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

LIST OF FIGURES

FIGURE PAGE

FIGURE PAGE

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

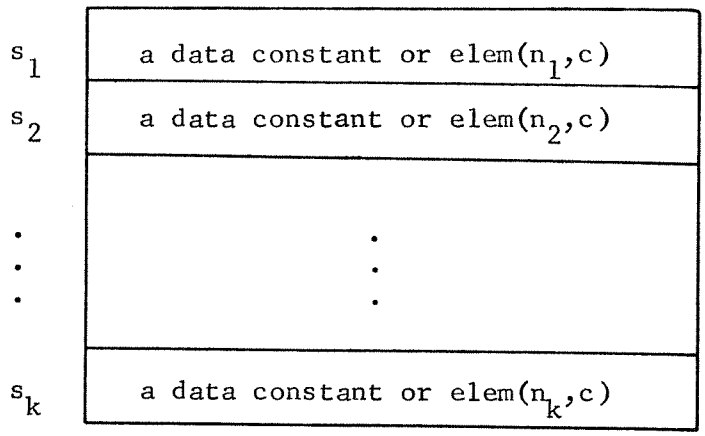
Formal definition of programming language semantics is a relatively new area. Debakker [5] provides a good (although dated) survey of research in the formal study of programming language semantics. Since the concern

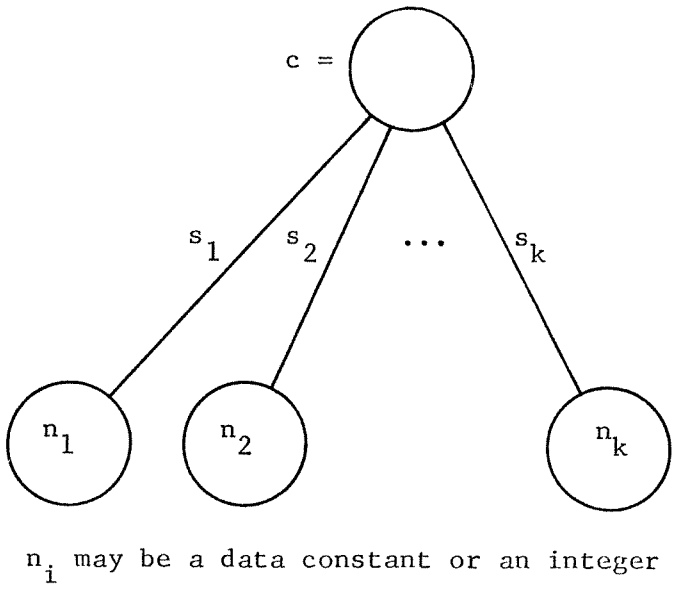
here is with objects and operations on objects, the discussion is limited to the treatment of this limited area of semantics. For each approach we are concerned with three basic questions. First, how is the total data space or "overall state" of the abstract machine conceived? Second, how are data structures 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





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

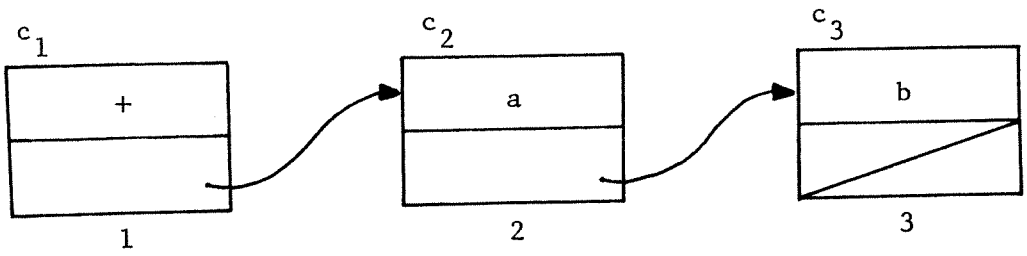


Figure 1.2. A LISP List

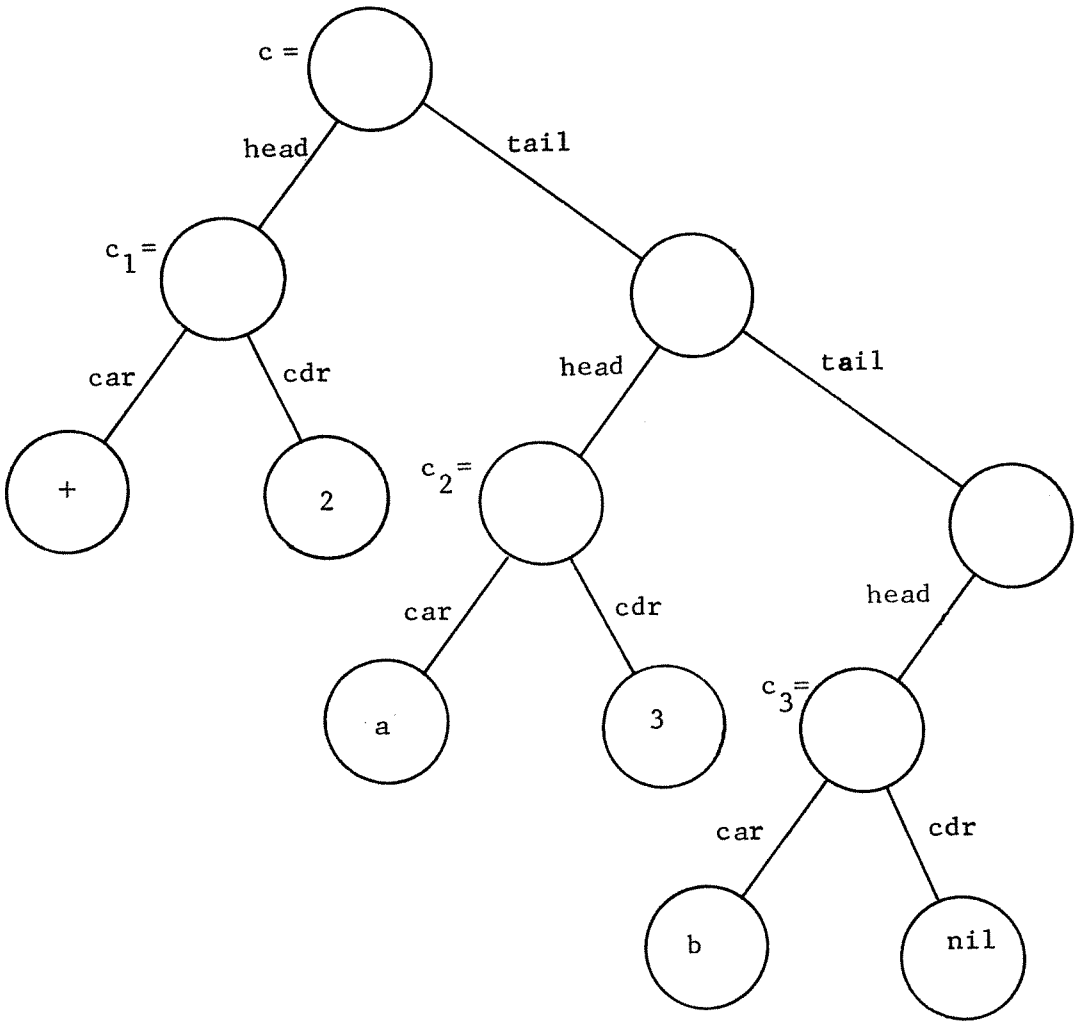


Figure 1.3. VDL Representation of a LISP List

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

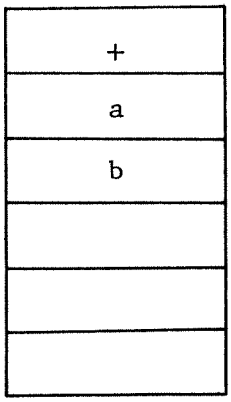


Figure 1.6. A Sample Stack

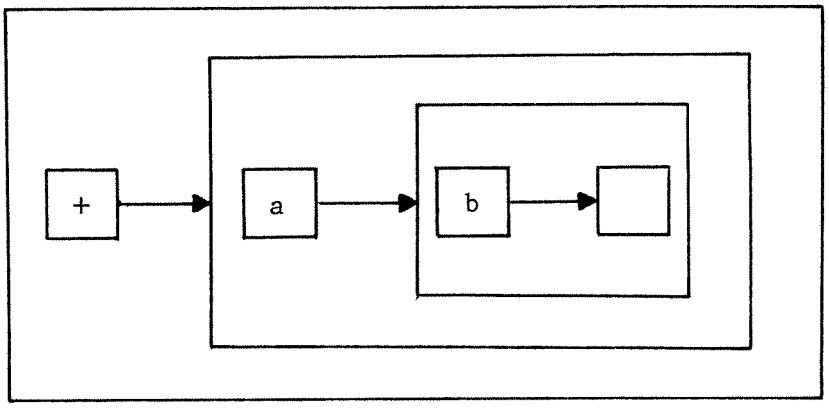


Figure 1.7. H-graph Representation of the Sample Stack

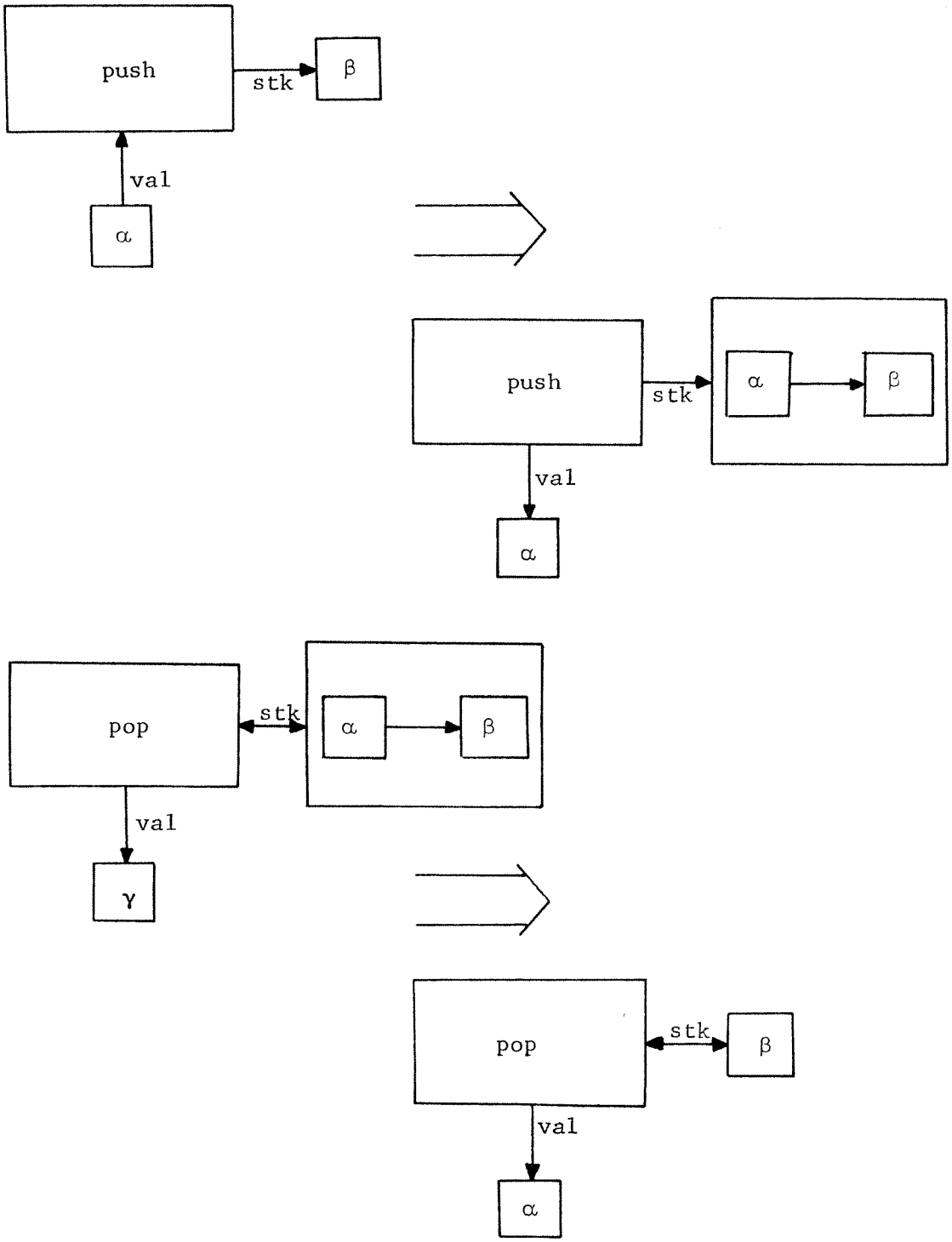


Figure 1.8. H-graph Representation of Stack Operations

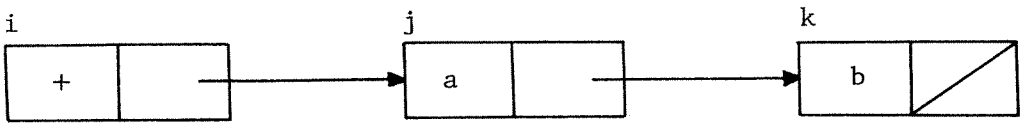
The overall state of the abstract machine in the axiomatic approach is represented by a sentence of the predicate calculus and a "state vector." A state vector, s, is an association of variables with their values. As a program is executed, the values in the state vector get altered (actually a new state 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 car and cdr 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 cons(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, $\underline{op}(H,x_1,x_2,\ldots,x_n) = (H',v)$. For purposes of convenience and naturalness, H and H' are considered implicitly as the underlying (overall) data space and the relationship becomes the familiar $\underline{op}(x_1,x_2,\ldots,x_n) = v$, 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.

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

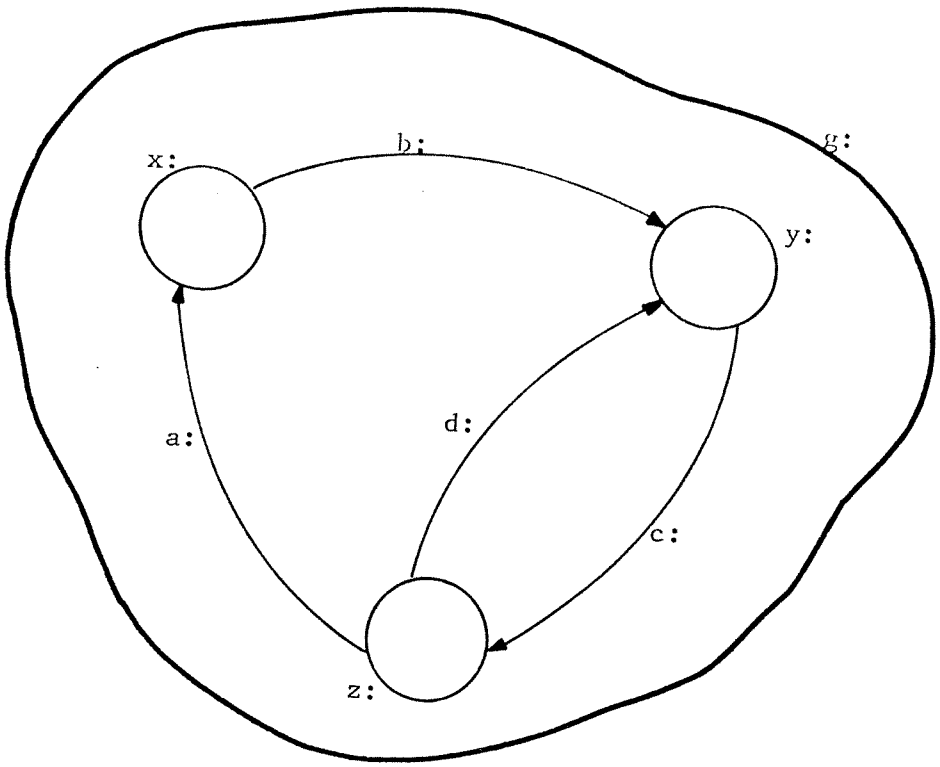


Figure 2.1. A Graph Skeleton

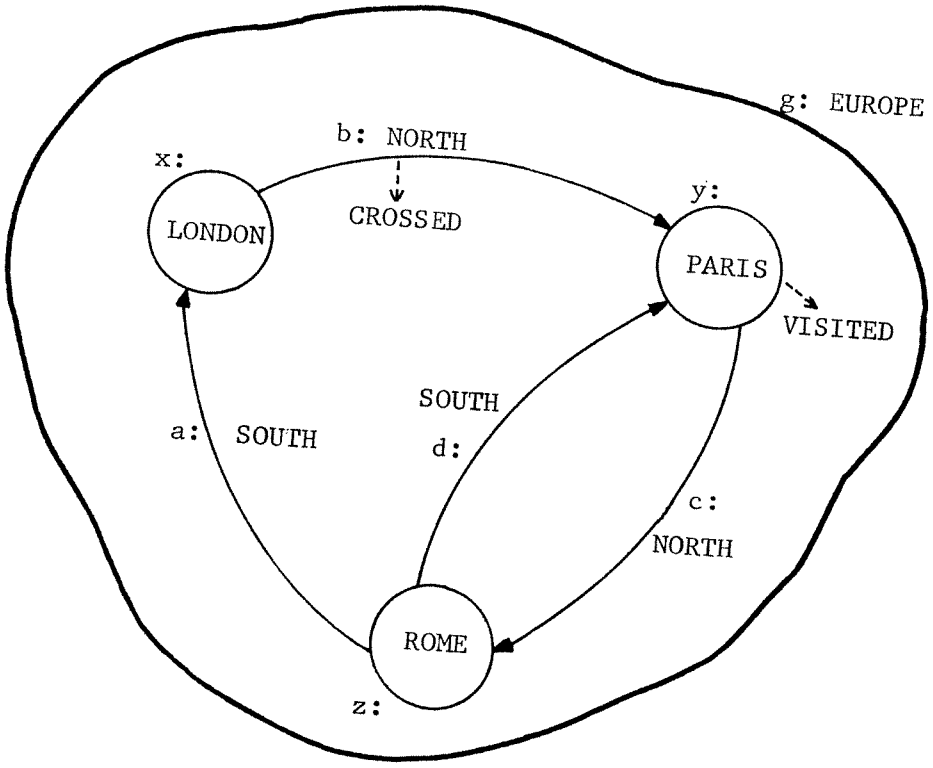


Figure 2.2. A Graph Data Structure

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.

System Sets

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 rseto (the set of all arcs emanating from a node), the rseti (the set of all arcs terminating at a node), the ndset (the set of all nodes on a graph), the nset (the set of all nodes with the same object), and the grset (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 frnode).

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 tonode).

These, in fact, are the criteria for membership in the rseto and rseti

	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

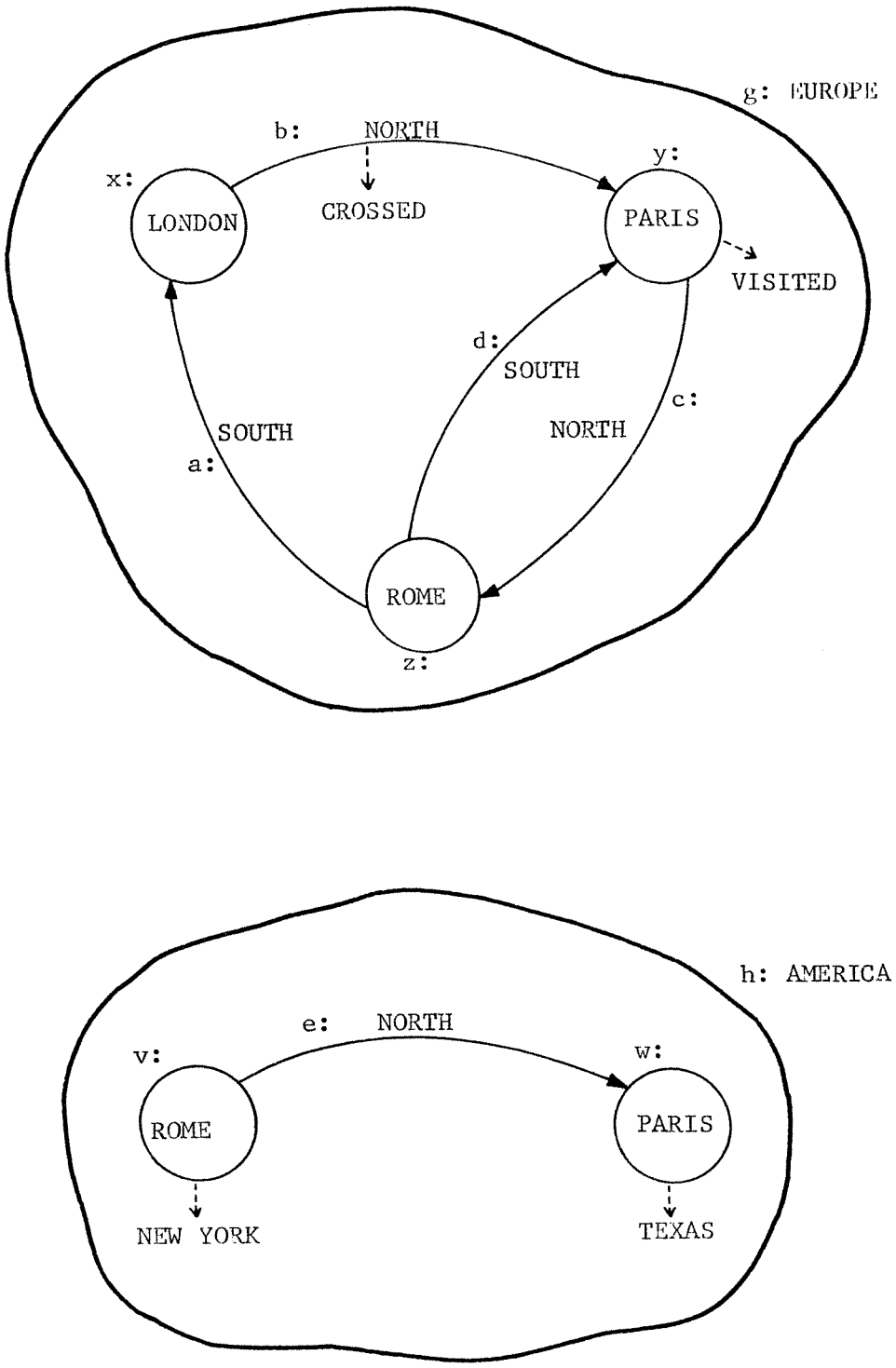


Figure 2.6. A Structure Which Emphasizes the nset and grset

		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

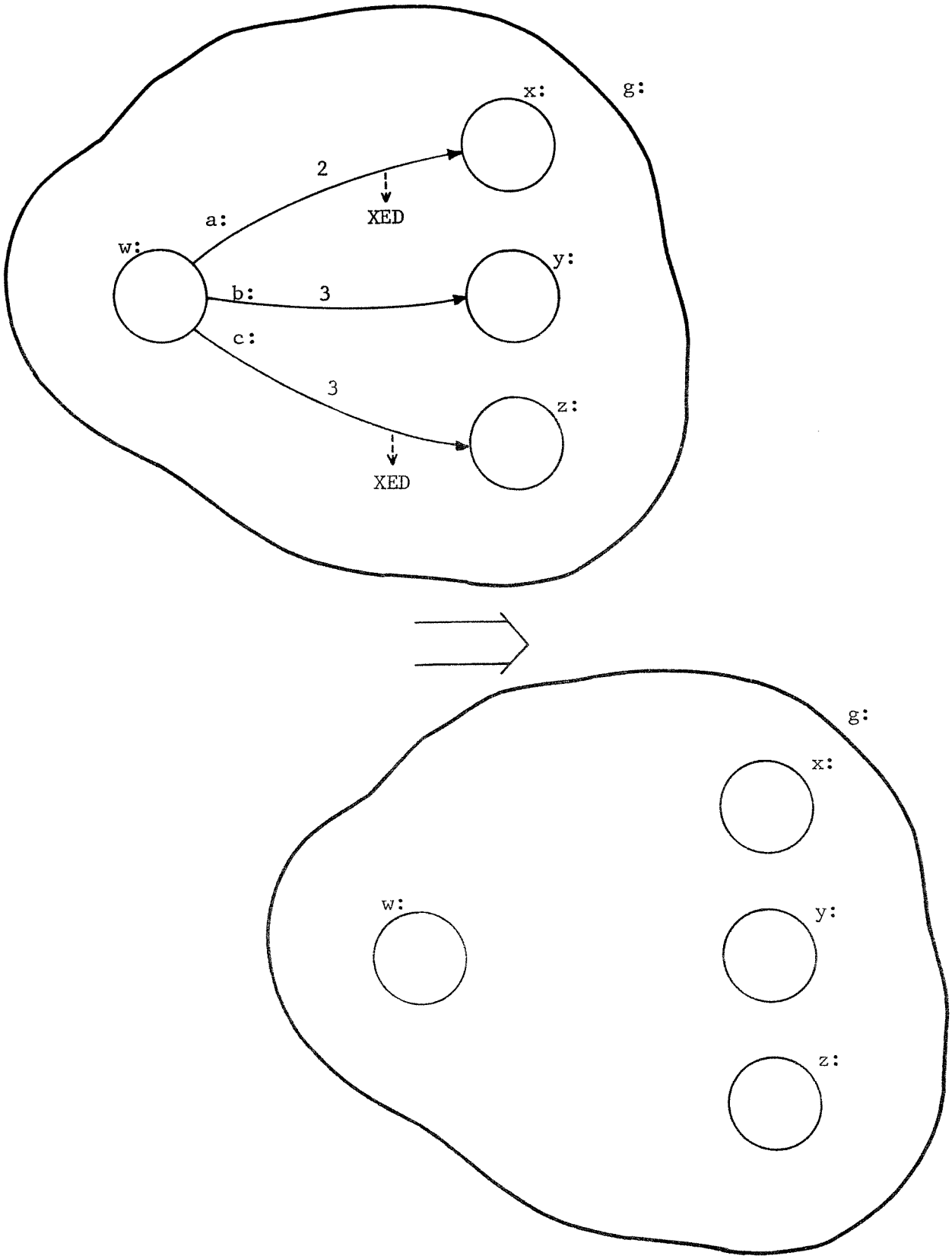


Figure 2.9. dmapft(rseto(w),true)

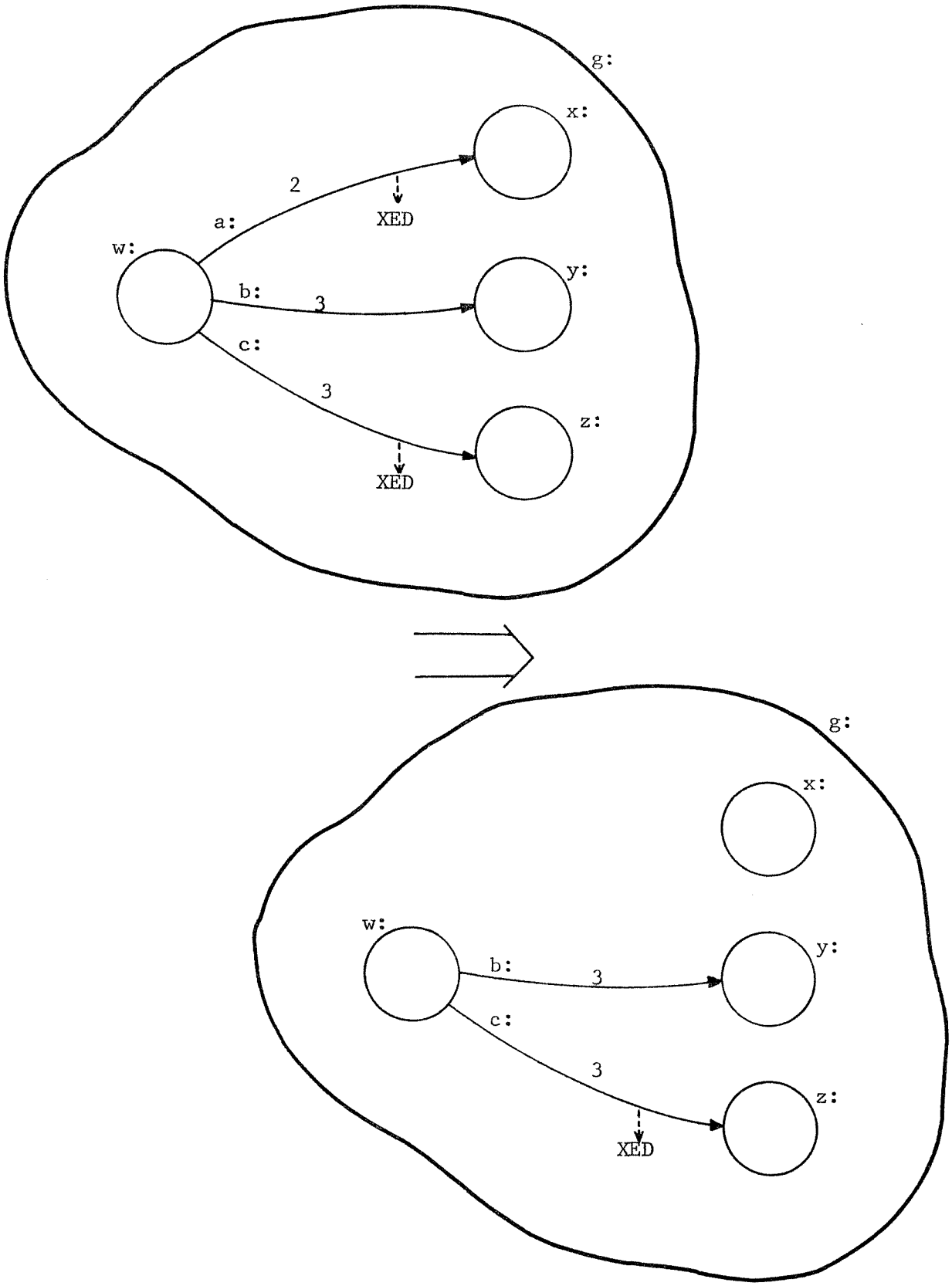


Figure 2.10. dorft(rseto(w),true)

f	v: false control	v: 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 n^{th} component of any system set. Let p be a system set; then Figure 2.13 represents an algorithm which terminates with t as the n^{th} component of p.

There are two ways to determine when a reader has read the m^{th} (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 end 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.