GROPE: A GRAPH PROCESSING LANGUAGE AND ITS FORMAL DEFINITION

by

Daniel Paul Friedman

August 1973 TR-20

This paper constituted the author's dissertation for the Ph.D. degree at The University of Texas at Austin, August 1973.

This work was supported in part by the following National Science Foundation grants: GJ-778, GJ-36424, and EC-509X.

Technical Report No. 20
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ACKNOWLEDGEMENTS

In a piece of research which consumes the better part of four years there are generally many people who contribute to the research effort. There is, however, one person to whom I am especially indebted--Jonathan Slocum, who almost single-handedly coded the many versions of GROPE. Proper design of a complex language requires a critical partner pointing out flaws, and certainly there are aspects of GROPE he inspired, initiated, or improved. I especially want to thank my advisor Professor Terrence W. Pratt for guiding me during every phase of the research and writing of this dissertation, Professor Robert F. Simmons for always finding time to talk with me and for his support of the development of GROPE, and my other two committee members, Professors Raymond T. Yeh and Norman M. Martin, for their critical reading. I wish to thank Gary Hendrix, Bary Gold, and Patrick Mahaffey for their suggestions and encouragement. I also wish to thank Juny Armus for all that he did for me while I was at the Lyndon Baines Johnson School of Public Affairs, Kathy Armus for the thoroughness in which she treated the art work in this dissertation, and Mrs. Dorothy Baker for her outstanding typing and attention to detail and for being such a pleasure to work with. Finally, I wish to thank all of the users for their patience and understanding during the development of GROPE.

This research was partially supported by National Science Foundation Grants GJ-778, GJ-36424, and EC-509X.

June 1973

ABSTRACT

This dissertation concerns the design of a programming language for efficient processing of directed graph data structures and the precise formal definition of the semantics of the language designed. The design handles data structures and operations rather than control structures. This emphasis at the semantics level gives rise to a somewhat different view of the problem of formal definition.

This research has resulted in the development of a graph processing language, GROPE, for efficient processing of directed graph structures. GROPE embodies some major new ideas about representation and processing of complex data structures. In addition, a new two-level definitional technique for programming language semantics has been introduced. One level develops user-oriented semantics and the other develops implementation-oriented semantics. As an illustration of this technique a major part of GROPE is formally defined.

TABLE OF CONTENTS

CHAPTER — PAGE

I. INTRODUCTION AND BACKGROUND 1

II. THE GROPE APPROACH TO GRAPH PROCESSING 28

III. MACRO-SEMANTICS OF GROPE 64

IV. MICRO-SEMANTICS OF GROPE 91

V. CONCLUSIONS 113

APPENDIX A 117

REFERENCES 134

LIST OF FIGURES

FIGURE		PAGE
1.1	A Multi-field Cell (plex) and Its Representation As a VDL Tree	14
1.2	A LISP List	15
1.3	VDL Representation of a LISP List	15
1.4	The LISP and VDL Function <u>member</u>	17
1.5	Definition of H-graph	18
1.6	A Sample Stack	19
1.7	H-graph Representation of the Sample Stack	19
1.8	H-graph Representation of Stack Operations	20
1.9	Representation of a LISP List in the Axiomatic Approach	22
1.10	Axiom for <u>cons</u>(x,y)	23
1.11	An Example of the "Abstract System" Approach: The Definition of a Hypergraph	24
1.12	The Definition of the GRASPE Function <u>cop</u>	26
2.1	A Graph Skeleton	30
2.2	A Graph Data Structure	30
2.3	Creation of a Graph Data Structure	31
2.4	Accesses from a Graph Data Structure	33
2.5	System Set Retrievals for a Graph Data Structure	35
2.6	A Structure Which Emphasizes the <u>nset</u> and <u>grset</u>	36
2.7	System-set Retrievals for the Structure Which Emphasizes the <u>nset</u> and <u>grset</u>	37
2.8	Table of Mapping Functions by <u>class</u> and <u>type</u>	40
2.9	dmapft(rseto(w),true)	41
2.10	dorft(rseto(w),true)	42

FIGURE PAGE

2.11 Definition of the Mapping Functions 43

2.12 Versatility of the Mapping Functions 44

2.13 The nth Component of p Component (p,n) 47

2.14 The Last Component of p 47

2.15 The Last Component of p Allowing for p to Be Altered 47

2.16 mapft(p,ft,arg_2,...,arg_k) 48

2.17 f(p,ft,arg_2,...,arg_k) 49

2.18 A Complex Graph-based Data Structure 52

2.19 Retrievals for a Complex Graph-based Data Structure 53

2.20 Another Complex Graph-based Data Structure 54

2.21 Retrievals for Another Complex Graph-based Data Structure . 55

2.22 Traversing with the Graph Reader 57

2.23 Data and Results of Algorithm 58

2.24 mapft(rseto(r),chafrn,w) 60

2.25 Changing the Graph of a Node 61

2.26 Representation of a Graph by Subgraphs 63

3.1 Formal Specification of the "Abstract System" Approach . . . 66

3.2 A Simple Graph (or gds) 70

3.3 H = gds of a Simple Graph 71

3.4 The States of Nodes and Arcs 76

3.5 Graph Structure for Semantic Examples 79

3.6 H = gds of Graph Structure for Semantic Examples 80

3.7 gds After State Changing Operations 81

3.8 gds After State and Structural Changing Operations 85

3.9 Graph for Illustrating "Subtle" Simple Traversal 87

FIGURE PAGE

3.10 The Complex Reader Mechanism 89
4.1 Arc Label Conventions 94
4.2 GDS of a Simple Graph 95
4.3 Diagram for CREATE-NODE(*,**) 98
4.4 Diagrams for RELATE(*) 101
4.5 Diagram for UNRELATE(*) 103
4.6 GDS for Semantic Examples 104
4.7 Tabular Form of GDS for Semantic Examples 105
4.8 Diagram for TRAVERSE-RELATED-SUCCESSOR(*) 109
4.9 Diagram for TRAVERSE-NODE-OUT(*) 111
4.10 Diagram for TRAVERSE-GRAPH-OUT(*) 112

CHAPTER I

INTRODUCTION AND BACKGROUND

Overview

This dissertation has two main concerns. The first is the design of a programming language for efficient processing of directed graph data structures. The second is the precise formal definition of the semantics of the language designed. In the design the concern is entirely with data structures and operations rather than control structures. This emphasis at the semantic level gives rise to a somewhat different view of the problems of formal definition.

This research has resulted in two major achievements. A programming language extension, GROPE, for efficient processing of directed graph structures has been designed and implemented. GROPE embodies some major new ideas about representation and processing of complex data structures. In addition, a new two-level definitional technique for programming language semantics has been introduced. One level develops user-oriented semantics and the other develops implementation-oriented semantics.

Graph processing is a new area of language design. The next section sheds some light on graph processing and discusses its relevant background. Likewise, formal semantic definition is relatively new, and the appropriate literature is discussed following the graph processing section.

Graph Data Structures

The term "graph" used in the previous section requires some explanation. Here, a graph is a data structure composed of nodes (vertices) and arcs (edges or branches). These graph data structures have labeled nodes and arcs, and they may be organized into sets, hierarchies, etc. The reader should not confuse our use of the term "graph" with the subject area of "computer graphics" which is directly concerned with picture construction.

Graph data structures are important data representations in many fields. In mathematics graph structures are studied for their static properties. In computer science graph structures are studied for their dynamic properties. In other fields "graph structures" have various names. For example, there are bonding structures in chemistry, Feynman Diagrams in physics, sociograms in sociology, circuit diagrams in electrical engineering, and flow networks in operations research.

Algorithms which process graphs are important. For example, algorithms which determine the maximal flow through a network, shortest path between two nodes, a Hamiltonian Path or optimal line balance are all usually formulated as graph processing algorithms. There are graph algorithms for the well-known "Traveling Salesman Problem," for finding a maximal spanning tree, and for information retrieval. Graph algorithms have also been applied to the "Four Color Problem," the solution of the "Knight's Tour," and the determination of transitive closures. Programs which involve graph processing are clearly an important class of programs.

Graphs are ordinarily represented in a computer in one of two ways. They are either simulated by using more primitive structures (e.g., arrays

in ALGOL [25] or property lists in LISP [23]) or they are simulated by using extensible data structures (e.g., programmer-defined data types in SNOBOL4 [12], "based variables" in PL/I [19] or plexes in AED [32]).

"Incidence arrays" are a well-known representation for graphs using more primitive data structures. A graph is represented by a two-dimensional square array A, having one row and one column for each node. An edge from node i to node j with label v is denoted by the array position $A_{i,j}$ having value v. The main drawback of this representation is the lack of flexibility for the representation of complex structures. For example, associating additional values with nodes and arcs or allowing parallel arcs requires additional storage. A lesser shortcoming of this representation is the relative inability to do dynamic processing. For example, if the graph contains k nodes, then it is difficult when using incidence arrays to let the graph grow to k+1 nodes through the creation of a new node, for few programming languages allow an array to grow by the addition of a row and column.

Another example of simulating graph structures by using primitive structures involves property lists (attribute-value pairs). Nodes are represented by "atoms" with attached property lists. If $A_{i,j}$ is the value of the arc from node i to node j, then there is an attribute-value pair $(A_{i,j}, j)$ in the property list of i. The property list representation causes graph algorithms to be inefficient in terms of time due to the necessity for property list searching for each arc access. In addition, the property list representation makes it difficult to traverse arcs in both directions, a property required in many graph algorithms (e.g., finding a critical path on a PERT network).

When graphs are simulated by using extensible data structures, the user defines blocks of core (plexes, records or based variables) as nodes. Arcs are represented by pointers from one block to another. The specified fields within a node are used to store the information associated with a node and with the arcs leaving the node. A number of programming languages have this ability as a built-in feature, e.g., PASCAL [43], PL/I [19], AED [32], and L^6 [17]. Each of these languages has the major difficulty that the burden is on the programmer to define a set of logical primitives for graph processing. In addition, the programmer must construct facilities for the storage management and input/output. In SNOBOL4 [12], using programmer-defined data types, some of these aspects disappear. For example, SNOBOL4 has a garbage collector for storage management, and some basic accessing and creating primitives are automatically created when a new data type is defined. Yet the responsibility of defining most of the appropriate graph processing primitives in SNOBOL4 still rests with the programmer.

In the preceding section certain shortcomings of using primitive structures or extensible structures to simulate graph structures and processes have been presented. Many of the arguments for choosing a true graph processing language over one of the simulations of graph structures mentioned above are reminiscent of the arguments for choosing a high-level language over assembly language. For example, in both the simulation and assembly language, input/output requires much software development whereas graph processing languages and high-level languages have (or should have) a well developed input/output facility. In addition, programs in assembly language tend to be error prone and have poor self-documentation (that is, the programs are difficult to follow). The same is true for many of the

simulation techniques mentioned above. In each case much user-supplied support software is required before considering the algorithm that is actually being programmed. Also, a new task for such a graph simulation may require a major redesign; however, for a graph processing language, little or no redesign should be necessary.

GROPE, the subject of this paper, is a general-purpose graph processing language in which graphs form the basic data structure. The general class of graph processing problems for which GROPE is designed is characterized by two aspects. First, the problems deal with sets of graph structures which are interrelated in complex ways and which contain symbolic as well as numeric data. Second, the problem solutions require the graph structures to grow, shrink, and be modified both dynamically and irregularly. These complex graph processing problems are precisely those for which the simple graph simulations described above are most inadequate.

GROPE is a graph processing language designed to provide appropriate structures and primitives for this class of problems. The GROPE design is based on three major design criteria. First there should be flexibility of structure for representing a variety of classes of data. There should be labeled nodes and labeled arcs and provision should be made for the representation of multiple arcs between two nodes. It should be possible to represent, in a natural manner, hierarchical graphs (graphs whose nodes can have values that are graphs) and other relationships between graphs. There should be supporting structures, such as simple list and set processing for maintaining information during graph searches. There should be special mechanisms for searching and processing graph structures.

Second, there must be operations which modify the structures dynamically. There must be operations which destroy and modify graphs, nodes, and arcs, e.g. for changing the labels of nodes and arcs.

Finally, the processes and storage management must be handled efficiently. The required efficiency is dictated by the combinatorial nature of algorithms for graph processing. Storage management must include automatic bookkeeping for the dynamic allocation and recovery of storage, e.g. using a free space list and garbage collector.

Related Graph Processing Literature

Since directed graphs are often used for informal description and analysis of 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.

The graph processing languages of interest are HINT [13], GRASPE [31,7,8,9], GEA [4], and LINKNET [3]. HINT and GRASPE were designed for symbolic structure manipulation problems, and each is associated with a list processing language. HINT is compiled into IPL-V [24]. GRASPE is a library of LISP functions. GEA and LINKNET were designed to perform numerical data analysis within a complex, but relatively static graph structure (problems in operations research, etc.), and each is associated with an algebraic language. GEA is a syntactic extension to ALGOL which is precompiled into ALGOL; LINKNET is a library of FORTRAN functions. Using the design criteria discussed above for the necessary characteristics of a

graph processing language, let us now compare and contrast these four languages with GROPE.

In terms of flexibility for representing a variety of structures, only HINT, GRASPE, and GROPE have provided for list processing as a supporting tool for graph processing. Only GROPE is concerned with more than one type of node and one type of arc. GEA and LINKNET deal only with numeric constants as values of nodes and arcs, whereas HINT, GRASPE, and GROPE provide for symbolic node and arc values as well as hierarchical structures.

In terms of operations for the dynamic creation of graph components, only HINT, GRASPE, and GROPE allow for the dynamic creation and destruction of graphs. Each language except LINKNET provides primitives for the dynamic creation and destruction of nodes and arcs. In LINKNET, these operations are the responsibility of the programmer, i.e. the programmer must produce code which correctly affects the appropriate fields to cause the creation and deletion of nodes and arcs.

In terms of efficiency of processes and storage management, GRASPE and HINT are tied to their respective hosts for their representation of graphs. Both use property lists. The efficiency of the processes in GRASPE and HINT is poor due to their internal representation of graphs as property lists and the cost of their primitives (which require property list searches). GEA uses lists to represent graphs. Little can be said about the efficiency of GEA as the details of the precompiler are unavailable. LINKNET and GROPE use plex structures for their representation of graphs. LINKNET does not have any graph processing primitives, only primitives to change the contents of fields in a plex. GROPE operations are very efficient (see

Chapter IV). GRASPE, GEA, and GROPE have a garbage collector. GRASPE's is that of its host, LISP. HINT uses the storage manager of IPL-V, and LINKNET has no storage management.

In the previous discussion of the graph processing languages, we noted what appeared as deficiencies in some of the languages. It should be pointed out that these were deficiencies in terms of our design criteria and not necessarily shortcomings of each language. On the contrary, each language appears to be a good model for the class of problems with which it is concerned, although in most instances the efficiency is very poor.

GROPE

GROPE is a successfully implemented graph processing extension to FORTRAN. In this sense, since it is a library of functions, GROPE parallels SLIP [40]. GROPE not only provides primitives for graph processing but also includes a number of other data structures and primitives which enhance and support graph processing.

There are a number of major new ideas embodied in the GROPE data structures and operations which are directly associated with graph processing. GROPE provides a set of building blocks (atoms, arcs, nodes, and graphs) and operations for putting these blocks together. The building blocks are used not only to form simple graphs but also complex graph-based structures (see Figures 2.18 and 2.20). In addition, because of the flexibility of the GROPE data structures, there are a number of graph modification primitives which perform unusual operations (for example, an operation to move a node from one graph to another). Arcs and nodes are partitioned

into four classes. Each class provides for a different level of structural information. For example, an arc between two nodes n and m may be accessible from n only, from m only, from both, or from neither. Although the structures a programmer can create are likely to be very complex, experience has shown the usage of the accessing primitives to be straightforward.

It is unreasonable, for our purposes, to consider a graph processing language as just a set of graph processing operations. The support operations are equally important to the development of efficient graph algorithms. Some of the support features are list, set, and array processing, and a large class of mapping functions which build or destroy structures by sequentially accessing elements in a set or list. In addition, there is an extensive input/output facility and a garbage collector. Throughout the design of GROPE, there has been a fanatical concern with efficiency and a serious endeavor to maintain generality.

GROPE has been a useful tool in many applications. Slocum [34], Hendrix [14,16], and Thompson [38] used GROPE in the area of natural language processing. In the area of programming language semantics, an ALGOL interpreter (see, for example, Wilson [42] or Wesson [41]), written using H-graphs [30], is being tested in GROPE. The Linguistics Research Center at The University of Texas at Austin has used GROPE to develop a central portion of its machine translation system [21,36,37]. Work in the analysis of programs (Griggs [11]), optimal overlay structures for LISP and FORTRAN programs (Greenawalt [10]), and robotics (Hendrix [15]) are further illustrations of the scope of GROPE usage.

GROPE has fostered the development of GROPE 2.0 (Slocum [35]). GROPE 2.0 is a complete, modular programming language with block structure which has a somewhat ALGOL-like syntax. The GROPE 2.0 compiler (written in GROPE) generates GROPE-FORTRAN code and thus serves as a very sophisticated FORTRAN preprocessor. GROPE has been implemented on the CDC 6600 and IBM 360 (Baron [1]).

For a complete description of GROPE, see Appendix A.

Programming Language Semantics

Techniques for formally defining programming languages generally follow a common pattern. First a translation is necessary which maps the program strings into some "internal form." This internal form is then considered as the "initial state" of an abstract machine. The abstract machine moves from state to state as a result of applying a primitive operation of the machine with a transition rule to the current state. A "final state" is encountered if the program terminates. This scheme is used in Landin's [18] definition of ALGOL, Lucas' [22] definition of PL/I, and Pratt's [30] definition of ALGOL.

This paper is also concerned with the formal definition of programming languages, in this case the definition of GROPE. However, because GROPE is defined as a language extension (a set of data structures and primitives operations), its formal definition presents somewhat different problems from those encountered in a definition of a language such as ALGOL or PL/I. Where our approach differs is in two aspects. First, because we are not concerned with syntax (i.e. with program strings), there is no concern with translation from strings into an internal form. Second,

because we are not concerned with control structure we simplify our problem at the "abstract machine" level. We need only be concerned with "states" composed of constants and data structures and "transition rules" defining operations on constants and operations on data structures.

By restricting our concern to the definition of data structures and operations we avoid many of the complexities of other definitional techniques. This allows us freedom to attack some problems which have as yet received scant attention in the literature. Stated informally, the concern here is with formal definitions which satisfy two particular criteria. First the formal definition should be such that a reader of the definition can obtain a conceptual understanding of the data structures and operations involved. Second, a reader should be able to understand how the data structures and operations can be implemented and to determine the relative efficiency of processing.

These criteria are of fundamental importance if formal definitions of languages are to be of practical value to language users and implementers. Existing definitional techniques tend to be either unintelligible to the programmer, impractical as the basis for an implementation, or both. In fact, we usually find that an implementation definition is too detailed for the development of a conceptual understanding of the data structures and operations and that on the other hand a conceptual definition is too simple for the development of an implementation which utilizes the structural subtleties that we find in a well-thought-out model. In point of fact, there really are two problems, and generally any approach which treats the definition of data structures and operations as only one problem has the shortcomings noted above.

The problem of finding a single definitional technique to display the external (conceptual) and internal (implementation) structure and hence satisfy both criteria is resolved in this paper by defining formally the same operations over two conceptually different formal systems. The two levels of definition are termed the "macro-semantics" and the "micro-semantics." The macro-semantics is the user-level semantics. Both a formal system to describe the data structures and the formal definitions of the set of operations over the data structures are included in the macro-semantics. Similarly, the micro-semantics is the implementer-level semantics. The micro-semantics is composed of a formal system to describe the data structures and a set of formal definitions of the operations over the data structures (storage structures).

The concern of the macro-semantics is to present the whole picture of the language model from the standpoint of the potential user who needs the answer to the following question: Notwithstanding storage and execution time requirements, are the structures and operations suited to my particular problem? The micro-semantics deals with the formalization of the implementation-level concepts. From these definitions an implementer can ferret out the "bits and pieces." In addition, for the potential user, the micro-semantics yields some approximation to the storage and execution time requirements to execute algorithms.

Formal Definition of Programming Language Semantics

Formal definition of programming language semantics is a relatively new area. Debakker [5] provides a good (although dated) survey of research in the formal study of programming language semantics. Since the concern

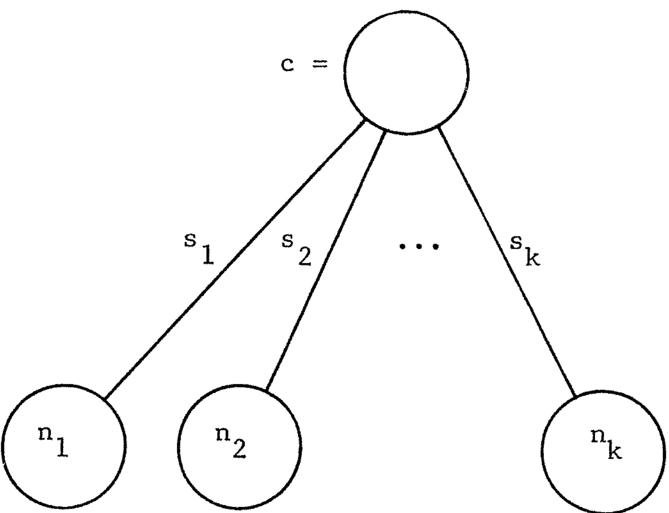
here is with objects and operations on objects, the discussion is limited to the treatment of this limited area of semantics. For each approach we are concerned with three basic questions. First, how is the total data space or "overall state" of the abstract machine conceived? Second, how are data structures represented? Third, how are primitive operations defined? The approaches of interest are the Vienna Definition Language [22,39,20], the H-graph approach [26,27,28,29,30], the axiomatic approach [2], and the author's "abstract system" approach [31,7,8].

The most well-known definitional technique is the Vienna Definitional Language (VDL). In VDL, the total data space is represented by a set of trees with labeled arcs. The overall "state" of all data structures at any point in a computation is represented by a single "state tree" which also contains components concerned with control structures.

VDL has the facility for handling data structures and operations over data structures. Consider the representation of a LISP list in VDL. Recall that two lists may have the same sublist and thus the simple tree representation of lists is inappropriate. The representation of a multi-field cell (plex) in VDL (see Figure 1.1) can be defined as a one-level tree where each s_i ($s_i \neq s_j$ for $i \neq j$) are the fields in the cell, and n_i (the leaves of the VDL tree) are integers (indices) or data constants. A LISP list (car,cdr) is a LISP-like list (head,tail), $c = (c_1, c_2, \ldots, c_k)$ where each c_i is a 2-field cell. We define a function elem(i,c) which maps to c_i. The VDL tree of Figure 1.3 is the representation of the LISP list of Figure 1.2.

Operations in VDL are presented using conditional expressions. As an example of the definition of a VDL operation, consider the LISP function

s_1	a data constant or elem(n_1,c)
s_2	a data constant or elem(n_2,c)
. . .	. . .
s_k	a data constant or elem(n_k,c)



n_i may be a data constant or an integer

Figure 1.1. A Multi-field Cell (plex) and Its Representation As a VDL Tree

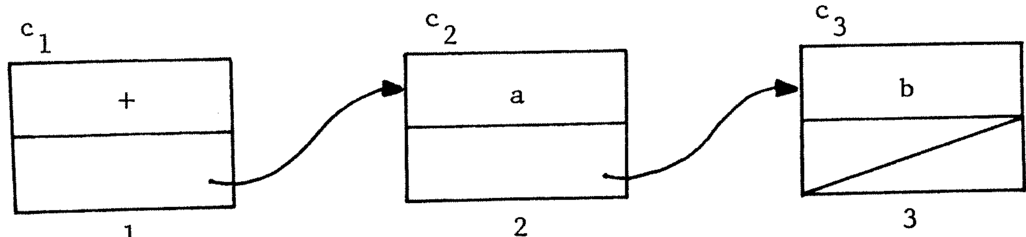


Figure 1.2. A LISP List

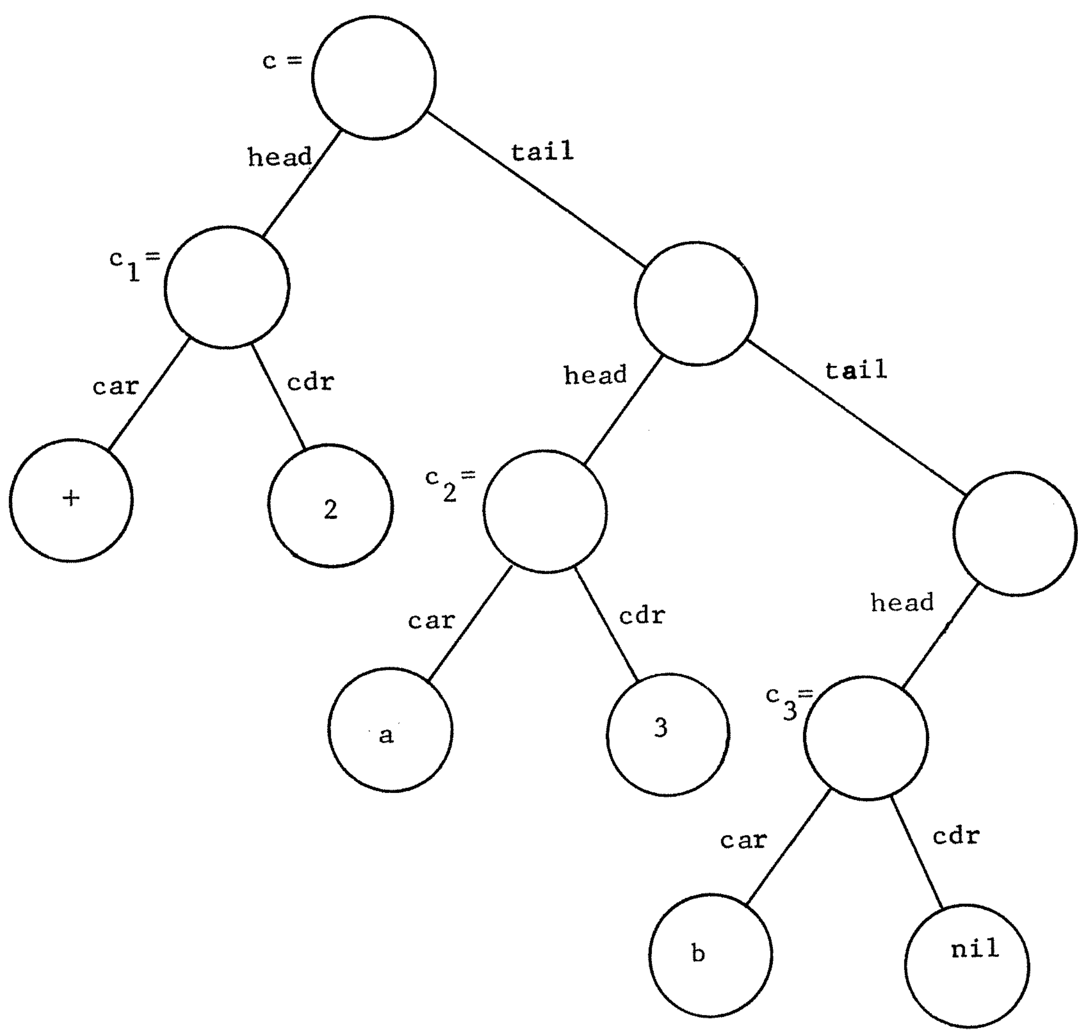


Figure 1.3. VDL Representation of a LISP List

member (see Figure 1.4) over the LISP lists defined above. Note that in the VDL member, i is initialized, in this case as 1.

Pratt [26,27,28,29,30] suggests a definitional technique based on "H-graphs" (see Figure 1.5 for its definition). The value of each node in an H-graph is a terminal or a graph, thus allowing the graphs to be organized into hierarchies. The total data space or the overall state of the "abstract machine" is an H-graph.

Data structures can be modeled as H-graphs. Consider Pratt's [30] representation of a stack. A stack is defined recursively to be a graph composed of two nodes. The first node is an arbitrary data node and the second is either null or a stack. (See Figure 1.6 for sample stack and Figure 1.7 for its representation as an H-graph.)

There are ways of defining operations using H-graphs which change the overall state. An operation in the H-graph approach is a transformation which maps an H-graph into an H-graph. Pratt [30] introduces a formal diagramatic approach to define the operations. Figure 1.8 illustrates the formal diagramatic approach for the operations--push and pop--over the stack defined above. In the figure, the push node and the pop node represent function references. An arc pointing into a function reference node implies that the node from which the arc emanates contains a parameter to the function and similarly an arc pointing out of a reference node implies that the node at which the arc terminates may have its contents altered.

Burstall [2] develops an axiomatic approach to programming language definition. This approach is based on the first-order predicate calculus; the axioms for a simplified ALGOL-like language are presented.

```
LISP definition

    member[a;c] = [
                        null[c] → NIL;
                        eq[a;car[c]] → T;
                        T → member[a;cdr[c]]    ]

VDL definition

    member(a,i,c) =
                        is-nil(i) → nil
                        car(elem(i,c)) = a → t
                        t → member(a,cdr(elem(i,c)),c)

    elem(i,c) =
                        i = 1 → head(c)
                        t → elem(i-1,tail(c))
```

Figure 1.4. The LISP and VDL Function member

An H-graph is a finite set of directed graphs over a common set of nodes, organized into a hierarchy. Assume a set A of basic data "atoms" and a set N of nodes.

DEFINITION: A graph over A and N is a triple (M,E,S) where M is a finite non-empty subset of N, the node set, E is a finite set of triples of the form (n,a,m) where $n,m \in M$ and $a \in A$, the arc set, and $S \in M$, the entry point node.

DEFINITION: An H-graph over A and N is a pair (M,V) where M, the node set, is a finite non-empty subset of N, and V, the value or contents function, is a function mapping M into $A \cup \{X | X$ is a graph over A and $M\}$.

Figure 1.5. Definition of H-graph

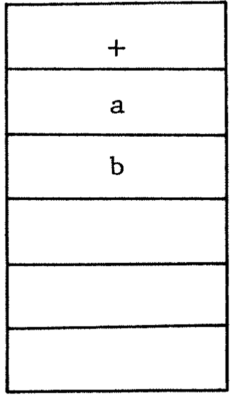


Figure 1.6. A Sample Stack

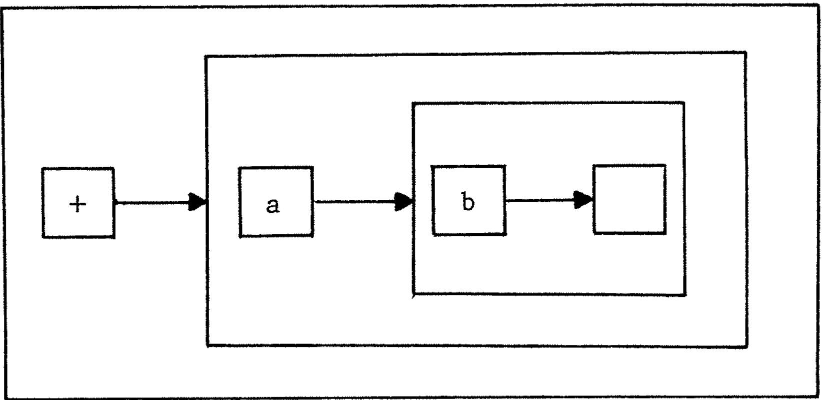


Figure 1.7. H-graph Representation of the Sample Stack

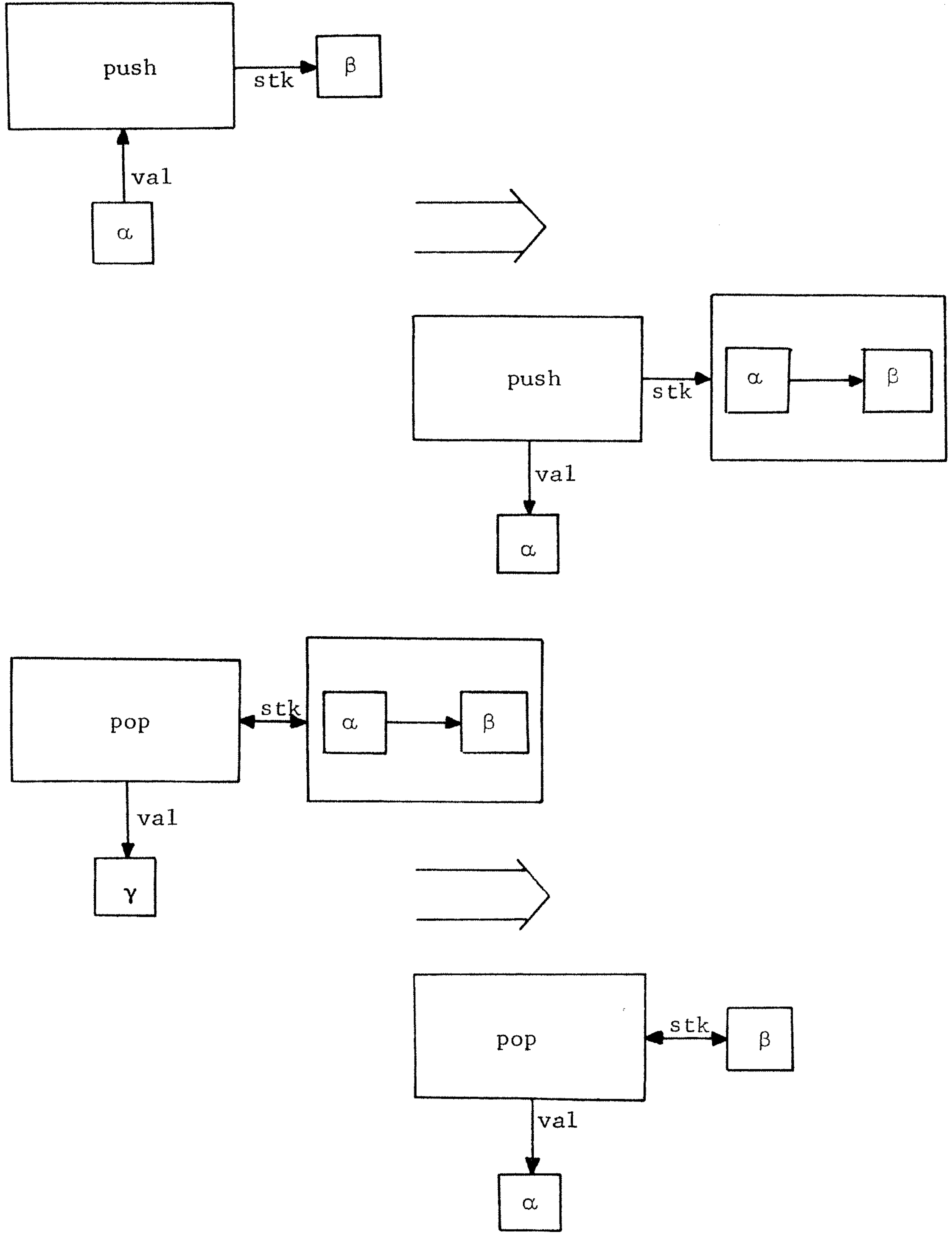


Figure 1.8. H-graph Representation of Stack Operations

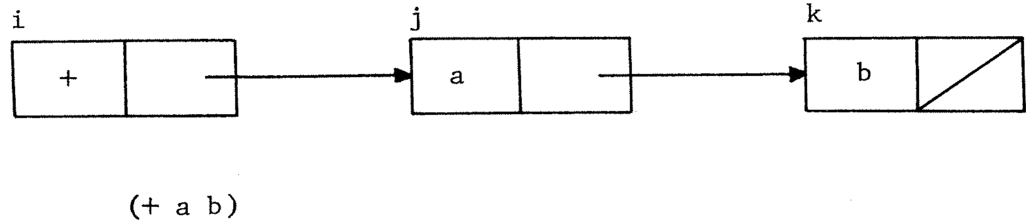
The overall state of the abstract machine in the axiomatic approach is represented by a sentence of the predicate calculus and a "state vector." A state vector, s, is an association of variables with their values. As a program is executed, the values in the state vector get altered (actually a new state vector, s*, is generated with perhaps some of the old values carried over as new values), and new sentences are concatenated (by the conjunctive connector, &) onto the old sentence.

Data structures can be represented using the axiomatic approach. In this technique each cell in a data structure is represented by the conjunction of the appropriate relational primitives (see Figure 1.9 for the representation of a LISP list).

Operations are defined in the axiomatic approach by showing what new axioms need to be added to the logical sentences which have thus far been built in order to describe how the state vector is to be altered. In order to give the reader the flavor of this approach, let us suppose that we want to add the LISP operation cons to an existing system. Figure 1.10 presents a possible axiom with a loose translation of its meaning for the operation cons.

The author [7,8,31] introduces the technique which employs an abstract system. The particular abstract system referred to is termed a "hypergraph" (see Figure 1.11). In this paper we define two other abstract systems: a gds for the definition of the macro-semantics and a GDS for the definition of the micro-semantics.

We can characterize the author's abstract system approach in the following fashion. An abstract system, call it H (hypergraph, gds, or GDS), is defined to represent the total data space. Each operation op (note that



(+ a b)

equal(fcar(i,s),+) & equal(fcdr(i,s),j) &

equal(fcar(j,s),a) & equal(fcdr(j,s),k) &

equal(fcar(k,s),b) & equal(fcdr(k,s),nil)

Figure 1.9. Representation of a LISP List in the Axiomatic Approach

i. equal(freespace(s),n) →

ii. equal(freespace(s*),next(n)) &

iii. equal(fcar(n,s*),value(x,s)) &
equal(fcdr(n,s*),value(y,s)) &

iv. for all m such that not equal(m,n) →
equal(fcar(m,s*),fcar(m,s)) &
equal(fcdr(m,s*),fcdr(m,s))

----

i. If while processing we encounter a statement cons(x,y) and if we denote n as the first available cell in the freespace stack, then

ii. n will be popped off the freespace stack, and

iii. the <u>car</u> and <u>cdr</u> of n will become the current binding (value) of x and the current binding of y respectively, and

iv. all other cells (different from n) will remain unaffected.

Figure 1.10. Axiom for <u>cons</u>(x,y)

A hypergraph is a quintuple (G,N,A,s,f) where

G is a finite set (of graphs)

N is a finite set (of nodes)

A is a finite set (of arc labels)

s: $G \rightarrow 2^{N}$, s defines the nodes which occur in each graph

f: $G \rightarrow 2^{N \times A \times N}$, and for each $g \in G$, $f(g) \subseteq s(g) \times A \times s(g)$

If $(n,a,m) \in f(g)$, then there is said to be an arc from node n to node m with label a in graph g. Note that any single graph is completely defined by the value of s(g) (giving its nodes) and f(g) (giving its arcs).

Figure 1.11. An Example of the "Abstract System" Approach: The Definition of a Hypergraph

op is defined over the total data space, yet op is not part of the total data space) is described in the following manner. Given op, its arguments $x_1, x_2, \ldots, x_n$ and H, then some entity from H, call it v, is the value of the operation, and H is transformed into a new abstract system H'. Mathematically, $op(H, x_1, x_2, \ldots, x_n) = (H', v)$. For purposes of convenience and naturalness, H and H' are considered implicitly as the underlying (overall) data space and the relationship becomes the familiar $op(x_1, x_2, \ldots, x_n) = v$, and the implicit argument H is now transformed into H'.

From the viewpoint of a state transition in an abstract machine, H is the structure of the state. Thus applying op to $x_1, x_2, \ldots, x_n$ is equivalent to making transitions from state to state in an abstract machine where the states (H's) are generated.

The hypergraph (see Figure 1.11) is an illustration of an abstract system where the total data space or overall state is any hypergraph (G,N,A,s,f). In the GRASPE description, the legal GRASPE data structures are presented. Figure 1.12 illustrates the definition of the GRASPE function cop which creates an arc. One important attribute of the definition of cop (which is true for all operations) is that only set operations (union, intersection, set difference, etc.) are required to specify the condition of the generated abstract system.

In this dissertation we present the formal semantics of GROPE using the abstract system approach. The technique of defining both levels--conceptual and implementation--in a single coordinated manner is a novel idea. None of the existing techniques has as yet been applied to more than one level.

$\underline{\text{cop}}(n,a,m,g)$ = true with the side effect of setting

$$G = G \cup \{g\}$$

$$N = N \cup \{n,m\}$$

$$s(g) = s(g) \cup \{n,m\}$$

$$f(g) = f(g) \cup \{(n,a,m)\}$$

Figure 1.12. The Definition of the GRASPE Function cop

Chapters III and IV present the two-level formal definition of GROPE. Our formal definitions differ from the actual programming language GROPE. In particular, the formal definitions do not include the supporting constructs such as set and list processing. Because each of the graph processing constructs in GROPE are not independent of the supporting constructs, we found it necessary to use terms that have a slightly different meaning in the actual programming language. Also, certain definitions were changed to bring out the essence of graph processing. Perhaps the clearest statement that can be made about the differences in the two languages is that the graph processing primitives in the abstract system are somewhat simplified versions of their equivalent in the actual programming language GROPE. For a complete description of the actual programming language, see Appendix A.

In the next chapter, the reader is introduced to the GROPE approach to graph processing. In Chapter III, there is a description of the GROPE model from the user's point of view (macro-semantics) and in Chapter IV, there is a description of the GROPE model from the implementer's point of view (micro-semantics).

CHAPTER II

THE GROPE APPROACH TO GRAPH PROCESSING

In this chapter most of the GROPE programming language is introduced. A complete description of GROPE is given in Appendix A. This chapter is composed of four sections. The first covers the elementary structures and operations. The second introduces the notion of "system set" as a constrained collection of elementary structures; the third section presents two natural mechanisms (mapping operations and system set readers) for searching and processing system sets; and in the final section there is an introduction to some of the generality and flexibility of GROPE's approach to graph processing.

As mentioned earlier, we believe that it is absolutely crucial that a graph processing language have a large class of support features. Besides the various supporting features described in this chapter, GROPE has a complete list and set processing facility including input/output and a garbage collector. The details are given in Appendix A.

Elementary Ideas

In this section the elementary data structures and constants are introduced. In addition, operations are presented for creating, detaching, and accessing information that has been associated with these structures. The elementary data items are atoms, arcs, nodes, and graphs; and the operations are crgraph, crnode, and crarc as the creation functions, detgraph, detnode, and detarc as the detaching functions and graph, frnode (from node), tonode (to node), object and value as the accessing functions.

Atoms are the legal constants in GROPE. In the FORTRAN implementation of GROPE, atoms are any integer or real numbers which are valid in FORTRAN, any arbitrary string of characters, one, two, and three dimensional arrays and functions created from FORTRAN externals. For purposes of this explanation, GROPE atoms may be considered similar to LISP atoms.

Elementary Graph Data Structures

The elementary data structures in GROPE, as might be expected in a graph processing language, are graphs, nodes, and arcs. In Figure 2.1, g is a graph, x, y, and z are nodes, and a, b, c, and d are arcs. The graph g is really only a graph skeleton as there are no data constants associated with any of the structures. In order to get a useful structure out of this graph skeleton, it is necessary to introduce constructs for associating constants with the individual structures.

There are operations for creating and detaching the elementary graph structures. The creation of a graph requires only an atom as a parameter. For nodes, the parameters for creation are an atom and a graph; and for arcs, an atom and two nodes are necessary. During the processing of graphs, it is often the case that we find an arc, node, or graph which we would like to detach. The philosophy used in GROPE is that an arc or node is destroyed when it has been detached (it is no longer accessible) from the structure.

There are operations for accessing information from the structures. Given a node, it is possible to determine upon which graph the node resides. Given an arc, there are operations for accessing the node from which the arc emanates and the node to which it points. All the data atoms are retrievable given a node, arc, or graph. Figure 2.3 presents the operations

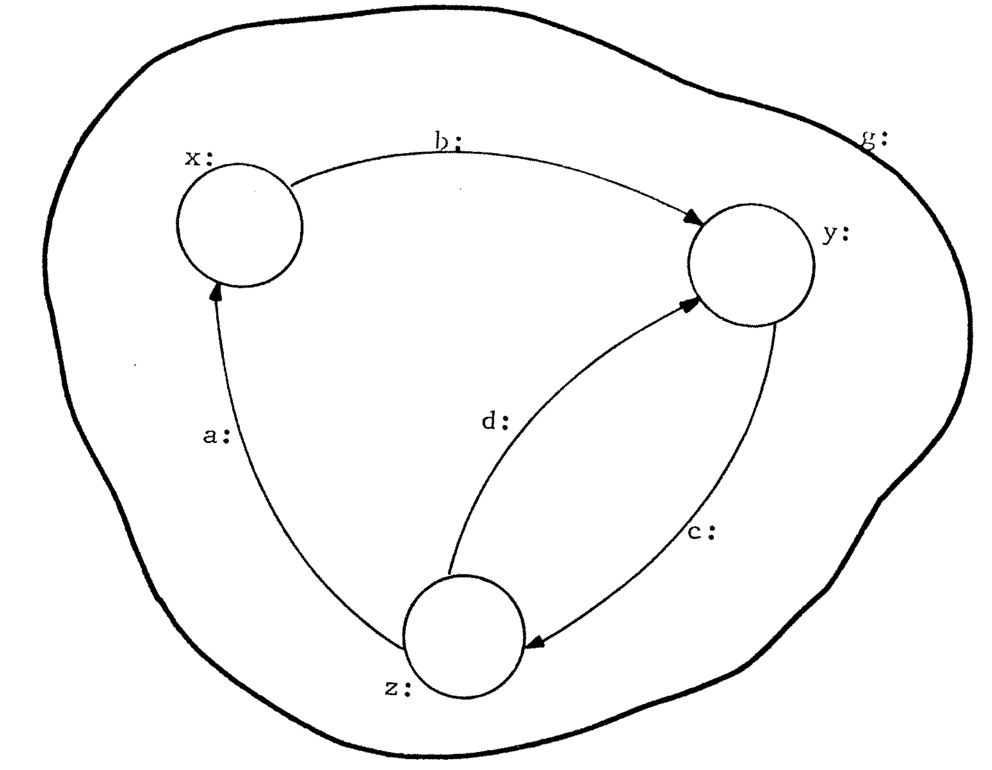


Figure 2.1. A Graph Skeleton

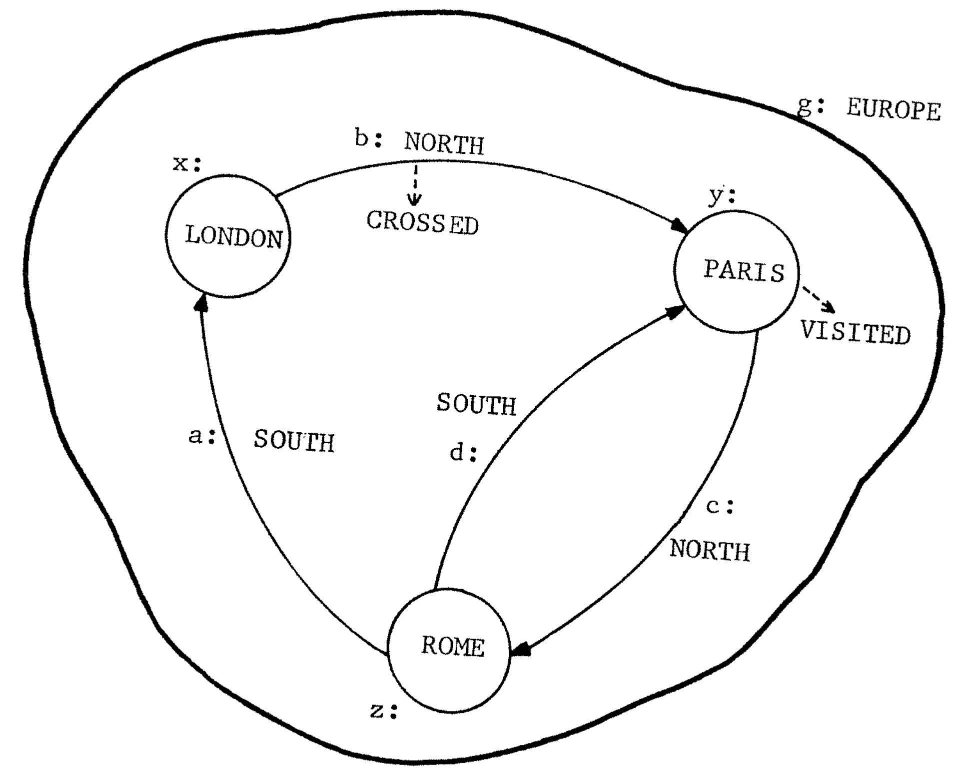


Figure 2.2. A Graph Data Structure

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

Figure 2.3. Creation of a Graph Data Structure

and arguments to the operations for the creation of Figure 2.2, and Figure 2.4 presents its accessing functions.

<u>System Sets</u>

In this section the "system sets" are introduced. A system set is an ordered collection of elementary structures which satisfy some predetermined specifications. These are called system sets because they differ from "user" sets and because no system set may contain more than one occurrence of the same elementary structure.

At this point all of the information has not been gleaned from the structures. There is an alternate way of viewing the structures of Figure 2.2. This perspective introduces us to the notion of a system set.

The system sets with which we are concerned are the <u>rseto</u> (the set of all arcs emanating from a node), the <u>rseti</u> (the set of all arcs terminating at a node), the <u>ndset</u> (the set of all nodes on a graph), the <u>nset</u> (the set of all nodes with the same object), and the <u>grset</u> (the set of all graphs).

A system set is a collection of elementary structures which have certain properties in common. For example, using Figure 2.2, the components of the set {a,d} have the following properties in common:

1. a and d are both arcs
2. a and d both emanate from the node z (have the same <u>frnode</u>).

Similarly, the components of the set {b,d} share the properties that:

1. b and d are both arcs
2. b and d both terminate at node y (have the same <u>tonode</u>).

These, in fact, are the criteria for membership in the <u>rseto</u> and <u>rseti</u>

	g	x	y	z	a	b	c	d
graph		g	g	g				
frnode					z	x	y	z
tonode					x	y	z	y
object	EUROPE	LONDON	PARIS	ROME	SOUTH	NORTH	NORTH	SOUTH
value			VISITED			CROSSED		

Figure 2.4. Accesses from a Graph Data Structure

respectively. Thus for some node n the rseto(n) is the set of all arcs emanating from the node n and the rseti(n) is the set of all arcs terminating at the node n.

In addition, the components of the set {x,y,z} share similar properties:

1. x, y, and z are nodes

2. x, y, and z reside on the same graph (have the same graph). Thus for some graph g, the ndset(g) is the set of all nodes on graph g. See Figure 2.5 for the system sets of the structure depicted by Figure 2.2.

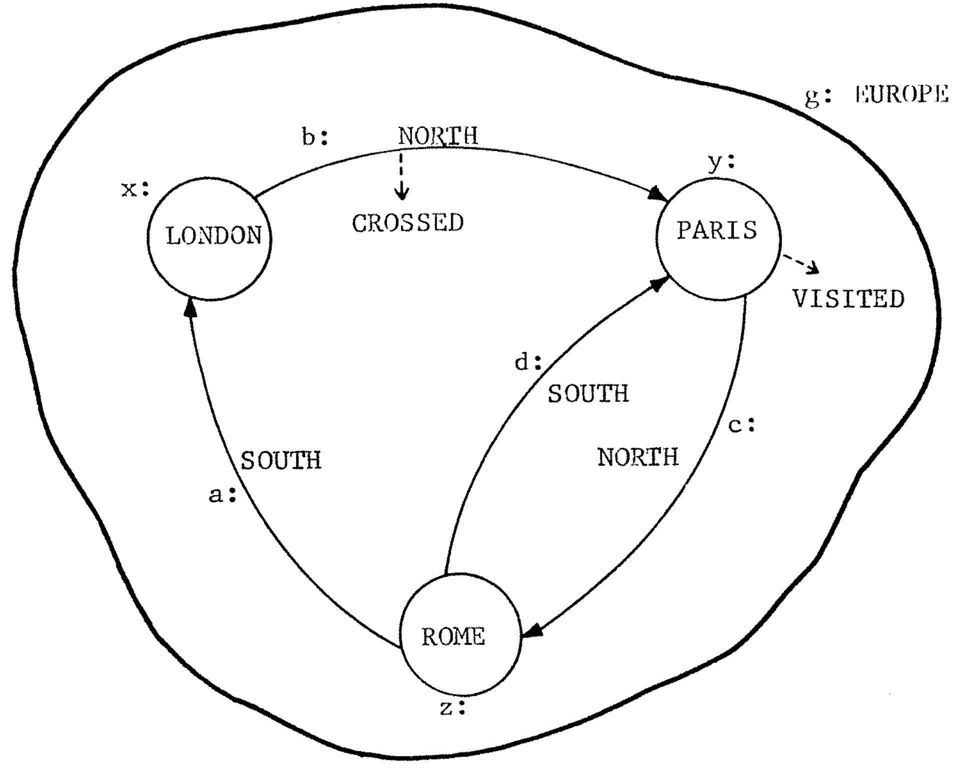
There are two other system sets; however, the structure of Figure 2.2 is inadequate for displaying the relationships associated with these sets (see Figure 2.6). These system sets are the nset (the set of all nodes with the same object) and the grset (the set of all graphs).

In the structure represented by Figure 2.6, we are now dealing with two graphs, g and h, as the entire structure. The system set, grset, is the set {g,h}. The final relationship which can be noticed from Figure 2.6 is the notion of two or more nodes having been created from the same atom. The nset(a) for some atom a is a system set which is the set of all nodes with the object a. Figure 2.7 illustrates all the system sets of this structure.

There appears to be little necessity to give motivation for the existence of the system sets rseto, rseti, ndset, and grset. However, the system set which is composed of all the nodes with the same object, the nset, requires some explanation. Consider once again Figure 2.2 and suppose we would like to find the node y given the graph g, and the atom PARIS. There are obviously two alternatives for finding the node y. One way is

	g	x	y	z
ndset	{x,y,z}			
rseto		{b}	{c}	{a,d}
rseti		{a}	{b,d}	{c}

Figure 2.5. System Set Retrievals for a Graph Data Structure



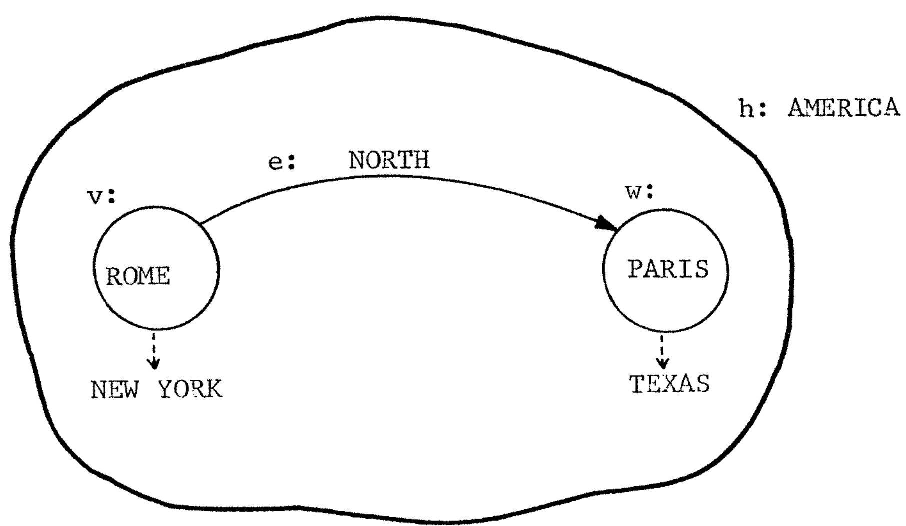


Figure 2.6. A Structure Which Emphasizes the nset and grset

		LONDON	PARIS	ROME	g	h	v	w	x	y	z
grset	{g,h}										
nset		{x}	{y,w}	{z,v}							
ndset					{x,y,z}	{v,w}					
rseto							{e}	{}	{b}	{c}	{a,d}
rseti							{}	{e}	{a}	{b,d}	{c}

Figure 2.7. System-set Retrievals for the Structure Which Emphasizes the nset and grset

to search all the nodes on graph g [ndset(g)], until we find one with object PARIS. The other way would be to search all the nodes with object PARIS [nset(PARIS)], until we find one on graph g. Since most graph structures contain many nodes, the search down the ndset (set of all nodes on a graph) is in general likely to be more expensive than the search down the nset (set of all nodes with the same object). It should be pointed out that there are diabolical structures where the reverse is true, but experience has shown that these diabolical structures rarely occur.

One motivating aspect of the system sets that cannot be overlooked is that the structures are woven into one another to form all the system sets and do not require any storage beyond that required for graphs, nodes, and arcs. Thus once the creations (crgraph, crnode, and crarc) have been accomplished, no new sets need be created in order to process the system sets. Details of this storage structure are presented in Chapter IV.

Processing Graphs

The primary mechanism for processing graphs is by accessing system sets and searching these system sets. There are two searching techniques in GROPE. The first is the utilization of the mapping functions and the second is the system set readers with their associated operations.

Mapping Functions

The mapping functions allow the user to selectively search a system set and sequentially process each individual element of the set. The mapping functions are distinguished by their class and by their type. There are four classes of mapping functions. First there is a class of mapping functions which simply process the elements of a set; second, there are those

that detach specified elements from the system sets; third, there are those which build a new list composed of specified values; and finally those which build a new set (user's set) composed of specified values.

There are three types of mapping functions. The three types are those that process every element of a set (map type), those that process until a value is false (and type) and those that process until a value is not false (or type). Every mapping function is in one class and is of one type. Thus we can describe the set of all mapping functions using a chart (see Figure 2.8).

A mapping function requires a system set p as its first parameter, a function ft as its second parameter, and zero or more additional parameters. Consider the following example of a mapping function which removes all of the arcs emanating from a particular node as a typical mapping operation. The transformation of Figure 2.9 would be accomplished by dmapft(p,true) where p = rseto(w) = {a,b,c} and true(x) = x for all x. What actually happens with dmapft is that for each $p_i \in p$, true(p_i) is processed and if true(p_i) $\neq$ false (as it always is) then p_i is removed from p. Note that had the function been dorft(p,true) then just the first arc would be removed (see Figure 2.10).

Figure 2.11 presents a definition of all the mapping functions, and Figure 2.12 shows some of the versatility of the mapping functions by illustrating each with six different functions. In Figure 2.12, the contents of each position is the value of f(p,ft) unless that position contains "p = ____" which means that p, the rseto of w, has perhaps been side-effected. The examples (dmapft,true) [dmapft(p,true)] and (dorft,true) [dorft(p,true)] from Figure 2.12 are illustrated by Figures 2.9 and 2.10. Note that the

class \ type	map	and	or
	mapft	andft	orft
d	dmapft	dandft	dorft
ℓ	ℓmapft	ℓandft	ℓorft
s	smapft	sandft	sorft

Figure 2.8. Table of Mapping Functions by class and type

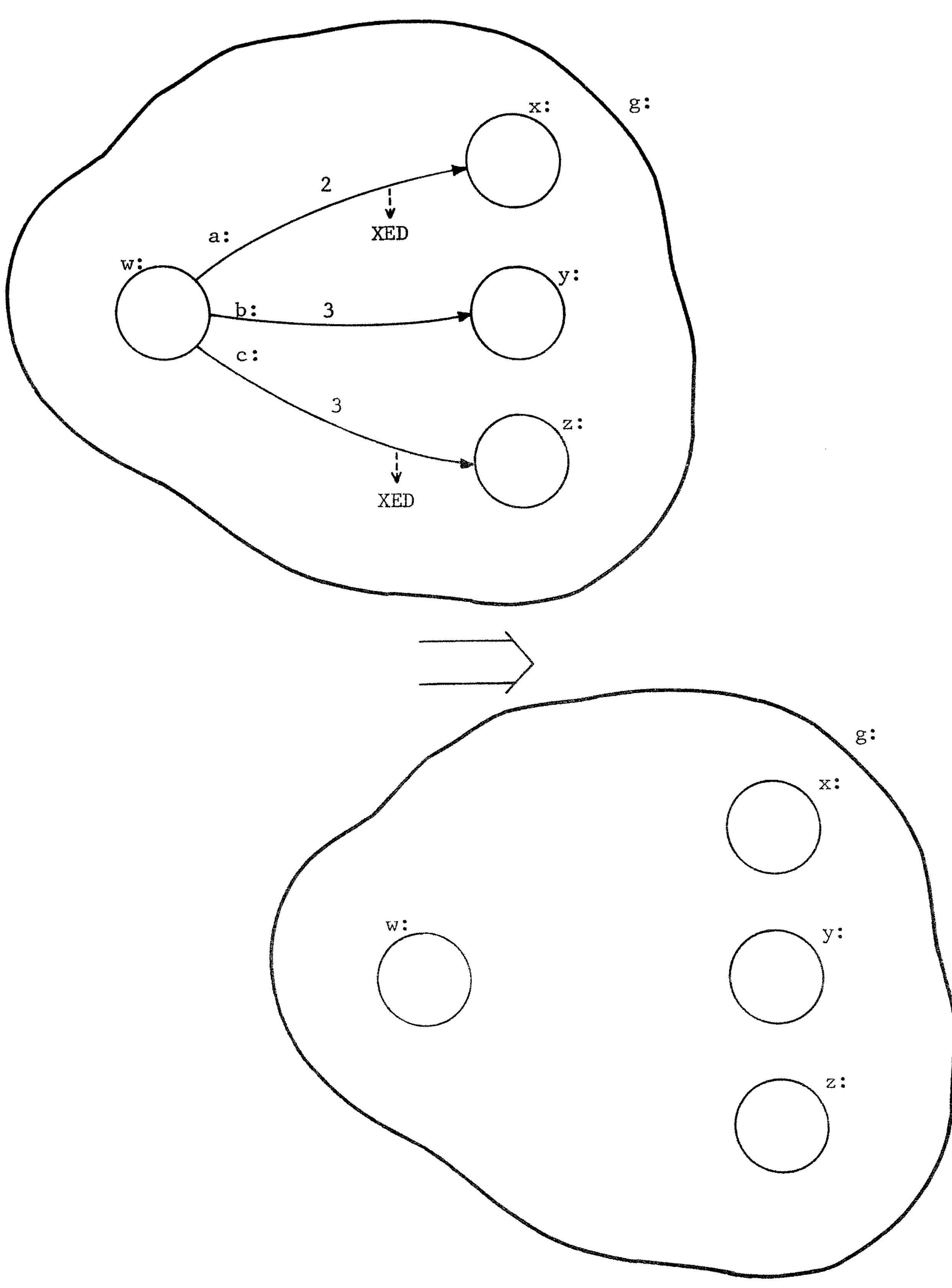


Figure 2.9. dmapft(rseto(w),true)

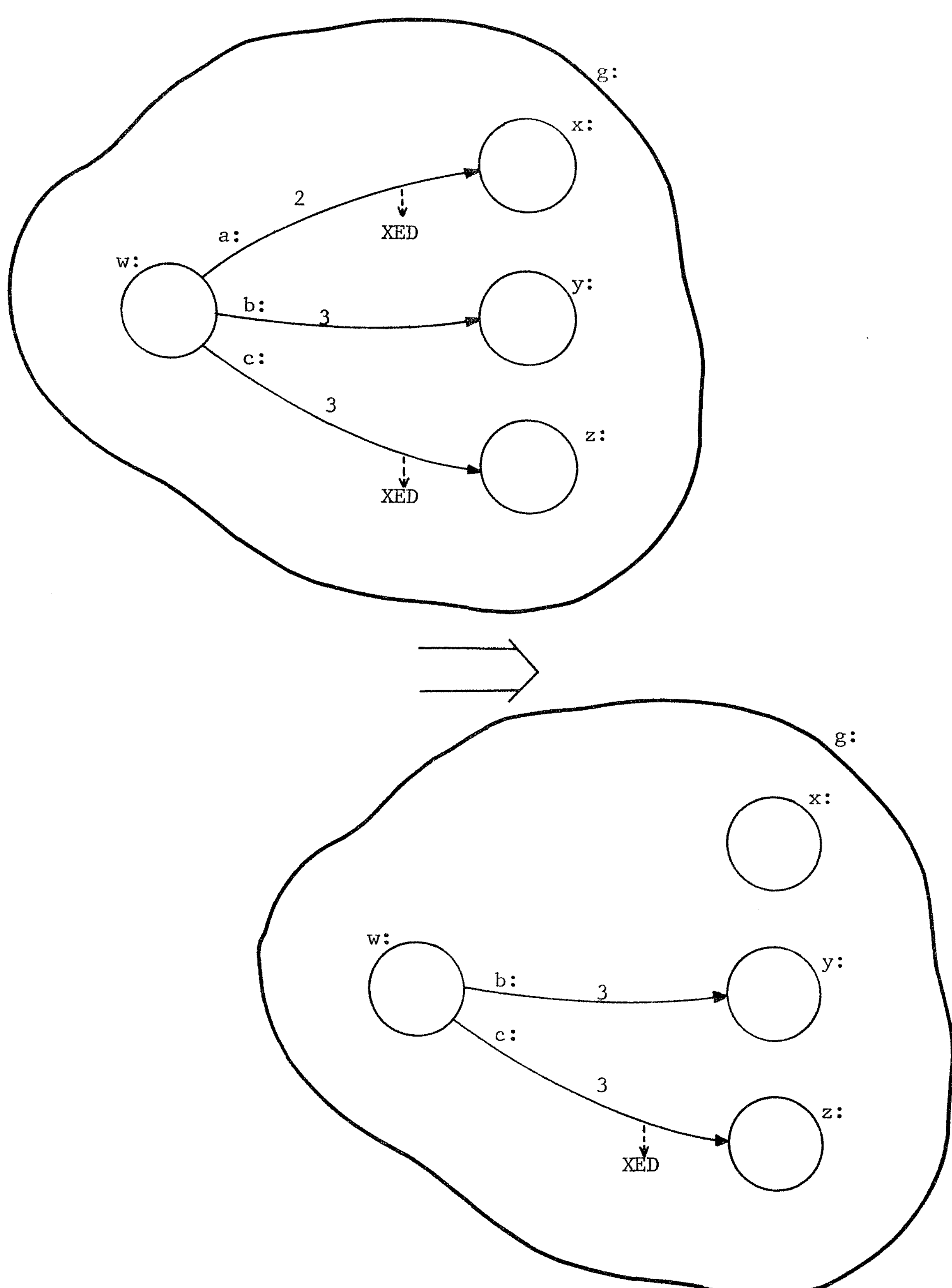


Figure 2.10. dorft(rseto(w),true)

f	v: false — control	v: not false — action	v: not false — control	q
mapft	-	no action	-	v or p if p = {}
dmapft	-	$p = p - \{p_i\}$	-	p
ℓmapft[1]	-	$t = t\ .\ (v)$	-	t
smapft[1]	-	$t = t \cup \{v\}$	-	t
andft	stop	no action	-	v or p if p = {}
dandft	stop	$p = p - \{p_i\}$	-	p
ℓandft[1]	stop	$t = t\ .\ (v)$	-	t
sandft[1]	stop	$t = t \cup \{v\}$	-	t
orft	-	no action	stop	v or false if p = {}
dorft	-	$p = p - \{p_i\}$	stop	p
ℓorft[1]	-	$t = t\ .\ (v)$	stop	t
sorft[1]	-	$t = t \cup \{v\}$	stop	t

Figure 2.11. Definition of the Mapping Functions

Each mapping function, f, has the same calling sequence, $q = f(p, ft, arg_2, \ldots, arg_k)$ where p is a system-set[2] $\{p_1, \ldots, p_m\}$. The elements of p are sequentially processed and return the value $v = ft(p_i, arg_2, \ldots, arg_k)$.

\- means that if all the elements of the system-set have been processed then stop, otherwise process the next element.

[1] t is a newly created list or set depending upon the class.

[2] Actually p can also be a list or set.

. is concatenation of two lists - $(v_1, v_2, \ldots, v_k)\ .\ (u_1, u_2, \ldots, u_j)$ forms the list $(v_1, v_2, \ldots, v_k, u_1, u_2, \ldots, u_j)$.

f \ ft	frnode	tonode	object	value	true	false
mapft	w	z	3	XED	c	false
dmapft	p = { }	p = { }	p = { }	p = {b}	p = { }	p={a,b,c}
ℓmapft	(w w w)	(x y z)	(2 3 3)	(XED XED)	(a b c)	()
smapft	{w}	{x y z}	{2 3}	{XED}	{a b c}	{ }
andft	w	z	3	false	c	false
dandft	p = { }	p = { }	p = { }	p = {b,c}	p = { }	p={a,b,c}
ℓandft	(w w w)	(x y z)	(2 3 3)	(XED)	(a b c)	()
sandft	{w}	{x y z}	{2 3}	{XED}	{a b c}	{ }
orft	w	x	2	XED	a	false
dorft	p = {b,c}	p = {b,c}	p = {b,c}	p = {b,c}	p = {b,c}	p={a,b,c}
ℓorft	(w)	(x)	(2)	(XED)	(a)	()
sorft	{w}	{x}	{2}	{XED}	{a}	{ }

$$\text{true}(p_i) = p_i$$

$$\text{false}(p_i) = \text{false}$$

Figure 2.12. Versatility of the Mapping Functions

user of GROPE may pass any function as the second parameter to any mapping function including those that the programmer writes himself.

Experience has shown that the mapping functions of GROPE are sufficient for most graph problems. There are two aspects which need to be emphasized about mapping functions. First, there is the aspect that graph processing is enhanced by the mapping functions and that in fact the mapping functions are merely support routines. Second, the mapping functions are built-in control structures represented functionally. It is this second fact which allows a programmer to use mapping functions and avoid numerous logical errors.

In the remainder of this section we discuss some aspects of the reader mechanism of GROPE. As an illustration of the reader mechanism we present an algorithm for the definition of all the mapping functions.

System Set Readers

In the event that the mapping functions are insufficient, there are mechanisms which allow for a step-by-step processing of structures. These mechanisms, called linear readers, exist for the purpose of searching system sets. Every system set has a finite number of components and the reader mechanism allows for their processing, one element at a time, in the analysis of an algorithm.

There is an operation which creates a reader of a system set. Thus r = creedr(p) where p is a system set causes r to be a reader of that system set. Next, given any reader of any system set, the operation traverse out (to), has two separate responsibilities. First, to advances the reader to the next component in p, and second the value returned by to is that component. For example, using t = to(r), the first execution causes t to be p_1,

the second causes t to be p_2, etc. Thus we can write an algorithm to find the nth component of any system set. Let p be a system set; then Figure 2.13 represents an algorithm which terminates with t as the nth component of p.

There are two ways to determine when a reader has read the mth (recall that $p = \{p_1, p_2, \ldots, p_m\}$) component in a system set. The first way is to note that there is a function which, given a system set p, determines the number of components in p. The function is called _length_. The second mechanism is a predicate that determines whether or not a reader has just read the last component in a system set. The predicate is termed _isatnd_ (which asks if the reader _is at_ the e_nd_ of the system set). Thus two ways of searching for every element in a non-empty system set p are given by the algorithms depicted by Figure 2.14 and Figure 2.15. Note that Figure 2.13 and Figure 2.14 are effectively the same.

In the definition of the mapping function, _mapft_, the algorithm using _isatnd_ is employed because, for example, creations (with the ft) might increase the length of the system sets. Recall that $x = mapft(p, ft, arg_2, \ldots, arg_k)$ is the standard parameter sequence of the function _mapft_. Figure 2.16 gives the definition of _mapft_. Note that in this definition, _length_ is used as a predicate to determine whether or not the system set p is non-empty. Figure 2.17 is the definition of all the mapping functions (recall Figure 2.8 for the definition of _type_ and _class_ predicates).

It is expected that for most problems, the mapping functions suffice and the user of GROPE should make every effort to become familiar with the mapping functions and to relegate the reader creation and _to_ operations to secondary consideration.

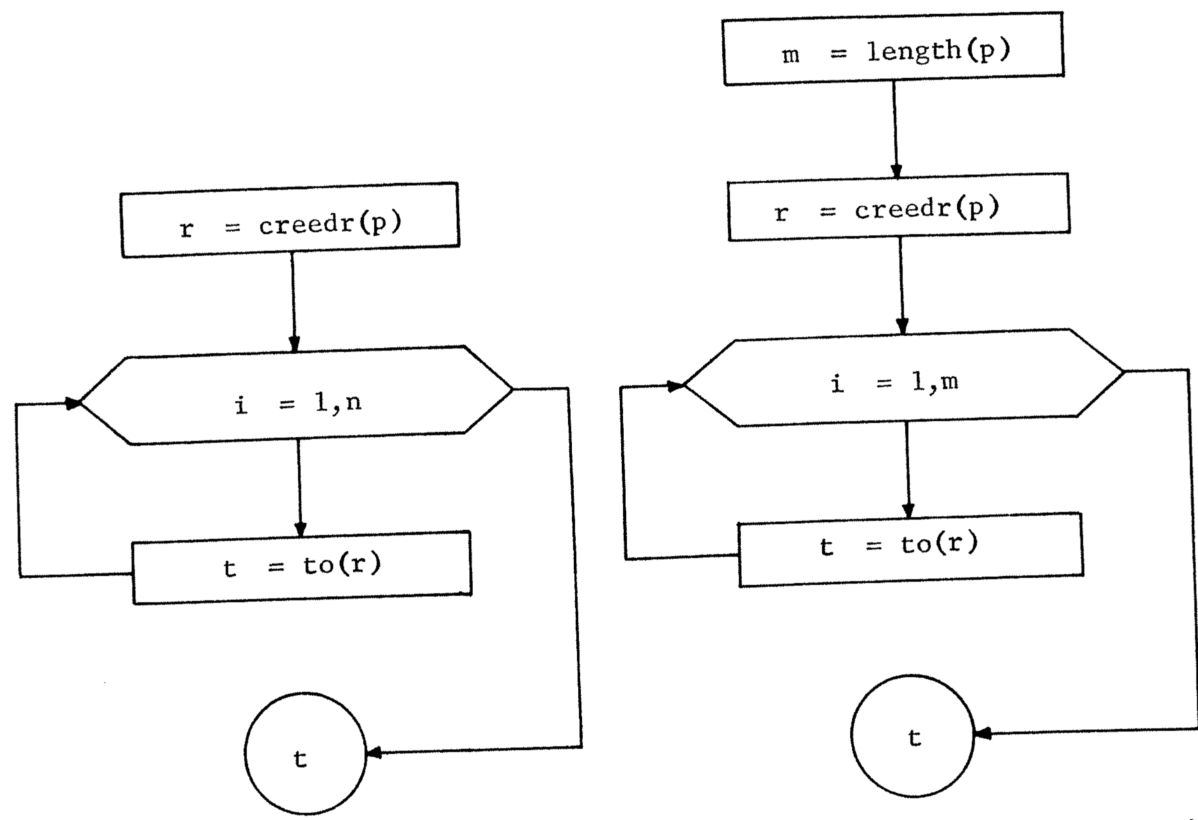


Figure 2.13. The nth Component of p
Component (p,n)

Figure 2.14. The Last Component of p

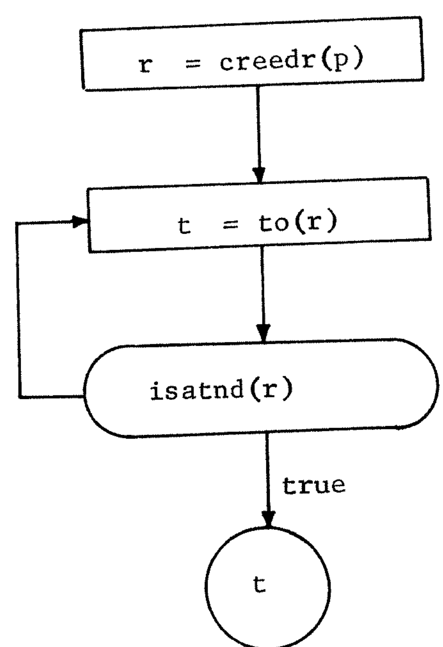


Figure 2.15. The Last Component of p Allowing for p to Be Altered

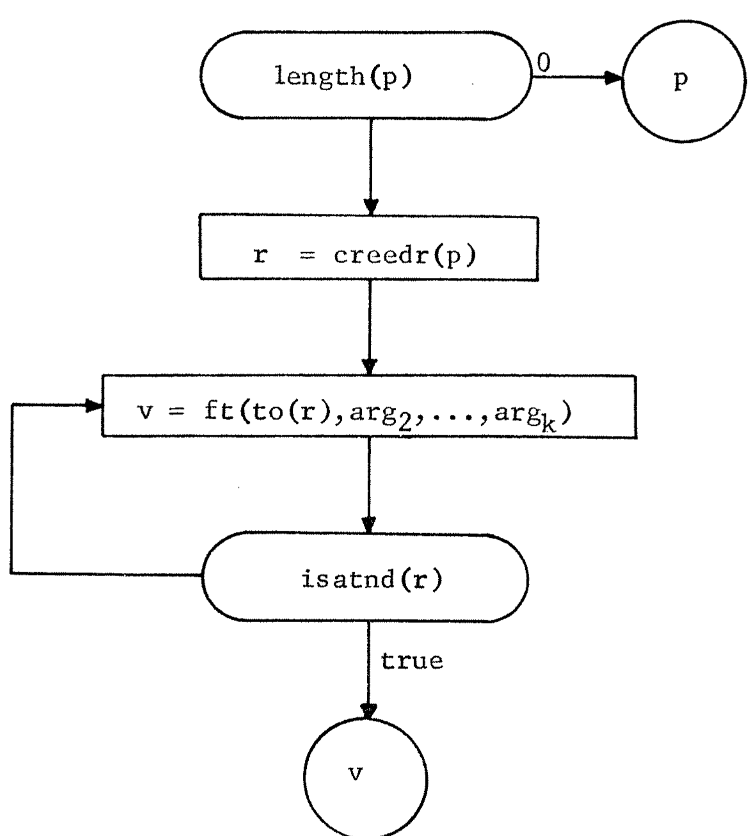


Figure 2.16. mapft(p,ft,arg_2,...,arg_k)

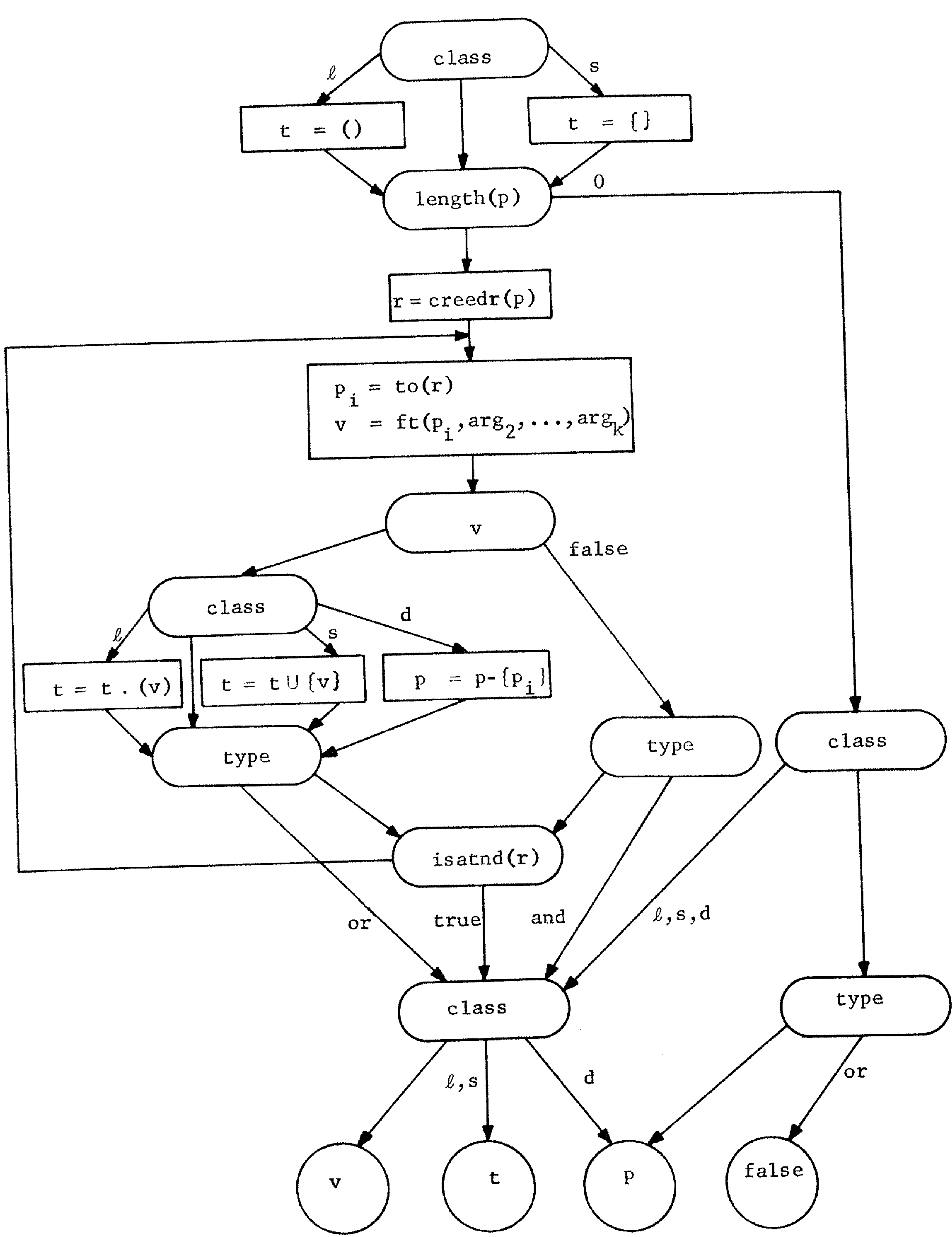


Figure 2.17. $f(p, ft, arg_2, \ldots, arg_k)$

Generalizations

In the preceding three sections we have been dealing primarily with very elementary relationships. In this section we present some of the more subtle aspects of GROPE. Specifically, we shall discuss the notion of a one-way arc, complex graph-based structures, sophisticated graph readers, and the graph modification operations. Most graph processing algorithms can exist without these features; however, it is these features of GROPE that make GROPE unique. It is these relationships which make graph processing an exciting, challenging, and rewarding (both conceptually and productively) experience. A programmer can master the first three sections and yet not fully understand what we mean by "graph processing."

One-way Arcs

The removal of restrictions is a very natural way to incorporate generality. This paradigm is one of the cornerstones of GROPE. In the example of one-way arcs we use this paradigm. The restriction is that an arc joining nodes n and m is *necessarily* in the *rseto*(n) (the set of out pointing arcs leaving the node n) and in the *rseti*(m) (the set of incoming arcs to the node m). Although for many algorithms this restriction causes no problem, by removing this restriction we obtain a more complete class of structures. We still maintain that an arc has a *frnode* (the node from which the arc emanates), a *tonode* (the node to which the arc points), an *object* and a *value*. But it is possible to find an arc in the *rseto* of some node which is not in the *rseti* of *any* node, and which thus conceptually can be traversed only in the direction it points. Thus, pictorially, we denote

this type of arc as n: ◯ a:→ ◯ m: where rseto(n) = {a} and rseti(m) = {}. A natural use for "one-way-out" arcs would be for interpreters of flow graphs or as a simulator of abstract automata represented by state transition graphs.

More unusual than the "one-way-out" arc is the "one-way-in" arc. This type of arc is pictured as n: ◯ a:→ ◯ m: where rseti(m) = {a} and rseto(n) = {}. Such arcs are useful when one wishes to traverse an arc and then make it impossible to traverse the arc again if the graph contains a loop.

Thus there are one-way-out arcs, one-way-in arcs, and regular arcs (where a ϵ rseto(n) and a ϵ rseti(m)). It is important to note that one graph may contain any combination of these arcs.

Complex Graph-based Structures

On the surface, nothing has been presented which directly allows for any kind of sophisticated data structures. But with the removal of one more restriction, we can enter the world of complex graph-based structures. We remove the restriction that objects and values must be atoms and allow objects and values to be arcs, nodes, and graphs as well. With this generalization, we can consider Figures 2.18 and 2.20 as typical illustrations. 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. Figure 2.19 and Figure 2.21 present the associated retrieval information for Figures 2.18 and 2.20, respectively.

These complex graph-based structures have a number of potential uses. For example, by allowing the values of nodes to be graphs, these structures

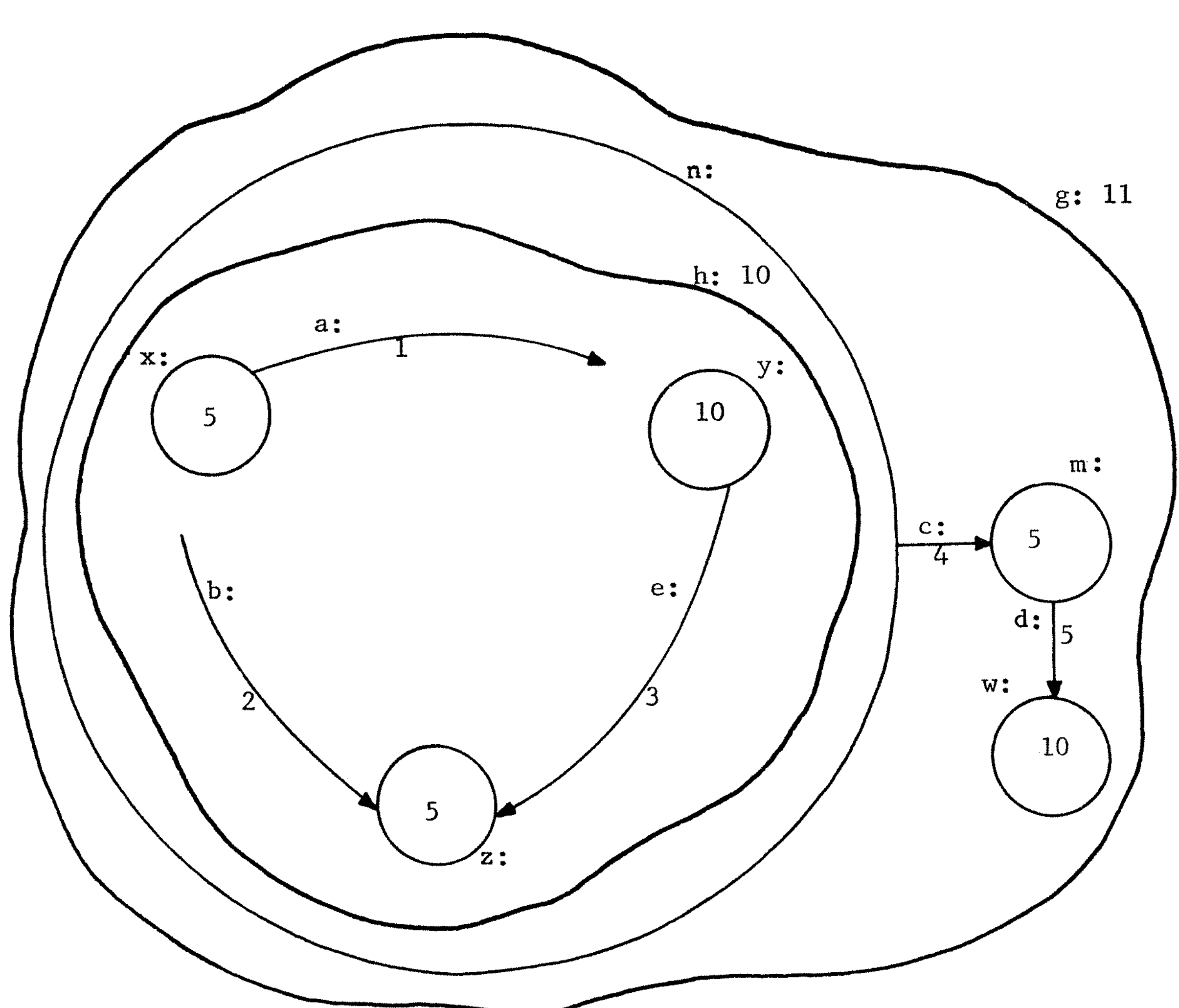


Figure 2.18. A Complex Graph-based Data Structure

		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		
object		11	10	10	5	10	5	h	5	1	2	4	5	3		
value																
grset	{g, h}															
ndset		{n,m,w}	{x,y,z}													
nset			{n}												{x,z,m}	{y, w}
rseto				{}	{a}	{e}	{}	{c}	{d}							
rseti				{d}	{}	{}	{b, e}	{}	{c}							

Figure 2.19. Retrievals for a Complex Graph-based Data Structure

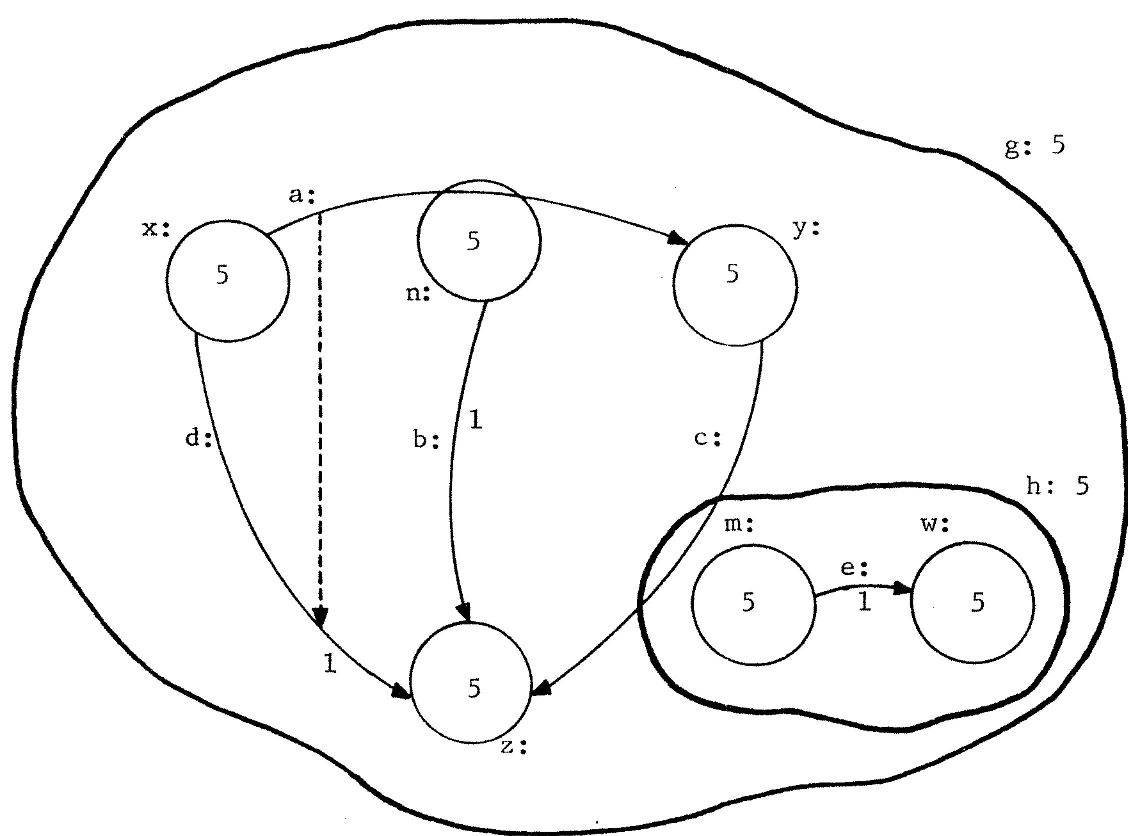


Figure 2.20. Another Complex Graph-based Data Structure

		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		
object		5	5	5	5	5	5	5	5	n	1	h	1	1		
value										d						
grset	{g,h}															
ndset		{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}	{}	{}							

Figure 2.21. Retrievals for Another Complex Graph-based Data Structure

simulate hierarchical graphs. Nested finite automata (the Woods machine [44]) can be represented by allowing the value of arcs to be graphs.

The Graph Reader Mechanism

The graph reader is a new idea in graph processing. This mechanism allows searching graph structures in a controlled manner. As a graph reader moves across an arc from one node to another node, it "ages" the arc it crosses. By this we mean that the arc which was crossed becomes the "oldest" arc leaving a node. The arc that the graph reader mechanism had chosen was the "youngest" arc.

Actually, every node which has a non-empty rseto, also has one of these arcs as the current-arc-out. Since the system sets are ordered, it is possible to retrieve the next arc after the current-arc-out. This next arc is chosen and it becomes the new current-arc-out. Similarly, when an arc is crossed in an in direction, the current-arc-in is affected. The graph traversal operation requires a graph reader. A graph reader is created with some node on the graph. The operation r = creedr(v) where v is some node, creates a graph reader r. The operation to(r) traverses the reader in an out direction and ti(r) traverses the reader in an in direction.

There is a function, curarc, which has as its value the latest arc crossed by any graph reader. If a + is placed on an arc when it is the current-arc-out and a - on the current-arc-in, then the algorithm in Figure 2.22 produces the results shown in Figure 2.23.

At this point very little experience has been gained using the graph reader, and it would be unreasonable to make any generalizations about this tool. Suffice it to suggest that the memory used within the node for the current-arc-out and current-arc-in appears to be a good way

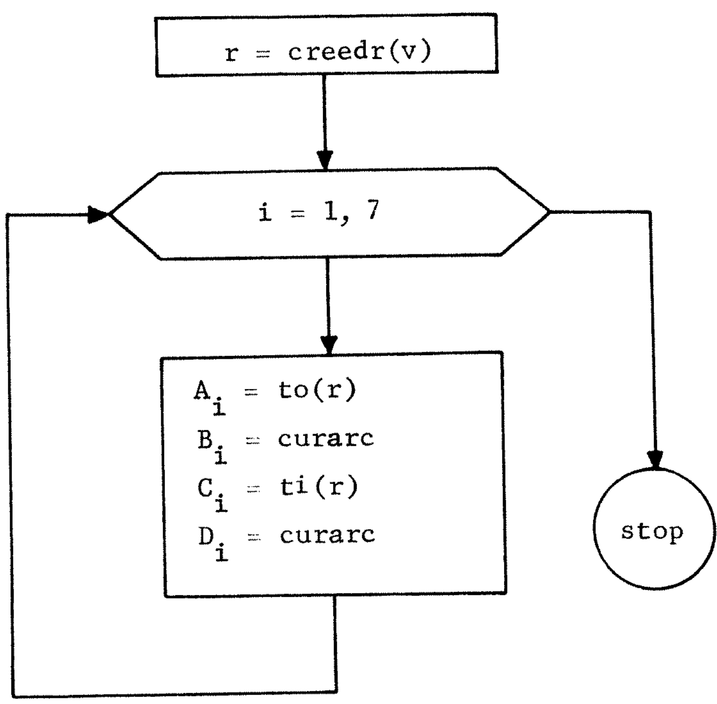
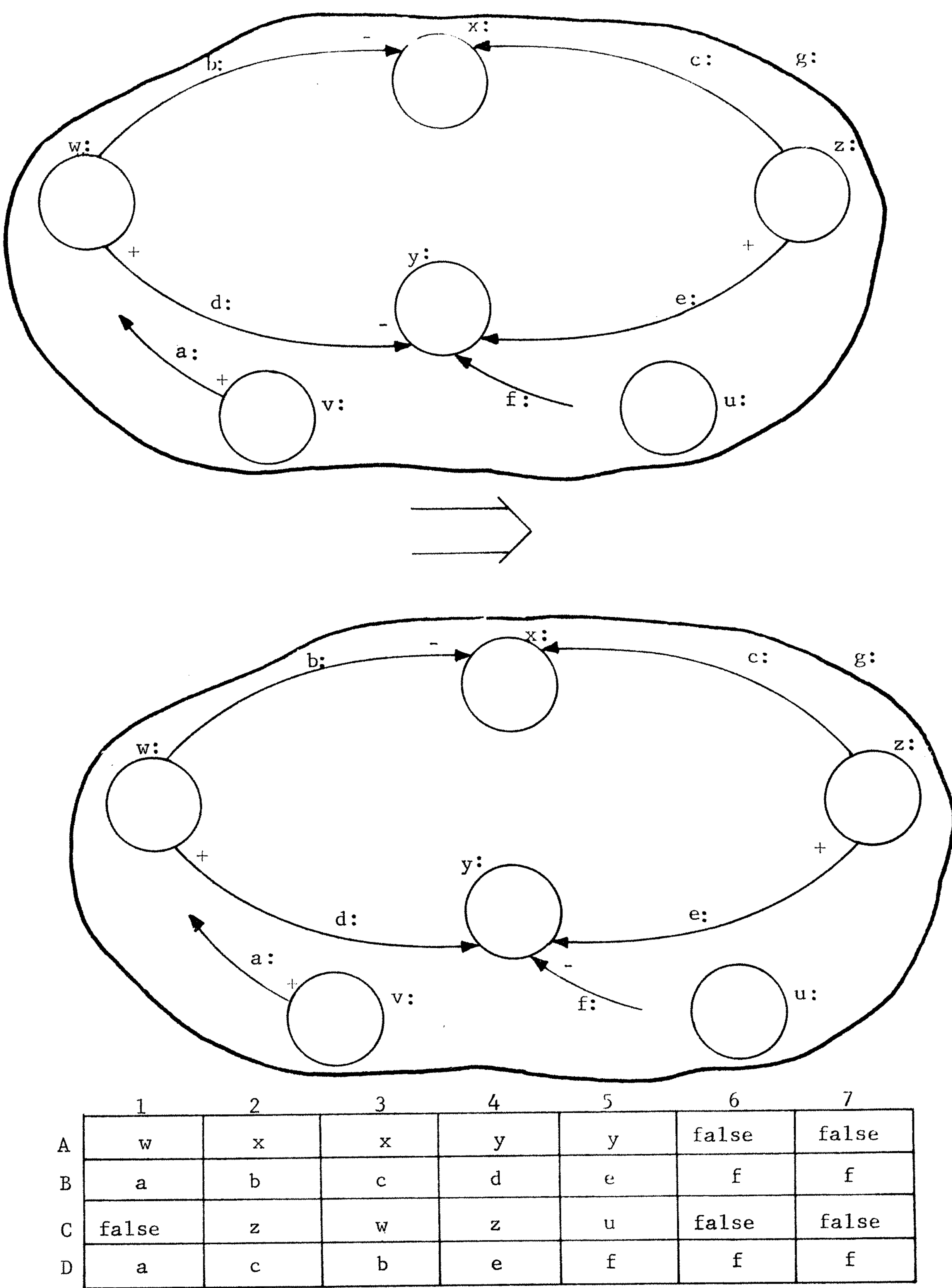


Figure 2.22. Traversing with the Graph Reader



	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 2.23. Data and Results of Algorithm

of automatically keeping a path between nodes on certain types of graphs. We look forward to some experimentation with graph readers to determine the relative flexibility of this tool within the context of graph processing.

Graph Modification Functions

The "changing functions" or structural modification functions introduce us to another aspect that makes GROPE very general. One of the obvious changing functions is the function hang. All that hang(x,y) does is simply hang the value y from the structure x. A more interesting changing function is the function chafrn, which changes the frnode of an arc. Here chafrn(x,y) causes y to be the node from which the arc x emanates. Consider the implication of this operation using mapft(p,chafrn,w) where p = rseto(n) = {a,b,c}. See Figure 2.24 for an illustration of this transformation.

When we consider that chafrn requires no searching, although possibly a deletion and an insertion, it is pleasing to note that the cost of this transformation is a small constant times the length(p). In fact, there are no searches in any of the changing functions.

Other changing functions change the tonode of an arc, the graph of a node, and the object of a graph, node, or arc. Consider the function chagr, which changes the graph upon which a node resides. Any arcs attached to the node on the original graph remain attached to the node on the new graph. Thus note that an arc from a node on one graph to a node on another graph is perfectly acceptable. See Figure 2.25 for an illustration of this transformation. One of the uses of such structures, as a natural generalization, is the representation of a graph by its subgraph structure with

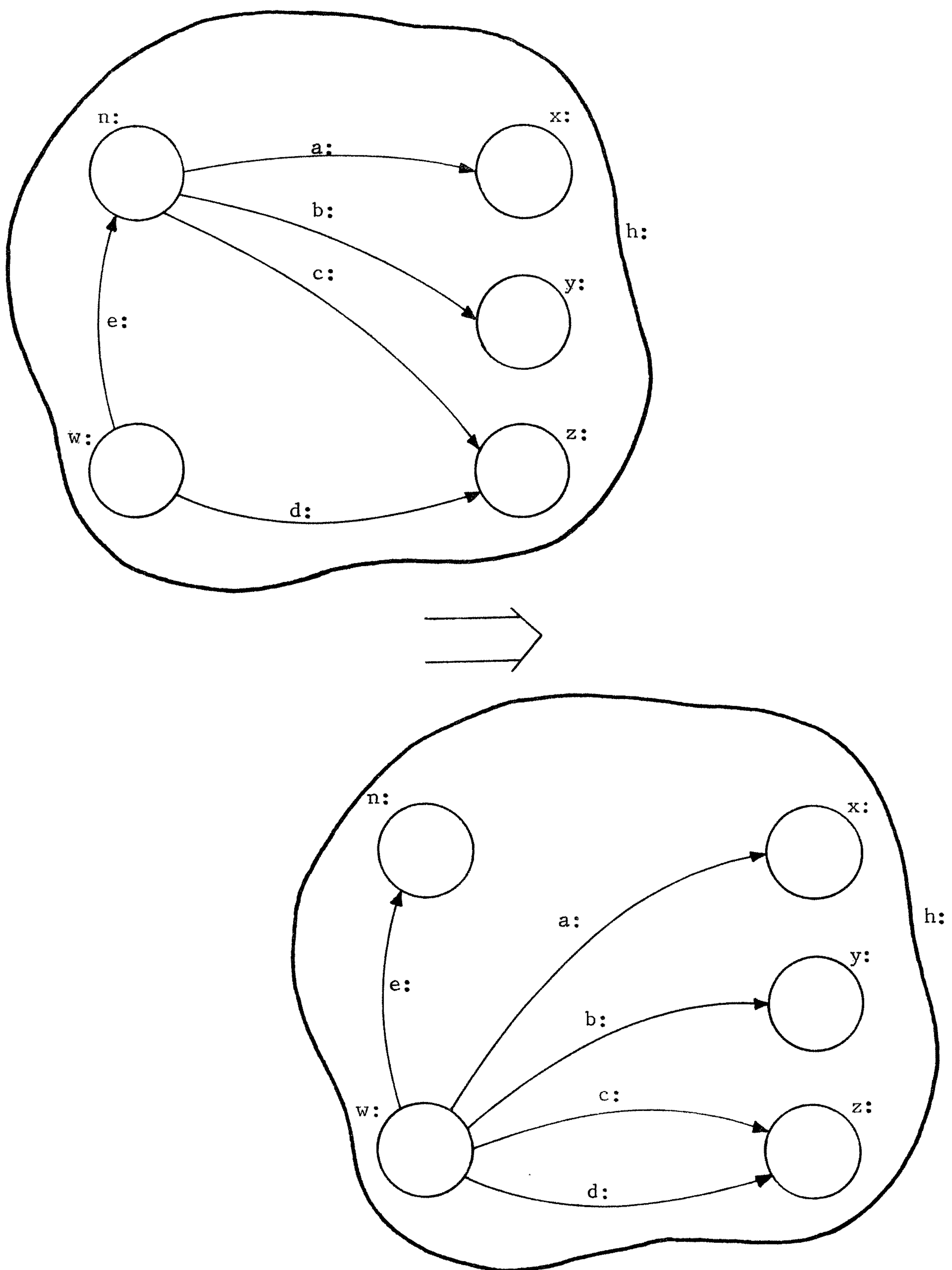


Figure 2.24. mapft(rseto(n),chafrn,w)

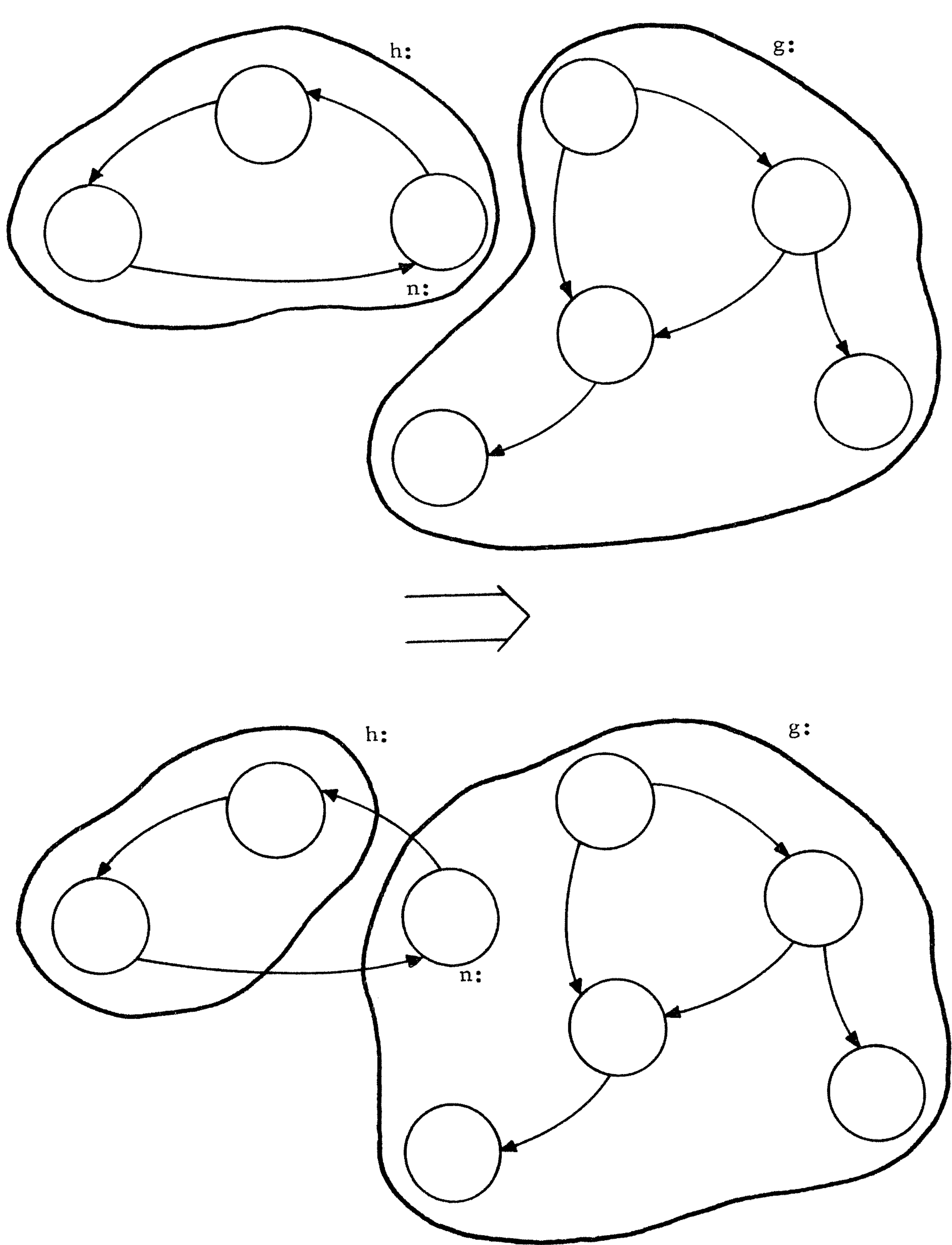


Figure 2.25. Changing the Graph of a Node

arcs connecting nodes from one subgraph to another. See Figure 2.26 for an illustration of a graph represented by subgraphs. We believe that, as graph processing algorithms get more sophisticated, these efficient changing (requiring no creations, no destructions, and no searches) functions--as well as all the generalizations mentioned in this section--will play a major role in reducing some of the combinatorial aspects of graph processing.

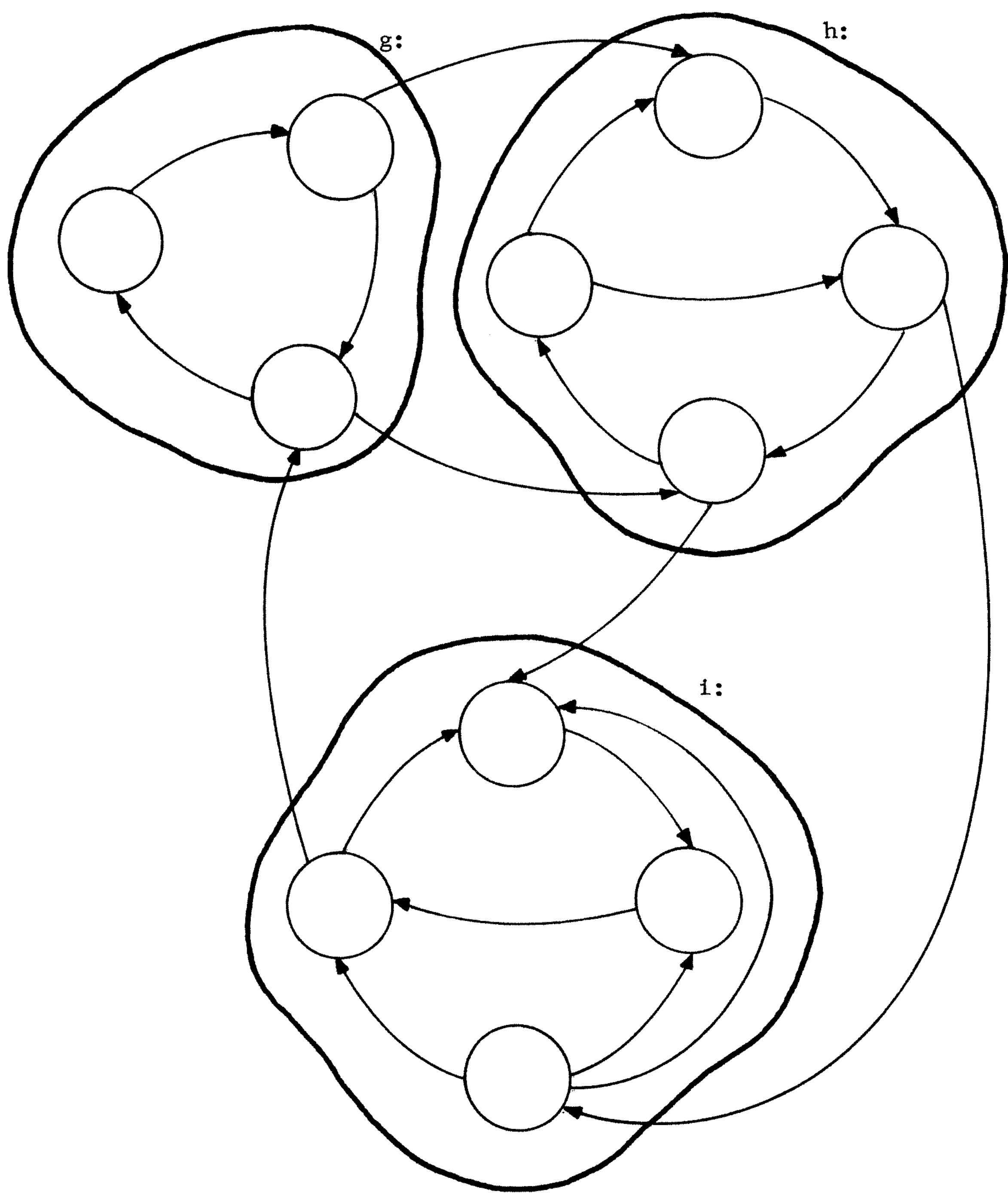


Figure 2.26. Representation of a Graph by Subgraphs

CHAPTER III

MACRO-SEMANTICS OF GROPE

In this chapter the macro-semantics of GROPE are developed. Recall that by "macro-semantics" we mean a formal definition which is designed to provide user understanding. Just as Chapter II is a description of GROPE from a user's point of view, so are the definitions presented in this chapter. In the next chapter we concern ourselves with the formal implementation level definitions, the micro-semantics.

As mentioned earlier, the GROPE extension that has been chosen for formal specification is somewhat different than the actual programming language extension, GROPE. For the most part, the formal GROPE is a slightly simplified version of the actual GROPE. From here on the term "GROPE" will be used for both GROPEs; however, in Chapters III and IV it will generally refer to the formal GROPE.

At this point we have discussed some of the highlights of GROPE (Chapter II) and have given an informal introduction to the "abstract system" approach (Chapter I) to formal definition of programming languages. In this chapter we present the macro-semantics of GROPE as an example of the "abstract system" approach. This chapter is composed of four sections. In the first section a formal statement of the "abstract system" approach is presented; in the second section a sample abstract system for the macro-semantics is defined; in the third section the data structures for the macro-semantics are described; and in the fourth section operations over these data structures are given.

The "Abstract System" Approach

Recall from Chapter I that "abstract system" definitions of programming languages are composed of three basic aspects. First, a total data space is defined; then data structures are described; and finally, operations over the data structures are presented. Figure 3.1 defines the concepts abstractly. Our treatment of the macro-semantics and micro-semantics follows a similar pattern.

In the macro-semantics the total data space is an abstract system, gds; the data structures are described by the definition of bodies (ordered n-tuples); and the fixed set of operations are developed by giving the transition rule for generating new states (gds's). In the micro-semantics the total data space is an "abstract system," GDS. The data structures are described by the definition of the functions ARCS and TYPE, and the fixed set of operations once again are developed by giving the transition rule for generating new states (GDS's). Thus our definitional technique defines the same fixed set of operations over two different "abstract systems" with two different conceptual data structures (i.e. one for the users and one for the implementers).

The Total Data Space of the Macro-semantics

The first phase of the formal definition of programming languages is to describe the total data space. The gds (graph data structure) defined below is the total data space of the macro-semantics.

A gds is a pair (L,body) where

i. L is a countably infinite set of location points. L is partitioned into two sets, L_0 (the unused locations) and

Let $H = (S_1, S_2, \ldots, S_j, F_1, F_2, \ldots, F_k)$ be an abstact system where

1. For all i such that $1 \leq i \leq j$, S_i is a set.
2. For all i such that $1 \leq i \leq k$, F_i is a function.

An operation f defined over H with parameters $x_1, x_2, \ldots, x_n$ is given by $f(H, x_1, x_2, \ldots, x_n) = (H', v)$ where $v, x_i \in S_1 \cup S_2 \cup \ldots \cup S_j$ and $H' = (S'_1, S'_2, \ldots, S'_j, F'_1, F'_2, \ldots, F'_k)$

For notational convenience and naturalness, H and H' are considered implicitly. Hence the operation, f, becomes the familiar $f(x_1, x_2, \ldots, x_n) = v$.

Figure 3.1. Formal Specification of the "Abstract System" Approach

a finite set V. V is further partitioned into sets E, G, N, A, C, and $\{\lambda\}$. Initially, V contains λ, the null value.

ii. body is a function such that

$$\text{body}: \begin{cases} E \rightarrow \{x : x \text{ is an element-body}\} \\ G \rightarrow \{x : x \text{ is a graph-body}\} \\ N \rightarrow \{x : x \text{ is a node-body}\} \\ A \rightarrow \{x : x \text{ is an arc-body}\} \\ C \rightarrow \{x : x \text{ is a cursor-body}\} \end{cases}$$

Elements of the sets E, G, N, A, and C are termed elements, graphs, nodes, arcs, and cursors, respectively. Note that if $x, x' \in v$ with $x \neq x'$, then it is still possible for body(x) = body(x'). For example, two arcs may have equivalent arc-bodies and still be different arcs.

The Data Structures of the Macro-semantics

The second phase of the formal definition of programming languages is the description of the data structures. The data structures of the macro-semantics of GROPE are bodies (ordered n-tuples). For each data type (element[1], graph, node, arc, and cursor[2]) there is a body type defined which contains the necessary components for describing the macro-semantics of GROPE. When a body contains an ordered k-tuple as a component, then the ordered k-tuple is a system set (Chapter II). The nset is denoted by N^-, the ndset is denoted by N^+, the rseti is denoted by A^- and the rseto is denoted by A^+.

---

[1]Element is synonymous with atom in the actual GROPE.

[2]Cursor is synonymous with reader in the actual GROPE.

The Bodies

An element-body is a pair (v,N^-) where

$v \in V$, the value

$N^- \in \{\lambda\} \cup \{(y_1,\ldots,y_k) : y_i \in N \text{ for all } 1 \leq i \leq k\}$, the attached set of nodes. For $i \neq j$, then $y_i \neq y_j$ and for an element, e, if $y \in N_e^-$, then b_y is e (where b_y is the b-component of the node-body of y).

A graph-body is a pair (v,N^+) where

$v \in V$, the value

$N^+ \in \{\lambda\} \cup \{(x_1,\ldots,x_k) : x_i \in N \text{ for all } 1 \leq i \leq k\}$, the related set of nodes. For $i \neq j$, then $x_i \neq x_j$ and for a graph, g, if $x \in N_g^+$, then u_x is g (where u_x is the u-component of the node-body of x).

A node-body is a septuple (v,u,b,A^+,a^+,A^-,a^-) where

$v \in V$, the value

$u \in G$, the origin (the graph of residence)

$b \in E$, the object

$A^+ \in \{\lambda\} \cup \{(x_1,\ldots,x_k) : x_i \in A \text{ for all } 1 \leq i \leq k\}$, the related set of arcs. For $i \neq j$ then $x_i \neq x_j$ and for a node, n, if $x \in A_n^+$, then u_x is n (where u_x is the u-component of the arc-body of x).

$a^+ \in \{\lambda\} \cup A^+$, the current-arc-out

$A^- \in \{\lambda\} \cup \{(y_1,\ldots,y_k) : y_i \in A \text{ for all } 1 \leq i \leq k\}$, the attached set of arcs. For $i \neq j$ then $y_i \neq y_j$ and for

a node, n, if $y \in A_n^-$, then b_y is n (where b_y is the b-component of the arc-body of y).

$a^- \in \{\lambda\} \cup A^-$, the current-arc-in.

An arc-body is a triple (v,u,b) where

$v \in V$, the value

$u \in N$, the origin (the node from which an arc emanates)

$b \in N$, the object (the node at which an arc terminates).

A cursor-body is a triple (v,u,b) where

$v \in V$, the value

$u \in N \cup A$, the origin

$b \in N \cup A$, the object.

If $b \in N$, then $u \in N$; and if $b \in A$, then $u \in A$.

In order to illustrate a simple gds consider a graph composed of two nodes and one arc. Figure 3.3 is a formal statement of the gds of Figure 3.2.

Operations in the Macro-semantics

The GROPE primitive operations are partitioned into five classes. The first class contains all of the creation operations along with their associated predicates which determine whether or not a value is of a certain type. The second class is composed of the basic retrieval operations. The third class contains the "state" changing operations. The fourth class contains the structural changing operations, and the fifth class is the cursor (reader) traversal operations.

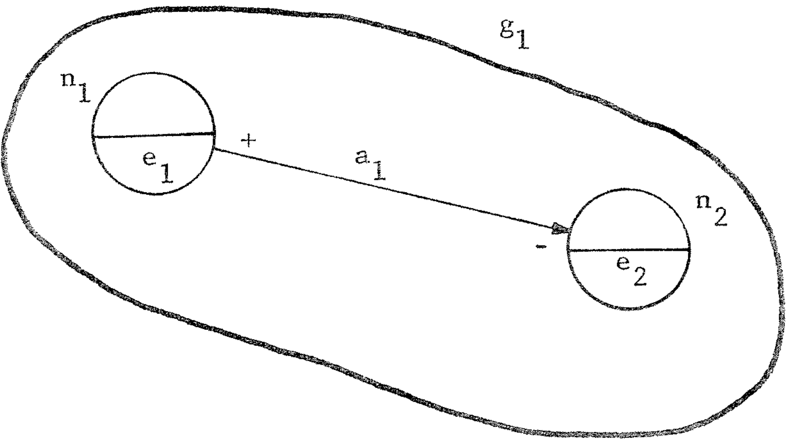


Figure 3.2. A Simple Graph (or <u>gds</u>)

$H = (L_0 \cup V, \underline{body})$ where $V = E \cup G \cup N \cup A \cup C \cup \{\lambda\}$, $C = \emptyset$

$E = \{e_1, e_2\}$

$\underline{body}(e_1) = (\lambda, (n_1))$

$\underline{body}(e_2) = (\lambda, (n_2))$

$G = \{g_1\}$

$\underline{body}(g_1) = (\lambda, (n_1, n_2))$

$N = \{n_1, n_2\}$

$\underline{body}(n_1) = (\lambda, g_1, e_1, (a_1), a_1, \lambda, \lambda)$

$\underline{body}(n_2) = (\lambda, g_1, e_2, \lambda, \lambda, (a_1), a_1)$

$A = \{a_1\}$

$\underline{body}(a_1) = (\lambda, n_1, n_2)$

Figure 3.3. $H = \underline{gds}$ of a Simple Graph

1. Creation operations and associated predicates

(Recall that each function definition has a gds, H, as an implicit argument and a modified gds, H', as an implicit result.)

a. create-element() = r where $r \in L$. Let $r \in L_0$. Place $r \in E$, remove r from L_0 and define body(r) = (λ,λ).

b. is-element(*) = $\begin{cases} * \text{ if } * \in E \\ \lambda \text{ otherwise} \end{cases}$.

c. create-graph() = r where $r \in L$. Let $r \in L_0$. Place $r \in G$, remove r from L_0 and define body(r) = (λ,λ).

d. is-graph(*) = $\begin{cases} * \text{ if } * \in G \\ \lambda \text{ otherwise} \end{cases}$.

e. create-node(*,**) = r where $* \in G$, $** \in E$, and $r \in L$. Let $r \in L_0$. Place $r \in N$, remove r from L_0, and define body(r) = $(\lambda,*,**,\lambda,\lambda,\lambda,\lambda)$.

f. is-node(*) = $\begin{cases} * \text{ if } * \in N \\ \lambda \text{ otherwise} \end{cases}$.

g. create-arc(*,**) = r where $*,** \in N$ and $r \in L$. Let $r \in L_0$. Place $r \in A$, remove r from L_0, and define body(r) = $(\lambda,*,**)$.

h. is-arc(*) = $\begin{cases} * \text{ if } * \in A \\ \lambda \text{ otherwise} \end{cases}$.

i. create-cursor(*) = r where $* \in N \cup A$ and $r \in L$. Let $r \in L_0$. Place $r \in C$, remove r from L_0, and define body(r) = $(\lambda,*,*)$.

j. is-cursor(*) = $\begin{cases} * \text{ if } * \in C \\ \lambda \text{ otherwise} \end{cases}$.

2. Retrieval operations

The group of operations referred to as the retrieval operations can, for the most part, be discerned directly from the data structures of the gds. There are, however, aspects that at this point could bear clarification.

The operation value is primarily to allow for the development of hierarchical structures. For example, a node whose value is a graph might very well be a representation of a list structure. The value of a cursor may be another cursor, thus allowing for the construction of a stack of cursors simulating a SLIP [40] reader. The operation, value, is also, of course, a natural mechanism for retrieving constants associated with nodes and arcs.

The operation object is of a more specialized nature than that of value. The object of a node must be an element and the object of an arc must be a node. The operation attach (in class 3) ties together those nodes and arcs with the same object.

Much that can be said about object is true also for origin with slight variations. The origin of a node must be a graph and the origin of an arc must be a node. The operation relate (in class 3) ties together those nodes and arcs with the same origin. A more complex situation exists for the origin and object of a cursor. When the origin of a cursor is a node, then the object of that cursor must be a node. Similarly, when the origin is an arc, then the object must be an arc.

The current-arc-out (in) represents the arc most recently crossed by a traverse-node (graph) -out (in) operation (see Class 5).

The last-of-related (attached) -set represents the fact that there is direct access to the last structure in the sets N^+ (N^-) and A^+ (A^-). These structures being circular and with related (attached) -successor and related (attached) -predecessor, one can access all the structures in the sets N^+ (N^-) and A^+ (A^-).

a. value(*) = v where $* \in V - \{\lambda\}$ and $v_{body(*)} \in V$.

b. origin(*) = u where $* \in N \cup A \cup C$ and $u_{body(*)} \in G \cup N$.

c. object(*) = b where $* \in N \cup A \cup C$ and $b_{body(*)} \in E \cup N$.

d. current-arc-out(*) = a^+ where $* \in N$ and $a^+_{body(*)} \in A \cup \{\lambda\}$.

In order to proceed we introduce some

notational conventions

(1) X_s^+ is N_s^+ if $s \in G$ and X_s^+ is A_s^+ if $s \in N$; similarly

Y_s^- is N_s^- if $s \in E$ and Y_s^- is A_s^- if $s \in N$.

(2) $s_{body(*)}$ is abbreviated to s for s being v, u, b, X^+, a^+, Y^-, or a^-.

(3) $i \oplus j = i + j \bmod k$

$i \ominus j = i - j + k \bmod k$

However, when we consider arithmetic of subscripts, the ordered k-tuple is renumbered $(x_0, x_1, x_2, \ldots, x_{k-1})$. Recall that k is the number of components in the tuple (system set) N^+, A^+, N^-, or A^-. Defining $\oplus$ and $\ominus$ this way should make the structure appear circular.

Continuing with class 2 (retrieval operations),

e. last-of-related-set(*) = $\begin{cases} x_k \in X^+ \text{ if } X^+ \neq \lambda \\ \lambda \text{ otherwise} \end{cases}$, where $* \in G \cup N$.

f. related-successor(*) = $\begin{cases} x_{i \oplus 1} \in X_u^+ \\ \lambda \text{ otherwise} \end{cases}$, where $* \in N \cup A$ and $* = x_i \in X_u^+$.

g. related-predecessor(*) = $\begin{cases} x_{i \ominus 1} \in X_u^+ \\ \lambda \text{ otherwise} \end{cases}$, where $* \in N \cup A$ and $* = x_i \in X_u^+$.

h. current-arc-in(*) = a^- where $* \in N$ and $a^- \in A \cup \{\lambda\}$.

i. last-of-attached-set(*) = $\begin{cases} y_k \in Y^- \text{ if } Y^- \neq \lambda \\ \lambda \text{ otherwise} \end{cases}$, where $* \in G \cup N$.

j. attached-successor(*) = $\begin{cases} y_{i \oplus 1} \in Y_b^- \\ \lambda \text{ otherwise} \end{cases}$, where $* \in N \cup A$ and $* = y_i \in Y_b^-$.

k. attached-predecessor(*) = $\begin{cases} y_{i \ominus 1} \in Y_b^- \\ \lambda \text{ otherwise} \end{cases}$, where $* \in N \cup A$ and $* = y_i \in Y_b^-$.

3. State changing operations

There are four possible states that a node or arc can be in: isolated, related-only, attached-only, or regular. Figure 3.4 shows the notation for the possible states of nodes and arcs. Regular structures are both attached and related; isolated structures are neither attached nor related. Freshly created nodes and arcs are obviously isolated. When a structure is related (attached) it is placed into a set, the elements of which share the same origin (object). When a structure is unrelated (detached) it is removed from the set; however, it maintains the same origin

states	nodes	arcs
isolated		
related-only		
attached-only		
regular		

Figure 3.4. The States of Nodes and Arcs

(*object*) for perhaps a later insertion. If a structure has been *unrelated* (*detached*), then at worst it is *isolated* and at best it is *attached-only* (*related-only*).

At this juncture we introduce a final

notational convention

(4) When *body*(z) = $(p_1, p_2, \ldots, p_m)$ and we want to redefine the *body*(z) = $(q_1, q_2, \ldots, q_m)$ then for all i such that $p_i = q_i$ we denote q_i as $\dot{p}_i$. Using this definitional notation allows us to emphasize what conceptually stays constant and what conceptually varies during an operation.

a. *relate*(*) = * where $* \in N \cup A$ and $* \notin X_u^+$. Define

body(u) = $(\dot{v}, (*))$ if $* \in N$ and $N_u^+ = \lambda$ or

$(\dot{v}, (*, \dot{x}_1, \ldots, \dot{x}_k))$ if $* \in N$ and $N_u^+ \neq \lambda$ or

$(\dot{v}, \dot{u}, \dot{b}, (*), *, \dot{A}^- \dot{a}^-)$ if $* \in A$ and $A_u^+ = \lambda$ or

$(\dot{v}, \dot{u}, \dot{b}, (*, \dot{x}_1, \ldots, \dot{x}_k), \dot{a}^+, \dot{A}^-, \dot{a}^-)$ if $* \in A$ and $A_u^+ \neq \lambda$.

b. *unrelate*(*) = * where $* \in N \cup A$ and $* = x_i \in X_u^+$. Define

body(u) = $(\dot{v}, \lambda)$ if $* \in N$ and $N_u^+ = (*)$ or

$(\dot{v}, (\dot{x}_1, \ldots, \dot{x}_{i \ominus 1}, \dot{x}_{i \oplus 1}, \ldots, \dot{x}_k))$ if $* \in N$ and $N_u^+ \neq (*)$ or

$(\dot{v}, \dot{u}, \dot{b}, \lambda, \lambda, \dot{A}^-, \dot{a}^-)$ if $* \in A$ and $A_u^+ = (*)$ or

$(\dot{v}, \dot{u}, \dot{b}, (\dot{x}_1, \ldots, \dot{x}_{i \ominus 1}, \dot{x}_{i \oplus 1}, \ldots, \dot{x}_k), \dot{a}^+, \dot{A}^-, \dot{a}^-)$

if $* \in A$ and $a_u^+ \neq *$ or

$(\dot{v}, \dot{u}, \dot{b}, (\dot{x}_1, \ldots, \dot{x}_{i \ominus 1}, \dot{x}_{i \oplus 1}, \ldots, \dot{x}_k), x_{i \ominus 1}, \dot{A}^-, \dot{a}^-)$

if $* \in A$ and $a_u^+ = *$.

c. <u>attach</u>(*) = * where $* \in N \cup A$ and $* \notin Y_b^-$. Define

<u>body</u>(b) = $(\dot{v},(*))$ if $* \in N$ and $N_b^- = \lambda$ or

$(\dot{v},(*,\dot{y}_1,\ldots,\dot{y}_k))$ if $* \in N$ and $N_b^- \neq \lambda$ or

$(\dot{v},\dot{u},\dot{b},\dot{A}^+,\dot{a}^+,(*),*)$ if $* \in A$ and $A_b^- = \lambda$ or

$(\dot{v},\dot{u},\dot{b},\dot{A}^+,\dot{a}^+,(*,\dot{y}_1,\ldots,\dot{y}_k),\dot{a}^-)$ if $* \in A$ and $A_b^- \neq \lambda$.

d. <u>detach</u>(*) = * where $* \in N \cup A$ and $* = y_i \in Y_b^-$. Define

<u>body</u>(b) = $(\dot{v},\lambda)$ if $* \in N$ and $N_b^- = (*)$ or

$(\dot{v},(\dot{y}_1,\ldots,\dot{y}_{i\ominus 1},\dot{y}_{i\oplus 1},\ldots,\dot{y}_k))$ if $* \in N$ and $N_b^- \neq (*)$ or

$(\dot{v},\dot{u},\dot{b},\dot{A}^+,\dot{a}^+,\lambda,\lambda)$ if $* \in A$ and $A_b^- = (*)$ or

$(\dot{v},\dot{u},\dot{b},\dot{A}^+,\dot{a}^+,(\dot{y}_1,\ldots,\dot{y}_{i\ominus 1},\dot{y}_{i\oplus 1},\ldots,\dot{y}_k),\dot{a}^-)$

if $* \in A$ and $a_b^- \neq *$ or

$(\dot{v},\dot{u},\dot{b},\dot{A}^+,\dot{a}^+,(\dot{y}_1,\ldots,\dot{y}_{i\ominus 1},\dot{y}_{i\oplus 1},\ldots,\dot{y}_k),y_{i\ominus 1})$

if $* \in A$ and $a_b^- = *$.

In order to show how the <u>gds</u> is changed as a result of these operations, consider Figure 3.5 as a typical data structure represented by the <u>gds</u> of Figure 3.6. Then the following sequence of operations generates a new <u>gds</u> depicted by Figure 3.7.

<u>relate</u>(a_6) makes a_6 a <u>related-only</u> arc by redefining

<u>body</u>(n_1) = $(\lambda,g_1,e_1,(a_6,a_2,a_3,a_4),a_4,\lambda,\lambda)$ and

<u>attach</u>(a_6) makes a_6 a <u>regular</u> arc by redefining

<u>body</u>(n_2) = $(\lambda,g_1,e_1,\lambda,\lambda,(a_6),a_6)$ and

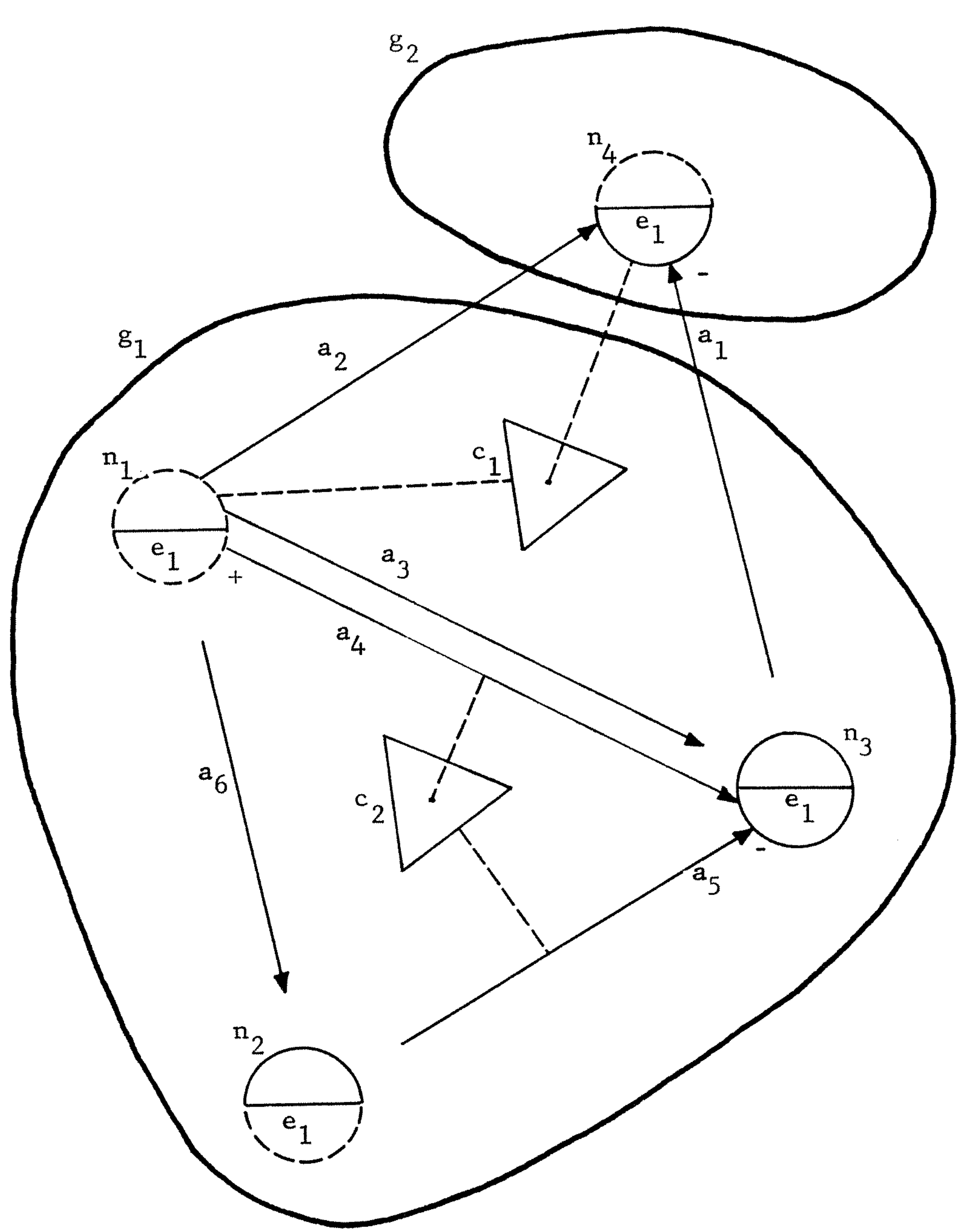


Figure 3.5. Graph Structure for Semantic Examples

$H = (L_0 \cup V, \underline{body})$ where $V = E \cup G \cup N \cup A \cup C \cup \{\lambda\}$

$E = \{e_1\}$

$\underline{body}(e_1) = (\lambda, (n_3, n_4))$

$G = \{g_1, g_2\}$

$\underline{body}(g_1) = (\lambda, (n_2, n_3))$

$\underline{body}(g_2) = (\lambda, \lambda)$

$N = \{n_1, n_2, n_3, n_4\}$

$\underline{body}(n_1) = (\lambda, g_1, e_1, (a_2, a_3, a_4), a_4, \lambda, \lambda)$

$\underline{body}(n_2) = (\lambda, g_1, e_1, \lambda, \lambda, \lambda, \lambda)$

$\underline{body}(n_3) = (\lambda, g_1, e_1, \lambda, \lambda, (a_4, a_5), a_5)$

$\underline{body}(n_4) = (\lambda, g_2, e_1, \lambda, \lambda, (a_1, a_2), a_1)$

$A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$

$\underline{body}(a_1) = (\lambda, n_3, n_4)$

$\underline{body}(a_2) = (\lambda, n_1, n_4)$

$\underline{body}(a_3) = (\lambda, n_1, n_3) = \underline{body}(a_4)$

$\underline{body}(a_5) = (\lambda, n_2, n_3)$

$\underline{body}(a_6) = (\lambda, n_1, n_2)$

$C = \{c_1, c_2\}$

$\underline{body}(c_1) = (\lambda, n_4, n_1)$

$\underline{body}(c_2) = (\lambda, a_4, a_5)$

Figure 3.6. $\underset{\cdot}{H}$ = gds of Graph Structure for Semantic Examples

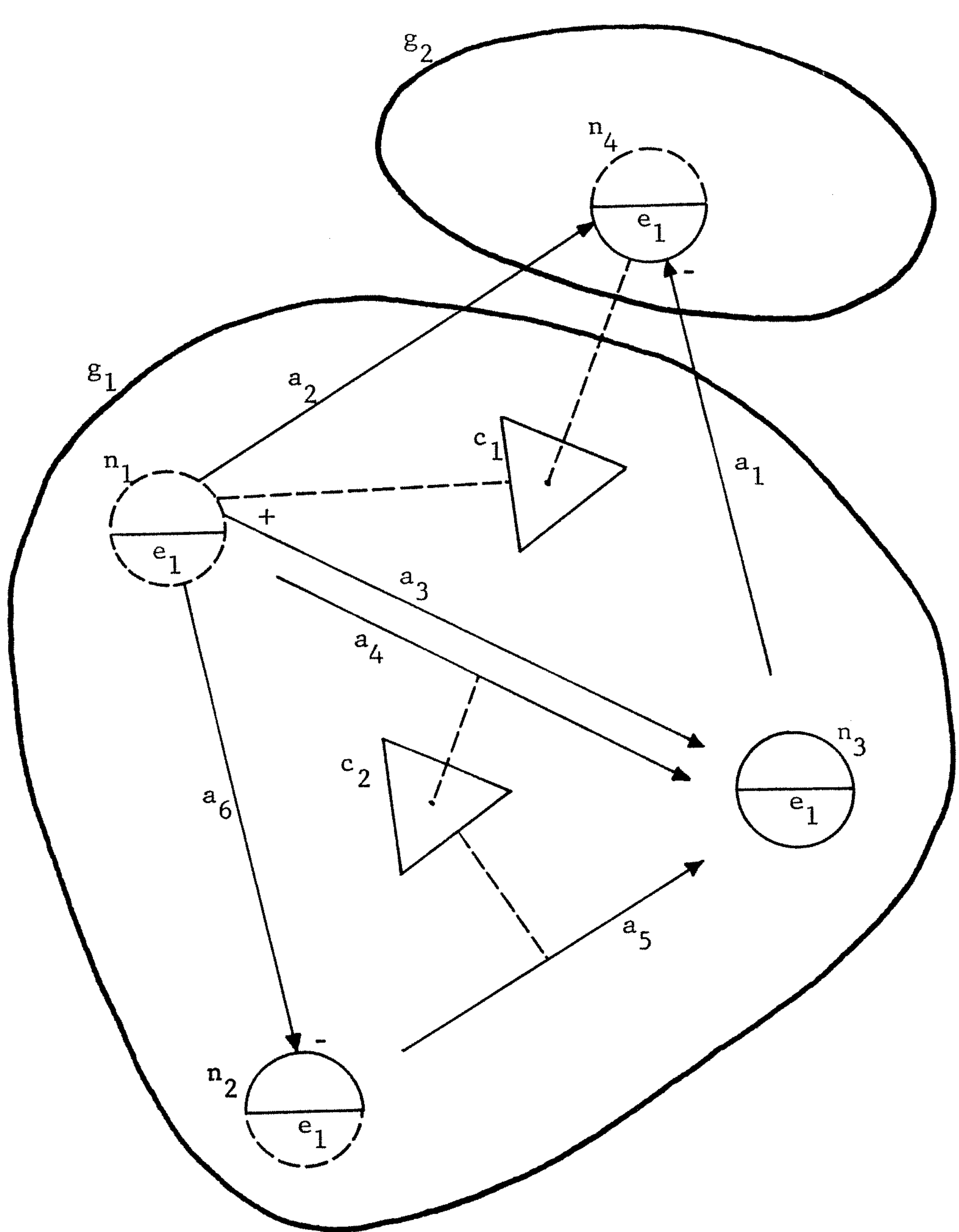


Figure 3.7. gds After State Changing Operations

unrelate(a_4) makes a_4 an attached-only arc by redefining

body(n_1) = $(\lambda, g_1, e_1, (a_6, a_2, a_3), a_3, \lambda, \lambda)$ and

detach(a_4) makes a_4 an isolated arc by redefining

body(n_3) = $(\lambda, g_1, e_1, \lambda, \lambda, (a_5), a_5)$ and

detach(a_5) makes a_5 an isolated arc by redefining

body(n_3) = $(\lambda, g_1, e_1, \lambda, \lambda, \lambda, \lambda)$.

4. Structural changing operations

The next operations are the structural changing operations. Included in these operations is the operation hang, which gives structures values. The operation change-current-arc-out (in) simply makes the arc which is its argument the current-arc-out (in) of the appropriate node. For change-last-of-related (attached) -set, the argument passed becomes the last structure in the set. Its related (attached) -successor becomes the first structure in the set, etc. In both change operations the argument must be related (attached). The operation change-origin (object) requires that its first argument not be related (attached) if it is a node or an arc. If it is a node, the node is moved to a new graph; if it is an arc, the arc emanates from a new node. If it is a cursor, then the operation effectively reinitializes the cursor. In the event that the origin of the cursor is the same type as the second argument, then only the origin (object) is changed.

a. hang(*,**) = * where $* \in V - \{\lambda\}$ and $** \in V$. Define

body(*) = $(**, \dot{N}^-)$ if $* \in E$ or

$(**, \dot{N}^+)$ if $* \in G$ or

$(**,\dot{u},\dot{b},\dot{A}^{+},\dot{a}^{+},\dot{A}^{-},\dot{a}^{-})$ if $* \in N$ or

$(**,\dot{u},\mathring{b})$ if $* \in A$ or

$(**,\dot{u},\mathring{b})$ if $* \in C$.

b. change-current-arc-out(*) = * where $* \in A_u^+$. Define

body(u) = $(\dot{v},\dot{u},\dot{b},\dot{A}^{+},*,\dot{A}^{-},\dot{a}^{-})$.

c. change-last-of-related-set(*) = * where $* = x_i \in X_u^+$. Define

body(u) = $(\dot{v},(\dot{x}_{i\oplus 1},\ldots,\dot{x}_k,\dot{x}_1,\ldots,\mathring{x}_i))$ if $* \in N$ or

$(\mathring{v},\dot{u},\dot{b},(\dot{x}_{i\oplus 1},\ldots,\dot{x}_k,\dot{x}_1,\ldots,\dot{x}_i),\dot{a}^{+},\dot{A}^{-},\mathring{a}^{-})$ if $* \in A$.

d. change-current-arc-in(*) = * where $* \in A_b^-$. Define

body(b) = $(\dot{v},\dot{u},\dot{b},\mathring{A}^{+},\dot{a}^{+},\dot{A}^{-},*)$.

e. change-last-of-attached-set(*) = * where $* = y_i \in Y_b^-$. Define

body(b) = $(\dot{v},(\dot{y}_{i\oplus 1},\ldots,\dot{y}_k,\dot{y}_1,\ldots,\dot{y}_i))$ if $* \in N$ or

$(\dot{v},\dot{u},\dot{b},\dot{A}^{+},\dot{a}^{+},(\dot{y}_{i\oplus 1},\ldots,\dot{y}_k,\dot{y}_1,\ldots,\dot{y}_i),\dot{a}^{-})$ if $* \in A$.

f. change-origin(*,**) = * where $* \in N \cup A \cup C$ and $** \in G \cup N \cup A$.

Define

body(*) = $(\dot{v},**,\dot{b},\dot{A}^{+},\dot{a}^{+},\mathring{A}^{-},\dot{a}^{-})$ if $* \in N$, $** \in G$ and $* \notin N_u^+$ or

$(\dot{v},**,\dot{b})$ if $* \in A$, $** \in N$ and $* \notin A_u^+$ or

$(\mathring{v},**,\dot{b})$ if $* \in C$ and $** \in N \cup A$ with b and ** in the same set or

$(\mathring{v},**,**)$ if $* \in C$ and $** \in N \cup A$ with b and ** in different sets.

g. change-object(*,**) = * where $* \in N \cup A \cup C$ and $** \in E \cup N \cup A$.

Define

<u>body</u>(*) = $(\dot{v},\dot{u},**,\dot{A}^+,\dot{a}^+,\dot{A}^-,\dot{a}^-)$ if * $\in$ N, ** $\in$ E and * $\notin N_b^-$ or

$(\dot{v},\dot{u},**)$ if * $\in$ A, ** $\in$ N and * $\notin A_b^-$ or

(v,u,**) if * $\in$ C and ** $\in N \cup A$ with u and ** in the <u>same</u> set or

(v,**,**) if * $\in$ C and ** $\in N \cup A$ with u and ** in <u>different</u> sets.

Once again, using the <u>gds</u> of Figure 3.5, the following sequence of state and structure changing operations forms a new <u>gds</u> depicted in Figure 3.8.

<u>change-origin</u>(a_5,n_1) makes a_5 have the <u>origin</u>, n_1, by redefining

<u>body</u>$(a_5) = (\lambda,n_1,n_3)$ and

<u>change-current-arc-out</u>(a_2) makes a_2 the <u>current-arc-out</u>(n_1) by redefining

<u>body</u>$(n_1) = (\lambda,g_1,e_1,(a_2,a_3,a_4),a_2,\lambda,\lambda)$ and

<u>relate</u>(a_5) makes a_5 a <u>regular</u> arc by redefining

<u>body</u>$(n_1) = (\lambda,g_1,e_1,(a_5,a_2,a_3,a_4),a_2,\lambda,\lambda)$ and

<u>change-last-of-related-set</u>(a_3) makes a_3 the <u>last-of-related-set</u>(n_1) by redefining <u>body</u>$(n_1) = (\lambda,g_1,e_1,(a_4,a_5,a_2,a_3),a_2,\lambda,\lambda)$.

5. <u>Traversing operations</u>

The traversing operations are partitioned into two types of operations. The simple traversing operations move around on the sets and the complex operations move around on the graph structures. The simplicity of the definitions for the simple traversals might cause the reader to overlook a powerful capability of this model. As an illustration, consider

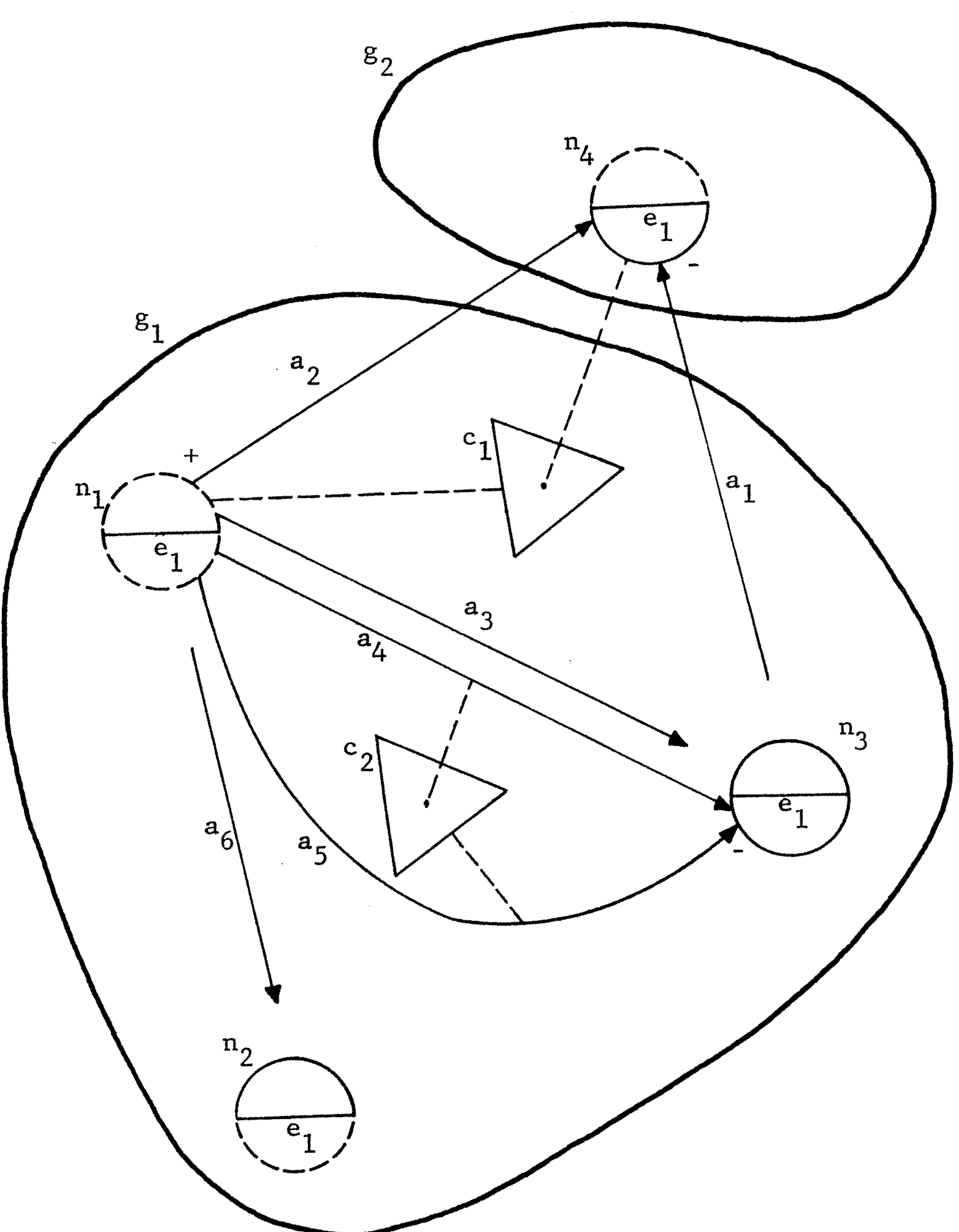


Figure 3.8. gds After State and Structural Changing Operations

Figure 3.9 with a cursor s where object(s) = origin(s) = a. Five executions of traverse-related-successor(s) makes object(s) = f by stopping at b', c', d', and e'. One more execution of traverse-related-successor(s) makes the object(s) = a once again. This is a simple linear structure search--we looked at all the arcs in the system set, A_u^+ where u = origin(a). Now consider three executions of the operation sequence (traverse-attached-successor(s), traverse-related successor(s)). Here, once again, object(s) becomes a, but this time by stopping at b, c, d, e, and f.

a. traverse-related-successor(*) = $\begin{cases} x_{i \oplus 1} \text{ if } b = x_i \in X^+_{u_b} \\ \lambda \text{ otherwise} \end{cases}$,

where * ∈ C and define body(*) = $(\dot{v},\dot{u},x_{i \oplus 1})$ if $b \in X^+_{u_b}$.

b. traverse-related-predecessor(*) = $\begin{cases} x_{i \ominus 1} \text{ if } b = x_i \in X^+_{u_b} \\ \lambda \text{ otherwise} \end{cases}$,

where * ∈ C and define body(*) = $(\dot{v},\dot{u},x_{i \ominus 1})$ if $b \in X^+_{u_b}$.

c. traverse-attached-successor(*) = $\begin{cases} y_{i \oplus 1} \text{ if } b = y_i \in Y^-_{b_b} \\ \lambda \text{ otherwise} \end{cases}$,

where * ∈ C and define body(*) = $(\dot{v},\dot{u},y_{i \oplus 1})$ if $b \in Y^-_{b_b}$.

d. traverse-attached-predecessor(*) = $\begin{cases} y_{i \ominus 1} \text{ if } b = y_i \in Y^-_{b_b} \\ \lambda \text{ otherwise} \end{cases}$,

where * ∈ C and define body(*) = $(\dot{v},\dot{u},y_{i \ominus 1})$ if $b \in Y^-_{b_b}$.

The more sophisticated cursor operations each require a cursor which has as its origin some node. These operations allow the cursor

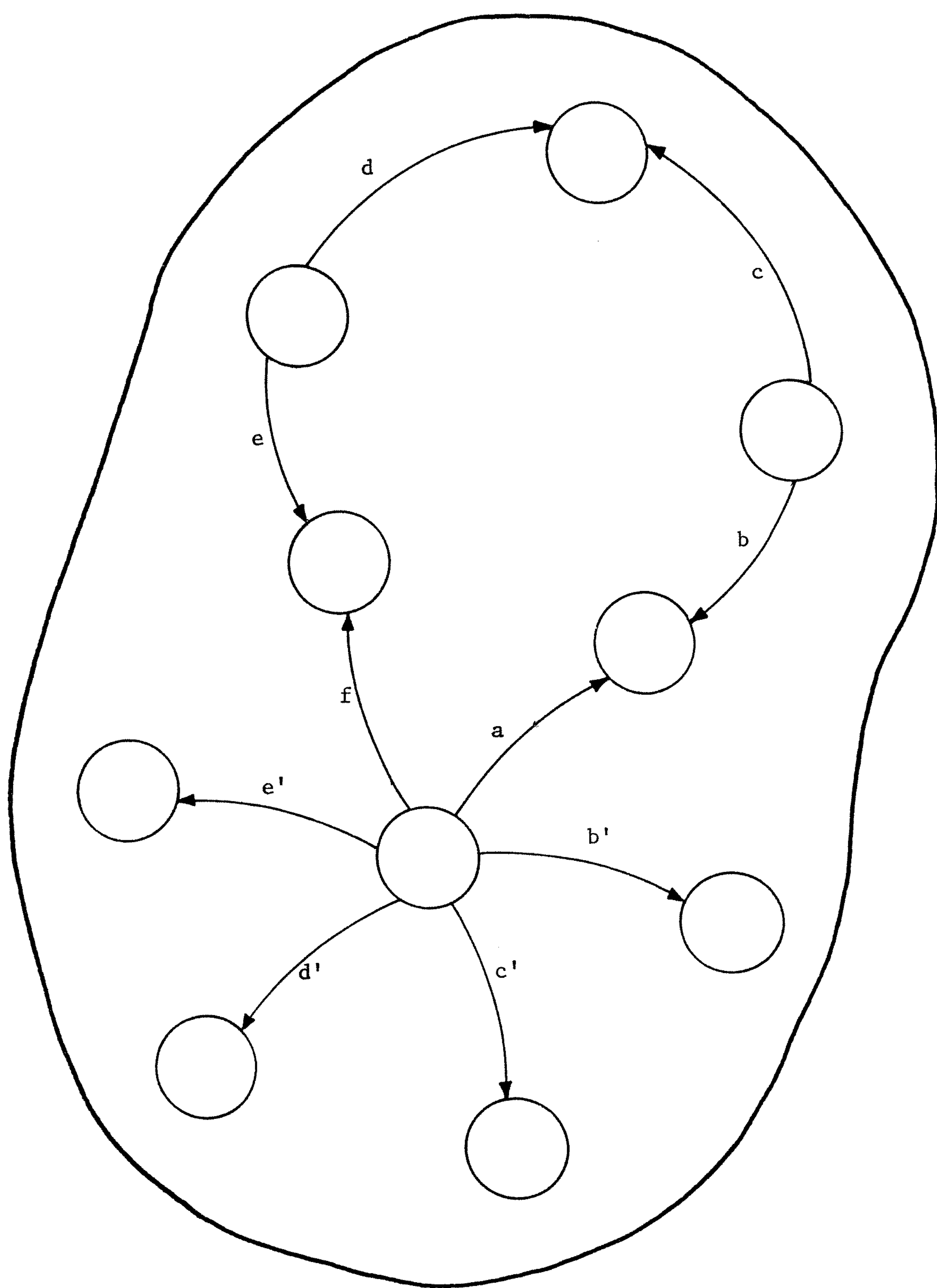


Figure 3.9. Graph for Illustrating "Subtle" Simple Traversal

to bounce from node to node so long as there exists an arc _related_ (attached) to the node which is the _object_ of the cursor. The value of these operations is the arc which is crossed or λ if none can be crossed. There are operations--_traverse-graph-out_ or _traverse-graph-in_--which cross only those arcs which lead to nodes with the same _origin_ (i.e., which are on the same graph) as the _object_ of the cursor.

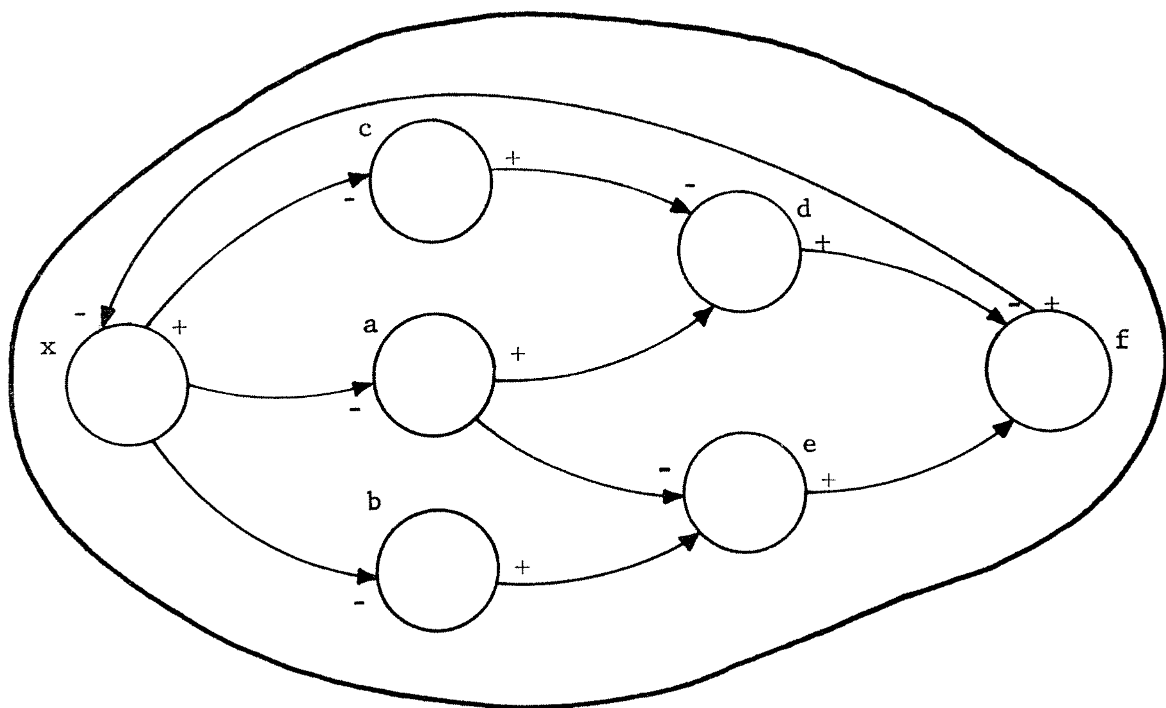
As arcs are crossed they become either the _current-arc-out_ (if crossed in an outward direction) or _current-arc-in_ (if crossed in an inward direction). For notational convenience we place a + (recall Figure 3.2) on the tail of a _current-arc-out_ and a - on the head of a _current-arc-in_.

For a cursor whose _origin_ and _object_ is the node x (see Figure 3.10) the execution of 16 _traverse-node-out_ operations stops at the nodes in this order (a, e, f, x, b, e, f, x, c, d, f, x, a, d, f, and x) while producing the _after_ diagram of Figure 3.10. The _current-arc-out_(x) has been changed. It should be noted that the _current-arc-out_ (_in_) is not crossed unless it is the only _related_ (_attached_) arc leaving (entering) a particular node. Also note that the _related_ (_attached_) _-successor_ of the _current-arc-out_ (_in_) is the arc chosen to cross each time.

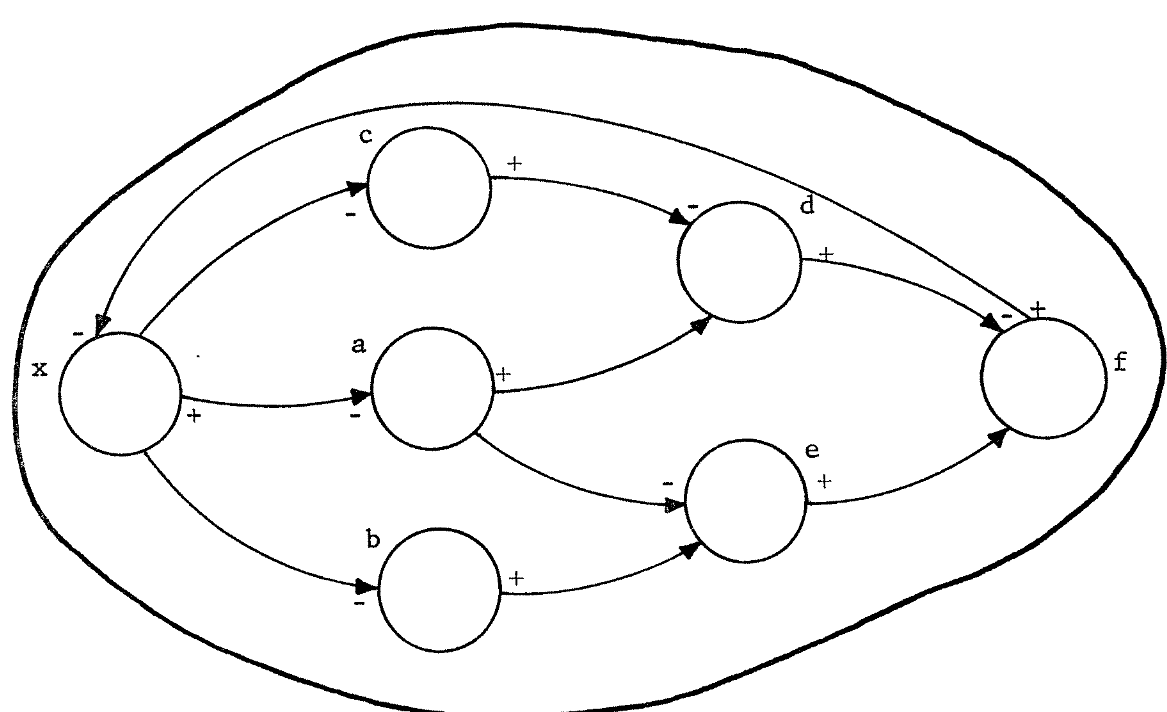
Traversing operations (complex)

a. _traverse-node-out_(*) = $\begin{cases} x_{i \oplus 1} \text{ if } a_b^+ = x_i \in A_b^+ \\ \lambda \text{ otherwise} \end{cases}$,

where * $\in$ C and b $\in$ N and define _body_(b) = $(\dot{v},\dot{u},\dot{b},\dot{A}^+,x_{i \oplus 1},\dot{A}^-,\dot{a}^-)$ and define _body_(*) = $(\dot{v},\dot{u},b_{x_{i \oplus 1}})$.



Before 16 <u>traverse-node-out</u> operations with the cursor residing on node, x



After 16 <u>traverse-node-out</u> operations

Figure 3.10. The Complex Reader Mechanism

b. traverse-graph-out(*) = $\begin{cases} x_{i \oplus j} & \text{if } a_b^+ = x_i \in A_b^+\text{, where } j \text{ is the smallest integer such that } 0 < j \leq k \text{ and } u_{b_{x_{i \oplus j}}} = u_b; \\ \lambda & \text{otherwise,} \end{cases}$

where $* \in C$ and $b \in N$ and define body(b) = $(\dot{v},\dot{u},\dot{b},\dot{A}^+,x_{i \oplus j},\dot{A}^-,\dot{a}^-)$ and define body(*) = $(\dot{v},\dot{u},b_{x_{i \oplus j}})$.

c. traverse-node-in(*) = $\begin{cases} y_{i \oplus 1} & \text{if } a_b^- = y_i \in A_b^- \\ \lambda & \text{otherwise} \end{cases}$,

where $* \in C$ and $b \in N$ and define body(b) = $(\dot{v},\dot{u},\dot{b},\dot{A}^+,\dot{a}^+,\dot{A}^-,y_{i \oplus 1})$ and define body(*) = $(\dot{v},\dot{u},u_{y_{i \oplus 1}})$

d. traverse-graph-in(*) = $\begin{cases} y_{i \oplus j} & \text{if } a_b^- = y_i \in A_b^-\text{, where } j \text{ is the smallest integer such that } 0 < j \leq k \text{ and } u_{u_{y_{i \oplus j}}} = u_b; \\ \lambda & \text{otherwise,} \end{cases}$

where $* \in C$ and $b \in N$ and define body(b) = $(\dot{v},\dot{u},\dot{b},\dot{A}^+,\dot{a}^+,\dot{A}^-,y_{i \oplus j})$ and define body(*) = $(v,u,u_{y_{i \oplus j}})$.

CHAPTER IV

MICRO-SEMANTICS OF GROPE

In this chapter we present the micro-semantics. The micro-semantics is the formal definition of GROPE from the implementation point of view (see, for example, Earley [6] or Shneiderman [33]). In the preceding chapter the macro-semantics were presented. One of the shortcomings of the macro-semantics is that no indication is given as to the execution time or storage cost of any of the data structures or primitive operations. A second more important weakness is that the macro-semantics does not display any notion of how GROPE might be implemented.

The micro-semantics of GROPE resolves both of these weaknesses. It can be correctly argued that the micro-semantics suffice and that the macro-semantics are superfluous. However, one need only consider the detailed descriptions of the micro-semantics (see Figure 4.6) to understand why we believe that the macro-semantics are important.

In the preceding chapter on macro-semantics, we introduced the notion of the "abstract system" approach using the gds as an illustration. Very little new need be presented here because direct parallels can be drawn for the micro-semantics from the macro-semantics. Once again we note that the "abstract system" follows the pattern of programming language definition by defining a total data space, data structures, and operations over the data structures. The total data space is the GDS; the data structures are defined by the functions ARCS and TYPE; and the operations are given as the transition rule from one state (GDS) to another state (GDS').

The Total Data Space and Data Structures of the Micro-semantics

The first and second phase of the formal definition of programming languages is to describe the total data space and the data structures. The GDS is the total data space, and the functions ARCS and TYPE present the data structures.

A GDS is a quintuple ($\bar{L}$,T-set,A-set,TYPE,ARCS) where

i. $\bar{L}$ is an infinite set of nodes partitioned into two sets, $\bar{L}_0$ (the unused nodes) and a finite set $\bar{V}$. Initially $\bar{V}$ contains λ, the null node.

ii. T-set is the finite set of types.

T-set = {element,graph,node,arc,cursor}.

iii. A-set is the finite set of arc labels.

A-set = $\{u,b,v,a^+,x_k,x,a^-,y_k,y\}$.

iv. TYPE is a partial function mapping $\bar{V} \rightarrow$ T-set.

As a point (node) in $\bar{L}_0$ is placed into $\bar{V}$, then its TYPE becomes defined.

v. ARCS is a partial function mapping $\bar{V}$ X A-set $\rightarrow \bar{V}$. More specifically,

$$\text{ARCS} : \left[\begin{array}{lll} \{n : \underline{\text{TYPE}}(n) = \underline{\text{element}}\} & \text{X } \{v,y_k\} & \cup \\ \{n : \underline{\text{TYPE}}(n) = \underline{\text{graph}}\} & \text{X } \{v,x_k\} & \cup \\ \{n : \underline{\text{TYPE}}(n) = \underline{\text{node}}\} & \text{X A-set} & \cup \\ \{n : \underline{\text{TYPE}}(n) = \underline{\text{arc}}\} & \text{X } \{u,b,v,x,y\} & \cup \\ \{n : \underline{\text{TYPE}}(n) = \underline{\text{cursor}}\} & \text{X } \{u,b,v\} & \end{array}\right] \rightarrow \bar{V}$$

If ARCS(n,a) = m then there is said to be an arc from node n to node m with label, a. Note that there may not be two arcs leaving a node with the same label.

To illustrate the micro-semantics of GROPE, let us consider, once more, a graph composed of two nodes and one arc (see Figure 3.2). Figure 4.1 shows the arc labels as kinds of arrowheads for the purpose of clarity, and Figure 4.2 is the GDS for that structure using the arc label conventions.

Let us focus our attention on the major differences in Figure 3.2 and Figure 4.2. Although the gds (macro-semantics) and GDS (micro-semantics) represent the same information, it appears that the simplicity of the gds for describing an arc is more natural for a user of a language than the complexity of the GDS for describing the same information (see also Figure 3.5 and Figure 4.6). On the other hand, if we consider the arc-labels as fields of a multi-word cell (plex), then the storage structure of GROPE is readily apparent, whereas it is possible (very likely) that an implementation developed from the gds (macro-semantics) would prove to be very inefficient in terms of storage and/or execution time.

Notational Conventions

If we choose an arc label a out of the A-set then

(1) if a(n) exists is the same as ARCS(n,a) is defined. In other words, if a(n) = m there is an arc with label a from node n to node m.

(2) a(n) becomes m means that an arc is created from node n to node m with label a. This means that an existing arc with label a from node n, if any, is removed.

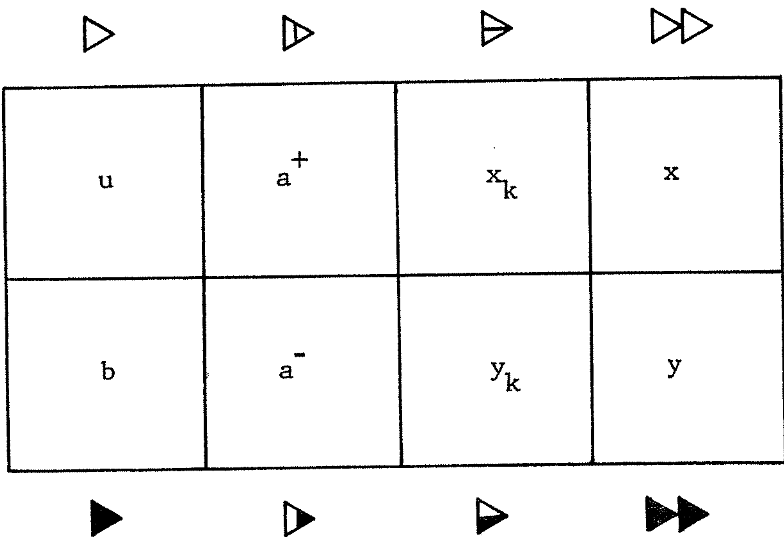


Figure 4.1. Arc Label Conventions

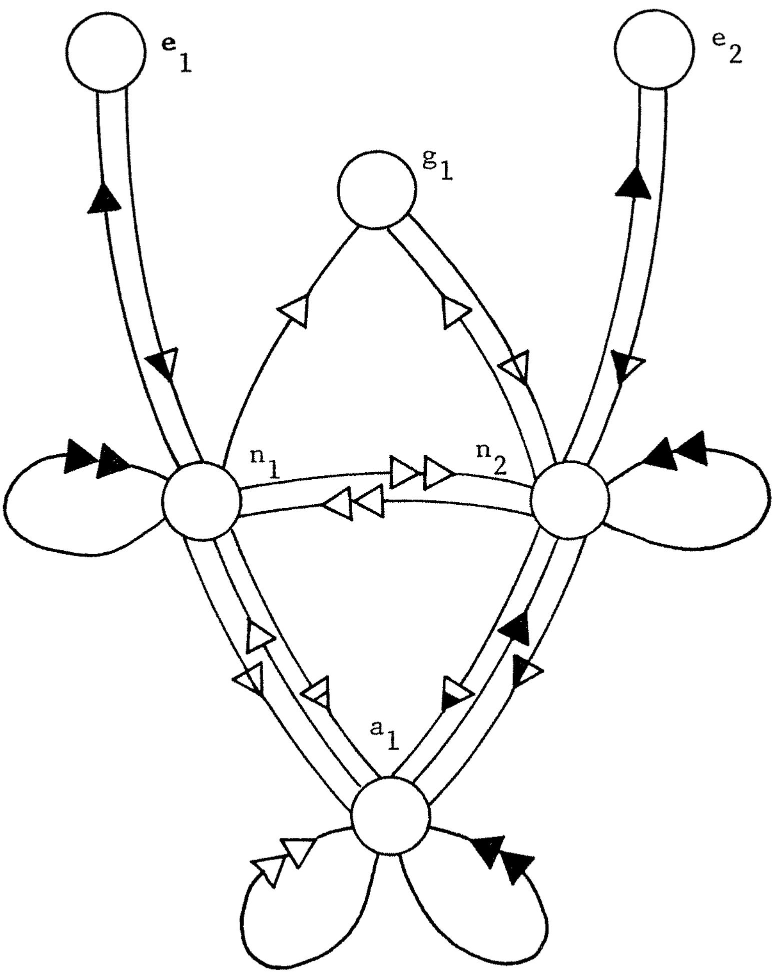


Figure 4.2. GDS of a Simple Graph

(3) a(n) is removed means that ARCS(n,a) is undefined.

Operation-diagram Conventions

These diagrams are before/after diagrams where

(1) Double thickness arrows are to attract attention.

(2) All darkened nodes are the same node.

(3) In the before diagram, darkened arrows do not exist.

(4) In the after diagram, dashed arrows do not exist.

Operations in the Micro-semantics

The GROPE primitive operations are partitioned into five classes as in Chapter III. The first class contains all of the creation operations along with their associated predicates which determine whether or not a value is of a certain type. The second class is composed of basic retrieval operations. The third class contains the "state" changing operations. The fourth class contains the structural changing operations and the fifth class is the cursor (reader) traversal operations.

1. Creation operations and associated predicates

   a. CREATE-ELEMENT() = R where $R \in \bar{L}$. There is a free[1] node $R \in \bar{L}_0$, define TYPE(R) = element. (Recall that if R is in the domain of TYPE, then R is placed in $\bar{V}$.)

From the implementation point of view, CREATE-ELEMENT() requires some explanation. In the definition of the operation CREATE-ELEMENT() there is no data constant (print image) associated with the created element.

[1]A node is free if it is not λ and if it is not in the domain of the function TYPE.

This is not quite the case. We actually would expect that CREATE-ELEMENT() would build a LISP-like atom from the host's simple structures. We would want to place this element into a bucket-sorted hash list (OBSET). Thus we might have the operation

PUT-ON-OBSET(*) = * where TYPE(*) is element.

There should also be an operation which removes elements from the OBSET. Hence

TAKE-OFF-OBSET(*) = * where TYPE(*) is element.

Finally, there should be a predicate which determines if an element is on the OBSET:

$$\text{IS-ON-OBSET}(*) = \begin{cases} * \text{ if } * \text{ is on the OBSET} \\ \lambda \text{ otherwise} \end{cases}, \text{ where TYPE}(*) \text{ is element.}$$

As in LISP, any element that is on the OBSET must be protected from the garbage collector.

(Continuing with creation operations and associated predicates)

b. $$\text{IS-ELEMENT}(*) = \begin{cases} * \text{ if TYPE}(*) \text{ is element} \\ \lambda \text{ otherwise} \end{cases}.$$

c. CREATE-GRAPH() = R where $R \in \bar{L}$. There is a free node $R \in \bar{L}_0$, define TYPE(R) = graph.

d. $$\text{IS-GRAPH}(*) = \begin{cases} * \text{ if TYPE}(*) \text{ is graph} \\ \lambda \text{ otherwise} \end{cases}.$$

e. CREATE-NODE(*,**) = R where TYPE(*) is graph, TYPE(**) is element and $R \in \bar{L}$. There is a free node $R \in \bar{L}_0$, u(R) becomes *, b(R) becomes **, and define TYPE(R) = node.

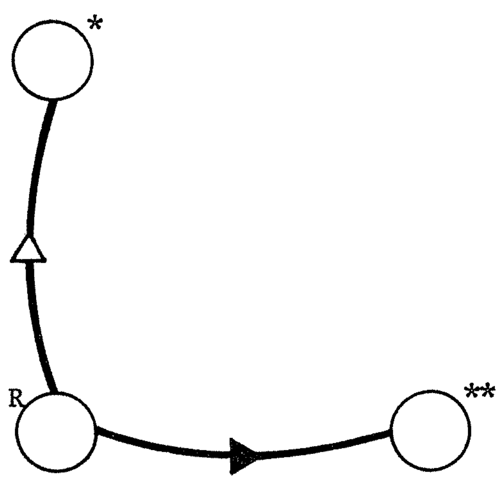


Figure 4.3. Diagram for CREATE-NODE(*, **)

f. IS-NODE(*) = $\begin{cases} * \text{ if TYPE}(*) \text{ is node} \\ \lambda \text{ otherwise} \end{cases}$.

g. CREATE-ARC(*,**) = R where TYPE(*) and TYPE(**) is node and $R \in \bar{L}$. There is a free node $R \in \bar{L}_0$, u(R) becomes *, b(R) becomes **, and define TYPE(R) = arc.

h. IS-ARC(*) = $\begin{cases} * \text{ if TYPE}(*) \text{ is arc} \\ \lambda \text{ otherwise} \end{cases}$.

i. CREATE-CURSOR(*) = R where TYPE(*) is node or arc and $R \in \bar{L}$. There is a free node $R \in \bar{L}_0$, u(R), and b(R) become * and define TYPE(R) = cursor.

j. IS-CURSOR(*) = $\begin{cases} * \text{ if TYPE}(*) \text{ is cursor} \\ \lambda \text{ otherwise} \end{cases}$.

2. Retrieval operations

a. VALUE(*) = $\begin{cases} v(*) \text{ if } v(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$, where TYPE(*) $\in$ T-set.

b. ORIGIN(*) = u(*) where TYPE(*) is node, arc, or cursor.

c. OBJECT(*) = b(*) where TYPE(*) is node, arc, or cursor.

d. CURRENT-ARC-OUT(*) = $\begin{cases} a^+(*) \text{ if } a^+(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$, where TYPE(*) is node.

e. LAST-OF-RELATED-SET(*) = $\begin{cases} x_k(*) \text{ if } x_k(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$,

where TYPE(*) is graph or node.

f. <u>RELATED-SUCCESSOR</u>(*) = $\begin{cases} x(*) \text{ if } x(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$,

where <u>TYPE</u>(*) is <u>node</u> or <u>arc</u>.

g. <u>RELATED-PREDECESSOR</u>(*) = $\begin{cases} x^{-1}(*) \text{ if } x^{-1}(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$,

where <u>TYPE</u>(*) is <u>node</u> or <u>arc</u>. (See note below.)

h. <u>CURRENT-ARC-IN</u>(*) = $\begin{cases} a^{-}(*) \text{ if } a^{-}(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$, where <u>TYPE</u>(*) is <u>node</u>.

i. <u>LAST-OF-ATTACHED-SET</u>(*) = $\begin{cases} y_k(*) \text{ if } y_k(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$,

where <u>TYPE</u>(*) is <u>element</u> or <u>node</u>.

j. <u>ATTACHED-SUCCESSOR</u>(*) = $\begin{cases} y(*) \text{ if } y(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$,

where <u>TYPE</u>(*) is <u>node</u> or <u>arc</u>.

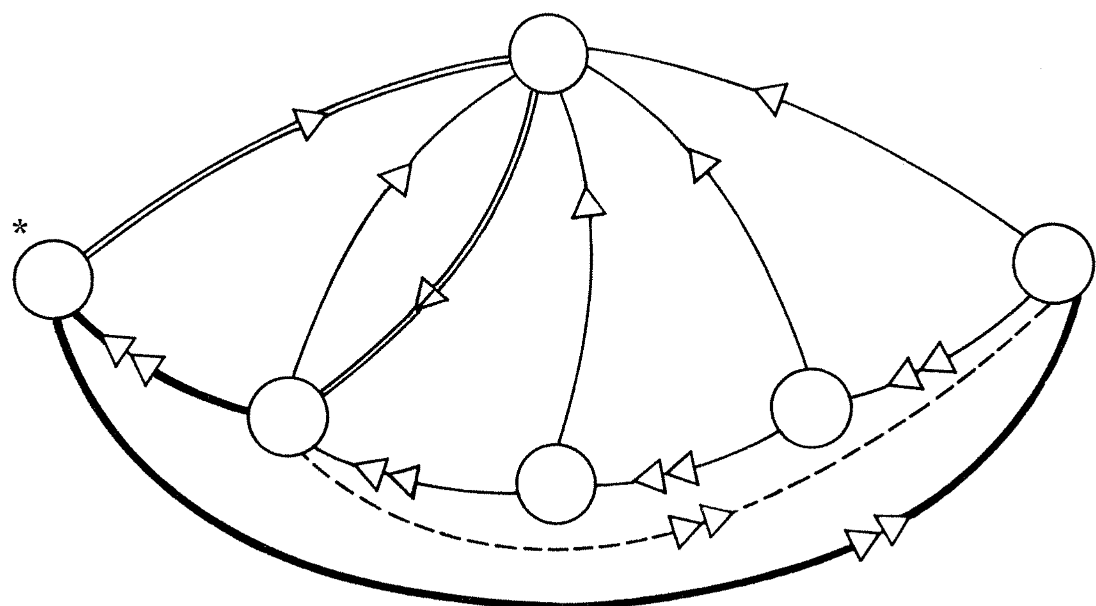
k. <u>ATTACHED-PREDECESSOR</u>(*) = $\begin{cases} y^{-1}(*) \text{ if } y^{-1}(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$,

where <u>TYPE</u>(*) is <u>node</u> or <u>arc</u>. (See note below.)

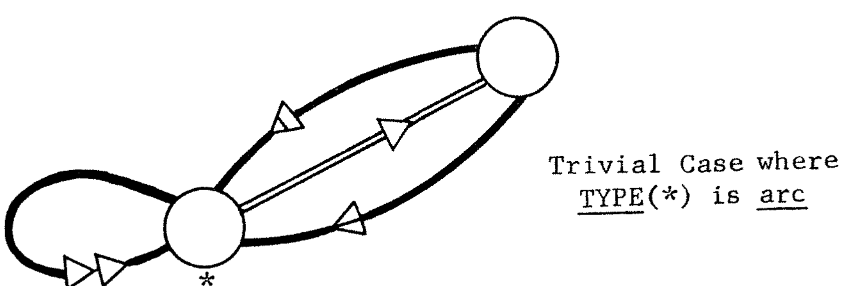
3. <u>State changing operations</u>

a. <u>RELATE</u>(*) = * where <u>TYPE</u>(*) is <u>node</u> or <u>arc</u> and x(*) does not exist. Set T = $x_k(u(*))$ if $x_k(u(*))$ exists. x(*) becomes x(T) and x(T) becomes *. If $x_k(u(*))$ does not exist, then x(*) and $x_k(u(*))$ become * and if <u>TYPE</u>(*) is <u>arc</u> then $a^{+}(u(*))$ becomes *.

<u>Note</u>: $x^{-1}(*) = *'$ such that $x(*') = *$ and $y^{-1}(*) = *'$ such that $y(*') = *$.



General Case



Trivial Case where
TYPE(*) is arc

Figure 4.4. Diagrams for RELATE(*)

b. UNRELATE(*) = * where TYPE(*) is node or arc and x(*) does exist.

If $x(x_k(u(*)))$ is *, then $x_k(u(*))$ and x(*) are removed.

If TYPE(*) is arc then $a^+(u(*))$ is also removed.

Otherwise, (X^+ has more than one component) set $T = x^{-1}(*)$.

If $x_k(u(*))$ is * then $x_k(u(*))$ becomes T.

If TYPE(*) is arc and $a^+(u(*))$ is * then $a^+(u(*))$ becomes T.

In any case, however, x(T) becomes x(*) and x(*) is removed.

c. ATTACH(*) = * where TYPE (*) is node or arc and y(*) does not exist.

Set T = $y_k(b(*))$ if $y_k(b(*))$ exists. y(*) becomes y(T) and y(T) becomes *. If $y_k(b(*))$ does not exist then y(*) and $y_k(b(*))$ become * and if TYPE(*) is arc then $a^-(b(*))$ becomes *.

d. DETACH(*) = * where TYPE(*) is node or arc and y(*) does exist.

If $y(y_k(b(*)))$ is *, then $y_k(b(*))$ and y(*) are removed.

If TYPE(*) is arc then $a^-(b(*))$ is also removed.

Otherwise, (Y^- has more than one component) set $T = y^{-1}(*)$.

If $y_k(b(*))$ is * then $y_k(b(*))$ becomes T.

If TYPE(*) is arc and $a^-(b(*))$ is * then $a^-(b(*))$ becomes T.

In any case, however, y(T) becomes y(*) and y(*) is removed.

In order to show how the GDS is changed as a result of these operations, consider Figure 3.5 as a typical data structure represented by the GDS of Figure 4.6 given by Figure 4.7. Then the following sequence of operations generates a new GDS.

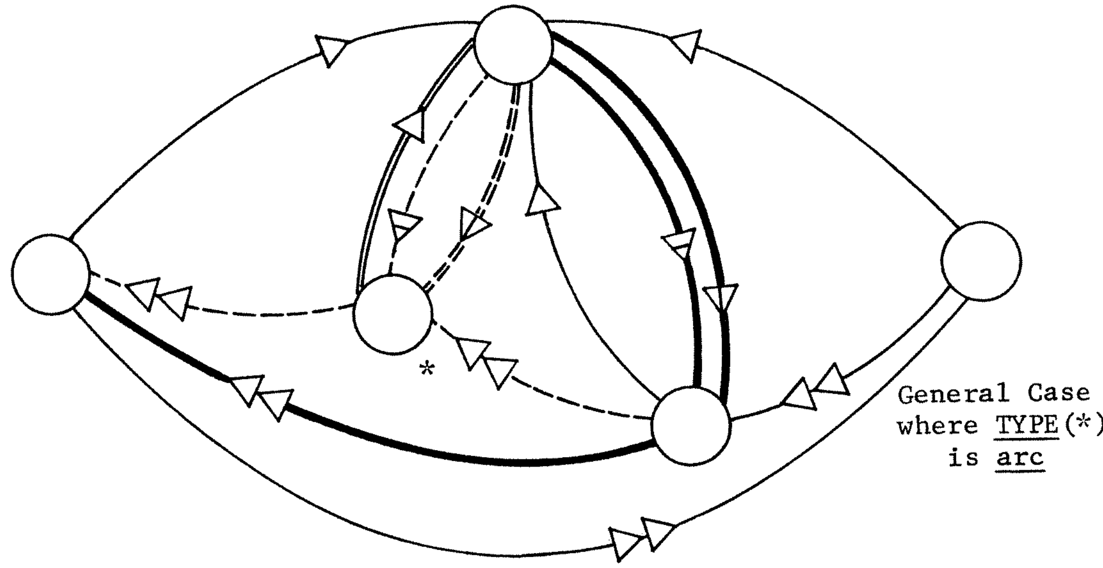


Figure 4.5. Diagram for UNRELATE(*)

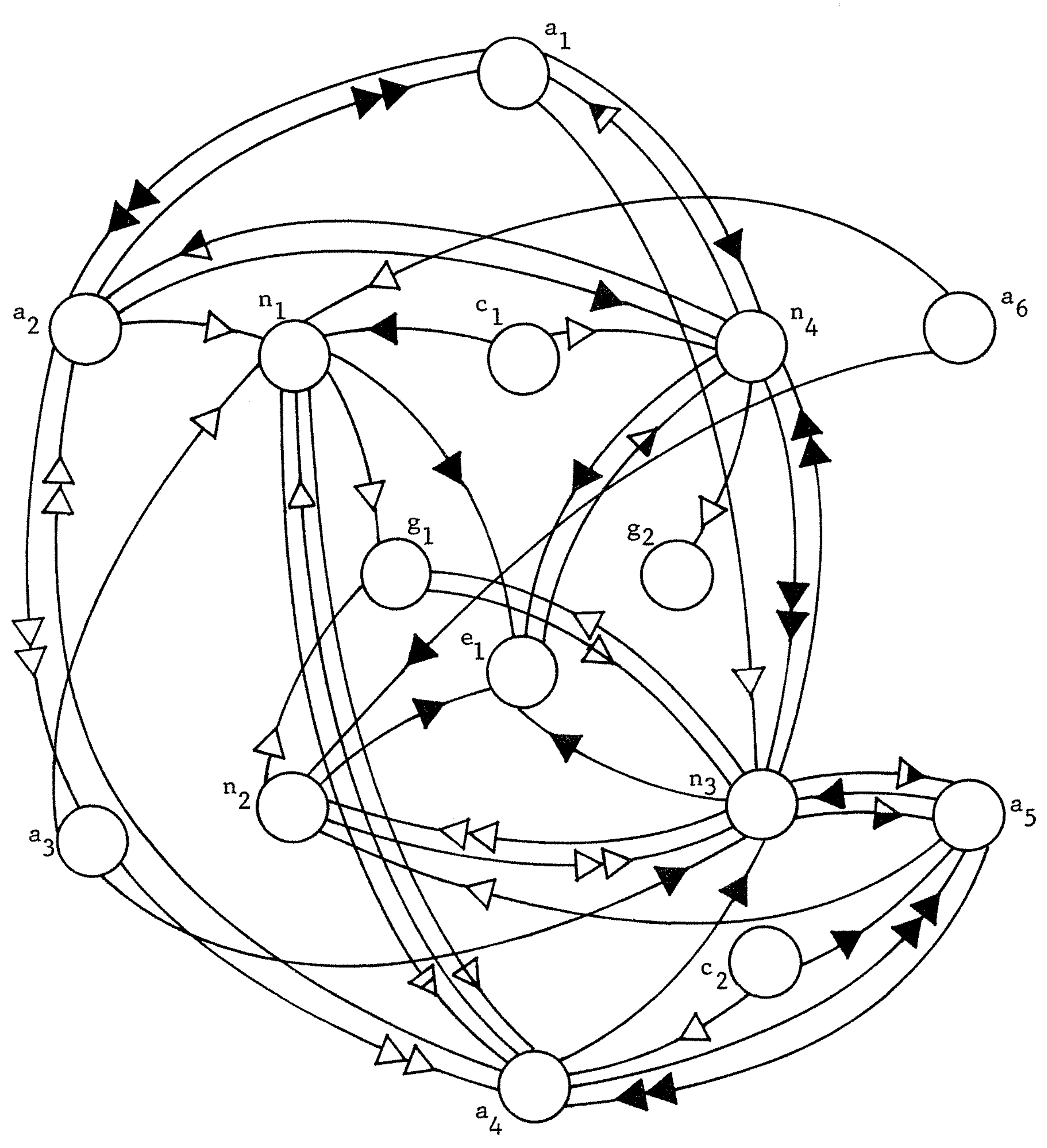


Figure 4.6. GDS for Semantic Examples

ARCS	u	b	x	x_k	a'	y	y_k	a^-	TYPE
e_1							n_4		element
g_1				n_3					graph
g_2									graph
n_1	g_1	e_1		a_4	a_4				node
n_2	g_1	e_1	n_3						node
n_3	g_1	e_1	n_2			n_4	a_5	a_5	node
n_4	g_2	e_1				n_3	a_2	a_1	node
a_1	n_3	n_4				a_2			arc
a_2	n_1	n_4	a_3			a_1			arc
a_3	n_1	n_3	a_4						arc
a_4	n_1	n_3	a_2			a_5			arc
a_5	n_2	n_3				a_4			arc
a_6	n_1	n_2							arc
c_1	n_4	n_1							cursor
c_2	a_4	a_5							cursor

Figure 4.7. Tabular Form of GDS for Semantic Examples

RELATE(a_6) makes a_6 a related-only arc by redefining

ARCS(a_4,x) = a_6 and ARCS(a_6,x) = a_2.

ATTACH(a_6) makes a_6 a regular arc by redefining

ARCS(n_2,y_k), ARCS(n_2,a^-), and ARCS(a_6,y) = a_6.

UNRELATE(a_4) makes a_4 an attached-only arc by redefining

ARCS(n_1,a^+) and ARCS(n_1,x_k) = a_3, ARCS(a_3,x) = a_6, and

ARCS(a_4,x) = λ.

DETACH(a_4) makes a_4 an isolated arc by redefining

ARCS(a_5,y) = a_5 and ARCS(a_4,y) = λ.

DETACH(a_5) makes a_5 an isolated arc by redefining

ARCS(a_5,y), ARCS(n_3,y_k), and ARCS(n_3,a^-) = λ.

Notice here that if x^{-1} is also stored, then removing structures from the sets is not a function of length. More importantly, a user ought to be able to state for a particular problem whether his algorithm requires detaching or unrelating. Whenever it is necessary to destroy the whole graph, all that must be done is to remove access to the graph, and the garbage colector gobbles up the whole structure. We would recommend for a first implementation that all x^{-1} and y^{-1} arc labels be stored explicitly. The reason for this recommendation is that graphs don't normally grow so large that they can't be stored; on the contrary, the difficult problems in graph processing are the result of the combinatorial nature of the graph algorithms.

4. Structural changing operations

a. HANG(*,**) = * where TYPE(*) ϵ T-set and TYPE(**) ϵ T-set or ** is λ.
If ** is λ, v(*) is removed. If not, v(*) becomes **.

b. CHANGE-CURRENT-ARC-OUT(*) = * where TYPE(*) is arc and x(*) exists.
a^+(u(*)) becomes *.

c. CHANGE-LAST-OF-RELATED-SET(*) = * where TYPE(*) is node or arc and
x(*) exists. x_k(u(*)) becomes *.

d. CHANGE-CURRENT-ARC-IN(*) = * where TYPE(*) is arc and y(*) exists.
a^-(b(*)) becomes *.

e. CHANGE-LAST-OF-ATTACHED-SET(*) = * where TYPE(*) is node or arc
and y(*) exists. y_k(b(*)) becomes *.

f. CHANGE-ORIGIN(*,**) = * where TYPE(*) is cursor, if TYPE(u(*)) is
TYPE(**) then u(*) becomes **; if not, b(*) also becomes **.
If TYPE(*) is node and TYPE(**) is graph or TYPE(*) is arc
and TYPE(**) is node then u(*) becomes ** if x(*) does not
exist.

g. CHANGE-OBJECT(*,**) = * where TYPE(*) is cursor, if TYPE(b(*)) is
TYPE(**) then b(*) becomes **; if not, u(*) also becomes **.
If TYPE(*) is node and TYPE(**) is element or TYPE(*) is arc
and TYPE(**) is node then b(*) becomes ** if y(*) does not
exist.

Once again, using Figure 4.6, the following sequence of state and structural changing operations forms a new GDS.

CHANGE-ORIGIN(a_5,n_1) makes a_5 have the ORIGIN, n_1, by redefining

ARCS(a_5,u) = n_1 and

CHANGE-CURRENT-ARC-OUT(a_2) makes a_2 the CURRENT-ARC-OUT(n_1) by redefining

ARCS(n_1,a^+) = a_2 and

RELATE(a_5) makes a_5 a regular arc by redefining

ARCS(a_4,x) = a_5 and ARCS(a_5,x) = a_2 and

CHANGE-LAST-OF-RELATED-SET(a_3) makes a_3 the LAST-OF-RELATED-SET(n_1) by

redefining ARCS(n_1,x_k) = a_3.

5. Traversing operations (simple)

a. TRAVERSE-RELATED-SUCCESSOR(*) = $\begin{cases} x(b(*)) \text{ if it exists} \\ \lambda \text{ otherwise} \end{cases}$,

where TYPE(*) is cursor. If x(b(*)) exists, then b(*) becomes x(b(*)).

b. TRAVERSE-RELATED-PREDECESSOR(*) = $\begin{cases} x^{-1}(b(*)) \text{ if it exists} \\ \lambda \text{ otherwise} \end{cases}$,

where TYPE(*) is cursor. If x^{-1}(b(*)) exists then b(*) becomes x^{-1}(b(*)).

c. TRAVERSE-ATTACHED-SUCCESSOR(*) = $\begin{cases} y(b(*)) \text{ if it exists} \\ \lambda \text{ otherwise} \end{cases}$

where TYPE(*) is cursor. If y(b(*)) exists then b(*) becomes y(b(*)).

d. TRAVERSE-ATTACHED-PREDECESSOR(*) = $\begin{cases} y^{-1}(b(*)) \text{ if it exists} \\ \lambda \text{ otherwise} \end{cases}$

where TYPE(*) is cursor. If y^{-1}(b(*)) exists, then b(*) becomes y^{-1}(b(*)).

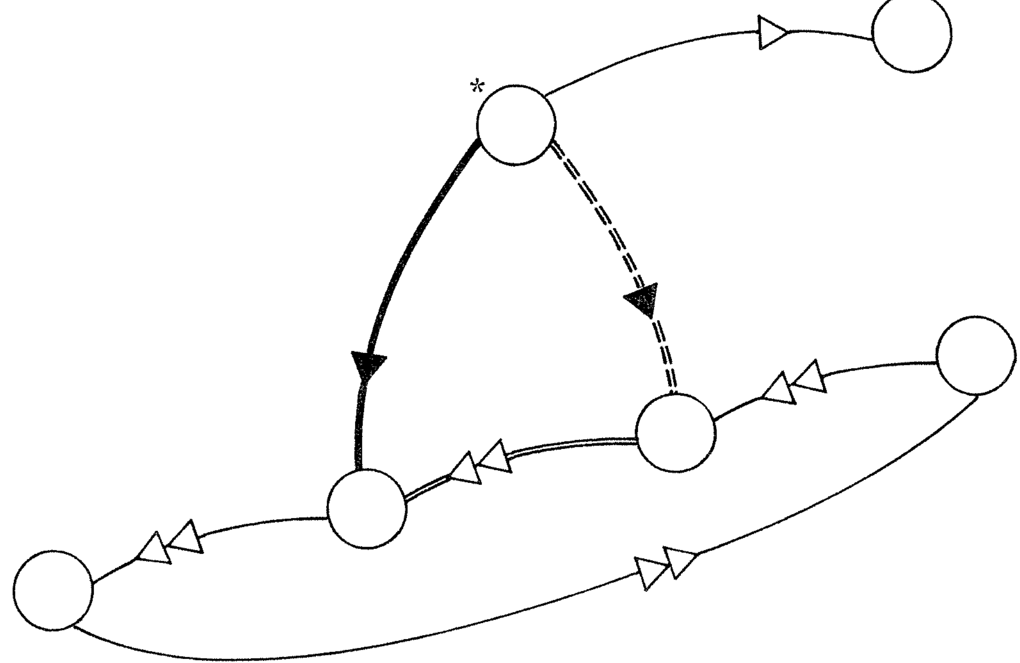


Figure 4.8. Diagram for TRAVERSE-RELATED-SUCCESSOR(*)

Traversing operations (complex)

e. TRAVERSE-NODE-OUT(*) = $\begin{cases} \alpha \text{ if it exists} \\ \lambda \text{ otherwise} \end{cases}$,

where TYPE(*) is cursor, TYPE(b(*)) is node, and α is $x(a^+(b(*)))$. If α exists then $a^+(b(*))$ becomes α and b(*) becomes $b(\alpha)$.

f. TRAVERSE-GRAPH-OUT(*) = $\begin{cases} \alpha \text{ if it exists} \\ \lambda \text{ otherwise} \end{cases}$,

where TYPE(*) is cursor, and TYPE(b(*)) is node. If there is an arc α such that $u(x(\alpha)) = b(*)$ and $u(b(\alpha)) = u(b(*))$ with minimum $\text{pathlength}_x(a^+(b(*)),\alpha)$ then $a^+(b(*))$ becomes α and b(*) becomes $b(\alpha)$. (See note below.)

g. TRAVERSE-NODE-IN(*) = $\begin{cases} \alpha \text{ if it exists} \\ \lambda \text{ otherwise} \end{cases}$,

where TYPE(*) is cursor, TYPE(b(*)) is node, and α is $y(a^-(b(*)))$. If α exists then $a^-(b(*))$ becomes α and b(*) becomes $u(\alpha)$.

h. TRAVERSE-GRAPH-IN(*) = $\begin{cases} \alpha \text{ if it exists} \\ \lambda \text{ otherwise} \end{cases}$,

where TYPE(*) is cursor and TYPE(b(*)) is node. If there is an arc α such that $b(y(\alpha)) = b(*)$ and $u(u(\alpha)) = u(b(*))$ with minimum $\text{pathlength}_y(a^-(b(*)),\alpha)$ then $a^-(b(*))$ becomes α and b(*) becomes $u(\alpha)$.

Note: $$\text{pathlength}_a(n,m) = \begin{cases} 1 \text{ if } \text{ARCS}(n,a) = m \\ \text{pathlength}_a(\text{ARCS}(n,a),m)+1 \text{ otherwise} \end{cases}$$

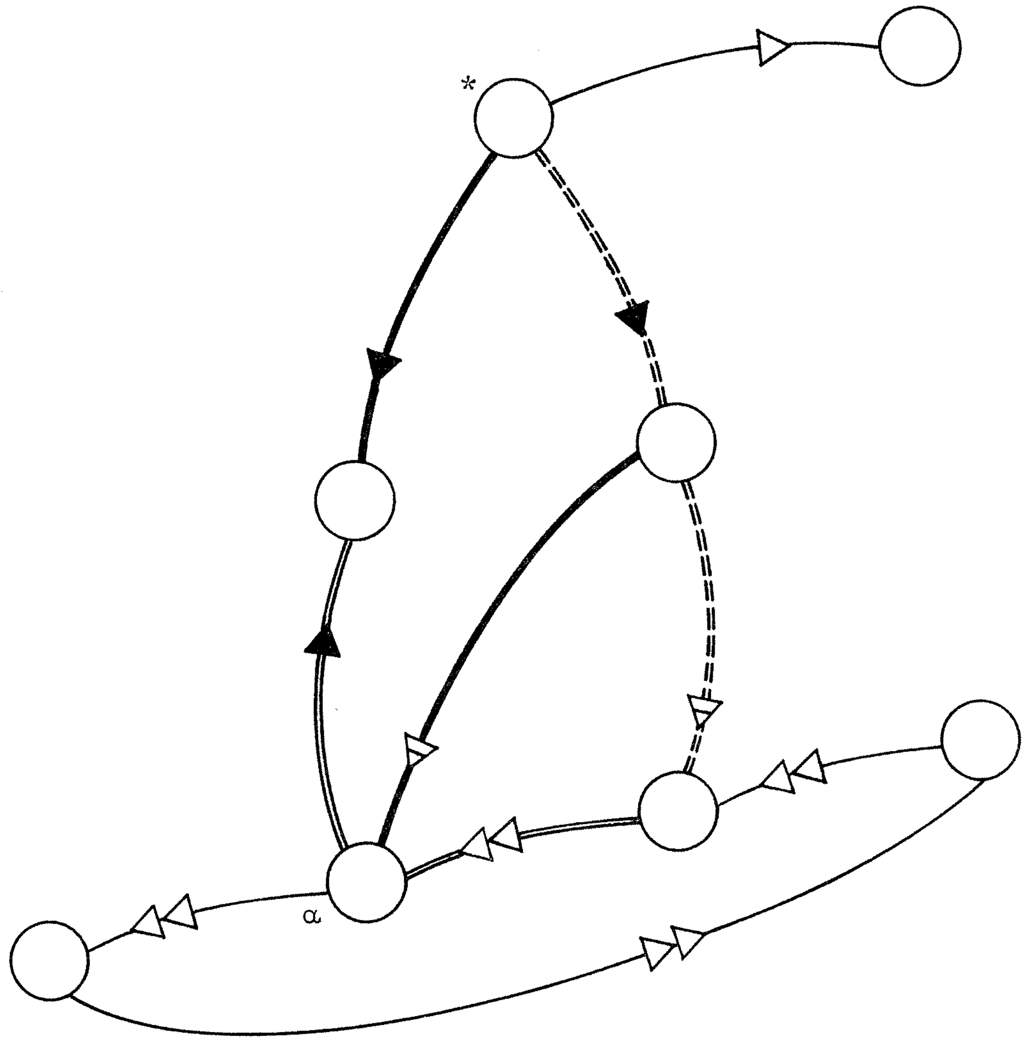


Figure 4.9. Diagram for TRAVERSE-NODE-OUT(*)

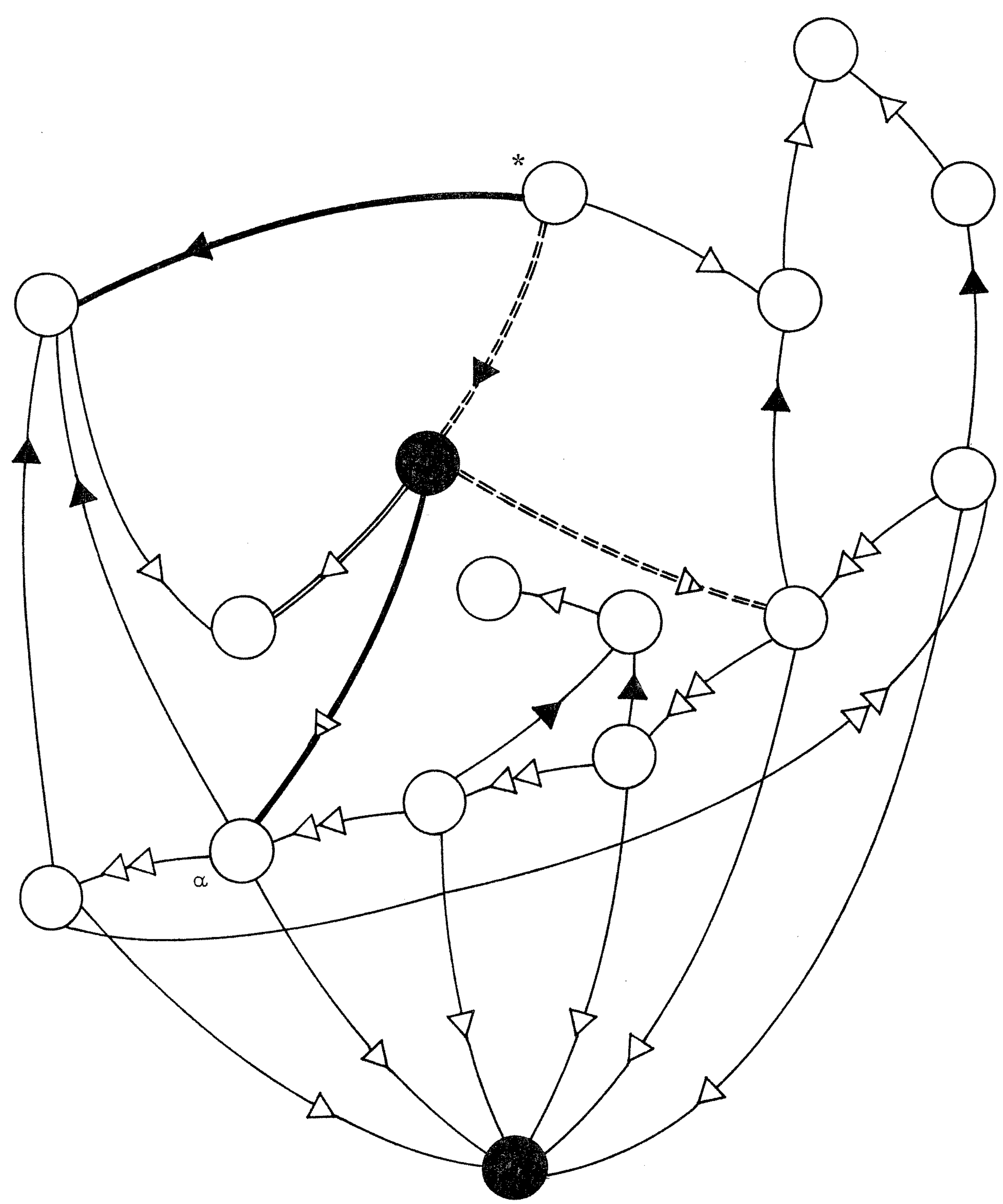


Figure 4.10. Diagram for TRAVERSE-GRAPH-OUT(*)

CHAPTER V

CONCLUSIONS

This dissertation has been concerned with two issues: the design of a graph processing language, and its formal definition. These two research efforts have resulted in the development of a graph processing language extension, GROPE, and the macro-semantics and micro-semantics two-level definitional technique.

GROPE embodies some major new ideas about representation and processing of complex structures while maintaining a serious concern with efficiency. GROPE is like LISP in that everything is dynamic. That is, the graph structures can grow, shrink, and be modified both dynamically and irregularly. In addition, GROPE has many support features which enhance the power of the graph processing primitives and special mechanisms for searching and processing graph structures. From the standpoint of the GROPE programming language extension, there is nothing new about embedding a set of operations in a high-level language. GROPE makes available to the programmer 154 operations and fifteen data types. Despite this complexity, there have been enough users of GROPE to warrant its development.

Considering the formal definitional technique, we accomplished what we set out to accomplish. That is, our formal definition is a relatively readable definition of the operations and structures available to the user (macro-semantics) and our formal definition provides an implementation design and an indication of the cost of the operations (micro-semantics). In both the macro-semantics and the micro-semantics we have used the same

approach, the abstract system approach. Thus we have shown that the abstract system approach is a viable alternative for programming language definition. One weakness of our model, however, is the lack of concern with such issues as control structure and parameter transmission.

Both the design of graph processing languages and the formal definition of programming languages have generated some interesting research problems.

Research Problems

The graph processing language GROPE has many interesting features that may or may not, in the final analysis, be considered useful graph processing tools. For example, graph readers appear to be clever mechanisms for searching graphs when combined with the current-arc-out and current-arc-in. An interesting problem would be to discover under what conditions this tool imparts a powerful algorithmic technique. Also, we introduced the notion of states of arcs and nodes. Recall that arcs and nodes can be related, attached, neither related nor attached, or both related and attached. Allowing a single graph to contain any of four types of nodes and any of four types of arcs could (and has) produced some interesting approaches to graph algorithms and the representation of graphs. Perhaps these structures could be the seed for some mathematical development as an extension of digraph theory. The wide range of flexibility as discussed in Chapter II could also lead to new schemes for representation. Finally, if the ideas of graph processing are further explored, a good research area would be to design a graph processing machine.

In the area of formal definition, there are a number of interesting research problems. The macro-semantics and the micro-semantics are both

abstract systems. An interesting research area would be to develop some mathematical results about these two systems. For example, a formal definition of a garbage collector could be defined, and one might prove that all and only those nodes which were available to become garbage would become so. One could show that by making simplifications or additions to the formal systems, other languages such as SLIP or LISP might be defined. Furthermore, an interesting result would be to prove that the micro-semantics and the macro-semantics were equivalent. Let us map out a strategy for this problem.

Define f as a one-to-one correspondence which shows for any gds (the formal system of the macro-semantics) how it can be mapped into a GDS (the formal system of the micro-semantics). For example, part of the correspondence would contain: "$x \in G$ if and only if TYPE(f(x)) = graph". In addition, we can extend f over the operations such that f(op) = OP where op is an operation defined over the macro-semantics and OP, similarly named, is defined over the micro-semantics. Thus we try to prove the following theorem.

THEOREM. The macro-semantics are f-equivalent to the micro-semantics if and only if for all op defined over the macro-semantics

$$op(gds, x_1, x_2, \ldots, x_n) = (gds', v) \quad \text{if and only if}$$

$$f(op)(f(gds), f(x_1), f(x_2), \ldots, f(x_n)) = (f(gds'), f(v)).$$

Similarly, we could prove that the application of op to a gds with legal parameters leaves the gds "well defined." The same analysis could be made on the micro-semantics.

It has long been understood that one of the major bottlenecks in the application of computers to old as well as new problems is the fact that in general each problem must undergo a number of transformations (some very complicated) to make it compatible with the language which the computer scientist decides to use. Languages such as GROPE, in which the data structures correspond more closely to those used to describe problems in applications areas, should hopefully lead to the eventual elimination of this bottleneck.

APPENDIX A

The listing presented here is the GROPE language manual for the CDC 6600 implementation. Although the manual is self-contained, a few words of direction are suggested.

The chart on page 120 is a VENN diagram. For example, a list is a lnsr, a rdsr an object, and a value whereas a node is not a lnsr. In addition, the v can be read as "only contain(s)." For example, the grset only contains graphs, the rseto and rseti only contain arcs. The chart on page 121 is also a Venn diagram.

Page 122 is a table of contents which does not give page numbers and is organized by classification of the function. The *'s are an indication of how important it is to understand a particular subclass of functions in order to program in GROPE (the more *'s, the more important). Because there are 154 functions in GROPE, we decided that placing the functions in alphabetical order was perhaps a reasonable way to organize the functions since the table of contents classification is available.

Also, the following correspondences hold:

Chapter II	Appendix A
mapft	mapft
orft	orft
andft	andft
dmapft	delft
lmapft	loft
smapft	soft
crgraph(x)	relate(cregr(x))
detgraph(x)	unrel(x)
detnode(x)	unrel(detnd(x))
detarc	detrc

Note that no other mapping functions that appear in Chapter II are in Appendix A.

```
*********
* GROPE *
*********
```

THERE ARE CERTAIN CONVENTIONS EMPLOYED IN THIS DESCRIPTION OF THE GROPE FUNCTIONS; THEY ARE BRIEFLY DEFINED AS FOLLOWS :

ARGUMENTS TO FUNCTIONS

ARG : SOME ARGUMENT -- NOT NECESSARILY A GROPE VALUE.

BITS : AN EXTERNAL REFERENCE OR A ONE-DIMENSIONAL ARRAY, POSSIBLY OF LENGTH 1 (HENCE A VARIABLE), WITH USER-SPECIFIED INFORMATION TO BE PASSED TO AN ATOM CREATION FUNCTION.

FUN : AN EXTERNAL REFERENCE OR A FUNCTION-ATOM.

G : A GRAPH.

L : A LIST.

LS : A LINEAR STRUCTURE [A LIST, SET, OR READER].

N : A NODE.

NG : A NODE OR A GRAPH.

NUM : A NUMBER.

OBJ : AN OBJECT; THE FUNCTION DEFINITION MAY STIPULATE RESTRICTIONS ON THE KINDS OF OBJECTS THAT ARE VALID.

P : A PAIR.

PS : A PSET.

Q : A DUMMY ARGUMENT; IT IS ALWAYS IGNORED.

RC : AN ARC.

RDR : A READER; THE FUNCTION DEFINITION MAY STIPULATE RESTRICTIONS ON THE KINDS OF READERS THAT ARE VALID.

RS : A READABLE STRUCTURE; X IS A READABLE STRUCTURE IF IT IS A LINEAR STRUCTURE OR A NODE OR A GRAPH.

TAPENUMBER : AN INTEGER TAPE REFERENCE.

VAL : SOME ARGUMENT -- NECESSARILY A GROPE VALUE.

VECTOR : A ONE-DIMENSIONAL ARRAY WITH NO USER-SPECIFIED INFORMATION WITHIN THE ARRAY.

CONCERNING MNEMONIC FUNCTION NAMES

ATT MEANS ATTACH[ED]

CH* OR CHA* MEANS CHANGE

CR* OR CRE* MEANS CREATE

CUR MEANS CURRENT

DEL* MEANS DELETE

DET* MEANS DETACH

*END MEANS END

*FT MEANS THE SECOND ARGUMENT IS A FUNCTION ** WHETHER AN EXTERNAL REFERENCE OR A FUNCTION*ATOM.

HS* MEANS HAS

*I MEANS IN

IS* DENOTES A PREDICATE FUNCTION THAT RETURNS ITS ARGUMENT [TRUE] OR ZERO [FALSE].

IS MEANS ISOLATED [UNATTACHED]

MA* MEANS MAKE

MOVE* MEANS MOVE [A READER] DIRECTLY

*O MEANS OUT

RC MEANS ARC

REL MEANS RELATE[D]

SERT* MEANS INSERT

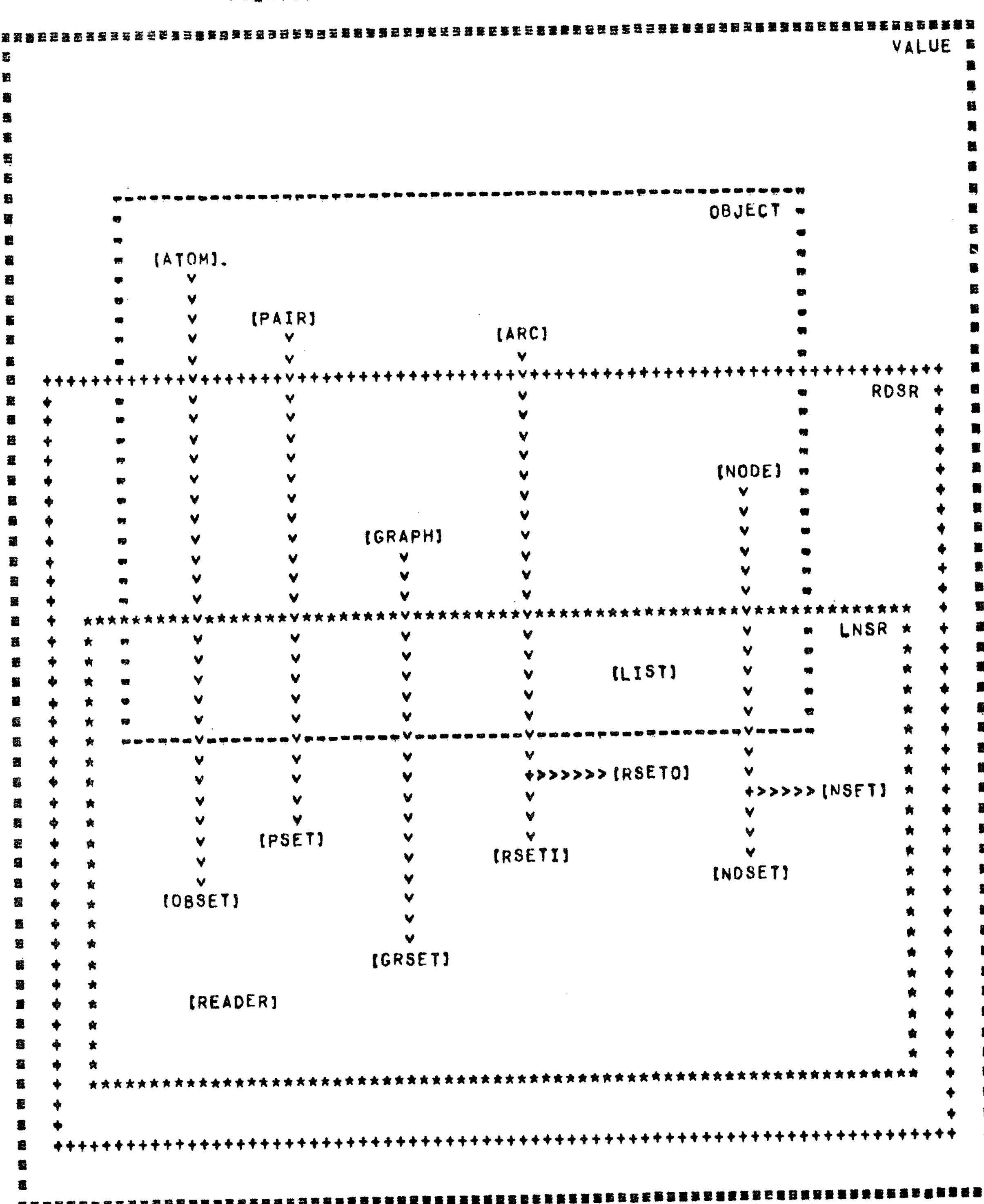
SET OR *ST* MEANS SET

*SR MEANS STRUCTURE

T* MEANS TRAVERSE [A READER]

GROPE ADDS ELEMENTS TO [SYSTEM] SETS BY EITHER STACKING [IF IN MODES] OR QUEUEING [IF IN MODEQ]; THE USER MAY CHANGE THE MODE.

RELATIONSHIPS AMONG THE GROPE DATA-TYPES



RELATIONSHIPS AMONG THE GROPE ATOM-TYPES

ATOM

HOLLERITH

FUNCTION

NUMBER

REAL

INTEGER

QUOTED

THE RELATIVE IMPORTANCE OF FUNCTION-GROUPS

ATOM AND LIST PROCESSING FUNCTIONS

Functions	Type tests	Importance
SETUP/SAVAR/SAVCOM/RETURN		********
ATOM/QUOTE	ISATOM/ISHOL/ISINT/ISREAL/ISFUN/ISQINT/ISNUM	********
IMAGE/LENATM		********
CREL/LIST/QUEUE/STACK/CONCAT/POP	ISLIST	********
IDENT/EQUAL		********
APPLY/EQULFT/COMPOS		********

GRAPH PROCESSING FUNCTIONS

Functions	Type tests	Importance
CREGR	ISGRAF	******
ANODE/CRNODE/CRISN	ISNODE	******
GRAPH/CHAGR		******
RELATE/UNREL	ISREL	******
CRARC/CRARCI/CRARCO/CRISR	ISARC	******
FRNODE/CHAFRN/TONODE/CHATON/REVRC		******
ATTRC/DETRC/ATTRCI/DETRCI/ATTRCO/DETRCO	ISRCI/ISRCO	****
ATTND/DETND	ISATTN	**
CURCI/MACURI/CURCO/MACURO		**
OBJECT/CHOBJ	ISOBJ	******
VALUE/HANG/UNHANG/BUMP	ISVAL	****

PROPERTY-SET FUNCTIONS

Functions	Type tests	Importance
PUT/GET		********
PSET/CREPS/HSPSET	ISPSET	****
CRISP	ISPAIR	**
ATTPR/DETPR	ISATTP	**
AFFIX/APPEND		**

LINEAR STRUCTURE FUNCTIONS

Functions	Type tests	Importance
FIRST/LAST		********
LENGTH	ISLNSR	********
MEMBER		********
ANDFT/ORFT/MAPFT/LOFT/SOFT/DELFT		********
GRSET/NDSET/NSET/HSNSET	ISGRST/ISNDSET/ISNSET	******
RSETI/RSETO	ISRSTI/ISRSTO	******
OBSET	ISOBST	**

READER PROCESSING FUNCTIONS

Functions	Type tests	Importance
CREEDR/ORIGIN/CHORIG/REED/RESTRT/COPYRD	ISRDR/ISRDSR	********
TO/TI		********
DSEND/DSNDTO/ASEND/REVERT/CONECT	ISDEEP	******
SERTO/SERTI/SUBST/DELETE/MERGE		******
CHEND/MOVEND/MOVETO	ISATND/ISATBG	****
TIFT/TOFT/CURARC		****

INPUT / OUTPUT AND MISCELLANEOUS FUNCTIONS

Functions	Type tests	Importance
RDFILE/RDEXP/ECHO		********
PRFILE/PRXPFT/PRINFT/TERPRI/MARGIN/TAB		********
MODEQ/MODES	ISMODS	****
INTGER/REALE/TRUE		******
MAXERR/MESAGE		**

AFFIX (OBJ,PS) : = OBJ. THE PSET PS IS AFFIXED TO OBJ (IN ADDITION TO ANY OTHER OBJECTS TO WHICH IT MAY BE AFFIXED). HENCEFORTH HSPSET(OBJ) = PS. [ONLY ONE PSET MAY BE AFFIXED TO ANY GIVEN OBJECT.]

ANDFT (LS,FUN,ARG2,ARG3,ARG4,ARG5) : = LS, PROVIDED THAT APPLY(FUN,ELEMENT, ARG2,ARG3,ARG4,ARG5) RETURNS TRUE [≠0] FOR EACH SUCCESSIVE ELEMENT IN THE LINEAR STRUCTURE LS; IF FUN RETURNS FALSE [=0], THE PROCESS IS TERMINATED AND ZERO IS RETURNED.
NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

ANODE (OBJ,G) : IF THERE IS A NODE N IN THE NSET(OBJ) WITH GRAPH(N) = G THEN THAT NODE IS THE VALUE. ELSE ANODE : = CRNODE(OBJ,G).

APPEND (OBJ,PS) : = OBJ. THE ELEMENTS OF THE PSET PS ARE QUEUED INTO THE PSET(OBJ) AND PS BECOMES EMPTY.

APPLY (FUN,ARG1,ARG2,ARG3,ARG4,ARG5) : = BITS(ARG1,ARG2,ARG3,ARG4,ARG5) WHERE FUN = ATOM(BITS,-3) FOR SOME EXTERNAL REFERENCE BITS; OR IF FUN IS AN EXTERNAL REFERENCE APPLY : = FUN(ARG1,ARG2,ARG3,ARG4,ARG5).
NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

ASEND (RDR) : THE VALUE IS THE READER PASSED IN THE CALL TO THE DESCEND FUNCTION WHICH RETURNED THE VALUE RDR; THIS CORRESPONDS TO THE SECOND READER IN THE STACK TO WHICH RDR REFERS. RDR IS NOT SIDE-AFFECTED.

ATOM (BITS,NUM) : RETURNS AN ATOMIC OBJECT ATM:
(1) IF NUM > 0 THEN ISHOL(ATM) IS TRUE AND NUM IS TAKEN TO BE AN INTEGER INDICATING THE MAXIMUM NUMBER OF CONTIGUOUS WORDS OF BITS CONTAINING THE DISPLAY-CODE PRINT-IMAGE OF THE ATOM, LEFT-JUSTIFIED, ZERO FILL. IF BLANK FILL IS GIVEN, ZERO FILL WILL BE SUBSTITUTED. THE VECTOR BITS IS SCANNED FROM RIGHT TO LEFT FOR THE FIRST NON-BLANK, NON-ZERO CHARACTER.
(2) IF NUM = 0 THEN ISINT(ATM) AND ISNUM(ATM) ARE BOTH TRUE.
(3) IF NUM IS -1 [OR -2] THEN BITS IS SINGLE [OR DOUBLE] PRECISION; ISREAL(ATM) AND ISNUM(ATM) ARE BOTH TRUE.
(4) IF NUM IS -3 THEN ATM IS A FUNCTION-ATOM. THIS IS A REFERENCE TO THE EXTERNAL FUNCTION BITS WHICH MAY BE STORED IN LISTS, ETC. AND MAY BE EXECUTED BY CALLING APPLY WITH ATM AS THE FIRST ARGUMENT. ISFUN(ATM) AND ISHOL(ATM) ARE BOTH TRUE.
(5) IN ALL CASES, ISATOM(ATM) IS TRUE.
[ONLY ONE ATOM WILL EXIST FOR ANY GIVEN BITS AND NUM.]

ATTND (N) : = N. IF ISATTN(N) IS TRUE, THERE IS NO EFFECT; ELSE THE NODE N IS STACKED OR QUEUED INTO THE NSET(OBJECT(N)).

ATTPR (P,PS) : = P. IF ISATTP(P) IS TRUE, THERE IS NO EFFECT; ELSE THE PAIR P IS STACKED OR QUEUED INTO THE PSET PS.

ATTRC (RC) : = ATTRCO(ATTRCI(RC)).

ATTRCI (RC) : = RC. IF ISRCI(RC) IS TRUE, THERE IS NO EFFECT; ELSE THE ARC RC IS STACKED OR QUEUED INTO THE RSETI(TONODE(RC)).
[IT CAN BE SAID THAT THE ARC IS ATTACHED IN.]

ATTRCO (RC) : = RC. IF ISRCO(RC) IS TRUE, THERE IS NO EFFECT; ELSE THE ARC RC IS STACKED OR QUEUED INTO THE RSETO(FRNODE(RC)).
[IT CAN BE SAID THAT THE ARC IS ATTACHED OUT.]

BUMP (OBJ,NUM) : = OBJ. THE [QUOTED INTEGER] VALUE(OBJ) IS INCREMENTED BY NUM. IF VALUE(OBJ) RETURNS ZERO [NO HANGING], THEN QUOTE(NUM) IS HUNG.

CHAFRN (RC,N) : = RC. THE FRNODE(RC) BECOMES N; THE ATTACHED RELATIONSHIP [ISRCO] BETWEEN THE OLD FRNODE AND THE ARC IS MAINTAINED BETWEEN THE NEW FRNODE AND THE ARC.

CHAGR (N,G) : = N. THE GRAPH(N) BECOMES G, AND THE RELATED CONDITION [ISREL] OF THE NODE N IS MAINTAINED. [IT MAY BE SAID THAT THE NODE N NOW RESIDES ON THE GRAPH G.]

CHATON (RC,N) : = RC. THE TONODE(RC) BECOMES N; THE ATTACHED RELATIONSHIP [ISRCI] BETWEEN THE OLD TONODE AND THE ARC IS MAINTAINED BETWEEN THE NEW TONODE AND THE ARC.

CHEND (RDR) : = RDR. THE SET OR LIST BEING READ BY RDR IS ALTERED SO THAT ISATND(RDR) BECOMES TRUE. THAT IS, THE RDR POINTS TO THE NEW END OF THE SET OR LIST.

CHOBJ (OBJ1,OBJ2) : = OBJ1. THE OBJECT(OBJ1) BECOMES OBJ2, AND ALL ATTACHED AND RELATED RELATIONSHIPS ARE MAINTAINED FOR OBJ1. NOTE : OBJ1 MUST NOT BE AN ATOM.

CHORIG (RDR,RS) : = RDR. FIRST THE READER IS RESTARTED [SEE RESTRT], AND THEN ITS ORIGIN IS CHANGED TO THE READABLE STRUCTURE RS.

COMPOS (VAL,FUN1,FUN2,ARG2,ARG3) : = APPLY(FUN1,APPLY(FUN2,VAL,ARG3), ARG2,ARG3).

CONCAT (L1,L2) : = L1. THE LISTS L1 AND L2 ARE CONCATENATED, AND L2 BECOMES EMPTY. THE LENGTH OF L1 BECOMES THE SUM OF THE FORMER LENGTHS OF L1 AND L2. LAST(L1) BECOMES THE FORMER LAST(L2).

CONECT (RDR1,RDR2) : = RDR1. ESSENTIALLY, RDR1 AND RDR2 ARE CONCATENATED. RDR2 IS UNAFFECTED, BUT RDR1 IS ENLARGED BY THE ADDITION OF THE ELEMENTS OF RDR2 AT THE END OF THE READER [STACK] RDR1. ISDEEP(RDR1) IS ALSO GUARANTEED TRUE.

COPYRD (RDR) : CREATES AND RETURNS A READER X SUCH THAT:
(1) ORIGIN(X) = ORIGIN(RDR)
(2) REED(X) = REED(RDR)
(3) ISDEEP(X) = 0.

CRARC (N1,OBJ,N2) : = ATTRC(CRISR(N1,OBJ,N2)).

CRARCI (N1,OBJ,N2) : = ATTRCI(CRISR(N1,OBJ,N2)).

CRARCO (N1,OBJ,N2) : = ATTRCO(CRISR(N1,OBJ,N2)).

CREEDR (RS) : CREATES AND RETURNS A READER X OF THE STRUCTURE RS; RS MAY BE A NODE, GRAPH, LIST, SET OR READER.
ORIGIN(X) = RS   REED(X) = 0   ISDEEP(X) = 0

CREGR (OBJ) : CREATES AND RETURNS A NEW, EMPTY, UNRELATED GRAPH WITH OBJECT OBJ.

CREL (OBJ) : CREATES AND RETURNS A NEW EMPTY LIST WITH OBJECT OBJ.

CREPS (Q) : CREATES AND RETURNS A NEW, EMPTY, NON-AFFIXED PSET.

CRISN (OBJ,G) : CREATES AND RETURNS A NEW ISOLATED NODE N :
OBJECT(N) = OBJ, GRAPH(N) = G, ISREL(N) = 0,
AND ISATTN(N) = 0.

```
CRISP (OBJ,VAL) : CREATES AND RETURNS A NEW ISOLATED PAIR P :
       OBJECT(P) = OBJ AND VALUE(P) = VAL.

CRISR (N1,OBJ,N2) : CREATES AND RETURNS A NEW ISOLATED ARC RC:
       FRNODE(RC) = N1     OBJECT(RC) = OBJ     TONODE(RC) = N2

CRNODE (OBJ,G) : = RELATE(ATTND(CRISN(OBJ,G))).

CURARC (Q) : RETURNS THE MOST RECENT ARC [CURRENT ARC] CROSSED BY A
       NODE OR GRAPH READER.

CURCI (N) : RETURNS THE CURRENT ARC INCOMING TO NODE N; ELSE Ø IF THERE IS
       NONE.  THE EXISTENCE OF ONLY ONE ARC IN THE RSETI(N) CAUSES IT TO BE
       THE CURRENT ARC INCOMING BY DEFAULT.

CURCO (N) : RETURNS THE CURRENT ARC OUTGOING FROM NODE N; ELSE Ø IF THERE IS
       NONE.  THE EXISTENCE OF ONLY ONE ARC IN THE RSETO(N) CAUSES IT TO BE
       THE CURRENT ARC OUTGOING BY DEFAULT.

DELETE (RDR) : = RDR. THE LIST READER RDR IS MOVED IN ONE POSITION [SEE TI],
       AND THE ELEMENT WHICH IT WAS PREVIOUSLY READING [SEE REED] IS
       REMOVED FROM THE LIST. IF THE FINAL REMAINING ELEMENT IS DELETED
       IN THIS MANNER [THE LENGTH(ORIGIN(RDR)) BECOMES Ø], THEN THE
       READER IS RESTARTED [SEE RESTRT]. AFTER DELETE HAS BEEN EXECUTED,
       TO(RDR) WILL MOVE RDR TO THE ELEMENT PREVIOUSLY JUST OUT FROM
       THE DELETED ELEMENT.

DELFT (LS,FUN,ARG2,ARG3,ARG4,ARG5) : = LS. THE ELEMENTS OF THE LINEAR
       STRUCTURE LS FOR WHICH APPLY(FUN,ELEMENT,ARG2,ARG3,ARG4,ARG5)
       IS TRUE ARE DELETED FROM THE STRUCTURE. WHILE IN DELFT, THE USER MAY
       NOT CHANGE THE STRUCTURE LS IN ANY WAY; ANY OTHER STRUCTURES MAY BE
       ARBITRARILY AFFECTED.
       NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

DETND (N) : = N. THE NODE N IS DETACHED FROM THE NSET(OBJECT(N)).

DETPR (P,PS) : = P. THE PAIR P IS DETACHED FROM THE PSET PS.

DETRC (RC) : = DETRCO(DETRCI(RC)).

DETRCI (RC) : = RC.  THE ARC RC IS DETACHED FROM THE RSETI(TONODE(RC)).
       [IT MAY BE SAID THAT THE ARC IS NO LONGER VISIBLE AT THE TONODE.]
       NOTE : IF THE ARC RC WAS THE CURCI(TONODE(RC)) THEN THE ARC
              IMMEDIATELY PRECEDING RC IN THE RSETI BECOMES THE NEW
              CURCI(TONODE(RC)).

DETRCO (RC) : = RC.  THE ARC RC IS DETACHED FROM THE RSETO(FRNODE(RC)).
       [IT MAY BE SAID THAT THE ARC IS NO LONGER VISIBLE AT THE FRNODE.]
       NOTE : IF THE ARC RC WAS THE CURCO(FRNODE(RC)) THEN THE ARC
              IMMEDIATELY PRECEDING RC IN THE RSETO BECOMES THE NEW
              CURCO(FRNODE(RC)).

DSEND (RDR) : = DSNDTO(RDR,REED(RDR)).

DSNDTO (RDR,RS) : = CONECT(CREEDR(RS),RDR).

ECHO (TAPENUMBER) : = TAPENUMBER. THE DATA READ BY RDEXP [SEE RDFILE]
       IS ECHO-PRINTED, A LOGICAL RECORD AT A TIME, ON FILE TAPENUMBER IF
       TAPENUMBER IS POSITIVE; OTHERWISE THE PRINTING IS SUPPRESSED.
       INITIALLY : ECHO(Ø).
```

EQUAL (OBJ1,OBJ2) : = OBJ1 IF OBJ1 AND OBJ2 ARE IDENTICAL, OR IF OBJ1 AND OBJ2 ARE LISTS SUCH THAT THEIR ELEMENTS ARE EQUAL.

EQULFT (ARG1,FUN,ARG2) : = ARG1 IF IDENT(APPLY(FUN,ARG1),ARG2) IS TRUE; ELSE EQULFT : = 0.

FIRST (LS) : RETURNS THE FIRST ELEMENT OF THE LINEAR STRUCTURE LS.

FRNODE (RC) : IN THE ARC <N1,OBJ,N2> , N1 IS THE FROM-NODE.

GET (OBJ1,OBJ2) : GENERALLY SPEAKING, GET : = VAL WHERE PUT(OBJ1,OBJ2,VAL) WAS LAST EXECUTED. MORE ACCURATELY, GET : = VALUE(P), WHERE P IS THE FIRST PAIR IN THE PSET AFFIXED TO OBJ1 SUCH THAT OBJECT(P) = OBJ2. IF NO SUCH PAIR OR PSET EXISTS, THE VALUE IS 0.

GRAPH (N) : RETURNS THE GRAPH ON WHICH NODE N RESIDES.

GRSET (G) : RETURNS THE SET OF ALL RELATED GRAPHS.

HANG (OBJ,VAL) : = OBJ. THE VALUE(OBJ) BECOMES VAL.

HSNSET (OBJ) : RETURNS THE SET [NSET] OF ATTACHED NODES WITH OBJECT = OBJ. IF NONE HAVE BEEN ATTACHED, HSNSET : = 0.

HSPSET (OBJ) : RETURNS THE PSET AFFIXED TO OBJ; IF NONE, HSPSET : = 0.

IDENT (ARG1,ARG2) : = ARG1 IF ARG1 AND ARG2 ARE IDENTICAL; ELSE IDENT : = 0. SINCE IDENT IS A BITWISE COMPARISON, ARG1 AND ARG2 NEED NOT BE GROPE VALUES.

IMAGE (ATM,VECTOR,NUM1) : RETURNS THE [FIRST WORD OF THE] BITS PASSED IN THE CALL ATM = QUOTE(BITS) OR ATM = ATOM(BITS,NUM2) -- UNLESS NUM2 IS -3, IN WHICH CASE THE (HOLLERITH) NAME OF THAT EXTERNAL REFERENCE IS RETURNED. IF MORE THAN ONE WORD IS REQUIRED TO CONTAIN THE IMAGE OF ATM, THEN (AT MOST) THE FIRST NUM1 WORDS OF BITS IS STORED INTO VECTOR.

INTGER (ARG) : = ARG. THE MOTIVATION FOR THIS FUNCTION IS AS FOLLOWS: M = ARG CAUSES MODE CONVERSION, AND THE EFFECT IS A CATASTROPHE IF M IS TO BE USED AS A GROPE VALUE; HENCE M = INTGER(ARG).

ISARC (VAL) : = VAL IF VAL IS AN ARC; 0 OTHERWISE.

ISATBG (RDR) : = RDR IF RDR IS POINTING AT THE FIRST ELEMENT OF ITS [SET, LIST, OR READER] ORIGIN; 0 OTHERWISE.

ISATND (RDR) : = RDR IF RDR IS POINTING AT THE LAST ELEMENT OF ITS [SET, LIST, OR READER] ORIGIN; 0 OTHERWISE.

ISATOM (VAL) : = VAL IF VAL IS AN ATOM; 0 OTHERWISE. VAL IS AN ATOM IF IT WAS GENERATED BY A CALL TO ATOM OR QUOTE.

ISATTN (N) : = N IF THE NODE N IS ATTACHED TO THE NSET(OBJECT(N)).

ISATTP (P) : = P IF THE PAIR P IS ATTACHED TO SOME PSET.

ISDEEP (RDR) : = RDR IF THE READER RDR MAY BE ASCENDED; 0 OTHERWISE.

ISFUN (VAL) : = VAL IF VAL IS THE RESULT OF SOME VAL = ATOM(BITS,-3); 0 OTHERWISE.

ISGRAF (VAL) : = VAL IF VAL IS A GRAPH; 0 OTHERWISE.

ISGRST (VAL) : = VAL IF VAL IS THE GRAPHSET [GRSET]; Ø OTHERWISE.

ISHOL (VAL) : = VAL IF VAL IS AN ATOM WITH A HOLLERITH [DISPLAY-CODE] IMAGE [THAT IS, IT WAS CREATED BY A CALL TO ATOM(BITS,NUM) WITH NUM ≥ 1 OR NUM = -3]; Ø OTHERWISE.

ISINT (VAL) : = VAL IF VAL IS AN ATOM WITH AN INTEGER [BINARY] IMAGE [THAT IS, IT WAS CREATED BY A CALL TO ATOM(BITS,Ø) OR BY A CALL TO QUOTE]; Ø OTHERWISE.

ISLIST (VAL) : = VAL IF VAL IS A LIST; Ø OTHERWISE.

ISLNSR (VAL) : = VAL IF VAL IS A LINEAR STRUCTURE [THE GRSET, THE OBSET, A LIST, PSET, RSETI, RSETO, NSET, NDSET OR READER]; Ø OTHERWISE.

ISMODS (Q) : = -1 IF GROPE IS IN THE STACK MODE; Ø OTHERWISE.

ISNDST (VAL) : = VAL IF VAL IS A NODESET [NDSET] OF SOME GRAPH; Ø OTHERWISE.

ISNODE (VAL) : = VAL IF VAL IS A NODE; Ø OTHERWISE.

ISNSET (VAL) : = VAL IF VAL IS AN NSET [OF SOME OBJECT]; Ø OTHERWISE.

ISNUM (VAL) : = VAL IF VAL IS A NUMERIC ATOM [FIXED- OR FLOATING-POINT]; Ø OTHERWISE.

ISOBJ (VAL) : = VAL IF VAL IS A GROPE OBJECT [ATOM, PAIR, ARC, NODE, GRAPH OR LIST]; Ø OTHERWISE.

ISOBST (VAL) : = VAL IF VAL IS THE OBSET; Ø OTHERWISE.

ISPAIR (VAL) : = VAL IF VAL IS A PAIR; Ø OTHERWISE.

ISPSET (VAL) : = VAL IF VAL IS A PSET; Ø OTHERWISE.

ISQINT (VAL) : = VAL IF VAL IS THE RESULT OF SOME VAL = QUOTE(BITS); Ø OTHERWISE. IF ISQINT(VAL) IS TRUE, THEN SO ARE ISINT(VAL) AND ISATOM(VAL).

ISRCI (RC) : = RC IF THE ARC RC IS IN THE RSETI(TONODE(RC)); Ø OTHERWISE.

ISRCO (RC) : = RC IF THE ARC RC IS IN THE RSETO(FRNODE(RC)); Ø OTHERWISE.

ISRDR (VAL) : = VAL IF VAL IS A READER; Ø OTHERWISE.

ISRDSR (VAL) : = VAL IF VAL IS A READABLE STRUCTURE [A LINEAR STRUCTURE, NODE OR GRAPH]; Ø OTHERWISE.

ISREAL (VAL) : = VAL IF VAL IS AN ATOM WITH A FLOATING-POINT [REAL] IMAGE; Ø OTHERWISE.

ISREL (NG) : = NG IF THE NODE OR GRAPH NG IS RELATED; Ø OTHERWISE.

ISRSTI (VAL) : = VAL IF VAL IS AN RSETI [OF SOME NODE]; Ø OTHERWISE.

ISRSTO (VAL) : = VAL IF VAL IS AN RSETO [OF SOME NODE]; Ø OTHERWISE.

ISVAL (ARG) : = ARG IF ARG IS A LEGAL GROPE VALUE : AN OBJECT, A SET, OR A READER; Ø OTHERWISE.

LAST (LS) : RETURNS THE LAST ELEMENT OF THE LINEAR STRUCTURE LS.

LENATM (ATM) : = THE NUMBER OF WORDS CONTAINING THE IMAGE OF THE ATOM ATM.

LENGTH (LS) : RETURNS THE [INTEGER] NUMBER OF TOP-LEVEL ELEMENTS IN THE LINEAR STRUCTURE LS. IF ANY OTHER TYPE OF ARGUMENT IS PASSED, LENGTH : = -1. FOR SETS AND LISTS, THE LENGTH IS IMMEDIATELY AVAILABLE [STORED].

LIST (ARG1,ARG2,ARG3,ARG4,ARG5) : RETURNS A NEW LIST L WITH OBJECT(L) = QUOTE(0). IF 5 ARGUMENTS [GROPE VALUES] ARE PASSED TO LIST, THE NEW LIST WILL CONTAIN THOSE 5 ELEMENTS. OTHERWISE THE LIST IS COMPOSED OF K-1 ELEMENTS WHERE ARGK IS THE FIRST ARGUMENT WHICH IS NOT A GROPE VALUE. [0 IS NOT A GROPE VALUE.] NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 5.

LOFT (LS,FUN,ARG2,ARG3,ARG4,ARG5) : RETURNS A NEW LIST L WITH OBJECT(L) = QUOTE(0). FUNCTION FUN IS APPLIED TO THE SUCCESSIVE ELEMENTS IN THE LINEAR STRUCTURE LS. IF FUN RETURNS A GROPE VALUE, VAL, THEN VAL IS QUEUED INTO THE NEW LIST. NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

MACURI (RC) : = RC. CURCI(TONODE(RC)) BECOMES RC. [THAT IS, RC BECOMES THE CURRENT ARC INTO THE TONODE(RC).]

MACURO (RC) : = RC. CURCO(FRNODE(RC)) BECOMES RC. [THAT IS, RC BECOMES THE CURRENT ARC OUT FROM THE FRNODE(RC).]

MAPFT (LS,FUN,ARG2,ARG3,ARG4,ARG5) : = LS. MAPFT APPLIES FUN TO EACH ELEMENT IN THE LINEAR STRUCTURE LS. [X = APPLY(FUN,ELEMENT,ARG2,ARG3, ARG4,ARG5) IS CALLED N TIMES WHERE N = LENGTH(LS).] NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

MARGIN (NUM) : THE MARGIN, INITIALLY SET TO 1, IS THE NUMBER OF COLUMNS ON THE LEFT OF THE GROPE OUTPUT BUFFER WHICH PRINFT WILL NOT FILL. IF NUM IS NONNEGATIVE, THEN THE MARGIN BECOMES NUM. IN ANY EVENT THE MARGIN IS RETURNED AS THE VALUE OF THE FUNCTION.

MAXERR (NUM) : = NUM. HENCEFORTH GROPE WILL ABORT THE PROGRAM AFTER NUM ERRORS.
INITIALLY : MAXERR(10)

MEMBER (VAL,LS) : = VAL PROVIDED THAT VAL IS ONE OF THE ELEMENTS IN THE LINEAR STRUCTURE LS; ELSE 0. IN MANY CASES MEMBERSHIP IN SETS [GRSET, NDSET, NSET, RSETI, RSETO, ETC.] MAY BE TESTED MORE EFFICIENTLY WITH AN APPROPRIATE IS-FUNCTION [ISREL, ISATTN, ISRCI, ISRCO, ETC].

MERGE (RDR,L) : = RDR. THE ELEMENTS IN THE LIST L ARE INSERTED IN ORDER TO THE RIGHT OF [OUT FROM] THE LIST READER RDR. L BECOMES EMPTY, AS IN CONCAT. THE EFFECT IS THAT THE NEXT TO(RDR) WILL RETURN THE VALUE WHICH WAS FORMERLY FIRST(L).

MESAGE (TAPENUMBER) : = TAPENUMBER. IF TAPENUMBER ≤ 0 THEN ERROR MESSAGES WILL NOT APPEAR, ELSE THEY WILL BE WRITTEN ON FILE TAPENUMBER.
INITALLY : MESAGE(6)

MODEQ (Q) : = 0; HENCEFORTH GROPE IS IN THE QUEUE MODE.

MODES (Q) : = -1; HENCEFORTH GROPE IS IN THE STACK MODE.
INITIALLY : MODES(Q)

MOVEND (RDR) : = RDR. THE LINEAR STRUCTURE READER RDR IS MOVED DIRECTLY TO THE LAST ELEMENT IN ITS ORIGIN. IF LENGTH(ORIGIN(RDR)) = 0, THEN RDR IS RESTARTED.

```
MOVETO (RDR,VAL) : = RDR. THE READER RDR MOVES DIRECTLY TO THE VALUE
       VAL WITHIN ITS ORIGIN.  [FOR A LIST READER, RDR IS RESTARTED,
       THEN MOVED TO THE FIRST OCCURENCE OF VAL.]  REED(RDR) BECOMES VAL.

NDSET (G) : RETURNS THE SET OF NODES ON GRAPH G THAT ARE RELATED.

NSET (OBJ) : RETURNS THE SET OF ATTACHED NODES WITH OBJECT = OBJ. [THE EMPTY
       NSET IS CREATED IF NECESSARY.]

OBJECT (OBJ) : IF OBJ IS A PAIR, THEN OBJECT : = X WHERE OBJ = CRISP(X,V).
       IF OBJ IS AN ARC, THEN OBJECT : = X WHERE OBJ = CRISR(N,X,M).
       IF OBJ IS A NODE, THEN OBJECT : = X WHERE OBJ = CRISN(X,G).
       IF OBJ IS A GRAPH, THEN OBJECT : = X WHERE OBJ = CREGR(X).
       IF OBJ IS A LIST, THEN OBJECT : = X WHERE OBJ = CREL(X).
       NOTE : OBJ CANNOT BE AN ATOM.

OBSET (Q) : RETURNS THE OBSET. THE OBSET IS THE SET OF ATOMS CREATED BY
       ATOM (BITS,NUM). THE SET IS NOT ORDERED, AND THE USER MAY NOT
       DIRECTLY AFFECT IT [SUCH AS WITH DELFT, SERTO, ETC], BUT IT
       MAY BE SEARCHED WITH THE READER MECHANISM [USING TO, LOFT, ETC].

ORFT (LS,FUN,ARG2,ARG3,ARG4,ARG5) : = 0 IF APPLY(FUN,ELEMENT,ARG2,ARG3,
       ARG4,ARG5) RETURNS FALSE [=0] FOR EACH SUCCESSIVE ELEMENT IN
       THE LINEAR STRUCTURE LS; IF FUN RETURNS TRUE [≠0], THE PROCESS IS
       TERMINATED AND THAT VALUE [≠0] IS RETURNED.
       NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

ORIGIN (RDR) : THE ORIGIN OF THE READER RDR IS THE READABLE STRUCTURE RS USED
       IN RDR = CREEDR(RS), WHETHER CALLED DIRECTLY [BY THE USER] OR BY
       DSNDTO. [THAT IS, THE ORIGIN IS LOCAL TO THE CURRENT LEVEL OF
       THE READER, AND [NORMALLY] CHANGES WHEN ASEND OR DSNDTO IS EXECUTED.]

POP (L) : REMOVES THE FIRST ELEMENT OF THE LIST L AND RETURNS THAT ELEMENT.

PRFILE (TAPENUMBER,NUM) : = TAPENUMBER.  TERPRI WILL WRITE THE CONTENTS OF
       THE GROPE OUTPUT BUFFER ON FILE TAPENUMBER OF COLUMN LENGTH NUM.
       INITALLY : PRFILE(6,136).

PRINFT (VAL,FUN,AGR2,ARG3,ARG4,ARG5) : = VAL. THE FOLLOWING ALGORITHM
       DETERMINES WHAT IS WRITTEN INTO THE GROPE OUTPUT BUFFER --
            SET X = VAL:
                 (1) IF X IS AN ATOM, WRITE ITS IMAGE INTO THE BUFFER.
            OR   (2) IF X IS A PAIR, ARC, NODE, GRAPH, OR READER, THEN SET X =
                     APPLY(FUN,X,ARG2,ARG3,ARG4,ARG5); IF ISVAL(X) IS TRUE
                     GO TO STEP (1), ELSE DO NOT WRITE X.
            ELSE (3) X MUST BE A LINEAR STRUCTURE [NOT A READER] :
                     WRITE THE CHARACTER (. IF X IS A LIST, WRITE ITS
                     OBJECT [AS ABOVE], DELIMITED BY THE CHARACTER ↓,
                     PROVIDED ITS OBJECT IS NOT A LIST OR QUOTE(0).
            AND  (4) SET X [SUCCESSIVELY] TO EACH ELEMENT IN THE
                     LINEAR STRUCTURE AND PROCEED AS (1) ABOVE; THEN
                     WRITE THE CHARACTER ) AND EXIT. THE GROPE
                     OUTPUT BUFFER IS PRINTED AND EMPTIED [TERPRI]
                     ONLY AS NECESSARY TO PREVENT BUFFER OVERFLOW.
       BE CAREFUL TO NOTE THAT THERE ARE MANY WAYS TO GENERATE
       INFINITE LOOPS. A RECOMMENDED CALL IS PRINFT(VAL,OBJECT) WHICH
       WORKS SUCESSFULLY IN MOST CASES.
       NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

PRXPFT (VAL,FUN,ARG2,ARG3,ARG4,ARG5) : = TERPRI(PRINFT(VAL,FUN,ARG2,ARG3,
       ARG4,ARG5)).

       NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.
```

PSET (OBJ) : RETURNS THE PSET AFFIXED TO OBJ. IF NONE EXISTS, ONE IS CREATED, AFFIXED TO OBJ AND RETURNED.
TO THE EXTENT TO WHICH OBJECTS SHARE THE SAME PSET, THEY SHARE THE SAME VALUE.

PUT (OBJ1,OBJ2,VAL) : = OBJ1. GET(OBJ1,OBJ2) BECOMES VAL. PUT STACKS OR QUEUES THE CRISP(OBJ2,VAL) ON THE PSET(OBJ1), UNLESS THERE ALREADY IS A PAIR P ON THE PSET(OBJ1) WITH THE OBJECT(P) = OBJ2, IN WHICH CASE HANG(P,VAL) IS EXECUTED.

QUEUE (VAL,L) : = L . THE VALUE VAL IS QUEUED ONTO THE LIST L.
THE QUEUE-STACK MODE IS UNAFFECTED BY QUEUE.

QUOTE (BITS) : RETURNS A QUOTED INTEGER X: ISINT(X), ISQINT(X), AND ISATOM(X) ARE TRUE. THE ABSOLUTE VALUE OF THE INTEGER BITS MUST BE ≤ 131,071. X DOES NOT APPEAR IN THE GROPE SPACE [VECTOR].
NOTE : EQUAL(ATOM(BITS,0),QUOTE(BITS)) IS FALSE.

RDEXP (Q) : RETURNS THE NEXT ATOM OR LIST [A BALANCED SET OF PARENTHESES AND ATOMS] IN THE INPUT BUFFER, READING LOGICAL RECORDS [CARDS] INTO THE BUFFER FROM THE CURRENT RDFILE AS NECESSARY TO COMPLETE THE OPERATION. NOTE : IF THE USER WISHES TO USE READ AND RDEXP ON THE SAME FILE, WHEN CHANGING THE READING MODE FROM READ TO RDEXP OR RDEXP TO READ ALWAYS START THE DATA ON A NEW LOGICAL RECORD [CARD]. WHEN RDEXP ENCOUNTERS AN END-OF-FILE, RDEXP : = 0. THE OBJECT OF ANY NEW LIST IS QUOTE(0).

RDFILE (TAPENUMBER,NUM) : = TAPENUMBER. THE FUNCTION RDEXP WILL READ FROM FILE TAPENUMBER OF [COLUMN] LENGTH NUM.
INITIALLY : RDFILE(5,80)

REALE (IARG) : = IARG. THE MOTIVATION FOR THIS FUNCTION IS AS FOLLOWS:
X = IARG CAUSES MODE CONVERSION, AND THE EFFECT IS A CATASTROPHE IF X IS TO BE USED AS A GROPE VALUE; HENCE X = REALE(IARG).

REED (RDR) : RETURNS THE VALUE AT WHICH THE READER RDR IS POINTING. IF THE READER IS UNMOVED [NOT READING ANYTHING], REED : = 0.
IF RDR IS AN OBSET READER, AN ATOM IS RETURNED.
IF RDR IS A PSET READER, A PAIR IS RETURNED.
IF RDR IS AN RSETI OR RSETO READER, AN ARC IS RETURNED.
IF RDR IS A NODE, GRAPH, NSET OR NDSET READER, A NODE IS RETURNED.
IF RDR IS A GRSET READER, A GRAPH IS RETURNED.
IF RDR IS A READER READER, A READER IS RETURNED.
IF RDR IS A LIST READER, THEN A GROPE VALUE IS RETURNED.

RELATE (NG) : = NG. THE NODE [OR GRAPH] NG IS STACKED OR QUEUED INTO THE APPROPRIATE NODESET [OR THE GRAPHSET] IF ISREL(NG) IS FALSE. HENCEFORTH ISREL(NG) IS TRUE.

RESTRT (RDR) : = RDR. THE REED(RDR) BECOMES 0. [THE READER BECOMES UNMOVED.]

RETURN (ARG) : WHEN CALL RETURN(ARG) IS THE LAST EXECUTED STATEMENT IN A FUNCTION, ALPHA, THEN THE FOLLOWING OCCURS:

(1) THE LAST VECTOR SAVED BY SAVAR IS NO LONGER PROTECTED FROM THE GARBAGE COLLECTOR.

(2)
```
      ALPHA = ARG
      RETURN
```

A TYPICAL FORTRAN-GROPE FUNCTION MIGHT BE:
```
FUNCTION ALPHA (A,B,C,D)
.
```

```
        .
        .
        CALL SAVAR(ALPHA,3)
        Y = F(A)
        ALPHA = F1(B,Y)
        Z = F2(C,D,ALPHA)
        CALL RETURN(Z)
        .
        .
        .
        END
NOTE : THE FUNCTION NAME AND THE TWO LOCAL VARIABLES, Y AND Z,
       ARE PROTECTED FROM THE CALL SAVAR(ALPHA,3) UNTIL THE
       CALL RETURN(Z) STATEMENT.
WARNING : WHEN RETURN IS USED, SAVAR MUST PROTECT ALL AND ONLY
          THE LOCAL VARIABLES.
```

REVERT (RDR) : = RDR IF ISDEEP(RDR) = 0; ELSE REVERT : = REVERT(ASEND(RDR)).
[REVERT RETURNS THE LAST READER IN THE STACK RDR.]

REVRC (RC) : = RC. THE ARC RC IS REVERSED - THE TONODE(RC) BECOMES THE FRNODE(RC), AND VICE-VERSA. THE ATTACHED RELATIONSHIPS ARE MAINTAINED:
IF ISRCO(RC) WAS TRUE, THEN IT IS STILL TRUE.
IF ISRCI(RC) WAS TRUE, THEN IT IS STILL TRUE.

RSETI (N) : RETURNS THE SET OF ARCS INTO NODE N THAT ARE ATTACHED IN.
[ISRCI IS TRUE FOR ALL ARCS IN THE RSETI.]

RSETO (N) : RETURNS THE SET OF ARCS FROM NODE N THAT ARE ATTACHED OUT.
[ISRCO IS TRUE FOR ALL ARCS IN THE RSETO.]

SAVAR (VECTOR,NUM) : = VECTOR. THE FIRST NUM VARIABLES IN VECTOR ARE SAVED - THAT IS, NONE OF THE GROPE STRUCTURES NAMED BY THESE VARIABLES AT GARBAGE-COLLECTION TIME WILL BE DESTROYED. [SEE SETUP FOR THE DESCRIPTION OF THE GARBAGE COLLECTOR.]
SAVAR ZEROES OUT THE FIRST NUM VARIABLES IN VECTOR.

SAVCOM (VECTOR,NUM) : IS THE SAME AS SAVAR, BUT SAVCOM DOES NOT ZERO OUT THE FIRST NUM VARIABLES IN VECTOR.

SERTI (RDR,VAL) : = RDR. IF RDR IS A LIST READER, THEN THE VALUE VAL IS INSERTED INTO THE LIST INWARD FROM RDR; THE NEXT TI(RDR) WILL PRODUCE VAL.
IF RDR IS A NODE OR GRAPH READER, THEN THE ARC VAL IS INSERTED IN THE RSETI(REED(RDR)) SO THAT THE NEXT TI(RDR) WILL CROSS THE ARC VAL [SUBJECT TO THE RESTRICTION ON THE GRAPH READER]. IN THIS CASE, ISRCI(VAL) MUST BE FALSE BEFORE SERTI, AND WILL BE TRUE AFTERWARDS. ALSO, THE TONODE(VAL) MUST = REED(RDR).

SERTO (RDR,VAL) : = RDR. IF RDR IS A LIST READER, THEN THE VALUE VAL IS INSERTED INTO THE LIST OUTWARD FROM RDR; THE NEXT TO(RDR) WILL PRODUCE VAL.
IF RDR IS A NODE OR GRAPH READER, THEN THE ARC VAL IS INSERTED IN THE RSETO(REED(RDR)) SO THAT THE NEXT TO(RDR) WILL CROSS THE ARC VAL [SUBJECT TO THE RESTRICTION ON THE GRAPH READER]. IN THIS CASE, ISRCO(VAL) MUST BE FALSE BEFORE SERTO, AND WILL BE TRUE AFTERWARDS. ALSO, THE FRNODE(VAL) MUST = REED(RDR).

SETUP (VECTOR,NUM1,NUM2,NUM3) : = 0. THIS SETS UP GROPE. NO OTHER GROPE FUNCTION MAY BE EXECUTED UNTIL AFTER SETUP; HOWEVER, THE SYSTEM MAY BE RE-INITALIZED [FOR EXECUTION OF A DIFFERENT PROGRAM, FOR EXAMPLE] BY CALLING SETUP AGAIN. SETUP USES VECTOR OF SIZE NUM1 DIMENSIONED

BY THE USER, WITH NUM1 * NUM2 [0≤NUM2<1] WORDS RESERVED FOR FULL WORDS AND NUM3 WORDS RESERVED FOR THE GARBAGE COLLECTOR STACK. THE FOLLOWING IS AN EXACT DESCRIPTION OF FULL WORD UTILIZATION IN THIS IMPLEMENTATION OF GROPE, ASSUMING AN ATOM IS ACTUALLY BEING CREATED:

(1) FOR ATOM(BITS,-NUM) = NUM FULL WORDS WHERE NUM IS 1 OR 2.
(2) FOR ATOM(BITS,0) = 1 FULL WORD IF BITS > 2 ↑ 18 - 1 .
(3) FOR NUM ≥ 1, AND MORE THAN 3 CHARACTERS IN BITS = NUM FULL WORDS.
(4) ALL OTHER CASES REQUIRE NO FULL WORDS.

THE FOLLOWING IS A DESCRIPTION OF WHAT GROPE VALUES ARE NOT DESTROYED BY THE GARBAGE COLLECTOR:

(1) THE GRSET AND THE OBSET.
(2) ALL VALUES NAMED BY THE SAVED VARIABLES. [THE USER SHOULD NOTE THAT EACH CALL TO SAVAR CONSUMES ONE WORD OF THE GARBAGE COLLECTOR STACK.]
(3) IF A LINEAR STRUCTURE IS SAVED, THEN SO ARE ITS ELEMENTS.
(4) IF A STRUCTURE X IS SAVED THEN [ASSUMING WELL DEFINED] SO ARE THE NDSET(X), HSNSET(X), RSETO(X), RSETI(X), HSPSET(X), GRAPH(X), TONODE(X), FRNODE(X), OBJECT(X), VALUE(X), ORIGIN(X), AND REED(X).

AN EXAMPLE INITIALIZATION IS :

```
DIMENSION ARRAY(2000)
BEGIN = SETUP(ARRAY,2000,0.04,200)
```

NOTE : ANY EXECUTION PARAMETER THE USER CAN ALTER [E.G. - RDFILE, PRFILE, MODEQ, ECHO, ETC.] IS NOT RE-INITIALIZED BY SETUP.

SOFT (LS,FUN,ARG2,ARG3,ARG4,ARG5) : THE EFFECT AND VALUE IS LIKE LOFT. HOWEVER, IF VAL IS ALREADY A MEMBER OF THE NEW LIST, VAL IS NOT INSERTED. THUS SOFT PRODUCES A LIST WITH NO DUPLICATED ELEMENTS. NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

STACK (VAL,L) : = L . THE VALUE VAL IS STACKED ONTO THE LIST L. THE QUEUE-STACK MODE IS UNAFFECTED BY STACK.

SUBST (RDR,VAL) : = RDR. THE VALUE VAL IS SUBSTITUTED FOR THE ELEMENT IN THE LIST WHICH IS BEING POINTED AT BY RDR. THAT IS, THE READER DOES NOT CHANGE POSITION, BUT REED(RDR) BECOMES VAL.

TAB (NUM) : IF NUM > 0, THE CURRENT OUTPUT TAB POSITION [AS ON A TYPEWRITER] BECOMES NUM - THAT IS, PRINFT BEGINS FILLING THE GROPE OUTPUT BUFFER AT COLUMN NUM. IN ANY EVENT, THE CURRENT TAB IS RETURNED AS THE VALUE OF THE FUNCTION.

TERPRI (ARG) : = ARG. TERPRI WRITES THE CONTENTS OF THE GROPE OUTPUT BUFFER ON THE CURRENT PRFILE, AND EMPTIES THE BUFFER.

TI (RDR) : = REED(RDR) ONCE THE READER RDR HAS MOVED AS DESCRIBED:
IF RDR IS A LIST READER, THEN RDR TRAVERSES THE LIST INWARD ONE ELEMENT. [THAT IS, RDR MOVES LEFT ONE POSITION.] LISTS ARE CIRCULAR, SO IF THE READER WAS ON THE FIRST ELEMENT BEFORE TI, IT WILL MOVE AROUND TO THE LAST ELEMENT.
IN THE CASE OF THE UNMOVED GRAPH READER RDR,
TI = REED(MOVETO(RDR,VALUE(ORIGIN(RDR)))) . [NOTE THAT IN THIS CASE THE VALUE HANGING FROM THE GRAPH MUST BE ONE OF ITS NODES.]
IF RDR IS A NODE OR GRAPH READER, LET N=REED(RDR) [OR ORIGIN(RDR) IF THE NODE READER RDR IS UNMOVED], THEN THE READER CROSSES THE NEXT ARC IN THE RSETI(N) AFTER THE CURCI(N), AND THE REED BECOMES THE NODE TO WHICH THE READER MOVES. [IN THE CASE OF A GRAPH READER, RDR WILL SEARCH THE ARCS UNTIL IT FINDS ONE WHICH COMES FROM A NODE ON THE GRAPH WHICH IS THE ORIGIN OF THE READER. IF NONE CAN BE FOUND, THE READER DOES NOT MOVE.] AFTER THE MOVE, THE FUNCTION CURARC(Q) WILL RETURN THE ARC CROSSED BY THE READER. THE ARC CROSSED BECOMES THE CURCI OF ITS TONODE.
IF FOR ANY REASON RDR CANNOT MOVE, TI : = 0.

TIFT (RDR,FUN,ARG2,ARG3,ARG4,ARG5) : IS LIKE TI(RDR) FOR THE NODE OR GRAPH READER RDR EXCEPT THAT APPLY(FUN,RC,ARG2,ARG3,ARG4,ARG5) MUST ANSWER TRUE FOR THE ARC UNDER CONSIDERATION [FOR BEING CROSSED], ELSE THE NEXT ARC IN THE RSETI WILL BE CONSIDERED. [THAT IS, FUN IS APPLIED TO THE ARCS IN THE RSETI STARTING AFTER THE CURCI, UNTIL FUN ANSWERS TRUE OR ELSE IT HAS BEEN UNSUCCESSFULLY APPLIED TO ALL, IN WHICH CASE NO MOVE IS MADE.] IF FUN ANSWERS TRUE, THEN TI IS EXECUTED ACROSS THAT ARC, BUT THE VALUE OF TIFT IS THAT VALUE RETURNED BY FUN. IN THE CASE OF THE UNMOVED GRAPH READER RDR, TIFT = REED(MOVETO(RDR,VALUE(ORIGIN(RDR)))) [IN THE GRAPH READER CASE, FUN IS ONLY APPLIED TO THOSE ARCS WHICH COME FROM A NODE ON THE GRAPH.] NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

TO (RDR) : = REED(RDR) ONCE THE READER RDR HAS MOVED AS DESCRIBED: IF RDR IS A LINEAR STRUCTURE READER, RDR TRAVERSES THE STRUCTURE OUTWARD ONE ELEMENT. [THAT IS, RDR MOVES RIGHT ONE POSITION.] SUCH STRUCTURES ARE CIRCULAR [WITH THE EXCEPTION OF THE READER], SO IF THE READER WAS ON THE LAST ELEMENT BEFORE TO, IT WILL MOVE AROUND TO THE FIRST ELEMENT [WITH THE EXCEPTION OF THE READER STRUCTURE, IN WHICH CASE NO MOVE IS MADE]. IN THE CASE OF THE UNMOVED GRAPH READER RDR, TO = REED(MOVETO(RDR,VALUE(ORIGIN(RDR)))) . [NOTE THAT IN THIS CASE THE VALUE HANGING FROM THE GRAPH MUST BE ONE OF ITS NODES.] IF RDR IS A NODE OR GRAPH READER, LET N=REED(RDR) [OR ORIGIN(RDR) IF THE NODE READER RDR IS UNMOVED], THEN THE READER CROSSES THE NEXT ARC IN THE RSETO(N) AFTER THE CURCO(N), AND THE REED BECOMES THE NODE TO WHICH THE READER MOVES. [IN THE CASE OF A GRAPH READER, RDR WILL SEARCH THE ARCS UNTIL IT FINDS ONE WHICH GOES TO A NODE ON THE GRAPH WHICH IS THE ORIGIN OF THE READER. IF NONE CAN BE FOUND, THE READER DOES NOT MOVE.] AFTER THE MOVE, THE FUNCTION CURARC(Q) WILL RETURN THE ARC CROSSED BY THE READER. THE ARC CROSSED BECOMES THE CURCO OF ITS FRNODE. IF FOR ANY REASON RDR CANNOT MOVE, TO : = 0.

TOFT (RDR,FUN,ARG2,ARG3,ARG4,ARG5) : IS LIKE TO(RDR) FOR THE NODE OR GRAPH READER RDR EXCEPT THAT APPLY(FUN,RC,ARG2,ARG3,ARG4,ARG5) MUST ANSWER TRUE FOR THE ARC UNDER CONSIDERATION [FOR BEING CROSSED], ELSE THE NEXT ARC IN THE RSETO WILL BE CONSIDERED. [THAT IS, FUN IS APPLIED TO THE ARCS IN THE RSETO STARTING AFTER THE CURCO, UNTIL FUN ANSWERS TRUE OR ELSE IT HAS BEEN UNSUCCESSFULLY APPLIED TO ALL, IN WHICH CASE NO MOVE IS MADE.] IF FUN ANSWERS TRUE, THEN TO IS EXECUTED ACROSS THAT ARC, BUT THE VALUE OF TOFT IS THAT VALUE RETURNED BY FUN. IN THE CASE OF THE UNMOVED GRAPH READER RDR, TOFT = REED(MOVETO(RDR,VALUE(ORIGIN(RDR)))) [IN THE GRAPH READER CASE, FUN IS ONLY APPLIED TO THOSE ARCS WHICH LEAD TO A NODE ON THE GRAPH.] NOTE : THIS FUNCTION ACCEPTS A VARIABLE NUMBER OF ARGUMENTS UP TO 6.

TONODE (RC) : IN THE ARC <N1,OBJ,N2> , N2 IS THE TO-NODE.

TRUE (ARG) : = ARG. [THUS IF ARG≠0, THEN ARG IS TRUE.]

UNHANG (OBJ) : = OBJ. THE VALUE(OBJ) BECOMES 0. [THE HANGING IS REMOVED.]

UNREL (NG) : = NG. THE NODE [OR GRAPH] NG IS REMOVED FROM THE APPROPRIATE NODESET [OR THE GRAPHSET], AND ISREL(NG) BECOMES FALSE.

VALUE (OBJ) : RETURNS THE MOST RECENT VALUE, VAL, SUCH THAT HANG(OBJ,VAL) WAS EXECUTED; IF NONE, VALUE : = 0.

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. Burstall, R. M., "Formal description of program structure and semantics in first-order logic," in Machine Intelligence 5 (B. Meltzer and D. Michie, eds.), American Elsevier Publishing Co., New York (1970).

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. deBakker, J. W., "Semantics of programming languages," in Advances in Information System Science (J. Tou, ed.), Vol. 2, Plenum Press, New York (1969).

6. Earley, J., "Toward an understanding of data structures," Comm. ACM 14, 10 (Oct. 1971), 617-627.

7. 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 RS1-69, University of Houston, Houston, Texas, 1969.

8. _______, "GRASPE: graph processing a LISP extension," Computation Center Report TNN-84, University of Texas, Austin, Texas 1968.

9. _______, "Use of the intersection rules in the development of new models within GRASPE 1.5," University of Texas, April, 1970 (unpublished manuscript).

10. Greenawalt, E. M., private communication.

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

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

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

14. Hendrix, G. G., "Question answering via canonical verbs and semantic models: a model of textual meaning," Technical Report NL12, January 1973, Department of Computer Science, The University of Texas at Austin.

15. _______, "Modeling simultaneous actions and continuous processes," to appear in Artificial Intelligence Journal.

16. _______, 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.

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

18. Landin, P. J., "Correspondence between ALGOL-60 and Church's lambda notation, Part I and II," Comm. ACM 8, 2 (February 1965), and Comm. ACM 8, 3 (March 1965).

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

20. Lee, J., Computer Semantics, Van Nostrand-Reinhold, 1972.

21. Lehmann, W. P., R. Stachowitz, and Bary Allan Gold, "German-English translation system," Technical Report of the Linguistics Research Center, The University of Texas at Austin (in preparation), 1973.

22. Lucas, P., and K. Walk, "On the formal description of PL/I," in Annual Review in Automatic Programming (L. Bolliet, et al., eds.), Vol. 6, Part 3, Pergamon Press,New York (1969).

23. McCarthy, J., et al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts, 1962.

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

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

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

27. _______, "Introduction to a theory of programming language semantics," Report TSN-4, University of Texas Computation Center, 1969, 10 pp.

28. _______, "Semantic modeling by hierarchical graphs," ACM SIGPLAN Symposium on Programming Language Definition, San Francisco, Calif., August 1969.

29. _______, "Pair grammars, graph languages, and string-to-graph translations," J. of Comp. Sys. Sci., 5, 6 Dec. 1971, 560-595.

30. _______, "A formal definition of ALGOL 60 using hierarchical graphs and pair grammars," Report TSN-33, University of Texas Computation Center, 1973, 82 pp.

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

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

33. Shneiderman, B., "Data Structures: Description, Manipulation, and Evaluation," unpublished Ph.D. dissertation, State University of New York at Stonybrook, 1973.

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

35. _______, "The Graph Processing Language GROPE 2.0," Master's thesis in preparation, The University of Texas at Austin.

36. Stachowitz, Rolf, Voraussetzungen für maschinelle Übersetzung: Probleme, Lösungen, Aussichten, Athenäum Verlag, Frankfurt/M, 1973.

37. _______, Ein Modell linguistischer Performanz, Athenäum Verlag, Frankfurt/M. (in vorbereitung).

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

39. Wegner, P., "The Vienna definition language," Computing Surveys 4, 1 (1972), 5-64.

40. Weizenbaum, J., "Symmetric list processor," Comm. ACM 6, 9 (Sept. 1963).

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

42. Wilson, James P., "Graphical representation of semantic structure," unpublished Master's thesis, The University of Texas at Austin, August 1972.

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

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