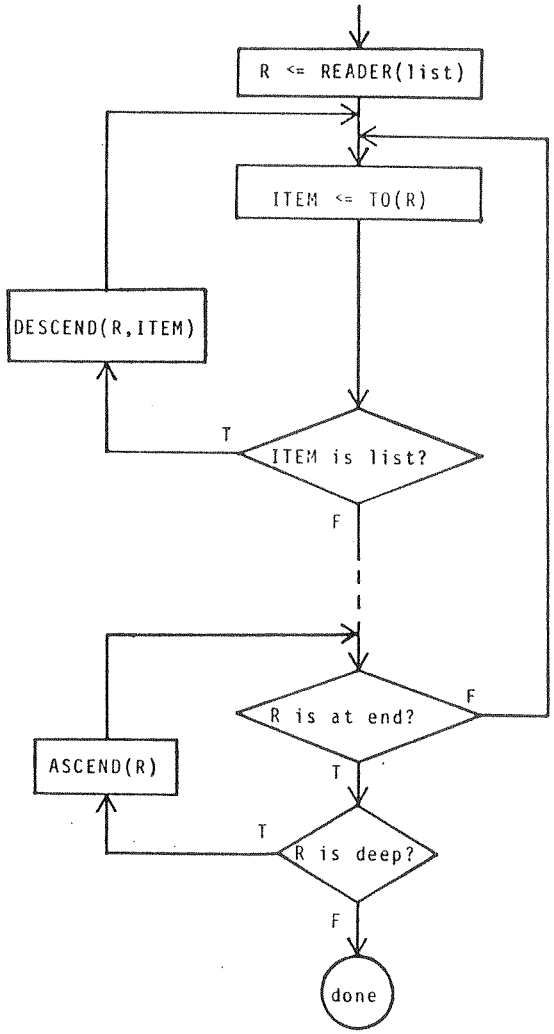


Figure 2.4 An in-depth list search

There are other operations that are associated with the reader: one may substitute a value (any GROPE structure) for the list element whose position is being indicated by a reader: SUBST(R,X) will store the value X in the list being read by the reader R, at the position indicated by R. R is unaffected. The user may retrieve the linear structure being read by a reader R with the GROPE function TEXT(R); the value within that structure which the reader is "pointing at" may be retrieved with TOKEN(R). That is, after X <= TO(R) and Y <= TOKEN(R), X and Y name the identical structure. SUBST(R,T) and X <= TOKEN(R) will cause X and T to name the identical structure; R is unmoved, but the list (the TEXT of R) has been affected in that T appears where another element used to be. The length of the list is unaffected.

The programmer may delete arbitrary items from a linear structure: again using the reader to indicate which item, DELETE(R) will remove that item from the linear structure. In order to prevent the reader from "dangling", GROPE moves it back one position (with TI). The length of the list is of course decremented by one. The programmer may also insert an item anywhere in a list. With the reader marking the position with respect to which the insertion is to be made, a new value may be inserted to the right of the reader's current position with INSERT(R,X). Since the new item is added to the list, the length of the list is incremented by one; the reader is unaffected. Note that INSERT(R,X) and Y <= TO(R) will cause X and Y to name the identical structure.

Two separate lists may be merged into one list, with the reader marking the position where the merge is to take place: MERGE(R,L) will not insert the list L to the right of the reader's position, like INSERT(R,L) would do, but rather insert all the elements of the list L to the right of the reader's position. The list L will become empty, and the list which is the TEXT(R) will grow by the number of elements originally in L. A list may also be split into two lists: SPLIT(R) will produce a new list, composed of the elements in the former TEXT(R) which were at and to the right of the reader's position. The reader R will have as its new text, the new list, and it will be stationed at the first element of this new list -- the identical element at which it was located in its former text. Thus the TOKEN(R) is unchanged, but the TEXT(R) is. The old text is shortened by the number of elements transferred to the new list.

The last major reader operation is the function RESTART(R), which resets the reader R to its "unmoved" state -- which it occupied when first created. An unmoved reader has a text (the structure it is reading), but no token: it is not pointing at any element in the text structure. From this unmoved state it may traverse out to the first element, or alternately traverse in to the last element, and continue searching from there.

Given the list and reader mechanisms, the user has a complete list processor: he can create and search lists, add and delete items, including sublists, and merge and split lists at any

desired location. The automatic "stack" feature of the reader mechanism allows complete in-depth processing of arbitrarily complex list structures without resorting to procedural recursion or user maintenance of a stack. Furthermore, the high-level operations provide the programmer with a conceptually simple yet elegant mechanism for the solution of list-processing problems. But of course GROPE is intended to be a graph processing system; the list feature is only for support. The next chapter returns to graph processing with a vengeance. All manner of generality is introduced, and a truly novel use for readers is presented: searching graph structures -- in which the ordering conventions are non-linear, leaving the reader with some element of choice as to which path it will choose, given the command to "traverse out", or "traverse in".

CHAPTER III

Generalizations

The preceding two chapters have dealt with elementary relationships. This chapter will present some of the more subtle aspects of GROPE. Specifically, the notions of complex graph-based structures, one-way arcs, graph readers, and sophisticated graph modification operations will be developed. Many graph processing algorithms can exist without these features; but then, many graph processing algorithms require considerable execution time -- which could likely be substantially reduced if the right data structures and operations were available. There is evidence to support the contention that GROPE is a good step in this direction. The special features to be discussed make graph processing an exciting, challenging, and rewarding experience, both conceptually and productively. A programmer can master the first two chapters (and indeed other graph processing systems) and still not fully understand what we mean by "graph processing".

Complex graph-based structures

On the surface, nothing has been presented which directly allows for truly sophisticated data structures. Arcs, nodes and graphs have an *object* attribute and a *value* attribute (atoms, lists and sets also have the latter): what has not been overtly mentioned is that the *object* attribute may be any of those grope structures known as objects -- atoms, graphs, nodes, arcs, lists, sets, pseudo

integers and readers; the value attribute may be any GROPE data structure -- objects and system sets. (Table 1.4 might profitably be studied again.) But the most significant restriction being (conceptually) removed here is that labels of graphs, nodes and arcs be atoms -- any graph, node, arc, list or set may also serve as a label for a graph, node or arc. Figures 3.1 and 3.2 provide indicative illustrations of this generalization. In each figure, g and h are graphs; n, m, w, x, y, and z are nodes; and a, b, c, d, and e are arcs.

These complex graph-based structures have a number of potential uses. For example, by allowing the labels of nodes to be graphs, these structures simulate hierarchical graphs. Nested finite automata (e.g., the Woods machine [42]) might be represented by allowing the labels of arcs to be graphs. This could also be simulated by using graphs for the objects or values of nodes and arcs, so GROPE might be criticized for providing overkill; but needless to say, many applications have been posited which involve heavy use of all three attributes -- even including simulating a LISP-like property list feature (something not directly available in GROPE) with an association list as the value of a structure so that more attributes might be available. The LABEL, OBJECT and VALUE functions have the advantage of no list searches: the attributes are available for the cost of an address computation. Since the label attribute is required at creation time, anyway,

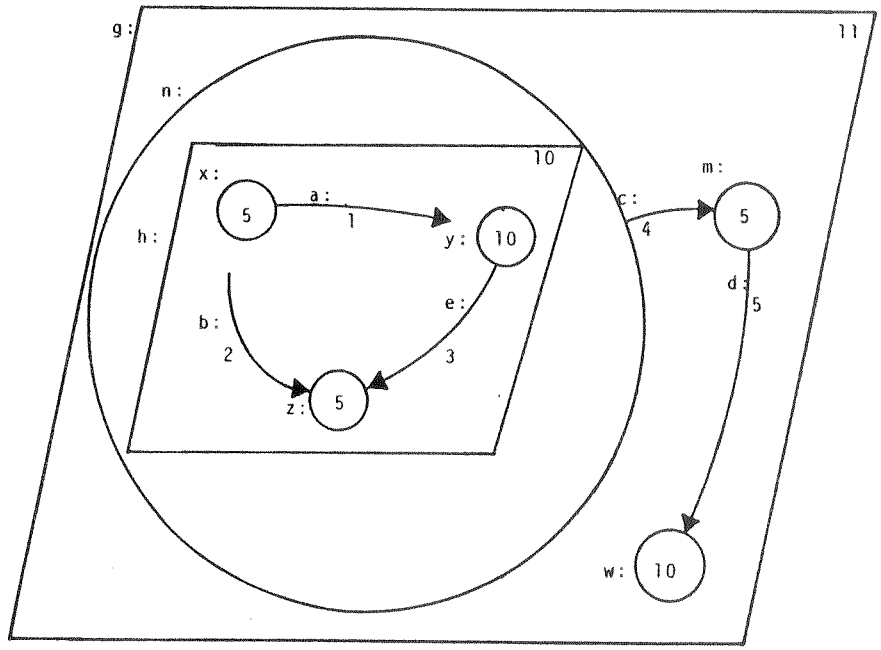


Figure 3.1 A Complex Graph-based Data Structure

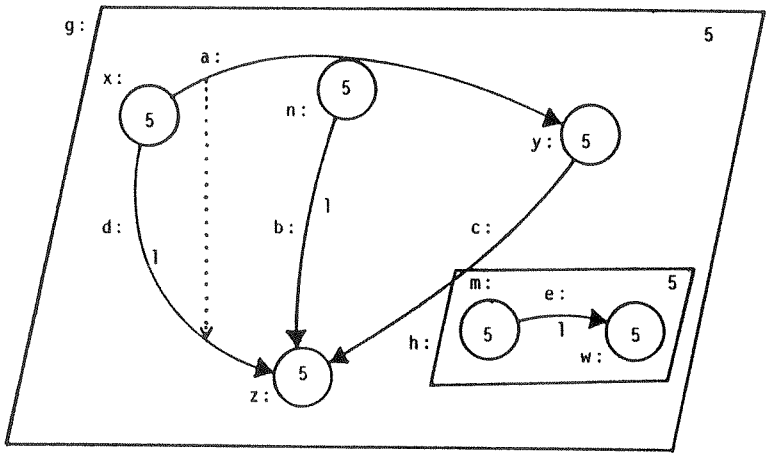


Figure 3.2 Another Complex Graph-based Data Structure

while the others are "extra", the generalized label features is also a matter of programmer convenience -- one of the more forceful arguments in the GROPE repertoire.

One-way arcs

The removal of restrictions is a natural way to incorporate generality -- and inefficiency, to be sure, but every effort has been made to win this particular battle; we feel we have done so. This paradigm is one of the cornerstones of the GROPE design, and one-way arcs is an example of this paradigm. The restriction being removed is that an arc from node n to node m is necessarily in the RSETO(n) and in the RSETI(m). For many algorithms, this restriction creates no problem; but by removing it nevertheless, we obtain a more complete and general class of structures. An arc always "knows" its frnode (from which it emanates) and its tonode (to which it points), as well as its label, object, and value. But it is now possible to find an arc in the rseto of its frnode, which is not in the rseti of its tonode. Conceptually, such an arc is "visible" from its frnode (we draw them touching), but invisible at its tonode (drawn not touching -- consider arc a in Figure 3.3). Such an arc is "one-way-out". Similarly, there may be "one-way-in" arcs, visible (touching) only at the tonode. These capabilities are exceptionally useful for "non-destructive" directed graph traversals in which re-traversal of arcs is undesirable: arcs can be disconnected (unrelated or detached, as

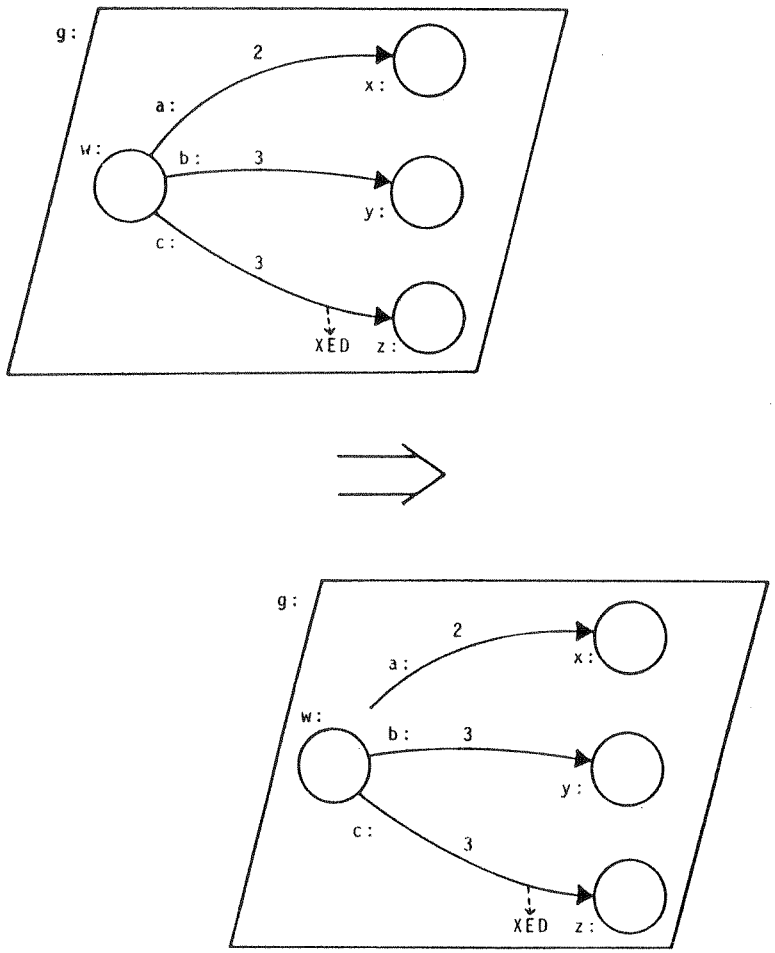


Figure 3.3 UNRELATE(a)

the case may be -- see Table 1.4), and yet not actually be destroyed, allowing the graph to be trivially regenerated. In extreme cases, arcs may be isolated -- unrelated and detached; but still they may be fully restored via the functions RELATE and ATTACH. Considering that no storage is going to be released (transformed into "garbage" which must be collected), or required at restoration time, and that no "list" searches are required for the detach and unrelate operations, the advantages of this scheme are obvious.

System sets

Need it be stated that the above capabilities extend to all system sets? No atom must be in the atomset; no graph, in the graphset; no node, in the nodeset of its graph (although it may not be in any other graph's nodeset), etc. In effect, the presence or absence of an arc, node or graph in its appropriate system set(s) has become a piece of information in itself: the programmer may desire to make something "invisible" for a period of time. The reader is left to consider the possibilities offered by this simple generalization, with the reminder that all these structures still "remember" to which system set(s) they "belong".

		g	h	w	x	y	z	n	m	a	b	c	d	e	5	10
graph				g	h	h	h	g	g							
frnode										x	x	n	m	y		
tonode										y	z	m	w	z		
label		11	10	10	5	10	5	h	5	1	2	4	5	3		
value																
graphset	{g,h}															
nodeset		{n,m,w}	{x,y,z}													
nset			{n}												{x,z,m}	{y,w}
rseto				{}	{a}	{e}	{}	{c}	{d}							
rseti				{d}	{}	{}	{b,e}	{}	{c}							

Table 3.1 Retrievals for Figure 3.1

		g	h	w	x	y	z	n	m	a	b	c	d	e	5	10
graph				h	g	g	g	g	h							
frnode										x	n	y	x	m		
tonode										y	z	z	z	w		
label		5	5	5	5	5	5	5	5	n	1	h	1	1		
value										d						
graphset	{g,h}															
nodeset		{n,x,y,z}	{m,w}													
nset															{w,x,y,z,n,m}	
rseto				{}	{a,d}	{c}	{}	{b}	{e}							
rseti				{e}	{}	{a}	{b,c,d}	{}	{}							

Table 3.2 Retrievals for Figure 3.2

The graph readers

Actually, there are two types of "graph" readers: one called a graph reader, and the other, a node reader. What is the difference? Consider the definition of an arc (Chapter I). Nothing says that the frnode and the tonode of an arc must be on the same graph! There is no reason in GROPE why this should be so, so it isn't. Therefore graph readers are restricted to a particular graph -- their text -- while node readers may traverse from node to node, heedless of any graph boundaries that may be crossed in the process. Conceptually, arcs that cross graph boundaries are invisible to graph readers. The function TO will move a node or graph reader outwards along an arc to its tonode; TI, inwards, to the frnode. However, the readers only see arcs in the rseto and rseti, respectively. (And a graph reader might not even see all of these, as explained above.)

The next question is: which arc will a reader cross? The graph reader mechanism is a new idea in graph processing, so naturally it must be coupled with another new idea for choosing arcs: the concept of "current" arc is hereby introduced. When a node or graph reader crosses an arc going outwards, it makes that arc the "current-arc-out" for that node. When some node or graph reader next traverses out from that node, it will not choose the current-arc-out (the curco) unless there is no other arc in the rseto; instead, it chooses one of the others -- actually the one

following the curco in the rseto. Thus successive departures from a node will choose different arcs in a cyclic fashion, each time making the arc crossed the curco. A similar argument applies to readers moving inwards along arcs: each node may also have a curci -- one of the arcs in its rseti -- and traversals will automatically cross different arcs among those available in the rseti. The function CURCO(N) will retrieve the current-arc-out for the node N, and CURCI(N) will retrieve its current-arc-in. The global variable CURARC will be set to the arc most recently crossed in any direction by any node or graph reader. Conceptually, a "+" may be placed on the tail of an arc (near its frnode) when it is crossed outwards and made the curco of its frnode, and on the head of an arc (near its tonode) when it is crossed inwards and made the curci of its tonode. The algorithm in Figure 3.4 will thus operate to produce the second graph in Figure 3.5, given that the graph g previously existed in the first state shown. (Note that the reader is sometimes unable to move.)

At this point not much experience has been gained using the graph reader mechanism due to its utter newness, and it would be unreasonable to make any strong statements about it. Suffice it to suggest that the curco and curci feature -- which is cheap in terms of storage, and which eliminates "list" searches for graph traversal with the reader mechanism -- appears to be a good means for automatically remembering a path between nodes. We look forward to some experimentation with graph readers to determine the

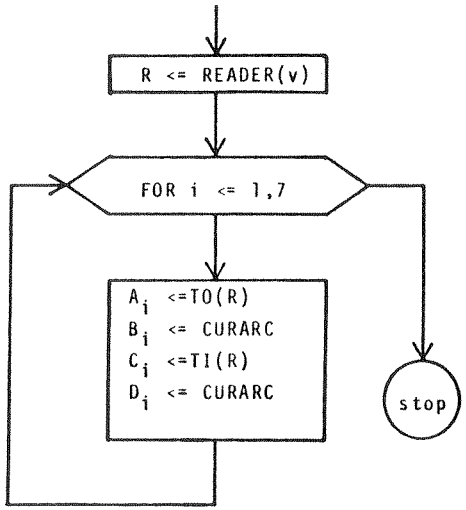
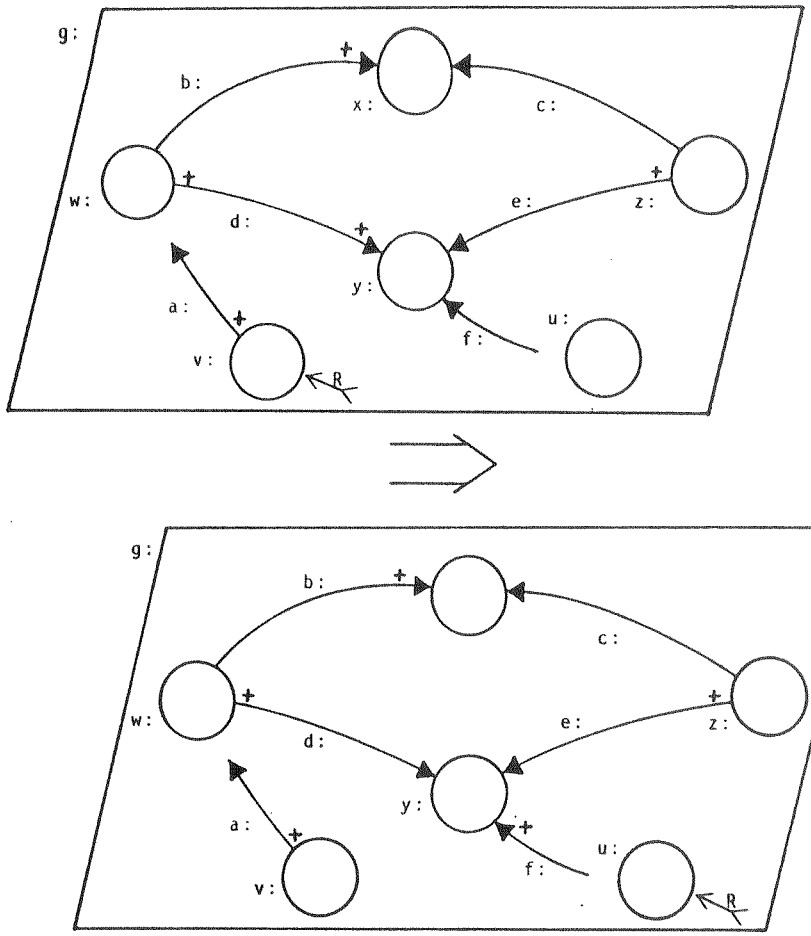


Figure 3.4 Traversing with the Node Reader



	1	2	3	4	5	6	7
A	w	x	x	y	y	false	false
B	a	b	c	d	e	f	f
C	false	z	w	z	u	false	false
D	a	c	b	e	f	f	f

Figure 3.5 Data and Results of Algorithm 3.4

relative utility and flexibility of this tool within the context of graph processing.

Graph modification operations

The structural modification operations introduce us to yet another generalization aspect of GROPE. One may of course change the values and objects of GROPE structures: OBJECT(X) <= Y will make the (object) structure Y the object of (the graph, node, or arc) X; VALUE(X) <= Y will make the GROPE structure Y the value of the (label) structure X. The labels may be changed, also: LABEL(X) <= Y will make Y the new label of (the graph, node, or arc) X. Note that this may force GROPE to (automatically) remove (the graph or node) X from the gset or nset of its old label; if X was indeed attached, then this will be the case, and X will be added to the gset or nset of its new label Y. If, however, X was detached, then it will not be attached to its new label Y. Thus the truth of <is X attached?> is unchanged by this operation.

An even more interesting change operation involves changing the tonode of an arc: TONODE(A) <= N will make the arc A point to a different node -- N. The truth of <is A attached?> (is it in the rseti of its tonode?) is the same before and after the operation, so this might involve moving the arc from one rseti to another -- automatically handled by GROPE, of course. Naturally, if one can change the tonode, then one should be able to change the frnode: FRNODE(A) <= N will perform this operation, and perform any

unrelating and relating that might have to be performed in the process -- rendering the logic of <is A related?> unchanged. See Figure 3.6

Surely the most interesting change operation of all, though, is changing the graph of a node: GRAPH(N) <= G will conceptually rip the node N out of its old graph and deposit it in the graph G -- but its arcs are unaffected. (They "stretch" like rubber bands!) Since arcs are allowed to cross graph boundaries, this poses no problem, other than perhaps deleting N from the nodeset of its old graph and adding it to the nodeset of its new graph G. (This is unnecessary if it was not related to begin with.) See Figure 3.7.

Considering that none of these "change" operations involve destroying any structures (leaving garbage), creating any structures (asking for more storage), or linear-structure searches (to find items in order to delete them from sets), we believe that the implied efficiency -- coupled with the other generalizations mentioned in this chapter -- will play a major role in reducing some of the combinatorial aspects of graph processing. The programmer now has available a truly high-level graph processor, encouraging the development of more and more sophisticated graph processing algorithms and data representation techniques. See Figure 3.8.

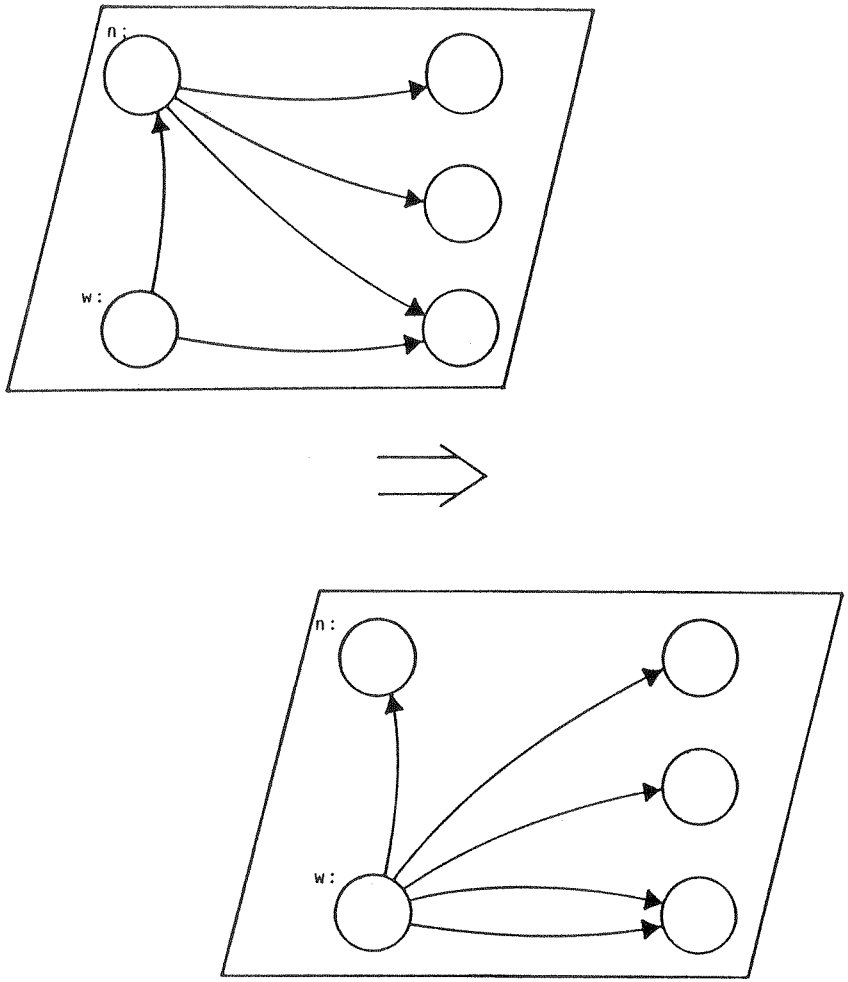


Figure 3.6 For each arc in RSETO(n), FRNODE(arc) <= w

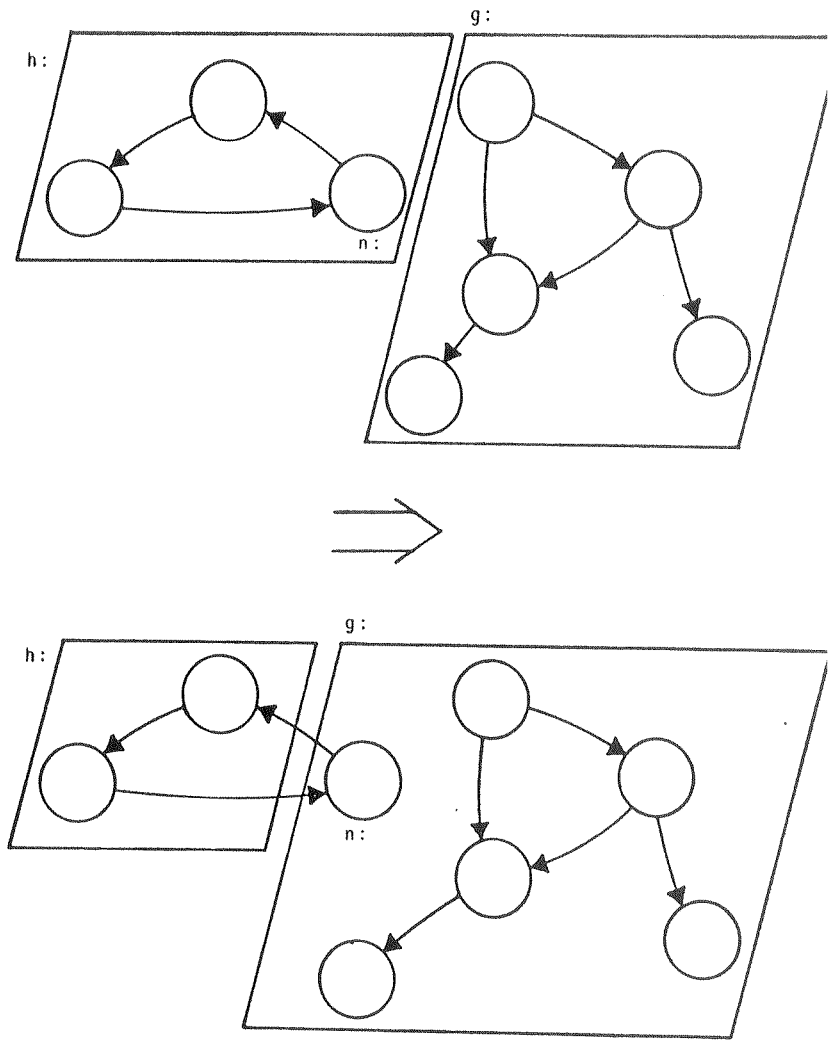


Figure 3.7 Changing the Graph of a Node
GRAPH(n) <= g

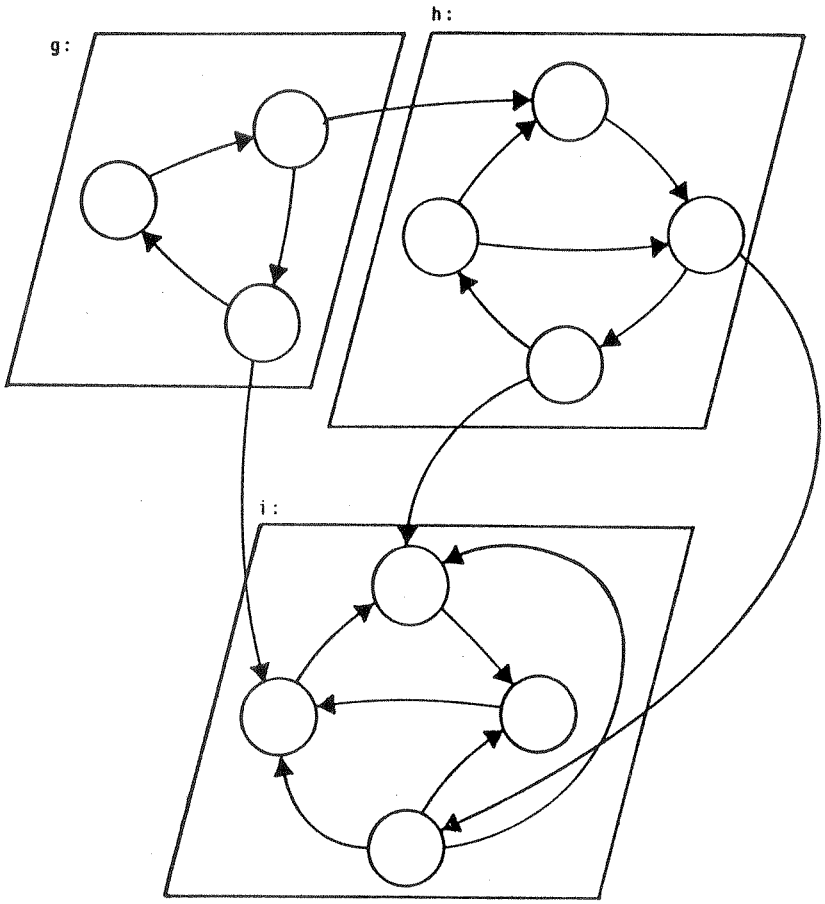


Figure 3.8 Representation of a Graph by Subgraphs

CHAPTER IV

The Data language

The Data language of GROPE 2 is the means of describing, in linear form, that subset of GROPE data structures known as labels. This facility allows a GROPE program to read "S-expressions" (in the LISP idiom) from data files, and to create or find the GROPE structures which they represent. Within the capabilities of the language, the programmer need not resort to I/O measures of his own, but rather he encodes the structures he desires to input in terms of Data language strings. LISP allows a programmer to input lists and atoms; in GROPE the possibilities are extended to include graphs, nodes, arcs, and sets as well. The Data language has also been incorporated into the Programming language by means of a powerful literal facility: a pair of quote (") symbols may enclose any readable (Data language) label, producing a "constant". Thus the programmer may refer to labels from within the program, with all the convenience of Data language expression.

The syntactic description of the Data language is in informal BNF, augmented with the Kleene star (indicating zero or more occurrences of an entity) and a superscript plus indicating one or more occurrences. Semantic descriptions are in English, augmented with GROPE-style pictures where necessary. (These pictures are of the form described earlier.)

<character>::= <letter> | <digit> | <special character>

<letter>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|
k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
<break> <digit> | <break> <special character>

<digit>::= 0|1|2|3|4|5|6|7|8|9

<special character>::= <delimiter> | <break> | ' | / | * | -

<break>::= ↑

<delimiter>::= <punctuation mark> | <bracket> | <blank> | # | = | + | ¬

<punctuation mark>::= " | $ | % | ¢ | & | ? | ! | ; | : | , | .

<bracket>::= (|) | [|] | (|) | < | >

<blank>::= <the usual space character>

With the exception of the cent-mark (included only as a punctuation mark), and the substitution of "¬" for "\", the language is defined in terms of a subset of the ASCII character set. The usual Roman alphabet is available in upper and lower case; the usual Arabic digits are available. The punctuation marks are those of English; brackets will always occur in the obvious pairs. Note that the break character transforms the semantics of any subsequent digit or special character into that of an unspecified letter; the two are considered to be one (letter) character.

Blanks may serve to separate lexical items and perhaps to enhance readability; they are significant only within strings.

Any number of "insignificant" blanks is equivalent to one (delimiter). Transitions from one line (card) to the next are of no special consequence.

<string>::= ' <any character sequence not containing '> '

<identifier>::= <any character sequence not including delimiters which cannot be interpreted as anything else>

<vector>::= < <scalar number>$^{+}$ >

<scalar number>::= + <unsigned number> | - <unsigned number> | <unsigned number>

<unsigned number>::= <decimal number> <exponent> | <decimal number> | <decimal number> K | <decimal number> M | <unsigned integer> / <unsigned integer> | <decimal number> % | <octal digit>$^{+}$ B

<unsigned integer>::= <digit>$^{+}$

<decimal number>::= <unsigned integer> . <unsigned integer> | . <unsigned integer> | <unsigned integer>

<exponent>::= E <integer>

<integer>::= + <digit>$^{+}$ | - <digit>$^{+}$ | <unsigned integer>

<octal digit>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Strings, identifiers, vectors and scalar numbers are self-explanatory. Unsigned numbers are FORTRAN-type numbers (real and integer), plus fractions, octal (base 8) integers, percentages, and decimal numbers with a scaling factor of 1000 (K), or 1,000,000 (M). The latter two are sometimes encountered in

engineering applications.

<atom>::= <string> | <identifier> | <vector> | <scalar number> | <punctuation mark>

Atoms are the basic building blocks in GROPE. Graphs, nodes, and arcs usually have atoms as their labels. Atoms are composed of strings, identifiers, vectors of scalar numbers, scalar numbers, or punctuation marks. The Data language is designed so that GROPE may read English text directly, so punctuation marks are delimiters except for their use in forming numbers; but note that the FORTRAN real-number representation in which the real number is terminated by its decimal point is illegal in GROPE. (The syntax is explicit about this point.) The decimal point must be followed by a digit, or else it will be interpreted as a period -- not part of the number.

<list>::= (<list-set value> <label>*)

<set>::= { <list-set value> <label>* }

<list-set value>::= ¬ <label> ¬ | <empty>

<empty>::=

Lists and sets are formed by the appropriate left delimiter, then the value of the structure if one is desired, then the elements of the structure, if any, then the closing delimiter. Remember that in GROPE the user may not control the ordering of elements in sets!

Therefore it is possible that the internal ordering of the elements of a set will not be the same as indicated when the set was input in the data stream.

<node>::= # <attach> <label> <object> <value> <relate> <graph> #

<attach>::= ¬ | <empty>

<object>::= :: <label> | <empty>

<value>::= : <label> | <empty>

<relate>::= ¬ | <empty>

A node is delimited by number-sign (#) pairs. Its label and graph must be specified; the object, the value, and the attach and relate signs are optional. The node will be attached to its label unless the "not" attach sign is present to indicate otherwise; the node will be related to its graph unless the "not" relate sign is present. If the node is to be attached, the input routine will first search the nset of the label for a "like" node and return it if it exists; only if such a node does not exist, or the node is not to be attached, does the input routine create a new node and return it. The test for a "like" node is: (1) the (nset) candidate must be on the same graph as the input node; (2) if the input node is to be related, then the candidate must be, and vice-versa; (3) if the input node specifies its object, then the object of the candidate node must be identical; and (4) if the input node specifies its value, then the candidate node must

<g-element>::= <node-in-graph-context> | <arc-in-graph-context>

<node-in-graph-context>::= # <attach> <label> <object> <value> <relate> #

<arc-in-graph-context>::= = <fnigc> # <rel-co> <label> <object>
<value> <att-ci> # <tnigc> =

<fnigc>::= <attach> <label> <object> <value> <relate>

<tnigc>::= <attach> <label> <object> <value> <relate>

A "graph element" is either a node or arc. In either case, the graph obviously need not be specified -- since it encompasses its elements. The syntactic description of <g-element> only serves to formalize what one could assume: the <graph> constituent has been deleted from the <node> and <arc> syntax to produce <node-in-graph-context> and <arc-in-graph-context>. GROPE prints a graph by printing its label, object, etc., then printing every arc in the rseto of each node in its nodeset, plus printing any "isolated" nodes (having no arcs). It is expected that graphs may be entered in the same fashion.

The Data language is not comprehensive in several respects: (1) some types of atoms are automatically hashed into the atomset, while others are not -- with no means for indicating otherwise; (2) only labels may be input -- no readers, pseudo integers (they become integer atoms), nodesets, etc.; (3) input values for atoms cannot be indicated as they may be for graphs, lists, etc.; (4) within the context of a graph, arcs to or from nodes on other

graphs may not be input; (5) input of graphs and nodes with lists (or sets, or non-integer numeric atoms, or arcs) as their labels is unreliable, since the automatic creation of the new label guarantees the creation of a new graph or node; (6) some applications might require a search before creating an arc; and (7) input of recursive structures is difficult or impossible, depending on circumstances. On the other hand, it is not expected that masses of data will be stored in Data language form -- GROPE provides a mechanism for maintaining a permanent internal data base, coupled with the virtual memory system. The Data language is rather intended to be a convenience item, with somewhat limited use: there is no I/O device or medium that would allow display, storage, or input of the more complicated GROPE structures in any "picture" form. It is hoped that this representation will suit most applications.

have the identical value.

<graph>::= [<attach> <label> <object> <value> <relate> <g-element>*]

A graph is delimited by square brackets. Its label must be specified; its object, value, "elements", and the attach and relate signs are optional. The graph will be attached to its label unless the "not" attach sign indicates otherwise; the graph will be related (to the graphset) unless the "not" relate sign is present. If the graph is to be attached, the input routine will first search the gset of the label for a "like" graph and return it if it exists; only if such a graph does not exist, or the graph is not to be attached, does the input routine create a new graph and return it. The test for a "like" graph is: (1) the (gset) candidate must be related if the input graph is to be, and vice-versa; (2) if the input graph specifies its object, then the object of the candidate graph must be identical; and (3) if the input graph specifies its value, then the value of the candidate must be identical.

Once a graph has been found or created, the graph elements (if any) are entered, and the graph is returned by the input routine.

<arc>::= = <frnode> # <rel-co> <label> <object> <value> <att-ci> # <tonode> =

<frnode>::= <attach> <label> <object> <value> <relate> <graph>

<rel-co>::= + | <relate>

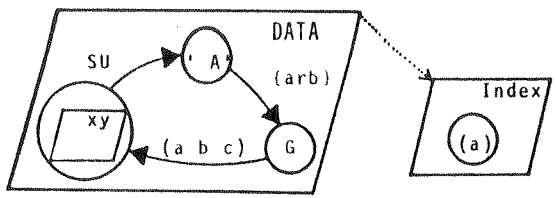
<att-ci>::= + | <attach>

<tonode>::= <attach> <label> <object> <value> <relate> <graph>

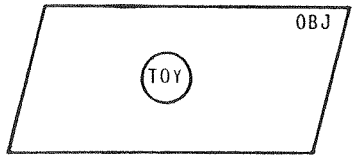
An arc must specify at least its frnode, its label, and its tonode; the object, the value, and the "relate-curco" and "attach-curci" signs are optional. If the <rel-co> is "+" then the arc will become the curco of its frnode (which necessarily relates the arc), else the arc will be related unless the "not" relate sign indicates otherwise. If the <att-ci> is "+" then the arc will become the curci of its tonode (which necessarily attaches it), else the arc will be attached unless the "not" attach sign is present. A new arc is always created and returned: no search for a "like" arc takes place.

<label>::= <atom> | <graph> | <node> | <arc> | <list> | <set>

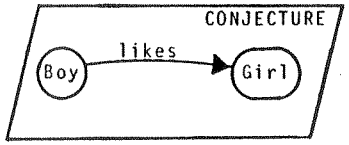
A label is either an atom, a graph, a node, an arc, a list, or a set. GROPE input insures unique atoms, except for those with non-integer numeric images, through a hashed search of the atomset; GROPE insures unique graphs and nodes within limits, as described. Arcs, lists, and sets, on the other hand, are automatically created -- without any search -- as are atoms with (type) real and vector images. Thus the input routine may duplicate arcs, lists, sets, and some atoms. Non-integer numeric atoms will not be related; all others will be.



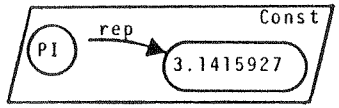
[DATA:[Index¬ #(a)#] =[xy]#SU#' A'= =' A'#{arb}#G= =G#(a b c)#[xy]=]



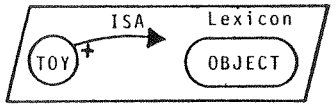
#TOY[OBJ]#



=Boy[CONJECTURE]#likes#Girl[CONJECTURE]=



=PI[Const]#¬rep#3.1415927[Const]=



=TOY[Lexicon]#+ISA¬#OBJECT[Lexicon]=

Figure 4.1 A Graph, a node, and three arcs

CHAPTER V

The Programming language statements

For clarity, we must depart from the traditional Backus-Naur form grammar in this chapter: there are simply too many options and choices to indicate with regards to the Programming language. While the language certainly can be described in BNF, to do so would render the description less concise and readable. The meta language to be employed, however, will be seen to have some similarities to BNF; it makes use of three new formalisms: (1) where the programmer has a (single) choice among multiple alternatives, the alternatives are listed vertically, enclosed within large brackets -- usually square brackets, but occasionally parentheses; (2) large parentheses indicate the enclosed item(s) to be optional -- it (they) need not appear in the statement to be generated; (3) superscripts are employed to indicate repetition of the bracket-enclosed item(s) -- the Kleene star has the traditional interpretation of "zero or more consecutive occurrences," the plus sign "+" indicates at least one or more occurrences, and the minus sign "-" indicates that on the last occurrence, the last symbol within the brackets (typically a comma) does not appear. (Commas serve to separate items, rather than terminate them.)

The names of non-terminals will be enclosed within angle-brackets (the symbols "<" and ">") as in BNF, and terminal keywords will be CAPITALIZED. The appearance of blanks is not accounted for in the syntax; rather, the familiar programming language con-

ventions apply: blanks are used as delimiters, and to enhance readability. Source statements of the language may be written in free-field format, eighty (80) columns per line if desired. Comments in the programming language are delimited by pairs of ampersands (&). Occasional abbreviations of non-terminal names may occur, such as "id" for "identifier", and "var" for "variable". Their interpretation should pose no problem to the reader, since English names are invariably employed.

The development will proceed top-down, starting with what constitutes a GROPE program, and ending with a description of the available character set.

<program>::= <procedure>* <main procedure> <procedure>*

A complete GROPE 2 program is a collection of modular procedures, including exactly one MAIN procedure to which control will be transferred by the operating system when execution of the program is initiated. This MAIN procedure corresponds to the main procedure in PL/I, the FORTRAN main program, etc.

Any number of procedures may be batched together and passed to the compiler at any one time; however, the user must bear in mind that the compiler is modular: no information will be carried over from one procedure to the next. The compiler is designed to provide very good error diagnostics. Every attempt will be made to correct an error, or if that is impossible, to recover with a

"fatal" flag (supressing the production of object code for that procedure) and continue compiling the procedure in order to inform the programmer of any other syntactic errors. If a serious error is encountered, though, the compiler may get completely lost and generate a cascade of "errors". In order to protect against this, the compiler will cease compilation after the tenth error in a procedure and scan ahead to the next procedure; this is accomplished by scanning for the reserved word FIN, which marks the end of a procedure. Compilation will then resume in normal mode. (Note: when at any time the terminating FIN is encountered, the rest of the line after the FIN is discarded -- meaning that each procedure definition must begin on a new line.)

<main procedure>::= MAIN PROCEDURE <procedure identifier>
(⟨ <file identifier> [<unsigned integer>], ⟩⁻) ;
⟨ <statement> ; ⟩⁺ FIN

The main procedure has a name (some identifier), and must list all files which the user intends to employ in the program. Each such file has a designated "record length" -- an unsigned integer indicating the number of characters (columns) per line in the file. Typically, this will be 80 for "data card" input files, and 132 or 136 for printer output files.

ex: MAIN PROCEDURE TEST (INPUT[80], OUTPUT[132], OTHER[80]); ...

The record size actually stipulates the right-hand margin that GROPE will "see", since the actual (physical) record may be longer (but not shorter).

<procedure>::= (RECURSIVE) (<type>) PROCEDURE <procedure identifier> (<formal parameter list>) ; ⟨<statement> ;⟩* FIN

The body of any procedure is composed of a sequence of statements (each terminated by a semicolon) terminated by the reserved word FIN. The statements may be declarative or executable in any order so long as any declaration of a variable precedes its use in an executable statement; of course it is highly recommended that all declarations precede all executables, just to be safe. Any undeclared identifiers encountered in an executable statement before being encountered in a declaration or an executable context obviously indicating them to be procedure identifiers or statement labels, are classified simple variables of LOCAL scope and GROPE type.

If a procedure is to be invoked recursively, the keyword RECURSIVE should occur first in its definition; if it is to return a non-GROPE value (GROPE is the default), then the type of value to be returned must be declared. Procedure arguments are passed by reference (as in FORTRAN); since the GROPE 2 translator does not check whether the number or types of actual parameters passed to a procedure corresponds to the number for which that

procedure is defined, disruptive results are likely to occur at run-time if these do not indeed correspond. These situations cannot be detected because the translator is modular, nor are run-time checks produced. The local FORTRAN compiler will define the semantics of these cases: at any rate their use is not recommended.

<formal parameter list>::= ([<variable specifiers> , (<type>)⟨<proc. id>((⟨<type>,⟩⁻)),⟩⁺]⁻)

A procedure (not the main procedure) may be defined with or without parameters. If extant, the formal (or dummy) parameters of a procedure are named and typed in the formal parameter list; FORTRAN, PL/I, and ALGOL, for instance, require the programmer to name the dummy variables in the parameter list, then "declare" them in separate statements. GROPE 2 is arranged so that no variable is mentioned more than once in the declarations of any one procedure; the formal parameter list is considered to be a declaration, except that it implies no particular storage allocation.

The EXTERNAL declaration here defines subsequent identifiers to be functional arguments -- procedures passed as actual parameters. Their types may also be declared, and must be declared if not GROPE.

<statement>::= [<declaration statement> | (: <sta. label> :) <executable statement>]

A statement is either a declaration or a (possibly labelled) executable statement. Statements are compiled in sequence (there

are no compile-time branches), and normal run-time flow-of-control is of course sequential. Branching is allowed via labelled statements and a GOTO, but the richness of control structures should obviate the need for this construct in all but the most extreme situations.

<declaration statement>::= [EXTERNAL ⟨(<type>)⟨<proc. id> ((⟨<type>⟩⁻)),⟩* | [GLOBAL | LOCAL | OWN] <variable specifiers> | COMMON < <common id> > <var. specifiers> | <type list>]

Identifiers have scope (EXTERNAL, GLOBAL, LOCAL, OWN, COMMON) and type (ALPHA, GROPE, REAL, etc.). EXTERNAL scope is reserved for procedure identifiers other than the one containing the declaration. The EXTERNAL declaration may be necessary if a procedure is to be passed as an actual parameter to another procedure (as in FORTRAN). In the corresponding position in the dummy parameter list of the called routine, the dummy parameter is also identified as an EXTERNAL.

GLOBAL variables have a scope encompassing the entire program, although they are accessible only to the procedures which declare them. Their storage allocation is fixed at compile time, since they are implemented via the FORTRAN labelled COMMON feature.

LOCAL variables are allocated storage at run-time (via a stack in "blank" COMMON) when the procedure declaring them is entered, and deallocated when the procedure is exited. Except for

any procedures to which individual LOCAL variables are passed as arguments, they are inaccessible outside the procedure declaring them. Their contents upon allocation is indeterminate, and is lost upon deallocation.

OWN variables are like (implemented as) FORTRAN local variables -- fixed allocation at compile time, accessible only to the procedure declaring them (other than when passed as arguments), and retaining their values when the procedure is exited.

Labelled COMMON behaves exactly as it does in FORTRAN. It is available primarily for communication with FORTRAN routines; however, it may be used by the GROPE programmer as desired. Of course, variables in COMMON are "global" by location with respect to the origin of the block, and so may be renamed and re-typed in different procedures -- naturally, not recommended.

The scope keywords apply until the next semicolon is reached (terminating the statement), but several type lists may be included. If a declaration begins with a type keyword, the default scope is LOCAL. If the identifier following a scope keyword is not a type keyword then type GROPE is the default.

<variable specifiers>::= [⟨<variable specifier> ,⟩⁻ | ⟨<type list> ,⟩⁻] ⟨, <type list>⟩*

Within a declaration statement the declared variables are separated by commas. Each type keyword applies until the next is encountered, or until the end of the statement (semicolon) is

reached.

<type list>::= <type> ⟨<variable specifier> ,⟩⁻

<type>::= [ALPHA | GROPE | INTEGER | LOGICAL | REAL]

A type list is a type keyword followed by a sequence of variable specifiers separated by commas. As mentioned, type lists may succeed one another in a single declaration. The type of a variable determines the nature of its contents once a value has been assigned to it: ALPHA variables contain character strings; GROPE variables, (pointers to) GROPE structures; INTEGER variables, binary integer numbers; LOGICAL variables, "true" or "false"; and REAL variables, single-precision real numbers.

<variable specifier>::= <variable description> (:= [<constant> | < ⟨<constant> ,⟩⁻ >])

A variable specifier describes the variable and specifies the initial value (if any) the user desires to assign to that variable at compile-time; the initial values must be "constants". Since storage for LOCAL variables is allocated at run-time, and this initializati feature is implemented via the DATA statement of FORTRAN, initial values for LOCAL variables may not be specified. An array variable may be assigned a vector of initial (constant) values. If the number of constant values is insufficient to

account for all the locations being allocated to an array variable, then the assignment sequence is repeated: the (first) constant being assigned to the next allocated location, etc., until all locations have been provided with an initial value. GROPE 2 allocates array storage by row, as opposed to by column, as in FORTRAN. In effect, one may view the rightmost subscript as varying most rapidly, then the next rightmost, etc., with the leftmost subscript varying least rapidly.

<variable description>::= [<array id> [⟨(<arith. exp>:) <arith. exp> ,⟩⁻] | <simple variable>]

As in ALGOL and some other languages, the user may declare both bounds of each subscript of an array, provided each first bound is the lowest. The default lower bound for a subscript is one; thus the default is FORTRAN-style 1-origin array indexing. The number of subscripts allowed is three for non-LOCAL arrays, and theoretically unlimited for LOCAL arrays. Note that there is no special keyword for array declaration; the only special provision is that the bounds be specified (within square brackets, as usual), and that the subscripts may be evaluable when storage is to be reserved for the array. This means that the subscripts for GLOBAL, OWN, and COMMON arrays must be integer constants (which the compiler can interpret); the subscripts for LOCAL arrays, on the other hand, may be arithmetic expressions involving constants and perhaps "dummy" parameters, since storage is reserved at run-time immediately upon entry of the procedure.

<array identifier>::= <identifier>

<common identifier>::= <identifier>

<file identifier>::= <identifier>

<procedure identifier>::= <identifier>

<statement label>::= <identifier>

<simple variable>::= <identifier>

<identifier>::= <letter> [<letter> | <digit>]*

Identifiers are formed by an arbitrary sequence of letters and digits, provided the first is a letter. The length of an identifier is theoretically unlimited, but in practice procedure identifiers, file identifiers, common identifiers and GLOBAL variable identifiers are truncated to six or seven character, as required by the local FORTRAN compiler and operating system. The user is responsible for insuring uniqueness at this six- or seven-character limit with respect to these types of identifiers, whose scope is above the procedure level. In the case of OWN variables, the translator provides new and guaranteed unique names to the FORTRAN compiler; LOCAL variables "disappear" into the run-time stack mechanism, which guarantees them unique run-time locations for each procedure entry. Because the GROPE library routines contain many procedures, COMMON blocks and file references to which the user must be denied access, all identifiers whose first two

characters are "G2" are reserved.

<constant>::= [<string> | <number> | " <label> " | <proc. id> | FALSE | TRUE]

<string>:: ' <any character sequence not containing '> '

<number>::= (+ | -) [[<digit>+ (. <digit>+) | . <digit>+ | <digit>+ B] (E (+ | -) <digit>+)]

<label>::= <a printable GROPE data structure, as defined in the Data language>

GROPE provides ALPHA constants (strings), INTEGER and REAL constants (in the FORTRAN sense), GROPE constants (atoms, arcs, nodes, graphs, lists, and sets), procedure identifier "constants" (EXTERNAL references), and the LOGICAL constants TRUE and FALSE. Strings are delimited by single quote marks (apostrophes), and numbers are as defined in FORTRAN, with in addition octal integers: some sequence of digits (0-7) terminated by the letter B. (The appearance of digits 8 or 9 in an octal integer is semantically illegal.)

The Data language entities may be included in the programming language delimited by double quote (") marks. Since GROPE 2 object code is actually FORTRAN, which must pass through some FORTRAN compiler and a loader before being executable, and since

GROPE structures are run-time entities, the translator cannot actually create (or "find") the linked data structures corresponding to the descriptor (the label). Instead, the translator generates code such that, upon the first invocation of a procedure containing GROPE "constants", those constants are read by the GROPE input routine. The resultant structure is then accessible wherever its descriptor appeared in the procedure. Note that one or more occurrences (exact copies) of a given descriptor in one procedure will cause the input and possible creation of only one structure, whereas the occurrence of identical descriptors in different procedures may, depending on the input routine and descriptor, cause more than one structure to be input and created -- possibly a different one for each procedure, though not more than one for any given procedure. The description of the operation of GROPE input in the Data language chapter should clarify this situation. In any case, the generated structure is guaranteed to correspond to the "picture"; but it is possible for there to be several copies of the structure generated. Note that it is possible (but not necessarily recommended) for the programmer to employ normal GROPE operations to subsequently modify such "constants" so that they no longer resemble their original descriptors.

We are now in a position to consider some examples of procedure headings and declarations.


```
ex:  RECURSIVE PROCEDURE INPUT;
     EXTERNAL HELP, INTEGER FUN1, ALPHA NEXTITEM;
     GLOBAL ALPHA RDFILE[7] <= 'INPUT';
     LOCAL X,Y,Z;
     OWN INTEGER POSITION <= 0, LIMIT <= 120;
     COMMON <IOBLOCK> ALPHA CLASS[30], INTEGER NUMBERS [70];
```

This starts the definition of a recursive GROPE procedure INPUT which takes no parameters; it may call (among others) procedures HELP (which is type GROPE), FUN1 (which is type INTEGER), and NEXTITEM (type ALPHA). Note that, unless HELP is to be passed as an argument to another procedure, its declaration here is extraneous since context would define it to be a procedure identifier (at the calling point), and it would default to type GROPE, anyway; however, FUN1 and NEXTITEM must be declared because they are not of type GROPE. INPUT may access the GLOBAL ALPHA variable RDFILE, whose initial contents is the string INPUT. X, Y, and Z are LOCAL GROPE variables -- their storage will be allocated each time INPUT is entered, and deallocated upon departure. Since INPUT is a recursive procedure, it is possible that several storage locations will be allocated to X, Y, and Z at the same time -- a different one for each variable, for each "level" of recursion. Storage for the OWN INTEGER variables POSITION and LIMIT, however, is fixed at one location each, no matter how many times INPUT is called recursively; thus it is possible for the different "levels" (invocations) of INPUT to intercommunicate via these variables. Before execution is begun, POSITION is initialized to 0, and LIMIT, 120. The labelled COMMON block IOBLOCK contains two variables: the ALPHA vector CLASS, which may contain a string of maximally thirty

characters as its value, and the integer vector (array) NUMBERS, which may of course have seventy distinct integer values. (RDFILE, by this example, could contain a string value of maximally seven characters.) ALPHA vectors are indexable to the character level, rather than the "word" level. The number of machine words required by ALPHA variables is of course a function of the particular hardware; GROPE avoids the problem of word indexing in this case by positing machine-independent character indexing. In the case of the other types, GROPE does not stipulate any particular word size (precision), leaving the determination of this characteristic to the FORTRAN compiler and the machine. (As a note of interest, it is intended that the number of words allocated to ALPHA variables be one word more than the number required to store the declared maximum number of characters -- to provide a binary zero terminator.)

ex: PROCEDURE APPLY (INTFUNCT(INTEGER) , ARG1,ARG2,ARG3);

The non-recursive procedure APPLY takes four parameters: the (type GROPE) functional argument INTFUNCT, and the three (type GROPE) arguments ARG1, ARG2, ARG3. Presumably it will invoke the procedure passed as its first argument, passing an integer argument to it; or it might pass INTFUNCT as one of the arguments to yet another procedure, or both.

ex: RECURSIVE INTEGER PROCEDURE FACTORIAL (INTEGER N);

This would be an appropriate heading for a procedure to compute n!, if one were to write it recursively. Note that the FORTRAN compiler

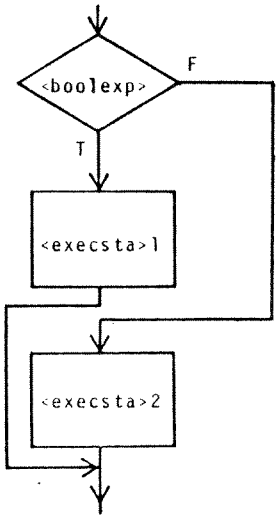
would see instead of the function FACTORIAL, the function FACTOR, or perhaps FACTORI. (Similarly, the labelled COMMON block in the first example might have its name shortened to IOBLOC, and the EXTERNAL ALPHA procedure, to NEXTIT or NEXTITE. Beware!) If an ALPHA variable is declared but unsubscripted, it is an ALPHA simple variable -- which may assume a zero-character value (the empty string), or a single-character value, but no more.

<executable statement>::= [<control statement> | <I/O statement> | <boolean expression>]

For canonical purposes, executable statements are of three types: control, I/O, and (boolean) expression. Control statements control the execution of statements through implicit tests and branches; I/O statements perform the obvious function; and (boolean) expressions may be simple assignments, procedure calls, or indeed any expression allowed in the language. When used as statements, the values of boolean expressions are not important, but rather their side effects are: assignment of a new value to a variable, alteration of GROPE data structures, etc.

<control statement>::= [<compound statement> | <IF statement> | <GO statement> | <NULL statement> | <CASE statement> | <INCRement statement> | <WHILE statement> | <REPEAT statement> | <FOR statement> | <SELECT statement>]

IF <boolexp> THEN <execsta>1 ELSE <execsta>2



UNLESS <boolexp> THEN <execsta>1 ELSE <execsta>2

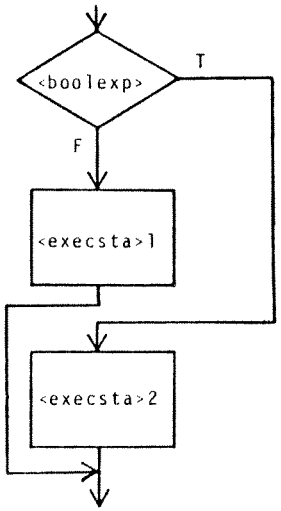
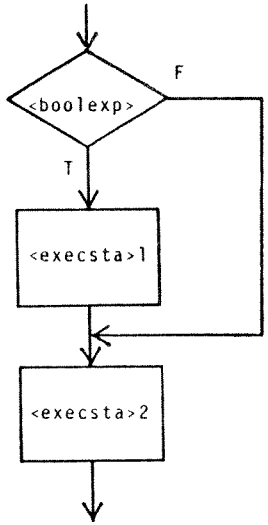


Figure 5.1 The IF statement

IF <boolexp> THEN <execsta>1 REGARDLESS <execsta>2



UNLESS <boolexp> THEN <execsta>1 REGARDLESS <execsta>2

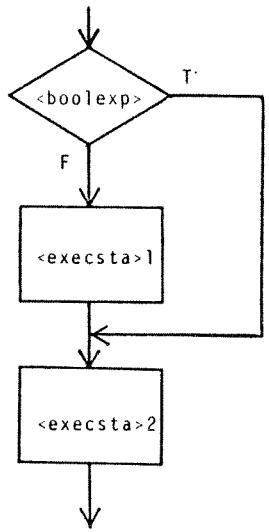


Figure 5.2 The IF statement

<compound statement>::= BEGIN ⟨ : <sta. label> : ⟩ <exec. sta> ; ⟩+ END

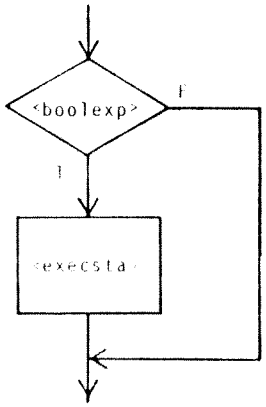
A compound statement is a sequence of (possibly labelled) executable statements initiated by the keyword BEGIN and terminated by the keyword END. Its purpose is to allow several statements to be treated (syntactically) as one. Any statement labels defined within the compound statement are undefined without -- as if it were a block in ALGOL. This is to prevent explicit transfer of control from outside a control structure to the inside by means of a GOTO.

<IF statement>::= [IF / UNLESS] <boolean exp> THEN <exec. sta> ([ELSE / REGARDLESS] <exec. sta>)

The IF statement in its form IF ... THEN ... (ELSE ...) maintains the traditional semantic interpretation: the THEN statement will be executed only provided the boolean expression is true; if it is false then instead the ELSE statement (if present) will be executed. In its form UNLESS ... THEN ... (ELSE ...) the logic is simply reversed: the THEN statement will be executed only provided the boolean expression is false, and the ELSE statement (if present) only provided it is true. In the form IF ... THEN ... REGARDLESS ..., the THEN statement will be excecuted only provided the expression is true, but the REGARDLESS statement will be executed (afterwards) in any case. Again, UNLESS reverses the logic concerning the execution of the THEN statement. In effect, the semantics follows that which the English speaker would expect from reading the

statement. Note that, in GROPE, the representations of integer zero, FALSE (which is implemented as integer zero), the empty string (also integer zero), and the null GROPE value (integer zero) are all treated as false under boolean interpretation. Floating-point zero will also be interpreted as boolean false, and anything else is interpreted as true. Again, the GROPE type LOGICAL is implemented as type INTEGER in FORTRAN: the logical constant TRUE is implemented as integer one. (This is so that the GROPE translator can produce code that is independent of any local test for "true" and "false". The translator-produced test is ".EQ.0" in FORTRAN.) The IF ... THEN IF ... ELSE ambiguity is resolved by matching the ELSE to the closest unmatched THEN to its left.

IF boolexp THEN execsta



UNLESS <boolexp> THEN <execsta>

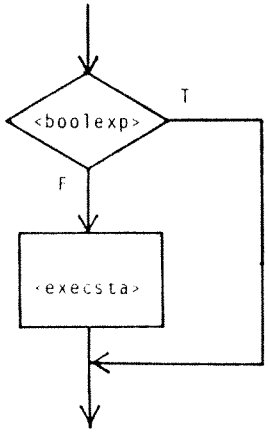


Figure 5.3 The IF statement

<GO statement>::= GO TO <statement label>

The GOTO is included in the language, but its use is somewhat restricted, and in any case is recommended only when no other control mechanism is appropriate (or possible). If used, it overrides all other mechanisms for flow-of-control and transfers control to the named statement label (if it is defined -- remember that a statement label defined within a control structure is undefined without.) A branch may not be effected to a statement label defined within a control structure not containing the GOTO.

<NULL statement>::= NULL

The NULL statement is a no-op: it exists for those occasions where syntax calls for a statement where the programmer has no useful operation to perform. (The ELSE NULL sequence will be seen to have its uses...)

<CASE statement>::= CASE <arith. exp> OF $\left\langle \left(:\text{<sta. label>:} \right) \left[\begin{matrix} \text{<exec. sta>} \\ \text{NEXT} \end{matrix} \right] ; \right\rangle^{+}$

$\left[\begin{matrix} \text{ELSE} \\ \text{REGARDLESS} \end{matrix} \right]$ <exec. sta>

The CASE statement causes execution of the n^{th} statement in the collection of (possibly labelled) executable statements, where n is determined by the arithmetic expression following CASE. The collection is syntactically terminated by ELSE or REGARDLESS; if ELSE, then the executable statement following it is executed iff n

does not correspond to any statement in the collection (n < 1, or n > k where there are k statements in the collection); if REGARDLESS, then the executable statement following it is executed in any case. If n does not correspond to any of the statements in the collection (as noted above), then none of the statements in the collection is executed. Note that the CASE statement does not "transfer control" to the n^{th} statement as does the FORTRAN computed GOTO, but rather causes the execution of that one particular statement. An exception of sorts is the "statement" NEXT: if the n^{th} statement is NEXT, then control will behave as if n had been n+1 -- the succeeding statement will be executed. A sequence of NEXT statements will cause continued "incrementing" of n until a statement other than NEXT is reached, whereupon that statement will be executed. This feature allows several "cases" to share identical executable code, if desired. It is semantically illegal for the last statement in a CASE collection to be NEXT. Any statement labels defined within the CASE statement are undefined outside the CASE statement: the programmer may not branch (via a GOTO) into the middle of a control structure.

CASE <arexp> OF <execsta>1; NEXT; <execsta>3; ... <execsta>n
ELSE <execsta>e

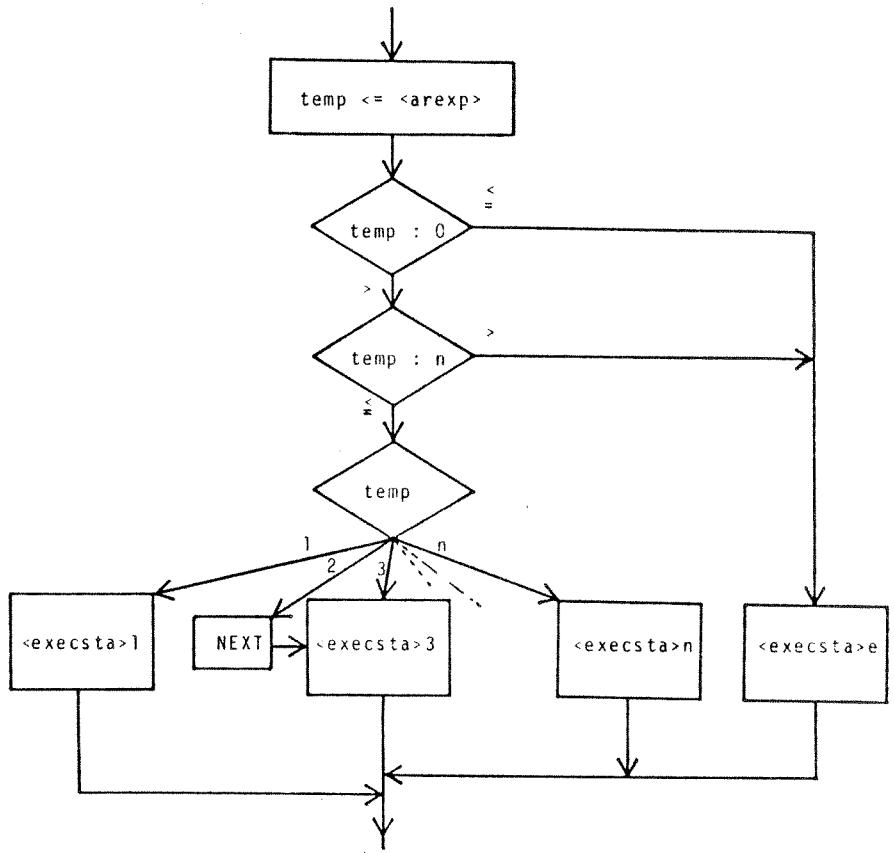


Figure 5.4 The CASE statement

CASE <arexp> OF <execsta>1; NEXT; NEXT; <execsta>4; ... <execsta>n

REGARDLESS <execsta>r

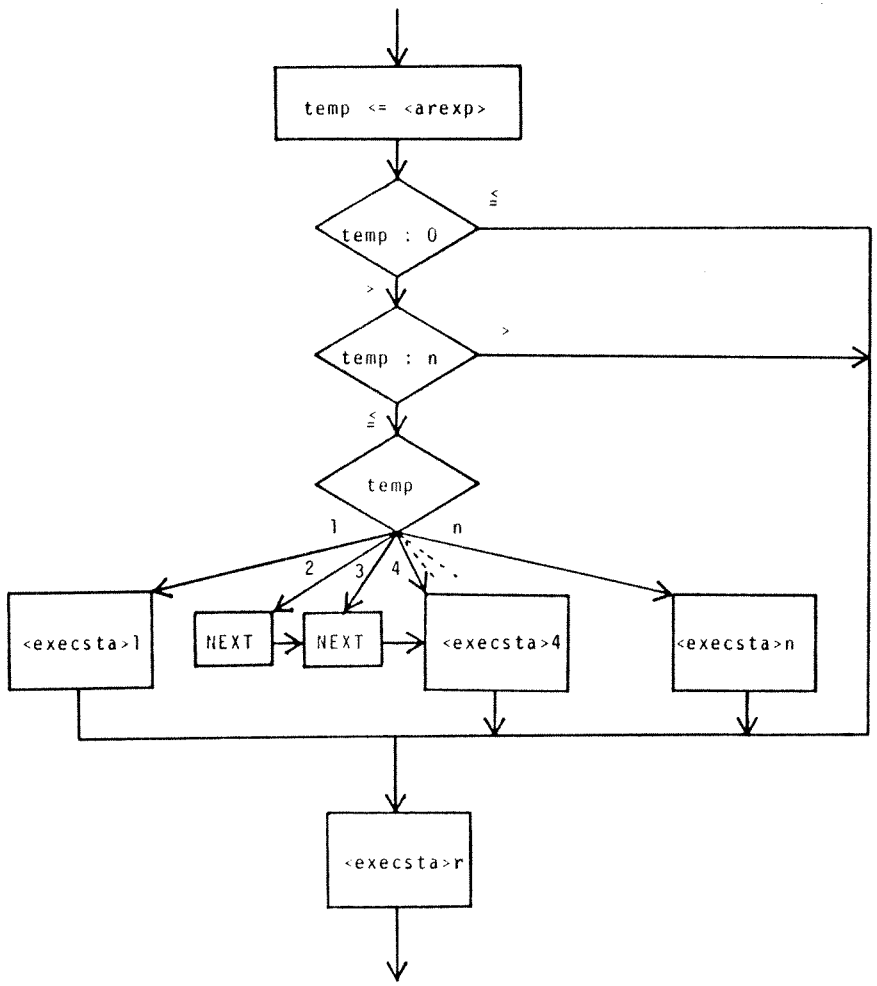


Figure 5.5 The CASE statement

<INCRement statement>::= [INCR / DECR] <simple var> (FROM <arith. exp>) (TO <arith. exp>) (BY <arith. exp>) <repeat loop> (THEN <exec. sta>)

<repeat loop>::= REPEAT ⟨(:<sta. label>:) <exec. sta> ;⟩+ LOOP

The INCRform of the INCRement statement is like FORTRAN's DO loop. However, any simple numeric variable may be used as the control variable, and all three arithmetic control expressions are optional (and default to 1 if absent). The statements in the repeat loop will not be executed if the initial value (the FROM expression) exceeds the terminal value (the TO expression). The optional THEN statement at the end will be executed once after the loop terminates, whether or not any statements in the loop were indeed executed. The value of the simple control variable after leaving the loop will be its value on the last iteration, or indeterminate if the loop is never executed.

The DECRement form of the statement will subtract the BY expression upon iteration, rather than add it as does INCR, and of course the completion test is whether or not the control variable is less than the TO expression. Again, the test is performed before the repeat loop is executed, and may prevent its being executed at all. The FROM, TO, and BY expressions are all evaluated once only, and the control variable will assume its successive values regardless of any manipulation performed within the loop. In no case will the repeat loop be executed if the BY expression is less than or equal to zero.

INCR <simvar> FROM <arexp>1 TO <arexp>2 BY <arexp>3
<repeat loop> THEN <execsta>

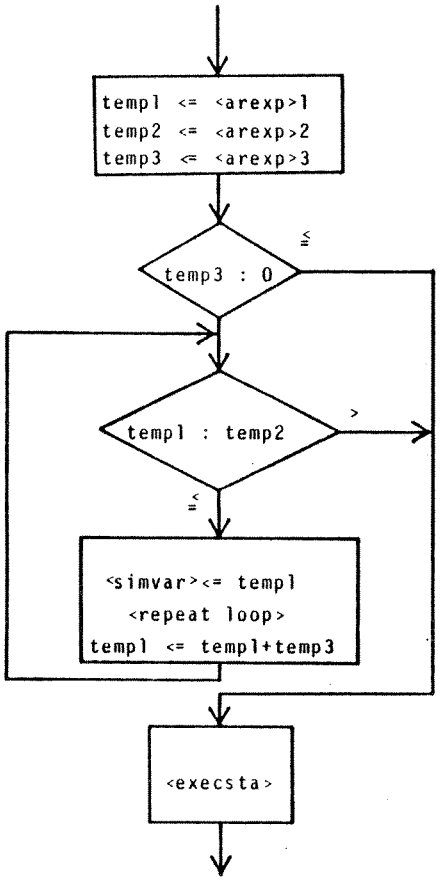


Figure 5.6 The INCR statement

DECR <simvar> FROM <arexp> TO <arexp>2 BY <arexp>3
<repeat loop>

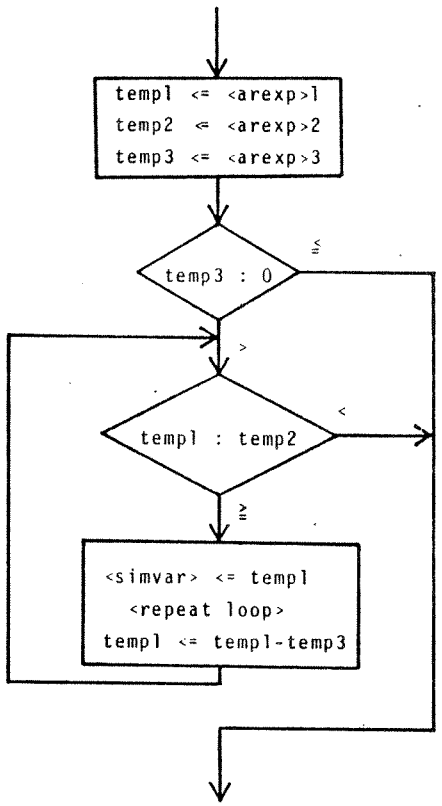


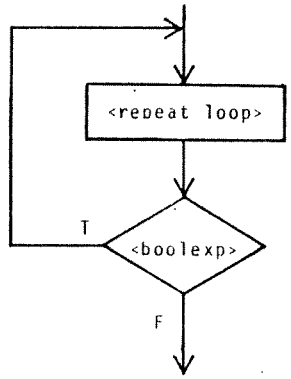
Figure 5.7 The DECR statement

<WHILE statement>::= [WHILE / UNTIL] <boolean expression> <repeat loop> (THEN <exec. sta>)

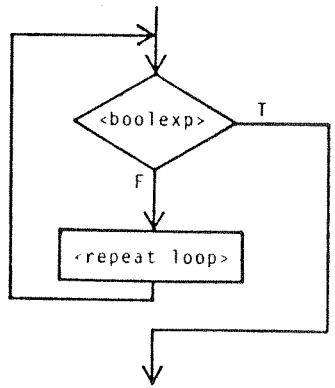
<REPEAT statement>::= <repeat loop> [WHILE / UNTIL] <boolean expression> (THEN <exec. sta>)

The WHILE and REPEAT statements differ only with respect to when the test for iteration is performed relative to execution of the repeat loop. In the former case, the test is performed first, and implies that the loop may not be executed at all; in the latter case, one execution of the loop is guaranteed before the test. The interpretation of UNTIL vs. WHILE is obvious. After the test indicates no (further) iteration is to take place, the optional THEN statement, if present, is executed once.

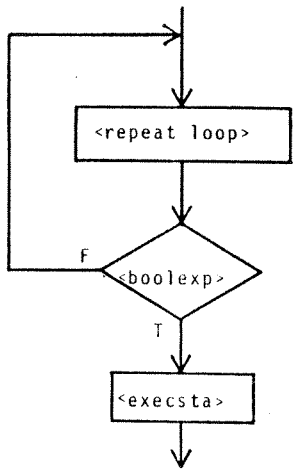
<repeat loop> WHILE <boolexp>



UNTIL <boolexp><repeat loop>



<repeat loop> UNTIL <boolexp>
THEN <execsta>



WHILE <boolexp><repeat loop>
THEN <execsta>

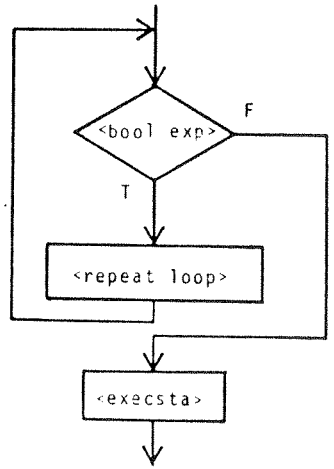


Figure 5.8 The REPEAT and WHILE statements

<FOR statement>::= FOR <quantified universe> <repeat loop
(THEN <exec. sta>)

<quantified universe>::= <quantifier> <simple var> <universe>
(<qualifying exp>)

<quantifier>:: = [ALL | EACH | EVERY | THE (<arith. exp>) FIRST]

<universe>::= [= [<atom descr> | <graph descr> | <node descr> | <arc descr>] IN <grope exp>]

<qualifying expression>::= SUCH THAT <boolean exp>

The FOR statement iteratively binds a GROPE control variable to GROPE values retrieved from some universe, then executes the repeat loop. The programmer indicates the type of element to be retrieved by means of a pattern which describes the form of GROPE structure desired (what type of graph, or arc, etc.), or alternately identifies the universe (some list, or other linear structure) which is to be sequentially scanned. The quantifier indicates how many such elements are to be considered: ALL, EACH, and EVERY are synonyms and indicate that all elements are of interest; THE FIRST indicates that only the first one is of interest, and THE <arith. exp> FIRST indicates that the first n are of interest, where the arithmetic expression produces n. The optional boolean qualifying expression (SUCH THAT ...) may place further constraints on the items. The universe will be searched by GROPE automatically,

but at most once; therefore it is possible that n elements will not actually be available, so n must be regarded as a maximal figure only. The optional THEN statement, if present, will be executed once after the loop terminates, whether or not any acceptable elements were actually found. The value of the control variable upon loop termination will be zero -- the null (false) GROPE value. The atom, graph, node, and arc descriptors will be discussed in Chapter 6, along with the expressions of the language.

FOR THE <arexp> FIRST <simvar> <universe> <qualexp>
<repeat loop> THEN <execsta>

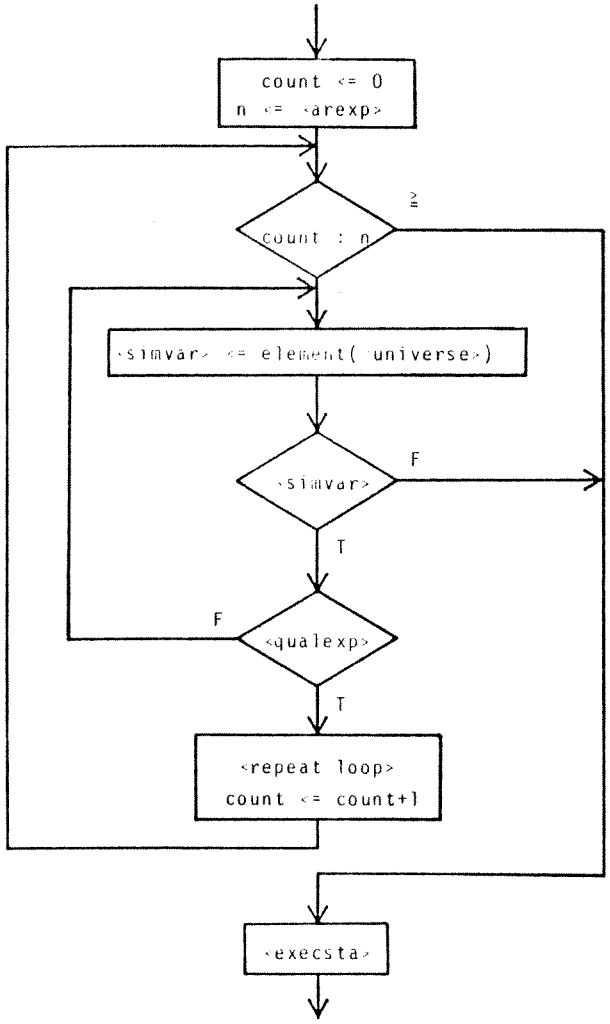


Figure 5.9 The FOR statement

<SELECT statement>::= SELECT <quantifier> <key> FROM ⟨<locked sta> ;⟩¹ [ELSE / REGARDLESS] <exec. sta>

<key>::= <expression>

<locked statement>::= : <lock> : [<exec. sta> / NEXT]

<lock>:: = <expression>

The SELECT statement is somewhat like the CASE statement, although more than one statement in the collection may be executed. In effect, the SELECT statement causes the execution of one or more statements, each of whose "lock" expression is matched by a "key" expression. The key expression is evaluated once only. Then the successive lock expressions in the collection are evaluated until one produces a result equal to that produced by the key; thereupon the corresponding executable statement is executed. If the quantifier ALL (or EACH, or EVERY) is present in the SELECT clause, the lock-evaluating and key-matching operations will be continued throughout the collection; otherwise, after execution of the n^{th} "unlocked" statement (where n is determined by the arithmetic expression in the quantifier), the lock evaluation and matching terminates. The ELSE statement, if present, is executed iff none of the locks are matched ("opened") by the key. The attempted execution of a NEXT statement -- because its lock is opened -- automatically allows the key to open the lock of the succeeding statement without evaluating the lock, and without "consuming" one of the n cases. Thus the role of NEXT here is

similar to its role in the CASE statement, and again NEXT may not be the last statement of the collection. (Due to this specialized usage, NEXT is not a true <statement>, but rather a keyword.)


```
SELECT ALL <key> FROM  :<lock>1: NEXT;  :<lock>2: <execsta>2;
       :<lock>3: <execsta>3; ... :<lock>n: <execsta>n;
ELSE <execsta>e
```

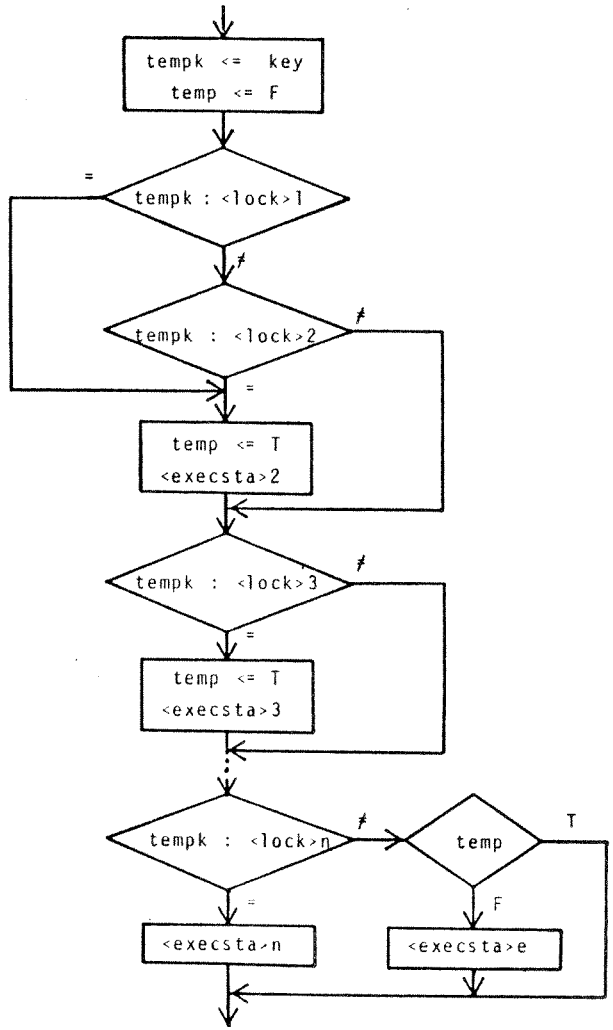


Figure 5.10 The SELECT statement

SELECT THE <arexp> FIRST <key> FROM :<lock>1: NEXT;
:<lock>2: <execsta>2; ... ; :<lock>n: <execsta>n;
ELSE <execsta>e

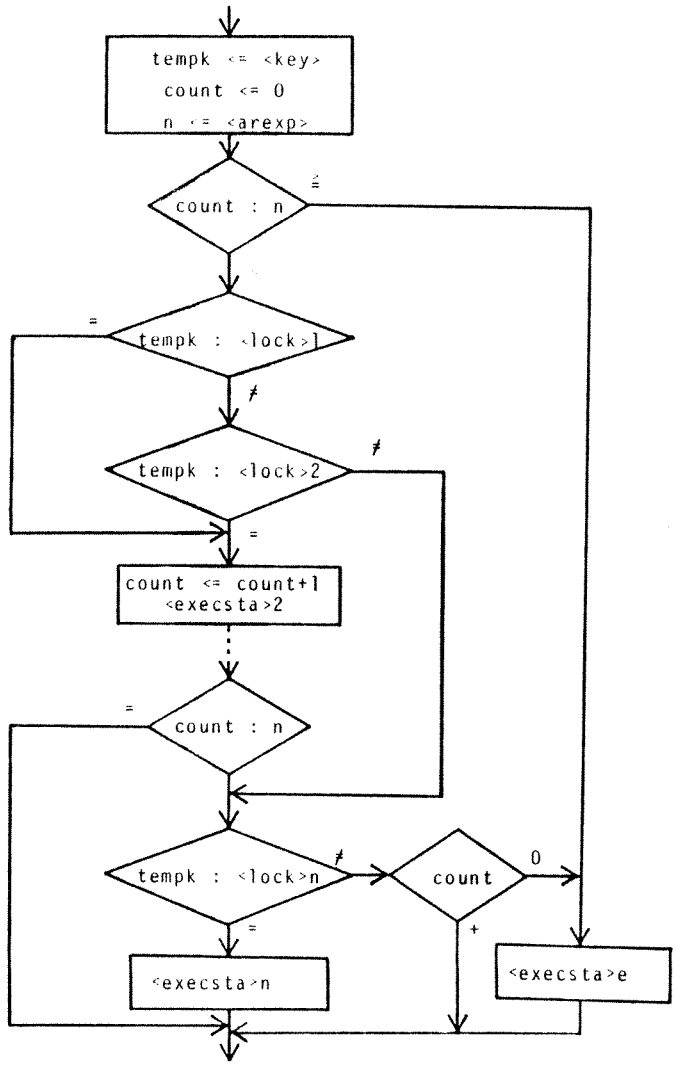


Figure 5.11 The SELECT statement

<I/O statement>::= [INPUT <format exp>⟨<variable> ,⟩⁻ | OUTPUT <format exp>⟨<expression> ,⟩⁻ | [PRIN | PRINT]⟨<expression> ,⟩⁻ | READ ⟨<location>,⟩⁻]

<format exp>::= <alpha exp>

The INPUT statement is roughly equivalent to the FORTRAN formatted READ statement. The (ALPHA) format expression produces a standard FORTRAN format, starting with a left parenthesis (lacking the word FORMAT) and ending with a right parenthesis. Any format which is acceptable in the local FORTRAN will do. The expressions will be read from the file named by the GLOBAL ALPHA variable INFILE.

The OUTPUT statement is similarly equivalent to the FORTRAN formatted WRITE statement; data is printed on the file named by the GLOBAL ALPHA variable OUTFILE.

GROPE provides free-field I/O via the PRIN, PRINT, and READ statements. The GLOBAL ALPHA variable PRFILE contains the logical file name (default 'OUTPUT') onto which PRIN and PRINT will write, and the GLOBAL ALPHA variable RDFILE contains the logical file name (default 'INPUT') from which READ will read. The GLOBAL ALPHA variable ECFILE names the file (default '', indicating no echo) on which READ will echo-print input lines. These may be changed as the user desires; any changes will be noted at the next I/O statement. The difference between PRIN and PRINT is that PRIN

will allow the next free-field output operation on the same file to continue on the same line at the point where PRIN stopped; PRINT will "dump the buffer" after printing all expressions, forcing a subsequent PRIN or PRINT operation to commence with the next (empty) line. The user should take care to use PRINT (or the provided procedure TERPRI) to terminate a line before he uses OUTPUT on the same file -- to avoid "mixing" results. READ will always continue scanning on the same line where it stopped last, unless the provided procedure TERRD is invoked to eject the remainder of an input line. A <location> may be regarded, for the moment, as a <variable>.

In conjunction with free-field I/O there are six GLOBAL INTEGER variables to allow some control over formatting: IMARGIN, ISPACE, ITAB, OMARGIN, OSPACE, and OTAB. I- denotes input, of course, and O- denotes output. The margins indicate the number of columns on the left from (in) which nothing will be read (printed). OMARGIN defaults to 1 (the leftmost column -- typically printer carriage control -- will be blank), while all five others default to zero. The value of (I/O)MARGIN is investigated only when each new input/output line is started. The other four I/O variables are interrogated immediately before respective READ/PRIN(T) statements are executed. If strictly positive, a suitable action is taken and the variable in question is reset to zero. If (I/O)TAB is positive, the read/print scanner is positioned at the column (1, 2, ...) corresponding to the value of the (I/O)TAB

variable, and the tab is reset to zero. Then (I/O)SPACE is investigated; if positive, its value is added to the read/print scanner pointer (thus advancing it to the right the number of spaces indicated), and the space variable is reset to zero. If the actions taken result in advancing the scanner beyond the end of the line, the line is terminated and the next is started at the specified (I/O)MARGIN. Note that the tab, but not the space, allows "backspacing" on the given line.

ex: INPUT <format> A[5],B,C,D[1,3,5:9],E;

This statement will read values for A[5], B, C, five values for the array D (to be stored in locations D[1,3,5], D[1,3,6], ... , D[1,3,9]), and a value for E; the GLOBAL ALPHA variable INFILE specifies from which file the input is to be read, and a FORTRAN format will be provided by the programmer.

ex: OUTPUT <format> 'THE ANSWER IS:',X;

This statement will print the string 'THE ANSWER IS', presumably in "A" format, then the value of the variable X in whatever format is provided, on the file specified by the GLOBAL ALPHA variable OUTFILE.

ex: PRIN 'THIS IS A STRING',1.23456E47,TRUE,X,A[1:10];

This statement will print (free-field -- with items separated by one blank space) the string 'THIS IS A STRING', then the constant 1.23456E47, then the logical constant TRUE, then the value of the

variable X (in some form appropriate to its type), then ten values from the array A (A[1], A[2], ... , A[10]) in the appropriate format. The print line will not be terminated after this statement, so the next free-field output statement may continue printing on the same line.

ex: PRINT A;

The contents of the variable A will be entered into the line buffer in some appropriate format, then the buffer will be printed and cleared. (Other items may have been in the buffer previously -- if PRIN was last used -- and if so they, too, will be printed along with A.)

All of the statements in GROPE 2 have now been presented. The flow-charts should answer any questions about the detailed logic of their implementation and operation, except for the evaluation of expressions and boolean expressions which is subsumed. The next chapter will cover the syntax of (boolean) expressions in GROPE 2.

CHAPTER VI

The programming language expressions

Although a variety of options allows the construction of quite complex expressions, it is not expected that the user will habitually avail himself of all this complexity. Nevertheless, the key to unraveling GROPE expressions is really very simple: within sub-expressions of equal precedence, everything is evaluated strictly left-to-right. Side-effects, if any, appear immediately. The general rule of precedence is: boolean (in the traditional order), followed by assignment (replacement), followed by catenation, followed by arithmetic (in traditional order), then primary expressions -- variables, procedure calls, constants, etc., which are of equal precedence.

<alpha exp>::= <expression>

<arithmetic exp>::= <expression>

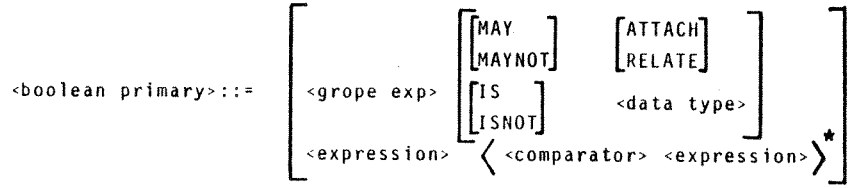
<grope exp>::= <expression>

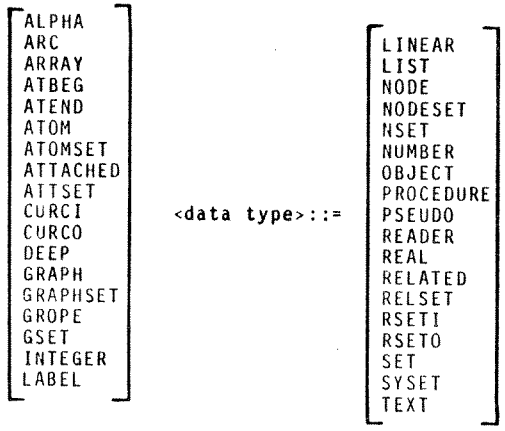
An alpha expression is any expression which produces a string as its result; an arithmetic expression, a number; and a grope expression, a grope value.

<boolean expression>::= <boolean factor> ⟨OR <boolean factor>⟩*
<boolean factor>::= <boolean secondary> ⟨AND <boolean secondary>⟩*
<boolean secondary>::= (NOT) <boolean primary>

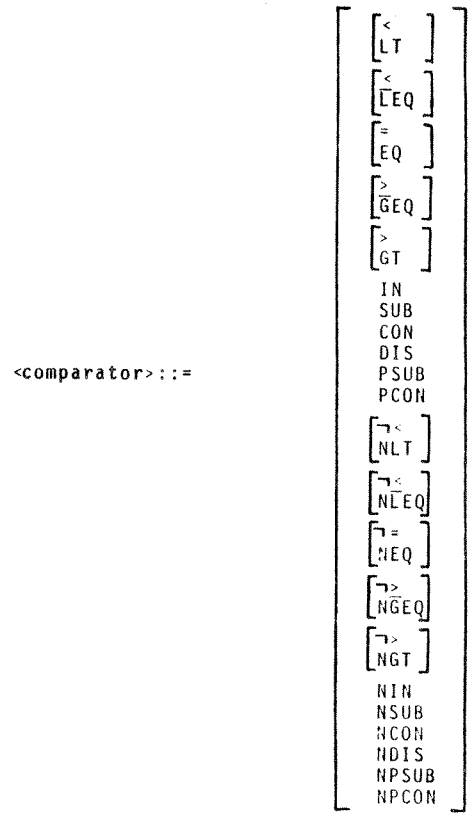
As in LISP, boolean expressions are evaluated only until such time as a true/false value for the expression may be ascertained.

This means that, unlike FORTRAN, not all components are guaranteed to be evaluated. For example, "a OR b" does not involve the evaluation of "b" if "a" is true, since the clause is necessarily true. Only if "a" is false, will "b" be evaluated; this argument generalizes to cases such as "a OR b OR c" and "a AND b". Clauses of equal precedence are evaluated left-to-right.





The programmer may employ certain forms of the boolean primary to test the values of grope expressions with regards to their current structural characteristics. For instance, one might ask if



a certain grope structure IS an ATOM, or if it MAY ATTACH, etc. ISNOT and MAYNOT simply negate the logic of IS and MAY, respectively, without requiring the programmer to use the boolean secondary NOT. (It "reads" better.)

The other form of the primary allows the programmer to compare expressions: some of the comparisons are numeric, and others are set-theoretic. In the case of numeric comparisons, there are two sets of comparators: symbolic and "literal", or FORTRAN-like. (For example, the symbols "<" and "LT" are synonyms for "less than", and "¬≥" and "NGEQ" are synonyms for "not greater than or equal to".) The "not" symbol "¬" and the prefix "N" both negate the logic of the comparison. In addition, the programmer may indicate a long AND-ed set of comparisons by using the (infix) comparators between successive expressions: "a ≤ b < c > d" is somewhat like "a ≤ b AND b < c AND c > d" except that the precedence is not that of AND, and "b" and "c" are evaluated only once, instead of twice each as they would be in the AND form. The set-theoretic comparisons are IN (member), SUBset, CONtain, and DISjoint; "P" is a prefix meaning "proper". For example, "a NPSUB b" means "a is not a proper subset of b". In addition to their numeric semantics, "=" and "¬=" will compare expressions of any type -- ALPHA, GROPE, or whatever. The operators "<", "≤", "≥", ">", and their "not" counterparts will also compare ALPHA expressions; this will, for instance, allow "alphabetization" of strings -- depending on the local FORTRAN collating sequence. That is, GROPE will simply employ

the local FORTRAN character-codes; however, note that strings are binary-zero filled, rather than blank filled as is the FORTRAN standard.

<expression>::= <catenation exp> ⟨<replacement op> <catenation exp>⟩¹

<replacement operator>::= [<= | <=> | =>]

GROPE allows "left" assignment, "exchange" assignment, and "right" assignment. The three characters "<", "=", and ">" are combined in three different ways, indicating the three different replacement operators, and of course "point" to the <location> (variable) receiving the new value. In this respect, the syntactic rule above is perhaps misleading because it indicates that assignment to a catenation expression is allowed; in point of fact it is not (semantically) allowed, but the strict syntactic rules to indicate what is and what is not allowed would cover half the page -- and would not facilitate (human) comprehension. Consider: <location> <= <catenation exp> , <location> <=> <location> , and <catenation exp> => <location> . This conveys some sense of what is allowed; however, such forms as these may be "strung together" much like comparisons. (Again, consider <location> to be <variable>.)

ex: A <= B <=> C => D

This is evaluated left-to-right, remember, and is "equivalent" to: A <= B ; B <=> C; C => D; -- which assigns the value of B to A, then exchanges the values of B and C, then assigns the value of C

(which was the old value of B) to D. So A is necessarily equal to C and D, after evaluation of the expression.

ex: A <= B <= C

This assigns the value of B to A, then assigns the value of C to B; thus A is not necessarily equal to B after evaluating the expression.

ex: A => B => C

This assigns the value of A to B, then assigns the value of B (and of A) to C. Finally, in the first example above (the one involving the exchange), neither A, B, C, nor D could have been a catenation expression since all received (new) values; in the second example, only C could have been an expression; in the third example, only A could have been an expression.

<catenation expression>::= <addition exp> ⟨<catenation op><addition exp>⟩*

<catenation operator>::= [<· | · | ·>]

A catenation expression allows the catenation of two (or more) strings to produce one string as the value of the expression. Each of the three operators (which are composed of some combination of the characters "<", "·" and ">") perform catenation. The first and last, however, also effect an assignment (to a <location>) in the process.

ex: A · B

This will concatenate the values of (the ALPHA sub-expression) A and (the ALPHA sub-expression) B to produce a third string; neither A nor B are affected.

ex: A <· B

This will also concatenate the values of A and B; but after the catenation, the resultant string will be assigned to the <location> A.

ex: A · B ·> C

This will concatenate the values of A and B and C, then assign the resultant string to C; A and B are unaffected.

ex: A <· B · C

This will concatenate A and B and assign the result to A; then that resultant string will be concatenated with the value of C to produce the value for the expression, but A will be unaffected by this operation -- that is, A will still contain the result of A · B.

<addition expression>::= (+ | -) <multiplication exp> ⟨<addition op> <multiplication exp>⟩*

<addition operator>::= [<- | <+ | - | + | -> | +>]

Again we see that the usual addition operators have been augmented so as to allow "simultaneous" assignment. As with the catenation expression, the value of the addition expression is independent of any of the "side-effects" that might take place, unless one of the variables to which a value is assigned by the special addition operators appears later in the evaluation sequence. Again, evaluation of sub-expressions of equal precedence is left-to-right, with side-effects (if any) appearing immediately.

ex: A + B <+ C -> D

The value of this expression is A + B + C - D, but B will assume the value A + B + C, while D assumes the value of the expression.

ex: A <- B + C + A

The value of this expression is A - B + C + (A - B). First A - B is assigned to A; then C is added to the result (of A - B); then the value of A is added, but note A was side-effected earlier, so its new value is the effective value added to the partial sum to produce the value for the expression. A, of course, must be a <location>, rather than just any multiplication expression.

<multiplication expression>::= <exponentiation exp> ⟨<multiplication op> <exponentiation exp>⟩*

<multiplication operator>::= [<* | </ | * | / | *> | />]

The multiplication operators are also augmented to allow side-effect replacement. It should be obvious by now how these will operate, and surely multiplication (and division) need no explanation, other than to note that integer division involves truncation.

<exponentiation expression>::= <primary exp> ⟨<exponentiation op> <primary exp>⟩*

<exponentiation operator>::= [<↑ | ↑ | ↑>]

By a similar argument, exponentiation should require no explication. Arithmetic in GROPE is allowed to be mixed-mode. If all operands are integer, all arithmetic is integer arithmetic; once a type real operand is encountered, type real arithmetic is performed. If an alpha operand is encountered in arithmetic, the character sequence of which it is composed is assumed to represent a number, and it will be duly converted to type REAL; if the representation is not that of a number, a diagnostic will be generated and the program aborted. If a grope operand is encountered in arithmetic, it is assumed to be either a pseudo integer or numeric atom, and its IMAGE will be retrieved, converted to type REAL if necessary, and used; if this is not the case, a diagnostic will be issued and the program aborted. If a logical operand is encountered, its arithmetic value will be zero if "false" or integer one otherwise.

<primary expression>::= [<pattern-directed search exp> | <list-building exp> | <set-building exp> | <vector-building exp> | <procedure invocation> | <extended variable> | <constant> | << <expression> >>]

<pattern-directed search expression>::= [? | ! | ¬] [<atom descr> | <graph descr> | <node descr> | <arc descr>]

The pattern-directed search expression signals a (possible) search of the GROPE universe for some atom, graph, node, or arc which "matches" the indicated pattern. The symbol "?" means that if the indicated item is not found, no operation is to be performed and the "false" value (zero) is to be returned as the value of the primary expression. The symbol "!" means that if the indicated item is not found, one is to be created. The symbol "¬" means that no search is actually performed, but the indicated item is created. In all cases, the found item (if any), or the created item (if any) is to be returned as the value of the expression -- unless the "false" value is returned, as described above. The following descriptor definitions -- and the type of search subsumed -- also apply to the FOR statement, except of course the FOR statement usage may find more than one entity matching the pattern.

<atom descriptor>::= <expression>

The expression, when evaluated, will produce a string or number or array or procedure identifier (constant) which is to be the image of the atom -- whether found or created. If a search is performed, it will only be through the atomset.

<graph descriptor>::= [(¬) [<grope exp> | *] (:: <grope exp>) (: <grope exp>) (¬) ,

The graph descriptor should be somewhat familiar -- it is essentially an un-quoted "graph constant" (from the Data language), with grope expressions allowed for the label, object (if any), and value (if any). The symbol "*" indicates an "I-don't-care" with regards to the label. The symbols "," represent an "I-don't-care" with regards to the attached and related conditions of the graph, while the symbols "¬" mean, as in the Data language, "not". The absence of one of these means, as in the Data language, the graph is to be related/attached. If a search is to take place, it will only be through the gset of the label (unless "don't attach" or "I don't care about the label" is indicated), and/or the graphset (unless "don't relate" is indicated). Note that no graph "elements" may be specified. At least one of these search possibilities must not be contra-indicated, if a search is to be performed.

<node descriptor>::= # (¬) [<grope exp> | *] (:: <grope exp>) (: <grope exp>) (¬) [<grope exp> | *] #

The node descriptor is also like its Data language equivalent. The symbols "*" and "," mean, as before, "I don't care", and "¬"

means "not". Any search to be performed will only be through the nset of the label (unless "I don't care about the label" or "don't attach" is specified), and/or the nodeset of the graph (unless "I don't care about the graph" or "don't relate" is indicated). At least one of these possibilities must not be contra-indicated, if a search is to take place.

<arc descriptor>::= =[<grope exp>/*]#(¬/,/+)<grope exp> (:: <grope exp>) (: <grope exp>)(¬/,/+)#[<grope exp>/*]=

Obviously the arc descriptor bears a resemblance to its Data language cousin, and the symbols "*", "¬", "+", and "," play the expected roles. If a search is to be performed, it will be through the rseto of its frnode (unless "I don't care about the frnode" or "don't relate" is indicated), and/or the rseti of its tonode (unless "I don't care about the tonode" or "don't attach" is specified). At least one of these possibilities must not be contra-indicated if a search is to be performed.

When these descriptors are employed in the FOR statement (cf. Chapter V), their interpretation is essentially the same; however, the FOR statement never implies the creation of one of these structures. If an item is found it is bound to the control variable in the "quantified universe" expression, then the boolean qualifying clause (if any) is evaluated to test for rejection of the item; then if the item is not rejected, the repeat loop is evaluated;

then the search may continue (if the quantifier allows) for the next item, and the process iterates until no more matching items are found (the end of the universe is reached) or the specified number of items has been found.

The test for "match" is as follows: an "I don't care" matches any value for its correspondent in the candidate structure; any fully-specified attribute must be identical to its correspondent's attribute; all attach/relate truth-values must be identical. Thus if the pattern specifies a particular value, the matching structure must have the identical value; if the pattern specifies "relate" then the matching structure must be related; if the pattern says "I don't care about the graph" then the graph of the (node) candidate is of no consequence in the match. The logical value "false" in the object or value position of the pattern indicates that the matching structure must not have any GROPE structure as the corresponding object or value attribute, while the absence of the object or value attribute in the pattern indicates an "I don't care" with regards to that attribute in the matching structure. Finally, note that either one or two system sets may be searched in this process.

ex: ? = N # , L : V , # M = or ?=N#,L:V,#M=

This says: "Find me an arc whose frnode is N, whose label is L, and whose tonode is M; I don't care whether it is attached or related (although it must be one or the other), or what its

object is, but its value must be V. If none exists, do nothing".

<list-building expression>::= ((¬<grope exp>¬) (<quant. univ> / <grope exp> ,)⁻)

The list-building expression is just that -- an expression which builds a list. No search is performed (there is no system set composed of lists), so this necessarily builds a new list each time it is evaluated. Its value may be specified if desired -- delimited by two symbols "¬" as in the Data language. The list's initial elements (if any) and their order in the list is indicated by evaluating the mixture of grope expressions and "quantified universes" encountered before the matching right parenthesis.

<set-building expression>::= { (¬<grope exp>¬) (<quant. univ> / <grope exp> ,)⁻ }

The set-building expression builds a set, of course. Its initial value may be specified, as well as any element -- but not the order of the elements, which is controlled by GROPE. A new set is created each time such an expression is encountered, since no system set of sets exists which could be searched.

<vector-building expression>::= < ⟨<expression> ,⟩⁻ >

A vector-building expression builds a vector of values; the types of these values must be uniform: all GROPE, all INTEGER, etc. The programmer will find that the utility of this feature is somewhat limited because it does not imply "permanent" allocation of storage space: its storage space is deallocated when the procedure

creating it is exited. Therefore a vector expression cannot be the value of a procedure, for instance. However, it may be passed as a parameter to another procedure.

<procedure invocation>::= <procedure name> ((<actual par. list>))

<procedure name>::= [<proc. id> / <grope exp>]

<actual parameter list>::= ⟨<expression> ,⟩⁻

A procedure invocation is a procedure "call". The name of the procedure (typically a procedure identifier) is given, followed by a pair of parentheses enclosing its arguments (if any). However, any GROPE expression which produces a "procedure atom" (an atom whose image is a procedure) may appear instead. This powerful feature allows the programmer to store the "name" of a procedure in a graph or list, say, and call that procedure with an argument list as desired. Thus the construction of interpreters in GROPE is almost unreasonably simple; the user can surely imagine many applications for this feature -- one of the more useful concepts in GROPE.

ex: G(X)(Y,Z)

This evaluates the function G with argument X; the result must be a procedure atom, and the procedure it references (as its image) is then evaluated with two arguments, Y and Z.

<extended variable>::= [<array name> [<subscripts>] | <simple var>]

<array name>::= [<array id> | <grope exp>]

<subscripts>::= ⟨ <arith. exp> , ⟩* [* | <arith. exp> (: [<arith. exp> | *])]

<variable>::= [<array identifier> [⟨ <arith. exp> , ⟩⁻] | <simple variable>]

Array accessing is non-standard in only two respects: (1) an array name may be either an array identifier or any GROPE expression which evaluates to produce an "array atom" -- one created with an array as its image; (2) the subscripts may reference a number of (contiguous) locations -- a portion of a row, an entire row, a portion of a plane, an entire plane, etc. Essentially, the programmer may "delete" subscripts from the right, substituting one symbol "*" to indicate this; or he may specify a range (two arithmetic expressions separated by a colon) as the rightmost un-deleted subscript, or he may specify a range but substitute the symbol "*" for the second arithmetic expression. Subscripts must be(come) integers, by means of truncation if necessary.

ex: A[10,10,10] , A[5,7,3:6] , A[2,6,4:*] , A[1,5,*] ,
A[3,9:10] , A[4,7:*] , A[6,*] , A[5:10] , A[7:*] , A[*]

The first example references one item at location 10, 10, 10; the second example, items 3 to 6 (inclusive) in row 5, 7; the third example, row 2, 6, from the 4th column to the end; the fourth example, the entire row 1, 5; the fifth example, rows 9 and 10 in

in plane 3; the sixth example, everything in plane 4 from row 7 to the end; the seventh example, all of plane 6; the eighth example, planes 5 through 10; the ninth example, everything in the array from plane 7 to the end; and the tenth example, the entire array A.

ex: F(X)[I,J]

This invokes the procedure F with one argument, X; the value returned by F must be an array atom, which is then accessed for its Jth item in row I.

In order to override the indicated precedence (to force addition before multiplication, for instance), the brackets "<<" and ">>" are provided. These are admittedly unusual, but since there is a shortage of characters in the alphabet, and this use of brackets is the most infrequent, it was decided that this usage would have to be the "ugly" one.

<location>::= [[CURCI | CURCO | FIRST | FRNODE | GRAPH | IMAGE | LABEL | LAST | LOCALE | OBJECT | TONODE | VALUE] (<grope exp>) | NTH (<grope exp> , <arith. exp>) <extended variable>]

At last we define "location". A location is a place to which one may assign a value; in the traditional usage, this has been a variable, whether simple or subscripted. GROPE adds the concept of "extended" variable, plus that of "functional replacement". PL/I has a form of the latter which is similar to GROPE's, except not for GROPE's type of data structures. The "change" operations in the last part of Chapter 3 are examples of what can be done using these "functional" locations.

ex: OBJECT(A) <= X

This replaces the object of A with the object X. A must be an arc, node, or graph, and X must be an object (or else the GROPE "false" value, in which case A will no longer have an object -- that is, OBJECT(A) will return the "false" value).

ex: READ VALUE(G)

This "reads" the new value for G "directly" into G's value field. G must be a label -- the only structure type that has a value attribute.

ex: NTH(L,4) <= Z

This replaces the fourth element of the list L with the GROPE structure Z. The other elements of the list, and the length of the list, are unaffected.

The character set of Programming language is the same as that of the Data language, with the addition of the characters

"≤", "≥" and "·", and the deletion of the characters "$", "%", and "¢". GROPE also defines "{" to be equivalent to "(:"; "}" is equivalent to ":)"; "[", to "<:"; and "]", to ":>". "≤" and "≥" may also be deleted from the character set because there are defined equivalents. Along with the deletion of lower-case letters, and the substitution of the decimal point "." for the catenation operator "·", the resultant set comprises 58 characters. Further substitutions and deletions would certainly begin to render the language unreadable, so this is not recommended; a minimum of 60 to 65 characters is preferable.

Provided procedures

The astute reader may have noticed in Chapter V that no means has yet been provided for exiting from procedures or terminating program execution. There is no statement in the language which effects these capabilities. Instead these features are implemented by means of the provided procedures RETURN, EXIT, STOP, and ABORT.

RETURN is a procedure that "never comes back" after being called; rather it causes the procedure calling it to be exited. The argument passed to RETURN is the value returned as the value of the calling procedure. The "execution" of FIN is RETURN(FALSE).

ex: RETURN(X) will exit the procedure containing this call to RETURN, and the value of the procedure will be X.

EXIT(procname,value), where procname is an ALPHA expression, and value is the value to be returned (as in RETURN), will cause the procedure whose symbolic name matches procname to be exited with the value. Procedure calls are of course handled with a run-time stack: EXIT searches this stack for the most recent invocation of the procedure whose name is procname, then pops all the stack to that point and returns value to the procedure calling procname. (If there is no instance of an invocation of procname, the program is terminated abnormally.)

ABORT and STOP each take one ALPHA argument and terminate program execution with a message (the argument); ABORT constitutes abnormal termination, while STOP is normal. Note that the program is also terminated "normally" (with no message) if MAIN procedure "falls through" to the end (FIN) by not invoking RETURN, EXIT, STOP or ABORT.

This completes the definition of GROPE 2. The design was intended to produce a clean, convenient, useful language for the GROPE programmer. Certain features of some other languages are not available in GROPE; certain features of GROPE are not available in other programming languages. In the end, only the user can judge the worth of the language, and then only with some experience in using it.

Appendix A contains the definitions of all GROPE procedures -- many more than have been defined up to this point. Given the

understanding of the GROPE data structures as presented in Chapters I through III, the reader should be able to understand the English definitions presented. Many of the procedures described there, however, need not necessarily be used as procedures in GROPE programs; the GROPE translator automatically generates calls to some of them when certain syntactic constructs (presented herein) are encountered. (For example, consider the "is" and "change" functions.) Even given the size of this document, not all of GROPE has been presented or necessarily adequately explained. The attempt has been to provide the user with a thorough introduction to GROPE, from which point one may experiment with more advanced features of the language and data structure until a complete understanding is attained. Nevertheless, it is felt that even partial comprehension of what has been introduced here will allow the production of "good" GROPE programs employing many of the advanced features of the language. It is hoped that this document will provide the incentive for such experimentation.

CHAPTER VII

Conclusions

There have been more than enough uses of GROPE1 to justify its development into GROPE 2. Slocum [31], Hendrix [13, 15], and Thompson [36] have implemented natural language processing programs in GROPE. The Linguistics Research Center at the University of Texas (Austin) has employed GROPE as a central portion of its automated translation system [20, 32, 33]. In the area of programming language semantics, an ALGOL interpreter (Wilson [40], and Wesson [39]), using H-graphs [25] is being tested in GROPE. Work in program analysis (Griggs [10]), optimal overlay structure for LISP and FORTRAN programs (Greenawalt [9]), and robotics (Hendrix [14]) are further illustrations of the utility of GROPE.

Sussman [34] makes the point that

> A higher level language derives its great power from the fact that it tends to impose structure on the problem solving behavior of the user. Besides providing a library of useful subroutines with a uniform calling sequence, the author of a higher level language imposes his theory of problem solving on the user. By choosing what primitive data structures, and operators he presents, he makes the implementation of some algorithms more difficult than others, thus discouraging some techniques and encouraging others. So, to be "good", a higher level language must not

> only simplify the job of programming, by providing features which package programming structures commonly found in the domain for which the language was designed, it must also do its best to discourage the use of structures which lead to "bad" algorithms.

The approach taken by GROPE 2 is to provide the user with a multiplicity of data structures which are very general in scope and yet efficiently implemented, then to advance a programming language which orients their usage in a convenient, readable manner which enhances their utility. Still, GROPE does not simply provide all known data structures in a new format, but rather advances some new structures and some old structures which, taken together, form a more powerful representation than any yet proposed for operating in the graph processing domain. GROPE embodies some major new ideas about representation and processing of complex data structures. No other language provides so many well-defined "system" sets which may be searched and modified by the user almost at will, while at the same time providing for their automatic maintenance so that the user is <u>never</u> required to perform any primitive operations upon them. No other programming language with so many data structures provides a means of expressing them in a linear format -- or of including them in the programming language, rather than restricting their expression to I/O operations only. GROPE data structures are everywhere dynamic: they can grow, shrink, and be modified irregularly. The presence of a garbage

collector relieves the user of the responsibility (and danger) of performing his own storage allocation/deallocation. The facility for creating atoms which are arrays and/or compiled procedures and distributing them about the data structures makes the creation and utilization of interpreters and simulation programs almost ridiculously simple -- so much so that programming a new interpreter for a new problem would be de rigeur, if it weren't for the fact that the programming language itself is so versatile.

The CASE and SELECT statements might of course be abandoned in favor of IF statements, but to do so could drastically reduce readability and associated reliability. (And in higher level programming languages, readability and reliability is the name of the game.) These statements are particularly useful in interpreters and compilers -- the GROPE 2 translator makes use of these statements in its own code.

The IF/UNLESS and ELSE/REGARDLESS choices are dictated by legibility and utility constraints: the former minimizes the use of NOT, while the latter sometimes allows the formation of one statement out of two without the use of BEGIN-END. The same argument applies to the <repeat loop> THEN <exec. sta> construct -- it is nice to have around.

Of all the programming language statements, certainly the FOR statement is the most novel. When used as a "mapping function",

performing operations upon each successive element of a linear structure, it is most readable:

```
FOR ALL X IN L SUCH THAT F(X) ...
```

When used as a "search function", performing operations upon each element matching some pattern, it is most powerful:

```
FOR EACH ARC =  =N#LAB#*=
        SUCH THAT F(TONODE(ARC)) ...
```

Most GROPE algorithms to date have relied upon the "mapping" facility (extant in GROPE 1 as the functions MAPFT, ANDFT, and ORFT) to process system sets; however, there was no corresponding search operation until GROPE 2 introduced it. Given that this new facility is reasonably legible, it would appear to allow a significant reduction in programming and debugging time -- since the user does not have to write his own search functions, but instead just specifies a pattern for the desired item(s). As Bobrow [2] stated:

> Most programming languages are universal in the sense that any algorithm that can be expressed by a program in one language can also be expressed in any of the other languages. However, the set of unique facilities provided by a language makes some types of programs easier to write in that language than any other. Indeed, the main reason for introducing new features into a programming language is to automate procedures that the user needs and would otherwise have to code explicitly;

> such features reduce the housekeeping details that distract the user from the algorithms in which he is really interested. Therefore, underlying the design of any programming language is a set of assumptions about the types of programs that users of that language will be writing.

In the Future

The reason for the existence of EXIT, RETURN, ABORT, and STOP as "functions" rather than statements is to pave the way for extensions into the realm of "non-deterministic" programming. Back-up features, etc., are easily implemented as "functions" which perform operations on the program's environment. The possibility of adding such operations to the GROPE repertoire of provided procedures is under investigation; their implementation as functions will allow the extension without any changes in the definition of the language or the translator. Balanced against this, however, is the knowledge that GROPE structures are "permanent" -- that is, much more so than in LISP and its descendents. There is the unanswered question: "Who (if anyone) will un-do all the data base changes made since the last fail-set, when backup is invoked"? Until there is some satisfactory solution, implementation will remain pending.

Finally one might ask, "What about a GROPE 2 interpreter"? One answer is that there is not likely to be one: The declarations in particular are oriented toward compilation -- not interpretation;

and the structure of the language gives no indication of any internal representation of a GROPE program -- so that the user might play with his program at run-time. The other answer is that there will not be a GROPE 2 interpreter: rather, the user may write his own if desired. It is more likely that the user will invent his own "mini-language", perfectly suited to the task at hand, then implement the interpreter for it in GROPE -- in a few minutes or hours. Herein lies the true power of GROPE: the wide range of data structures and powerful programming language features promote straightforward solutions to previously difficult problems such as the implementation of ultra-specialized systems.

GROPE combines the speed of compiled code, the utility of LISP, the flexibility of sophisticated graph-based structures, and the over-all system control and portability of FORTRAN in a single self-documenting language to produce a truly powerful and convenient programming tool.

REFERENCES

[1] Baron, R., L. Shapiro, D. P. Friedman, and J. Slocum, "Graph processing using GROPE/360", University of Iowa Computer Science Technical Report (in preparation).

[2] Bobrow, D. G., "New Programming Languages for AI Research", Tutorial Lecture presented at 3IJCAI, Stanford University (1973).

[3] Cashin, P. M., M. R. Mayson, and R. Podmore, "LINKNET -- A structure for computer representation and solution of network problems", Australian Computer Journal 3 (August 1971).

[4] Crespi-reghizzi, S., and R. Morpurgo, "A language for treating graphs", Comm. ACM 13 (1970), 319-323.

[5] Feldman, J. A., and P. D. Rovner, "An ALGOL-based Associative Language", CACM 12, 8, p. 439 (August 1969).

[6] Friedman, D. P., D. Dickson, J. Fraser, and T. W. Pratt, "GRASPE 1.5: a graph processor and its application", Department of Computer Science Report RSI-69, University of Houston, Houston, Texas (1969).

[7] _____, "GRASPE: graph processing a LISP extension", Computation Center Report TNN-84, University of Texas, Austin, Texas (1968).

[8] Friedman, D. P., GROPE: A Graph Processing Language and its Formal Definition, Ph.D. dissertation, University of Texas (June 1973).

[9] Greenawalt, E. M., private communication.

[10] Griggs, Eric R., "Automatic Data Flow Analysis of Computer Programs", unpublished Master's thesis, University of Texas at Austin (May 1973).

[11] Griswold, R. E., J. F. Poage, and I. P. Polonsky, The SNOBOL4 Programming Language, Englewood Cliffs, New Jersey: Prentice-Hall, Inc. (1968).

[12] Hart, R., "HINT: a graph processing language", Institute for Social Science Research Technical Report, Michigan State University, East Lansing, Michigan (1969).

[13] Hendrix, G. G., "Question answering via canonical verbs and semantic models: A model of textual meaning", Technical Report NL12, Department of Computer Science, The University of Texas at Austin (January 1973).

[14] _____, "Modeling simultaneous actions and continuous processes", to appear in Artificial Intelligence Journal.

[15] _____, C. W. Thompson, and J. Slocum, "Language processing via canonical verbs and semantic models", in Proceedings of the Third Annual Joint Conference on Artificial Intelligence (August 1973).

[16] Iverson, K. E., A Programming Language, John Wiley & Sons, New York (1962).

[17] Kiviat, P. J., Introduction to the SIMSCRIPT II Programming Language, RAND Corp., P-3314, Santa Monica, Calif. (February 1966).

[18] Knowlton, K. C., "A programmer's description of L^6, Bell Telephone Laboratories low-level linked list language", Comm. ACM 9, 8 (August 1966).

[19] Lawsen, Harold W., Jr., "PL/I list processing", Comm. ACM 6 (June 1967), 385-367.

[20] Lehmann, W. P., and R. Stachowitz, "German - English translation system", Technical Report of the Linguistics Research Center, The University of Texas at Austin. In preparation (1973).

[21] McCarthy, J., et al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts (1962).

[22] Naur, P. (ed.), "Report on the algorithmic language ALGOL 60", Comm. ACM 6, pp. 1-17 (January 1963).

[23] Newell, Allen (ed.), Information Processing Language-V Manual, Prentice-Hall, Englewood Cliffs, New Jersey (1961).

[24] Pohl, Ira, "A method for finding Hamilton paths and Knight's tours", Comm. ACM 7 (July 1967).

[25] Pratt, T. W., "A hierarchical graph model of the semantics of programs", Proceedings of AFIPS SJCC, 813-825 (1969).

[26] _____, "Semantic modeling by hierarchical graphs", ACM SI GPLAN Symposium on Programming Language Definition, San Francisco, Calif. (August 1969).

[27] _____, "Pair grammars, graph languages, and string-to-graph translations", J. of Comp. Sys. Sci., 5, 560-595 (6 Dec. 1971).

[28] _____, "A formal definition of ALGOL 60 using hierarchical graphs and pair grammars", Report TSN-33, University of Texas Computation Center, 82 pp. (1973).

[29] _____, and D. P. Friedman, "A language extension for graph processing and its formal semantics", Comm. ACM 14, 460-467 (1971).

[30] Ross, Douglas T., "The AED free storage package", Comm. ACM 8, 481-492 (August 1967).

[31] Slocum, J., "Question answering via canonical verbs and semantic models: generating English for the model", Technical Report NL13, Department of Computer Science, The University of Texas at Austin (January 1973).

[32] Stachowitz, Rolf, Voraussetzungen für maschinelle Übersetzung: Probleme, Lösungen, Aussichten, Athenäum Verlag, Frankfurt/M. (1973).

[33] _____, Ein Modell linguistischer Performanz, Athenäum Verlag, Frankfurt/M. (in preparation).

[34] Sussman, G. J., and D. V. McDermott, "Why Conniving is better than Planning", AI Memo No. 255A, MIT Project MAC (April 1972).

[35] Swinehart, D. C., and R. F. Sproull, "SAIL Manual", Stanford AI Projecy Operating Note No. 57.2, Artificial Intelligence Project, Stanford University (1971).

[36] Thompson, C. W., "Question answering via canonical verbs and semantic models: Parsing to canonical verb forms", Technical Report NL13, Department of Computer Science, The University of Texas at Austin (January 1973).

[37] USA Standard X3.23-1968 COBOL.

[38] Weizenbaum, J., "Symmetric list processor", Comm. ACM 6, 9 (September 1963).

[39] Wesson, Robert B., "A pair grammar based string to graph translator writing system", Master's thesis in preparation, The University of Texas at Austin.

[40] Wilson, James P., "Graphical representation of semantic structure", unpublished Master's thesis, The University of Texas at Austin (August 1972).

[41] Wirth, N., "The programming language, PASCAL", Acta Informtica 1, 35-63 (1971).

[42] Woods, W. A., "Transition network grammars for natural language analysis", Comm. ACM 13, 10, 591-606 (October 1970).

[43] Wulf, W. A., D. B. Russell, and A. N. Haberman, "BLISS: A Language for Systems Programming", CACM 14, 12, pp 780-790 (December 1971).

APPENDIX A

ALL THE LITTLE GROPE FUNCTIONS

(OTHERWISE KNOWN AS GROPE 1.5)

NOTE: THE ASTERISKS MARK THE PROCEDURES NOT AVAILABLE IN THE PROGRAMMING LANGUAGE VIA SPECIAL SYNTACTIC CONSTRUCTS

* ABORT (MESSAGE) NEVER RETURNS. THE PROGRAM IS TERMINATED ABNORMALLY, AND THE ALPHA STRING MESSAGE IS PRINTED.

* ADD (VALUE,SET) := SET. IF THE VALUE IS NOT ALREADY A MEMBER OF THE SET, THEN IT IS ADDED TO THE SET AS A NEW ELEMENT. OTHERWISE, NO ACTION IS PERFORMED.

* ASCEND (READER) := READER. THE READER'S STACK IS POPPED ONE LEVEL - THAT IS, THE READER ASCENDS OUT OF THE MOST RECENT SUB-STRUCTURE TO WHICH IT DESCENDED.

* ATTACH (ARC-NODE-OR-GRAPH) := ARC-NODE-OR-GRAPH. THE ARC, NODE, OR GRAPH PASSED AS AN ARGUMENT IS ATTACHED TO THE APPROPRIATE (RSETI, NSET, OR GSET) ATTSET -- THAT IS, IT BECOMES AN ELEMENT OF THE PROPER SYSTEM SET, IF IT WAS NOT SO ALREADY. THE ADDITION IS MADE BY STACKING OR QUEUING, ACCORDING TO THE POSITION OF THE GLOBAL ALPHA "QSMODE" SWITCH.

* CARTES (SET1,SET2) := A NEW SET WHICH IS THE CARTESIAN PRODUCT OF THE TWO SETS PASSED AS ARGUMENTS. EACH ELEMENT OF THE NEW SET IS A LIST OF TWO VALUES, THE FIRST OF WHICH IS AN ELEMENT OF SET1, AND THE SECOND OF WHICH IS AN ELEMENT OF SET2.

CHACURI(NODE,ARC) := NODE. THE ARC BECOMES THE CURRENT-ARC-INCOMING TO THE NODE. CHATON IS INVOKED IF NECESSARY.

CHACURO(NODE,ARC) := NODE. THE ARC BECOMES THE CURRENT-ARC-OUTGOING FROM THE NODE. CHAFRN IS INVOKED IF NECESSARY.

CHAFIR (LIST,VALUE) := LIST. THE VALUE IS SUBSTITUTED FOR THE FIRST ITEM IN THE LIST, WHICH MUST NOT BE EMPTY.

CHAFRN (ARC,NODE) := ARC. THE NODE BECOMES THE FRNODE OF THE ARC. (CONCEPTUALLY, THE TAIL OF THE ARC IS SWUNG AROUND SO THAT THE ARC NOW ORIGINATES FROM THE NEW NODE.) IF THE ARC WAS RELATED (IN THE RSETO), IT WILL ALSO BE IN THE NEW RSETO.

CHAGR (NODE,GRAPH) := NODE. THE NODE IS REMOVED FROM ITS OLD GRAPH AND DEPOSITED IN THE NEW (ARGUMENT) GRAPH. IF THE NODE WAS RELATED (IN THE NODESET OF ITS OLD GRAPH), IT WILL BE IN THE NODESET OF ITS NEW GRAPH.

CHALAB (ARC-NODE-OR-GRAPH,LABEL) := ARC-NODE-OR-GRAPH. THE ARC, NODE, OR GRAPH ACQUIRES THE NEW LABEL. IN THE CASE OF NODES AND GRAPHS, THIS MIGHT ENTAIL REMOVING THE NODE OR GRAPH FROM THE NSET OR GSET OF ITS OLD LABEL AND ADDING IT TO THAT OF ITS NEW LABEL (BUT ONLY IF THE NODE OR GRAPH WAS ALREADY ATTACHED).

CHALAS (LIST,VALUE) := LIST. THE VALUE IS SUBSTITUTED FOR THE LAST ITEM IN THE LIST (WHICH MUST NOT BE EMPTY).

CHALEX (CHARACTER,INTEGER) := CHARACTER. THE "LEXICAL CLASS" OF THE (STRING) CHARACTER BECOMES THE CLASS DESIGNATED BY THE SMALL POSITIVE INTEGER. (SEE THE LEX FUNCTION.)

CHALOC (READER1,READER2) := READER1. THE LOCALE (TEXT AND TOKEN) OF READER1 IS ALTERED TO BECOME IDENTICAL TO THAT OF READER2.

CHANTH (LIST,VALUE,INTEGER) := LIST. THE VALUE IS SUBSTITUTED FOR THE NTH ITEM IN THE NON-EMPTY LIST, AS INDICATED BY THE SMALL POSITIVE INTEGER. (CHANTH(LIST,VALUE,1) IS EQUIVALENT TO CHAFIR(LIST,VALUE).)

CHATON (ARC,NODE) := ARC. THE NODE BECOMES THE TONODE OF THE ARC. (CONCEPTUALLY, THE HEAD OF THE ARC IS SWUNG AROUND TO POINT TO THE NEW NODE.) IF NECESSARY, THE ARC IS DETACHED FROM ITS OLD TONODE AND ATTACHED TO ITS NEW TONODE (BUT ONLY IF IT WAS ALREADY IN THE RSET1 OF ITS OLD TONODE).

CHAVAL (LABEL,VALUE) := LABEL. THE LABEL ACQUIRES THE NEW VALUE, LOSING ITS OLD ONE IF ANY. IF THE SECOND ARGUMENT IS THE *FALSE* GROPE VALUE (BINARY ZERO), THE LABEL WILL HAVE THE *FALSE* VALUE -- THAT IS, NO VALUE AT ALL.

* CHEND (READER) := READER. THE LIST BEING READ BY THE READER IS ALTERED SO THAT THE ELEMENT INDICATED BY THE READER BECOMES THE LAST IN THE LIST. THE ORDER OF ELEMENTS IS NOT CHANGED (THAT IS, THE INDICATED ITEM IS NOT MOVED), BUT RATHER THE INTERNAL POINTER TO THE *FIRST* LIST ELEMENT IS CHANGED TO POINT TO THE ELEMENT FOLLOWING THE INDICATED ITEM.

CHOBJ (ARC-NODE-OR-GRAPH,OBJECT) := ARC-NODE-OR-GRAPH. THE ARC, NODE, OR GRAPH ACQUIRES A NEW OBJECT, LOSING ITS OLD ONE. IF THE SECOND ARGUMENT (OBJECT) IS THE *FALSE* GROPE VALUE (BINARY ZERO), THEN THE ARC, NODE, OR GRAPH WILL HAVE NO OBJECT.

CHTEXT (READER,TEXT) := READER. THE READER, WITHOUT ASCENDING OR DESCENDING, IS CAUSED TO HAVE A NEW TEXT (LINEAR STRUCTURE NODE, OR GRAPH). IT IS RESTARTED -- SO THAT IT HAS NO CURRENT TOKEN UNTIL THE READER HAS BEEN MOVED.

* COMDIF (SET1,SET2) := A NEW SET COMPOSED OF THE ELEMENTS OF SET1 THAT ARE NOT ALSO IN SET2. (SET1 - INTERSECTION(SET1,SET2))

* CONCAT (LIST1,LIST2) := LIST1. LIST2 BECOMES EMPTY, AND ITS ELEMENTS ARE APPENDED TO THE END OF LIST1, WHICH THUS GROWS BY THE NUMBER OF ITEMS FORMERLY IN LIST2.

* CONNECT(READER1,READER2) := READER1. READER1 IN THIS MANNER ACQUIRES A NEW HISTORY (STACK), AND IN THE PROCESS LOSES ITS OLD ONE. AFTER THIS OPERATION, ASCEND(READER1) WILL PRODUCE READAR2. (AMONG OTHER THINGS, THIS ALLOWS FOR THE EASY CREATION OF A CONCORDANCE.)

* CURCI (NODE) := THE CURRENT-ARC-INCOMING TO THE NODE, IF ANY. IF THERE IS NONE (PROBABLY BECAUSE NO READER HAS EXECUTED A TI FROM THIS NODE), THEN THE *FALSE* VALUE IS RETURNED.

* CURCO (NODE) := THE CURRENT-ARC-OUTGOING FROM THE NODE, IF ANY.

* DELETE (READER) := READER. THE ITEM IN THE LIST INDICATED BY THE READER IS ELIMINATED FROM THE LIST -- THUS SHRINKING IT BY ONE ITEM. THE READER IS MOVED BACK ONE STEP (WITH TI) BEFORE THE OPERATION, SO AS NOT TO LEAVE IT DANGLING.

* DESCEND(READER,TEXT) := A NEW READER FOR THE TEXT, HAVING PUSHED THE ARGUMENT READER DOWN ON ITS STACK. (THAT IS, THE CURRENT STATUS OF THE ARGUMENT READER IS REMEMBERED.) OPERATIONALLY, THIS IS EQUIVALENT TO CONNECT(READER(TEXT),READER).

* DETACH (ARC-NODE-OR-GRAPH) := ARC-NODE-OR-GRAPH. THE ARC, NODE, OR GRAPH ARGUMENT IS REMOVED FROM THE APPROPRIATE ATTSET (RSET1, NSET, OR GSET).

* DISCON (READER) := READER. THE READER LOSES ITS HISTORY (STACK). HENCEFORTH IT MAY NOT ASCEND, SINCE IT IS NO LONGER *DEEP*.

* ENDFILE(FILENAME) := FILENAME. AN *END-OF-FILE* IS WRITTEN ON THE FILE INDICATED BY THE ALPHA STRING FILENAME.

* EXIT (PROCNAME,VALUE) := VALUE, BUT EXIT DOES NOT SIMPLY RETURN TO THE PROCEDURE CALLING IT. INSTEAD, IT CAUSES THE PROCEDURE INDICATED BY THE ALPHA STRING PROCNAME TO BE EXITED WITH THE INDICATED VALUE AS ITS FUNCTIONAL VALUE. THIS MAY POP THE CONTROL STACK BY AN ARBITRARY AMOUNT, BUT ONLY UNTIL THE LAST (MOST RECENT) INVOCATION OF PROCNAME IS FOUND.

* FIRST (LINEAR-STRUCTURE) := THE FIRST ELEMENT IN THE LINEAR STRUCTURE (LIST, SET, OR SYSTEM SET), ELSE THE *FALSE* VALUE IF THE STRUCTURE IS EMPTY.

* FRNODE(ARC) := THE NODE FROM WHICH THE ARC EMANATES.

* GRAPH (NODE) := THE GRAPH ON WHICH THE NODE RESIDES.

* GSET (LABEL) := THE SYSTEM SET COMPRISING ALL OF THE ATTACHED GRAPHS SHARING THE IDENTICAL LABEL.

IMAGE (ATOM) := THE CHARACTER STRING, NUMBER, ETC., WHICH IS THE *PRINT* IMAGE OF THE ATOM.

* INSERT (READER,VALUE) := READER. THE VALUE IS INSERTED INTO THE LIST INDICATED BY THE READER, JUST TO THE RIGHT OF THE READER'S POSITION. THUS THE LIST GROWS BY ONE ITEM, AND THE NEXT TO(READER) WILL ADVANCE THE READER PTO THE ITEM INSERTED.

* INTERSECT (SET1,SET2) := A NEW SET COMPOSED OF THE ELEMENTS IN SET1 WHICH ARE ALSO IN SET2.

ISALPHA(ATOM) := ATOM IF THE IMAGE(ATOM) IS A CHARACTER STRING.

ISARC (ARG) := ARG IF ARG IS AN ARC; OTHERWISE ISARC := FALSE.

ISARRAY(ATOM) := ATOM IF THE IMAGE(ATOM) IS AN ARRAY (OF ANY TYPE).

ISATBEG(READER) := READER IF THE READER IS POINTING AT THE FIRST ITEM

IN THE LINEAR STRUCTURE THAT IS ITS TEXT.

ISATEND[READER] := READER IF THE READER IS POINTING AT THE LAST ITEM IN THE LINEAR STRUCTURE THAT IS ITS TEXT.

ISATOM [ARG] := ARG IF ARG IS AN ATOM; OTHERWISE ISATOM := FALSE.

ISATSET[ARG] := ARG IF ARG IS THE ATOMSET; OTHERWISE FALSE.

ISATT [ARC-NODE-OR-GRAPH] := ARC-NODE-OR-GRAPH IF IT IS ATTACHED (IF IT IS IN THE APPROPRIATE ATTSET); OTHERWISE FALSE.

ISATTST[ARG] := ARG IF ARG IS AN ATTSET (RSETI, NSET, OR GSET).

ISCURCI[ARC] := ARC IF ARC IS THE CURCI OF ITS TONODE.

ISCURCO[ARC] := ARC IF ARC IS THE CURCO OF ITS FRNODE.

ISDEEP [READER] := READER IF THE READER HAS A NON-EMPTY STACK (THE READER MAY BE ASCEND); OTHERWISE FALSE.

ISDISJ [SET1,SET2] := SET1 IF THE TWO SETS ARE DISJOINT (HAVE NO ELEMENTS IN COMMON); OTHERWISE FALSE

ISGRAPH[ARG] := ARG IF ARG IS A GRAPH; OTHERWISE ISGRAPH := FALSE.

ISGROPE[ATOM] := ATOM IF THE IMAGE[ATOM] IS ITSELF A GROPE STRUCTURE -- THAT IS, AN ARRAY OF GROPE STRUCTURES.

ISGRSET[ARG] := ARG IF ARG IS THE GRAPHSET; OTHERWISE FALSE.

ISGSET [ARG] := ARG IF ARG IS A GSET (OF SOME LABEL).

ISINT [ATOM] := ATOM IF THE IMAGE[ATOM] IS AN INTEGER.

ISLAB [ARG] := ARG IF ARG IS A LABEL (AN ATOM, ARC, NODE, GRAPH, LIST, OR SET).

ISLIST [ARG] := ARG IF ARG IS A LIST; OTHERWISE ISLIST := FALSE.

ISLNSR [ARG] := ARG IF ARG IS A LINEAR STRUCTURE; OTHERWISE FALSE.

ISNODE [ARG] := ARG IF ARG IS A NODE; OTHERWISE ISNODE := FALSE.

ISNODES[ARG] := ARG IF ARG IS A NODESET (OF SOME GRAPH).

ISNSET [ARG] := ARG IF ARG IS AN NSET (OF SOME LABEL).

ISNUM [ATOM] := ATOM IF THE IMAGE[ATOM] IS NUMERIC (INTEGER OR REAL), WHETHER AN ARRAY OR A SCALAR.

ISOBJ [ARG] := ARG IF ARG IS AN OBJECT (A LABEL, READER, OR PSEUDO INTEGER); OTHERWISE ISOBJ := FALSE.

ISPROC [ATOM] := ATOM IF ATOM IS A PROCEDURE-ATOM (ONE WHOSE "IMAGE" IS A PROCEDURE); OTHERWISE ISPROC := FALSE.

ISPSEUD[ARG] := ARG IF ARG IS A PSEUDO-INTEGER; OTHERWISE FALSE.

ISPSUB [SET1,SET2] := SET1 IF SET1 IS A PROPER SUBSET OF SET2 (A SUBSET, BUT NOT EQUAL); OTHERWISE FALSE.

ISRDR [ARG] := ARG IF ARG IS A READER; OTHERWISE ISRDR := FALSE.

ISREAL [ATOM] := ATOM IF THE IMAGE[ATOM] IS A REAL NUMBERP.

ISREL [ATOM-ARC-NODE-OR-GRAPH] := ATOM-ARC-NODE-OR-GRAPH IF IT IS RELATED (IT IS IN THE APPROPRIATE RELSET); OTHERWISE FALSE.

ISRELST[ARG] := ARG IF ARG IS A RELSET (THE ATOMSET, THE GRAPHSET, A NODESET, OR AN RSETO); OTHERWISE ISRELST := FALSE.

ISRSETI[ARG] := ARG IF ARG IS AN RSETI (OF SOME NODE).

ISRSETO[ARG] := ARG IF ARG IS AN RSETO (OF SOME NODE).

ISSET [ARG] := ARG IF ARG IS A (USER, NOT SYSTEM) SET.

ISSUB [SET1,SET2] := SET1 IF SET1 IS A SUBSET OF SET2; OTHERWISE FALSE.

ISSYSET[ARG] := ARG IF ARG IS A SYSTEM SET (AN ATTSET OR A RELSET).

ISTEXT [ARG] := ARG IF ARG IS A LINEAR STRUCTURE, A NODE, OR A GRAPH (ARG MAY BE SEARCHED WITH A READER).

ISVAL [ARG] := ARG IF ARG IS A GROPE STRUCTURE; IF ARG IS THE "FALSE" VALUE, OR A STRING, OR NUMBER, ETC., ISVAL := FALSE.

* LABEL [ARC-NODE-OR-GRAPH] := THE LABEL ASSIGNED TO THE ARC, NODE, OR GRAPH. (EVERY ARC, NODE, AND GRAPH HAS A LABEL.)

* LAST [LINEAR-STRUCTURE] := THE LAST ITEM IN THE LINEAR STRUCTURE.

* LENGTH [LINEAR-STRUCTURE] := THE (INTEGER) NUMBER OF ITEMS IN THE LINEAR STRUCTURE.

* LEX [CHARACTER] := THE (INTEGER) "LEXICAL CLASS" OF THE ALPHA STRING CHARACTER. THIS IS THE CODE WHICH THE GROPE I/O ROUTINES EMPLOY TO DETERMINE HOW TO TREAT EACH CHARACTER -- FOR EXAMPLE, THE LEFT PARENTHESIS HAS DEFAULT CLASS 19; ANY CHARACTER IN CLASS 19 IS BY DEFINITION A LEFT-SIDE LIST DELIMITER. WHEN THE GROPE INPUT ROUTINE FINDS A CHARACTER OF CLASS 19, IT EXPECTS A LIST TO FOLLOW, TERMINATED BY SOME CHARACTER FROM CLASS 20. CONVERSELY, WHEN THE GROPE OUTPUT ROUTINE NEEDS TO PRINT A LIST, IT CHOOSES ONE OF THE CHARACTERS IN CLASS 19 TO PRINT AS THE LEFT DELIMITER, AND ONE OF THOSE IN CLASS 20 AS THE RIGHT DELIMITER, ETC. THUS THE USER MAY EXAMINE THE CURRENT LEXICAL CLASS OF A CHARACTER BY MEANS OF THE LEX FUNCTION, AND HE MAY CHANGE IT WITH THE CHALEX ROUTINE MENTIONED EARLIER. BELOW IS A LIST OF THE CHARACTERS AND THEIR DEFAULT CLASSES. IN ORDER TO FULLY UNDERSTAND THE IMPLICATIONS OF THIS TABLE, THE READER MUST BE

FAMILIAR WITH THE DATA LANGUAGE.

CLASS	MEMBERS
0	<END-OF-LINE>
1	* A C F G H I J L N O P Q R S T U V W X Y Z
2	B
3	K
4	D
5	E
6	M
7	0 1 2 3 4 5 6 7 8 9
8	-
9	/
10	'
11	+
12	%
13	:
14	,
15	.
16	" $ & ? ! ↑ ¢
17	{
18	}
19	(
20	)
21	[
22	]
23	<
24	>
25	#
26	=
27	¬
28	;
29	<BLANK>

* LIMAGE (ATOM) := THE LENGTH OF THE IMAGE(ATOM), IN CHARACTERS OR WORDS AS APPROPRIATE.

LIST (ARG) := A NEW EMPTY LIST. IF ARG IS A GROPE VALUE, THEN IT WILL BE THE VALUE OF THE NEW LIST; OTHERWISE, THE LIST HAS NO VALUE.

* LOCALE (READER) := READER. THIS IDENTITY FUNCTION ACTUALLY EXISTS ONLY FOR THE SAKE OF COMPLETENESS -- LOCALE IS A <LOCATION> FUNCTION, AND AS SUCH IT HAS A CH- COUNTERPART: CHALOC.

* MACURI (ARC) := ARC. THE ARC BECOMES THE CURCI OF ITS TONODE.

* MACURO (ARC) := ARC. THE ARC BECOMES THE CURCO OF ITS FRNODE.

MAYATT (VALUE) := VALUE IF THE GROPE VALUE (STRUCTURE) MAY BE ATTACHED -- THAT IS, IF IT IS A GRAPH, NODE, OR ARC.

MAYREL (VALUE) := VALUE IF THE GROPE VALUE (STRUCTURE) MAY BE RELATED -- THAT IS, IF IT IS AN ATOM, GRAPH, NODE, OR ARC.

MEMBER (VALUE,SET) := VALUE IF THE VALUE IS IN THE SET; ELSE FALSE.

* MERGE (READER,LIST) := READER. THE ARGUMENT LIST IS MERGED INTO THE LIST INDICATED BY THE READER, TO THE RIGHT OF THE READER'S POSITION. THE ARGUMENT LIST BECOMES EMPTY, AND THE LENGTH(TEXT(READER)) GROWS BY THE FORMER LENGTH(LIST). THUS THE NEXT TO(READER) WILL CAUSE THE READER TO ADVANCE TO THE ITEM THAT WAS FORMERLY THE FIRST ITEM IN THE ARGUMENT LIST.

* MOVETO (READER,VALUE) := READER. CONCEPTUALLY, THE READER IS ADVANCED (VIA TO) UNTIL THE TOKEN(READER) = VALUE. IN THE CASE WHERE THE TEXT(READER) IS A LIST, THIS IS PRECISELY THE CASE; HOWEVER, WHERE THE TEXT(READER) IS ANY OTHER STRUCTURE THAN A LIST, MOVETO IS MUCH FASTER THAN CALLING TO ITERATIVELY -- THAT IS, THE MOVETO ALGORITHM IS NOT ITERATIVE FOR STRUCTURES OTHER THAN LISTS. EFFECTIVELY, MOVETO WILL MOVE THE READER DIRECTLY TO THE ARGUMENT VALUE WITHIN THE TEXT(READER).

* NODESET(GRAPH) := THE NODESET OF THE GRAPH. THE NODESET OF A GRAPH IS COMPOSED OF THOSE NODES ON THE GRAPH WHICH ARE RELATED.

* NSET (LABEL) := THE NSET OF THE LABEL. THE NSET OF A LABEL IS COMPOSED OF THOSE NODES SHARING THE IDENTICAL LABEL AND WHICH ARE ATTACHED.

* NTH (LINEAR-STRUCTURE,INTEGER) := THE INTEGER-TH ITEM IN THE LINEAR STRUCTURE. NTH(LSR,1) = FIRST(LSR).

* OBJECT (ARC-NODE-OR-GRAPH) := THE OBJECT OF THE ARC, NODE, OR GRAPH, IF ANY; IF THERE IS NONE, OBJECT := FALSE.

* POP (LIST) := FIRST(LIST), AFTER WHICH THE FIRST ITEM IN THE LIST IS DELETED FROM THE LIST. (THAT IS, POP REMOVES THE FIRST ITEM FROM THE LIST, AND RETURNS THAT ITEM AS ITS FUNCTIONAL VALUE.)

* PSEUDO (INTEGER) := THE PSEUDO INTEGER WHOSE IMAGE IS THE INTEGER.

* PULL (LIST) := LAST(LIST), AFTER WHICH THE LAST ITEM IN THE LIST IS DELETED FROM THE LIST. (THAT IS, PULL REMOVES THE LAST ITEM FROM THE LIST, AND RETURNS THAT ITEM AS ITS FUNCTIONAL VALUE.)

* QUEUE (VALUE,LIST) := LIST. THE VALUE IS ADDED TO THE LIST BY INSERTING IT TO THE RIGHT OF ALL OTHER LIST ITEMS; THUS THE LENGTH(LIST) IS INCREMENTED BY ONE AND THE LAST(LIST) BECOMES VALUE.

* READER (READABLE-STRUCTURE) := A NEW READER FOR THE LINEAR-STRUCTURE, GRAPH, OR NODE PASSED AS THE ARGUMENT. THE READER IS UNMOVED AND NOT DEEP -- THAT IS, TOKEN(READER) AND ISDEEP(READER) ARE BOTH FALSE WITH RESPECT TO THE NEW READER.

* RELATE (ATOM-ARC-NODE-OR-GRAPH) := ATOM-ARC-NODE-OR-GRAPH. THE ATOM, ARC, NODE, OR GRAPH PASSED AS THE ARGUMENT IS RELATED -- INSERTED INTO THE APPROPRIATE (ATOMSET, RSET0, NODESET, OR GRAPHSET) RELSET BY STACKING OR QUEUING, ACCORDING TO THE POSITION OF THE GLOBAL ALPHA *QSMODE* SWITCH -- IF IT WAS NOT SO ALREADY.

* REMOVE (VALUE,SET) := VALUE. THE VALUE IS DELETED FROM THE SET; THUS THE SET SHRINKS BY ONE ITEM, PROVIDED THE VALUE WAS INDEED IN THE SET. (IF NOT, NO ACTION IS PERFORMED.)

* RESTART(READER) := READER. THE READER IS RESTARTED: THE TEXT IS UNCHANGED, BUT THE TOKEN(READER) BECOMES FALSE. HENCE THE NEXT TO(READER) WILL ADVANCE THE READER TO THE FIRST ITEM IN ITS TEXT.

* RETURN (ARG) := VALUE, BUT RETURN DOES NOT SIMPLY RETURN CONTROL TO THE PROCEDURE CALLING IT, BUT INSTEAD RETURNS CONTROL TO THE PROCEDURE WHICH CALLED THE PROCEDURE CALLING RETURN; THUS THE ARGUMENT WILL BE THE FUNCTIONAL VALUE OF THE PROCEDURE CALLING RETURN. (THE RECURSION STACK IS POPPED ONE LEVEL.)

  REVARC (ARC) := ARC. THE ARC IS REVERSED -- THE TONODE BECOMES THE FRNODE, AND THE FRNODE, THE TONODE. IF THE ARC WAS ATTACHED BEFORE REVARC, IT WILL BE AFTERWARDS; IF IT WAS RELATED BEFORE REVARC, IT WILL BE AFTERWARDS. OTHERWISE IT WILL NOT BE ATTACHED OR RELATED, RESPECTIVELY.

* REWIND (FILENAME) := FILENAME. THE FILE NAMED BY THE ALPHA STRING FILENAME IS REWOUND.

* RSETI (NODE) := THE SET OF ARCS INCOMING TO THE NODE THAT ARE ATTACHED.

* RSETO (NODE) := THE SET OF ARCS OUTGOING FROM THE NODE THAT ARE RELATED.

  SET (ARG) := A NEW EMPTY SET. IF ARG IS A GROPE VALUE, THEN ARG WILL BE THE VALUE OF THE NEW SET; OTHERWISE THE NEW SET WILL HAVE NO VALUE.

* SPLIT (READER) := READER. THE ELEMENTS AT AND TO THE RIGHT OF THE READER'S POSITION IN THE TEXT(READER) (WHICH MUST BE A LIST) ARE DELETED FROM THAT TEXT (LIST) AND FORM A NEW LIST. THE READER'S TEXT WILL BE CHANGED TO BE THE NEWLY CREATED LIST, AND THE READER WILL BE POSITIONED ON THE FIRST ELEMENT OF THE NEW LIST -- THUS THE TOKEN(READER) WILL BE UNCHANGED BY THIS OPERATION. IN EFFECT, THE ORIGINAL TEXT(READER), A LIST, IS SPLIT INTO TWO LISTS AT THE LOCATION MARKED BY THE READER.

* STACK (VALUE,LIST) := LIST. THE VALUE IS ADDED TO THE LIST BY INSERTING IT TO THE LEFT OF ALL OTHER LIST ITEMS; THUS THE LENGTH(LIST) IS INCREMENTED BY ONE AND THE FIRST(LIST) BECOMES VALUE.

* STOP (MESSAGE) NEVER RETURNS. THE PROGRAM IS TERMINATED NORMALLY, AND THE ALPHA STRING MESSAGE IS PRINTED.

* SUBST (READER,VALUE) := READER. THE VALUE IS SUBSTITUTED FOR THE ITEM IN THE LIST MARKED BY THE LOCATION OF THE READER. THUS LENGTH(LIST) IS UNCHANGED, WHILE THE TOKEN(READER) DOES CHANGE -- TO VALUE. THE READER IS NOT MOVED.

* SYMDIF (SET1,SET2) := A NEW SET WHICH IS THE (SYMMETRIC) DIFFERENCE OF SET1 AND SET2 (UNION(SET1,SET2) - INTERSECTION(SET1,SET2)).

* TEXT (READER) := THE READABLE STRUCTURE PASSED TO THE FUNCTION READER WHICH RESULTED IN THE CREATION OF THE READER -- INCLUDING VIA THE DESCEND FUNCTION. (THE TEXT OF A READER IS THE OVERALL STRUCTURE BEING READ BY THE READER, AS OPPOSED TO THE PARTICULAR ITEM WITHIN THE TEXT WHICH THE READER IS POINTING TO -- THE TOKEN.)

* TI (READER) := THE TOKEN(READER) AFTER THE READER HAS MOVED ONE STEP TO THE LEFT IN THE LINEAR STRUCTURE TEXT, OR HAS CROSSED AN ARC INCOMING TO THE NODE WHICH WAS THE FORMER TOKEN IN THE CASE OF GRAPH AND NODE READERS.

* TO (READER) := THE TOKEN(READER) AFTER THE READER HAS MOVED ONE STEP TO THE RIGHT IN THE LINEAR STRUCTURE TEXT, OR HAS CROSSED AN ARC OUTGOING FROM THE NODE WHICH WAS THE FORMER TOKEN IN THE CASE OF GRAPH AND NODE READERS.

* TOKEN (READER) := THE ITEM WITHIN THE TEXT(READER) WHICH THE READER IS CURRENTLY POINTING AT; BUT IF THE READER IS UNMOVED, TOKEN := FALSE. THE READER IS NOT MOVED.

* UNION (SET1,SET2) := A NEW SET COMPOSED OF ALL OF SET1 PLUS ANY ITEMS IN SET2 WHICH WERE NOT IN SET1.

* UNRELATE (ATOM-ARC-NODE-OR-GRAPH) := ATOM-ARC-NODE-OR-GRAPH. THE ATOM, ARC, NODE, OR GRAPH ARGUMENT IS REMOVED FROM ITS RELSET IF IT WAS RELATED.

* VALUE (LABEL) := THE VALUE OF THE ATOM, ARC, NODE, GRAPH, LIST, OR SET ARGUMENT (LABEL), UNLESS THE LABEL HAS NO VALUE IN WHICH CASE VALUE := FALSE.

APPENDIX B

THE SPECIAL GLOBAL VARIABLES

(IMPLEMENTED AS LABELLED COMMON BLOCKS IN FORTRAN)

NOTE: EXCEPT FOR THE ATOMSET AND GRAPHSET, THE USER MAY ASSIGN A NEW VALUE TO ANY OF THESE VARIABLES, AT ANY TIME

GLOBAL ATOMSET; THE ATOMSET
THIS VARIABLE NAMES THE ATOMSET -- THE HASH-CODED SYSTEM SET OF RELATED ATOMS. THE USER MAY NOT ASSIGN A VALUE TO THIS VARIABLE.

GLOBAL CURARC; INITIALLY FALSE
WHEN ANY ARC IS CROSSED BY ANY NODE OR GRAPH READER, GROPE WILL ASSIGN THAT ARC TO CURARC.

GLOBAL ALPHA ECFILE[7]; INITIALLY ""
ECFILE NAMES THE FILE ON WHICH ECHO-PRINTING OF ANY LINE(S) READ BY A <READ STATEMENT> IS TO TAKE PLACE. THE EMPTY STRING INDICATES NO ECHO-PRINT.

GLOBAL REAL FREEPCT, FULLPCT; INITIALLY 1.0
THESE VARIABLES INDICATE THE RESPECTIVE FRACTIONS OF IN-CORE FREE (POINTER-CELL) SPACE AND FULL (ATOM-IMAGE) SPACE THAT WERE AVAILABLE FOR USE IMMEDIATELY AFTER THE LAST GARBAGE COLLECTION.

GLOBAL GRAPHSET; THE GRAPHSET
THIS VARIABLE NAMES THE GRAPHSET -- THE SYSTEM SET OF RELATED GRAPHS. THE USER MAY NOT ASSIGN A VALUE TO THIS VARIABLE.

GLOBAL INTEGER IMARGIN; INITIALLY 0
IMARGIN INDICATES THE NUMBER OF COLUMNS ON THE LEFT OF AN INPUT LINE THAT A <READ STATEMENT> WILL IGNORE.

GLOBAL ALPHA INFILE[7]; INITIALLY "INPUT"
INFILE NAMES THE FILE FROM WHICH THE <INPUT STATEMENT> WILL READ.

GLOBAL INTEGER ISPACE; INITIALLY 0
ISPACE, WHICH IS RESET TO ZERO AFTER EACH <READ STATEMENT>, INDICATES THE NUMBER OF COLUMNS TO BE SKIPPED BEFORE THE READ STATEMENT BEGINS SCANNING THE CURRENT INPUT LINE. A NEGATIVE VALUE WILL CAUSE BACKSPACING, BUT AT MOST TO THE BEGINNING OF THE CURRENT LINE (OVER-RIDING THE MARGIN). SPACING BEYOND THE RIGHT-HAND END OF THE CURRENT LINE WILL CAUSE A NEW LINE TO BE READ, AND SCANNING WILL START AT THE IMARGIN.

GLOBAL INTEGER ITAB; INITIALLY 0
ITAB, WHICH IS SET TO IMARGIN+1 AS A NEW LINE IS READ, INDICATES THE EXACT POSITION (COLUMN) OF THE CURRENT INPUT LINE WHERE SCANNING WILL RESUME WHEN THE NEXT <READ STATEMENT> IS EXECUTED -- SUBJECT TO MODIFICATION BY ISPACE. IF, WHEN A READ STATEMENT IS EXECUTED, ITAB+ISPACE IS GREATER THAN THE LINE LENGTH OF THE CURRENT RDFILE (AS SPECIFIED ON THE MAIN PROCEDURE CARD), A NEW LINE IS READ.

GLOBAL INTEGER MAXERR; INITIALLY 10
MAXERR IS DECREMENTED BY ONE EVERY TIME GROPE DETECTS A RECOVERABLE RUN-TIME ERROR CONDITION. WHEN MAXERR AS DECREMENTED BECOMES NEGATIVE, ABORT IS CALLED WITH AN APPROPRIATE DIAGNOSTIC. THUS MAXERR AT ANY TIME REFLECTS THE

NUMBER OF ALLOWABLE ERRORS REMAINING.

GLOBAL ALPHA MSFILE[7]; INITIALLY *OUTPUT*
MSFILE NAMES THE FILE ON WHICH ERROR MESSAGES ARE TO BE PRINTED. THE EMPTY STRING INDICATES NO PRINTING OF SUCH MESSAGES.

GLOBAL INTEGER NFREEGC, NFULLGC; INITIALLY 0
THESE VARIABLES INDICATE THE RESPECTIVE NUMBER OF FREE SPACE AND FULL SPACE GARBAGE COLLECTIONS THAT HAVE OCCURRED.

GLOBAL INTEGER OMARGIN; INITIALLY 1
OMARGIN INDICATES THE NUMBER OF COLUMNS ON THE LEFT OF AN OUTPUT LINE THAT A PRIN OR PRINT STATEMENT WILL FILL WITH BLANKS AS THE NEW LINE IS STARTED, BUT OTHERWISE IGNORE.

GLOBAL INTEGER OSPACE; INITIALLY 0
OSPACE, WHICH IS RESET TO ZERO AFTER EACH PRIN OR PRINT STATEMENT, INDICATES THE NUMBER OF COLUMNS TO BE SKIPPED BEFORE THE PRIN OR PRINT STATEMENT BEGINS FILLING THE CURRENT OUTPUT LINE. A NEGATIVE VALUE WILL CAUSE BACKSPACING (BUT NOT OVER-PRINTING), BUT AT MOST TO THE BEGINNING OF THE CURRENT LINE (OVER-RIDING THE OMARGIN). SPACING BEYOND THE RIGHT-HAND END OF THE CURRENT LINE WILL CAUSE THE CURRENT LINE TO BE PRINTED, AND A NEW ONE INITIATED AT THE OMARGIN.

GLOBAL INTEGER OTAB; INITIALLY 0
OTAB, WHICH IS SET TO OMARGIN+1 AS A NEW LINE IS STARTED, INDICATES THE EXACT POSITION (COLUMN) OF THE CURRENT OUTPUT LINE WHERE FILLING WILL RESUME WHEN THE NEXT PRIN OR PRINT STATEMENT IS EXECUTED -- SUBJECT TO MODIFICATION BY OSPACE. IF, WHEN A PRIN OR PRINT STATEMENT IS EXECUTED, OSPACE+OTAB IS GREATER THAN THE LINE LENGTH OF THE CURRENT PRFILE (AS SPECIFIED ON THE MAIN PROCEDURE CARD), A NEW LINE IS STARTED.

GLOBAL ALPHA OUTFILE[7]; INITIALLY *OUTPUT*
OUTFILE NAMES THE FILE ON WHICH THE <OUTPUT STATEMENT> WILL WRITE.

GLOBAL ALPHA PRFILE[7]; INITIALLY *OUTPUT*
PRFILE NAMES THE FILE ON WHICH THE PRIN AND PRINT STATEMENTS WILL WRITE LINES AS NECESSARY.

GLOBAL ALPHA QSMODE[5]; INITIALLY *QUEUE*
THE ADDITION OF ITEMS TO SYSTEM SETS (BY ATTACHING OR RELATING) IS BY STACKING OR QUEUING -- ACCORDING TO WHETHER THE VALUE OF QSMODE IS *STACK* OR *QUEUE*.

GLOBAL ALPHA RDFILE[7]; INITIALLY *INPUT*
RDFILE NAMES THE FILE FROM WHICH THE <READ STATEMENT> WILL READ LINES AS NEEDED.

GLOBAL ALPHA TRFILE[7]; INITIALLY *OUTPUT*
TRFILE NAMES THE FILE ON WHICH ANY TRACE MESSAGES ARE WRITTEN. THE EMPTY STRING INDICATES NO TRACE MESSAGES.