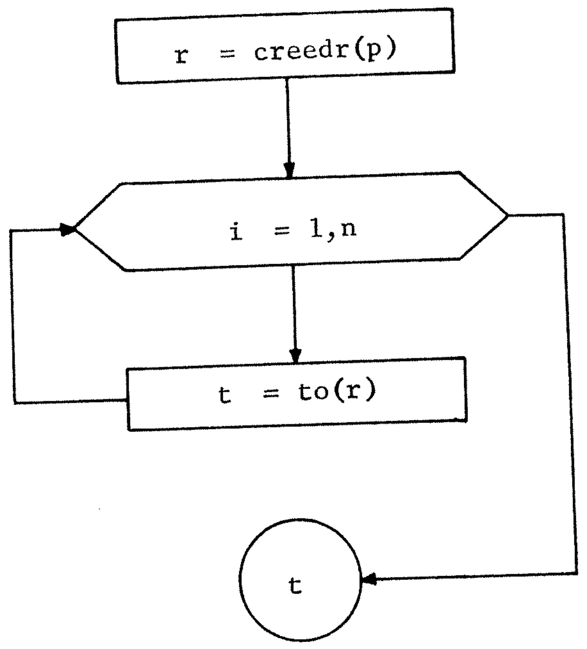


Figure 2.13. The nth Component of p
Component (p,n)

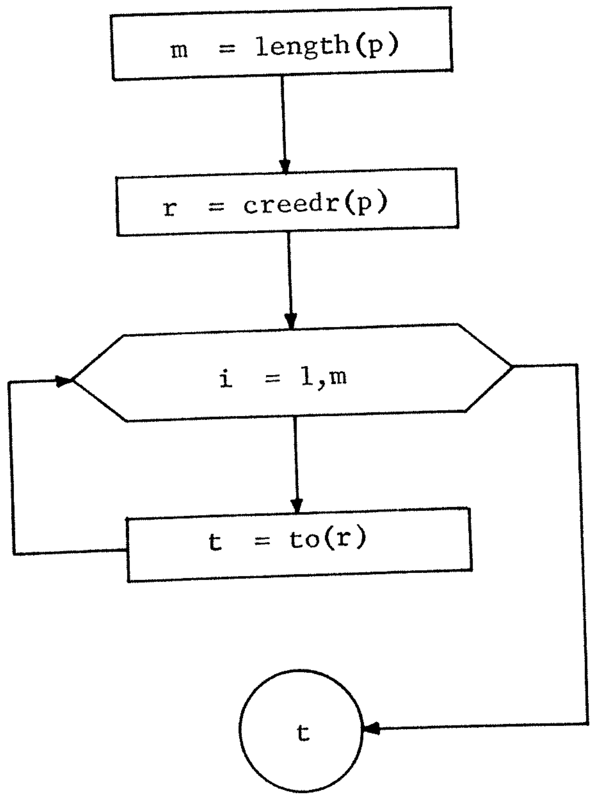


Figure 2.14. The Last Component of p

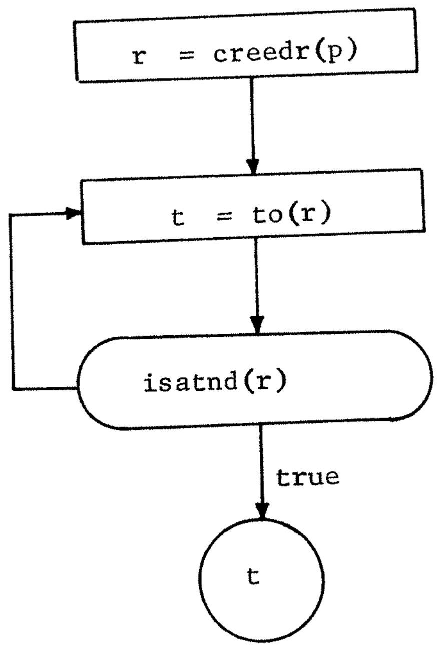


Figure 2.15. The Last Component of p Allowing for p to Be Altered

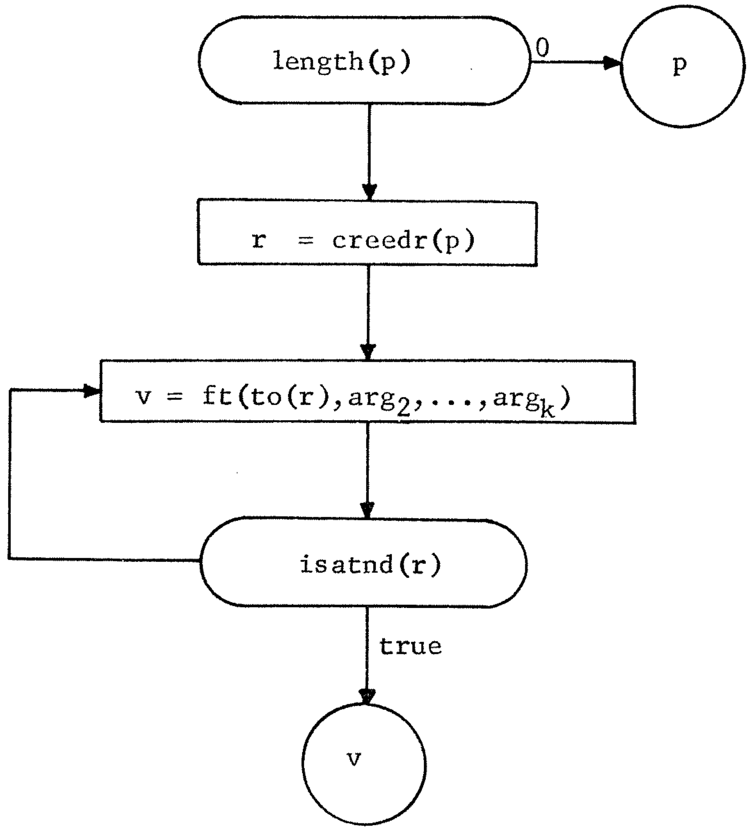


Figure 2.16. mapft(p,ft,arg$_2$,...,arg$_k$)

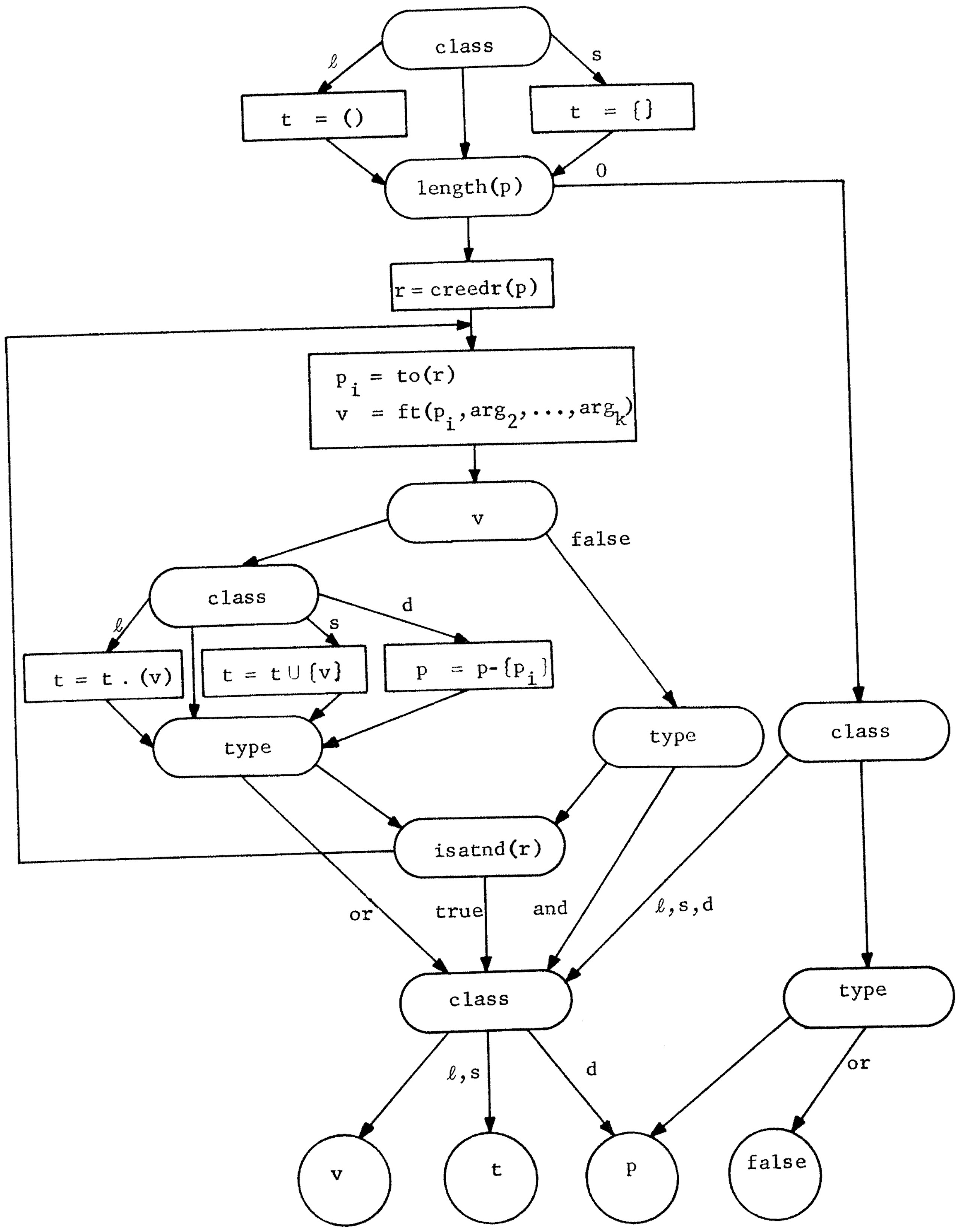
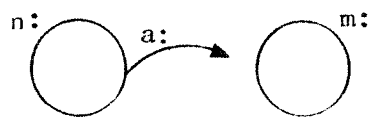


Figure 2.17. f(p,ft,arg_2,...,arg_k)

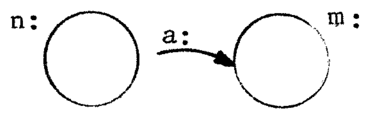
Generalizations

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  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  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 $\in$ rseto(n) and a $\in$ 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

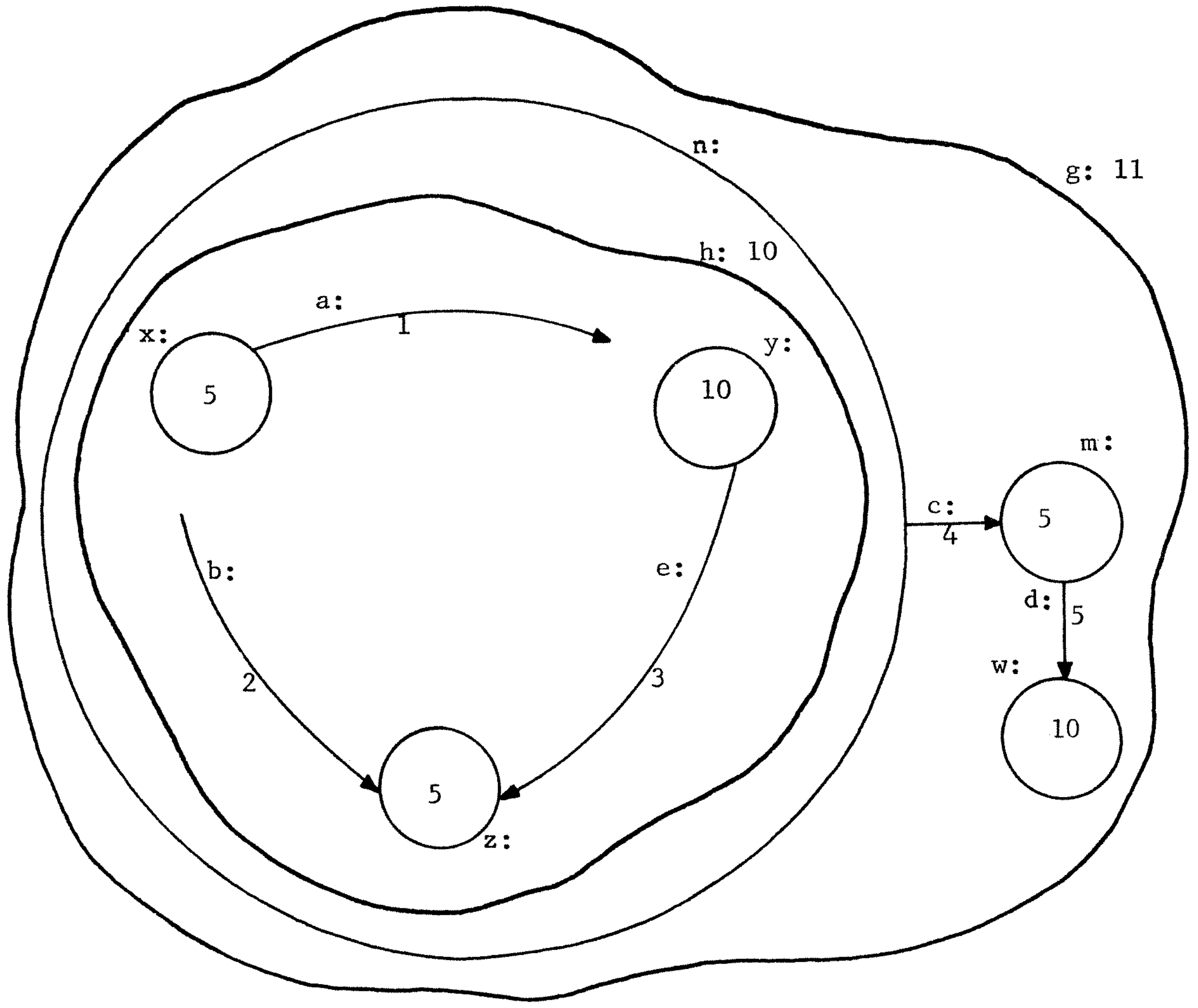


Figure 2.18. A Complex Graph-based Data Structure

		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

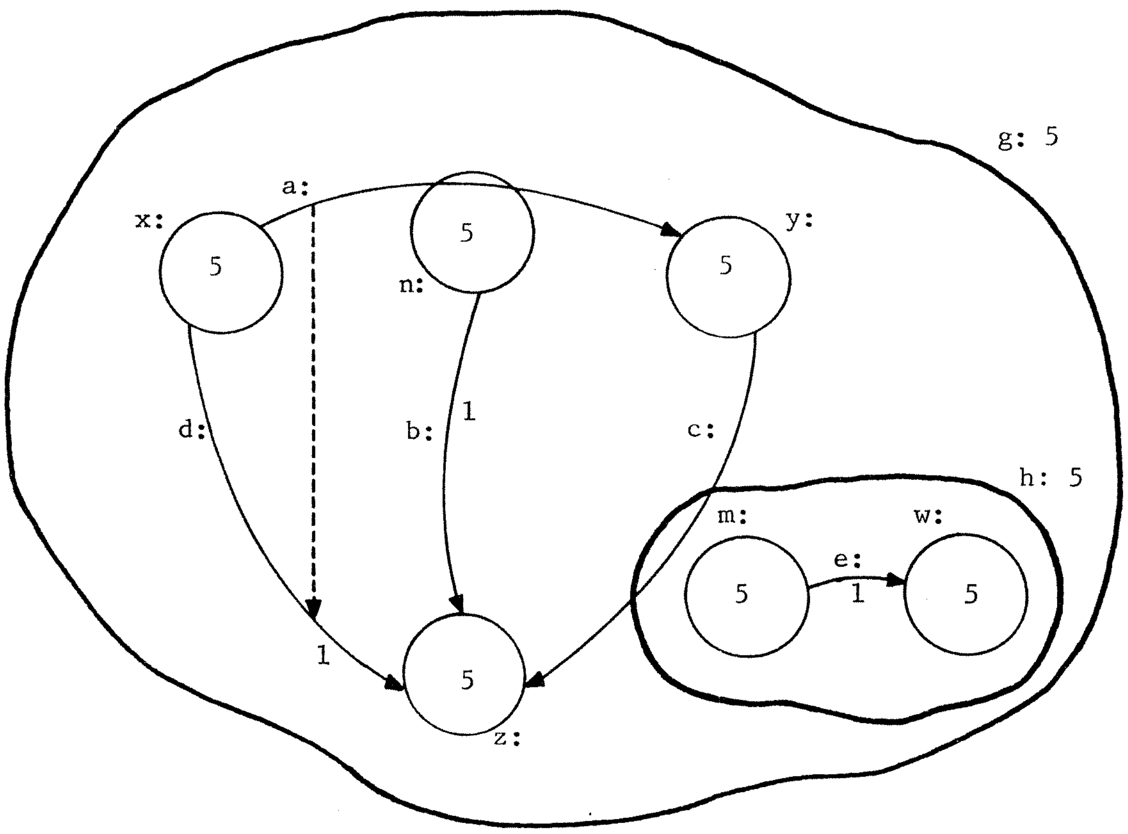


Figure 2.20. Another Complex Graph-based Data Structure

		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

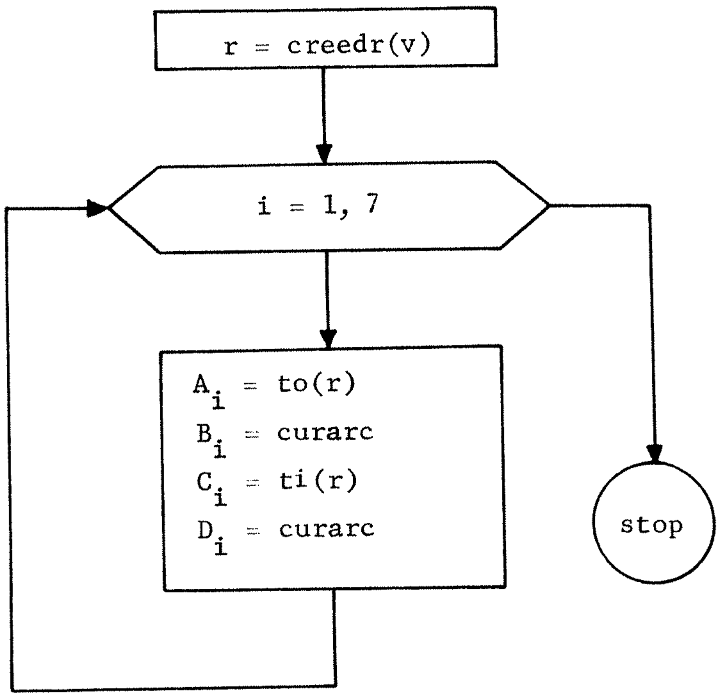
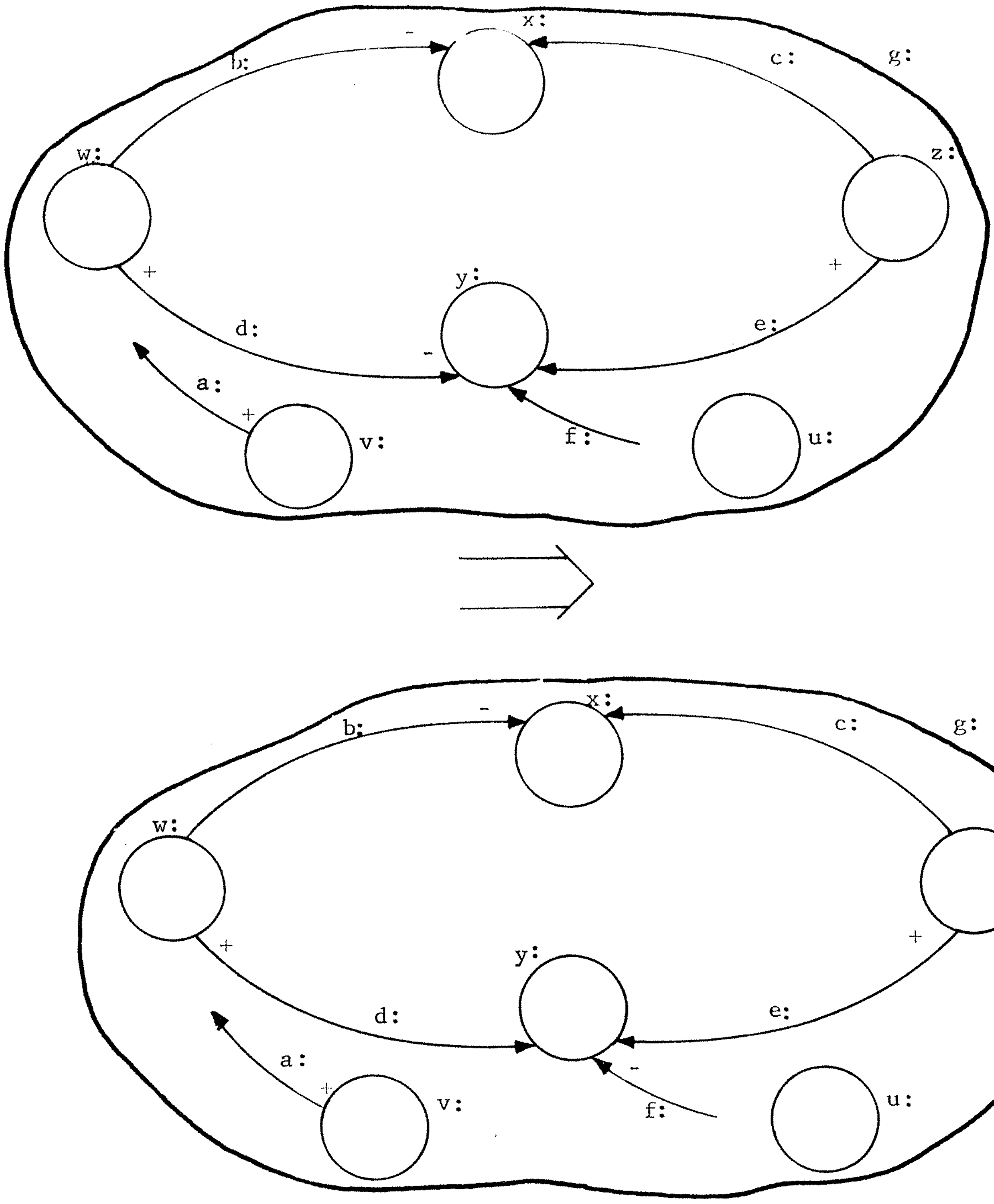


Figure 2.22. Traversing with the Graph Reader



	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

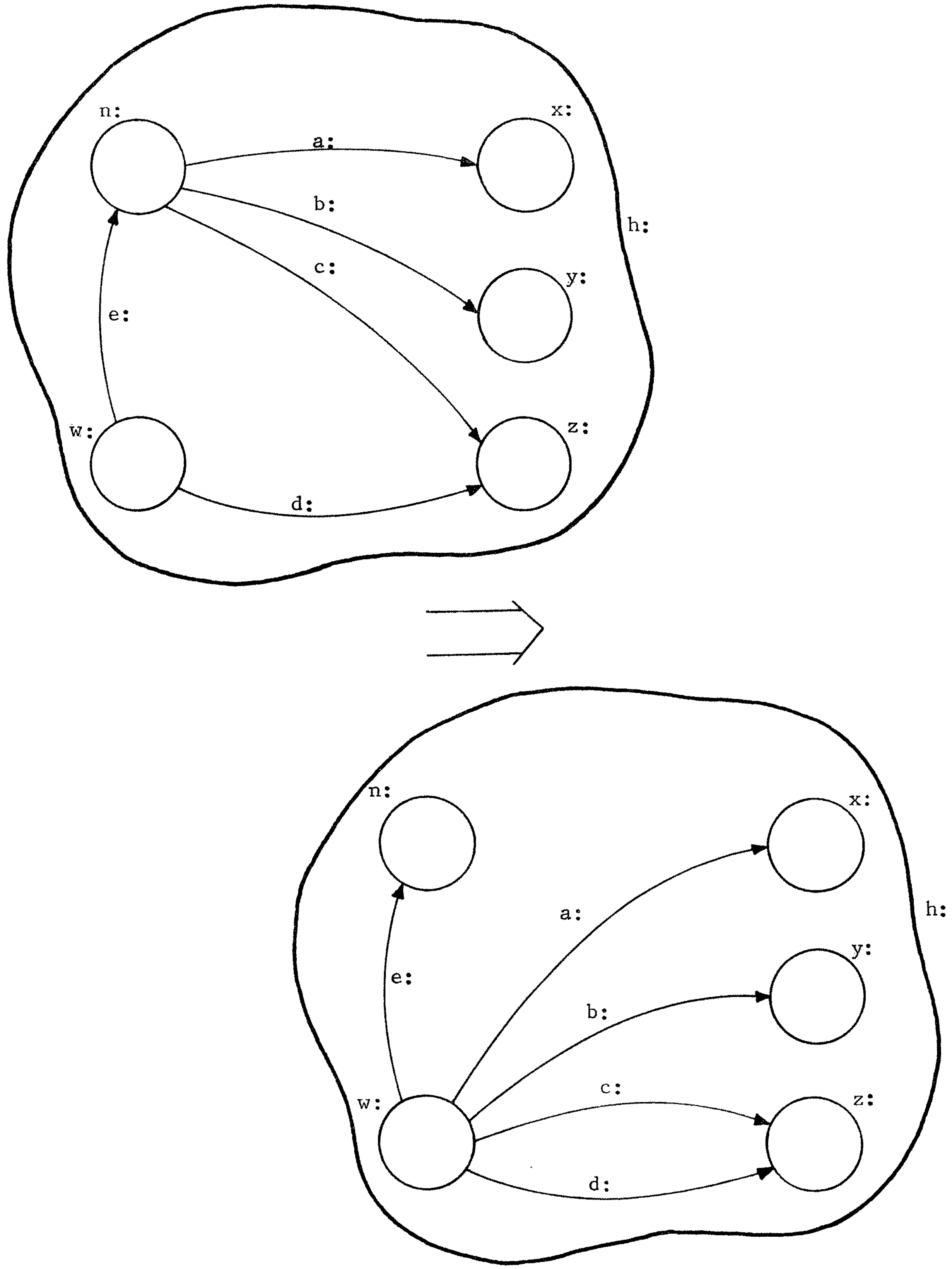


Figure 2.24. mapft(rseto(n),chafrn,w)

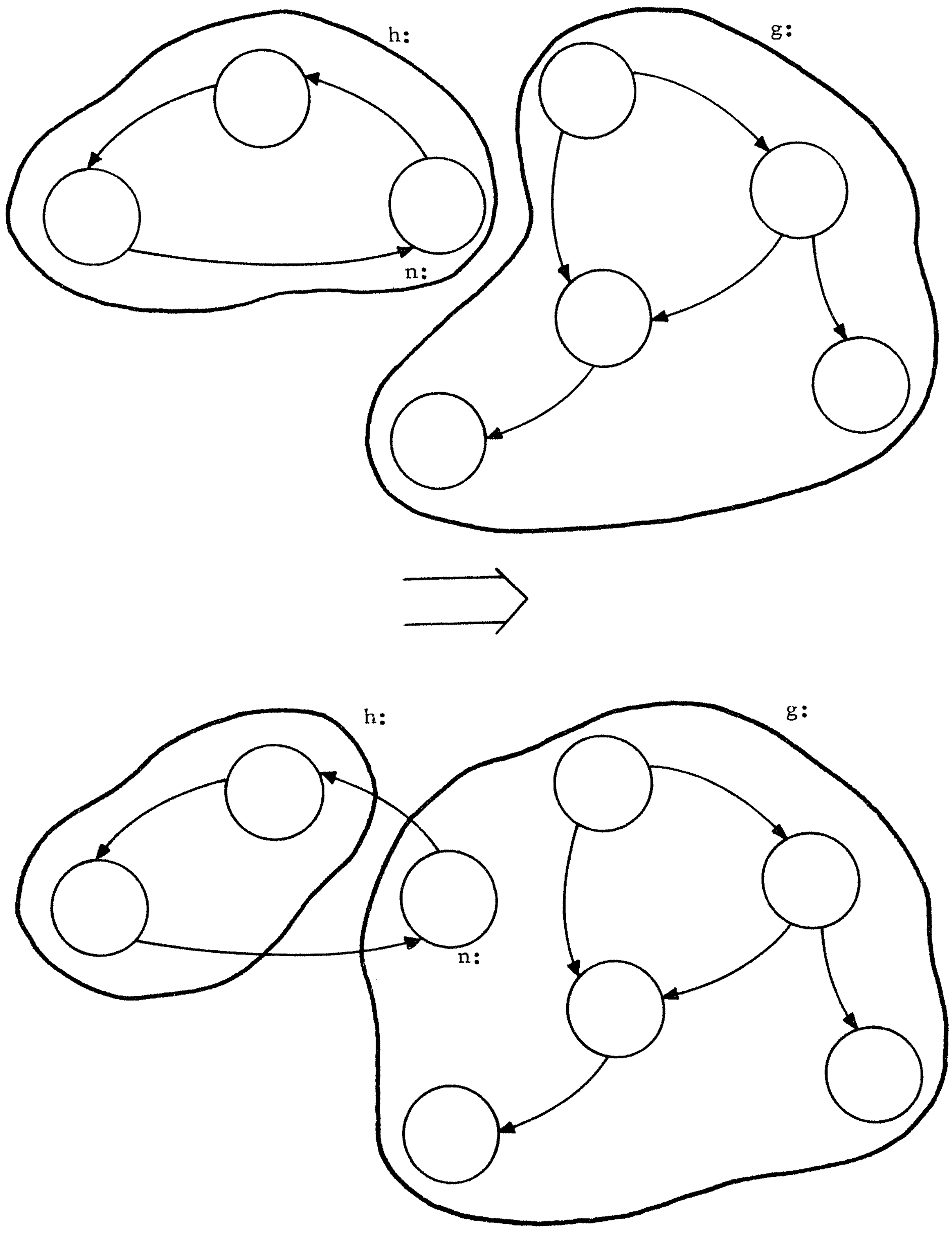


Figure 2.25. Changing the Graph of a Node

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.

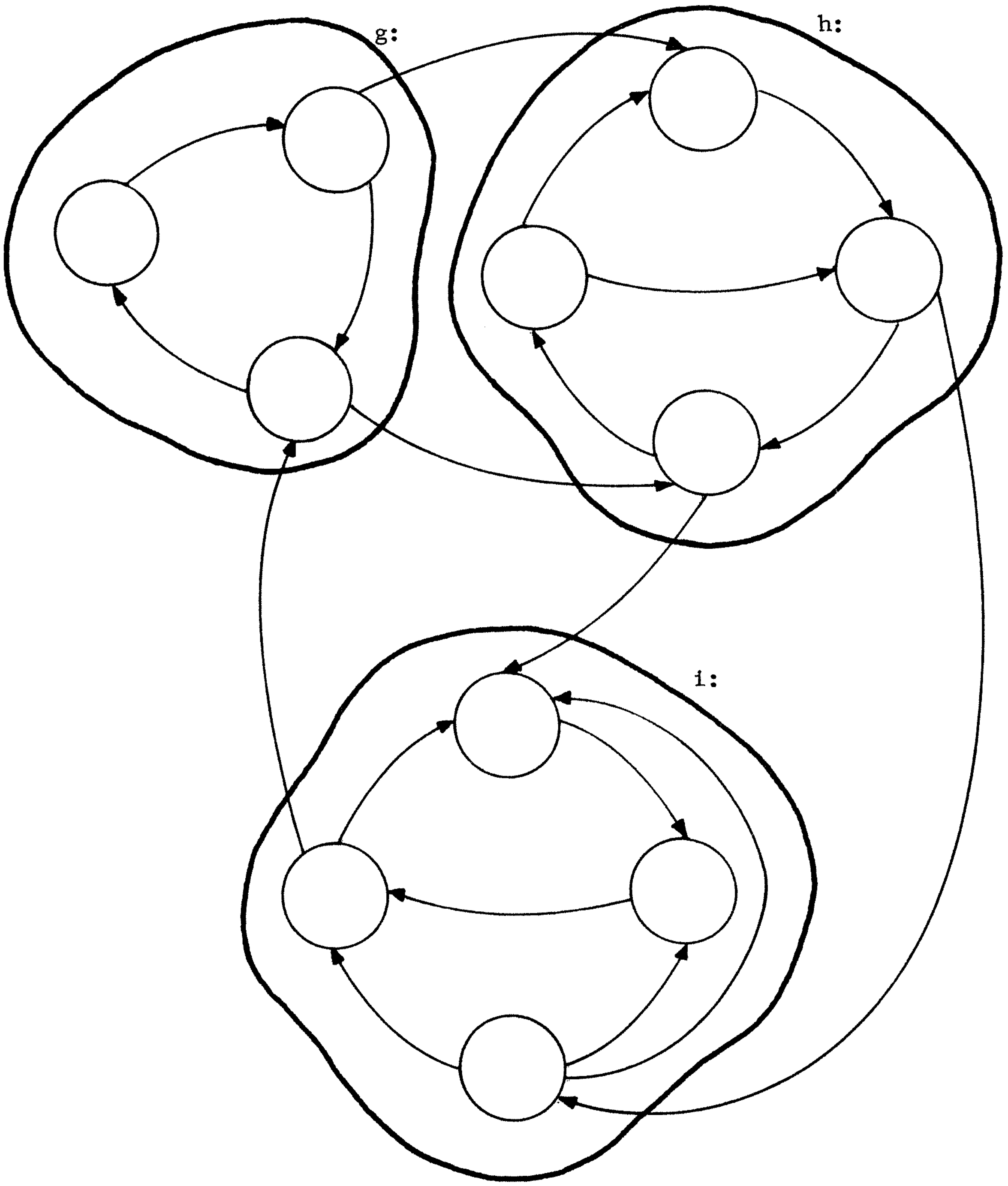


Figure 2.26. Representation of a Graph by Subgraphs

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 $\text{body}(x) = \text{body}(x')$. 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 <u>current-arc-in</u>.

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

$v \in V$, the <u>value</u>

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

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

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

$v \in V$, the <u>value</u>

$u \in N \cup A$, the <u>origin</u>

$b \in N \cup A$, the <u>object</u>.

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

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

<u>Operations in the Macro-semantics</u>

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.

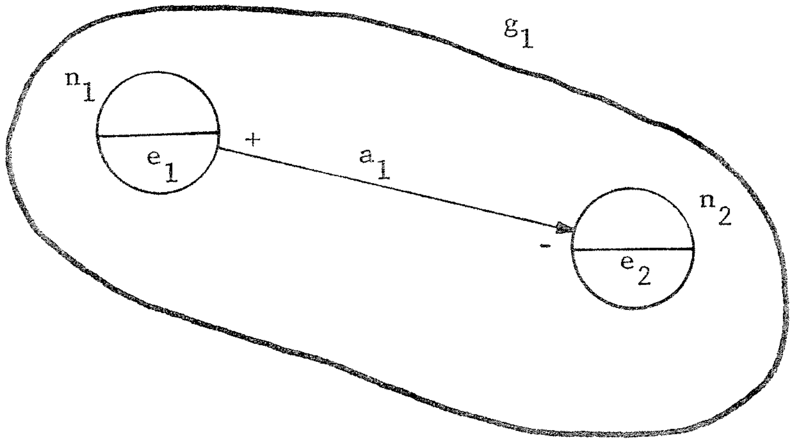


Figure 3.2. A Simple Graph (or gds)

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

$E = \{e_1, e_2\}$

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

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

$G = \{g_1\}$

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

$N = \{n_1, n_2\}$

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

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

$A = \{a_1\}$

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

Figure 3.3. H = 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. <u>is-cursor</u>(*) = $\begin{cases} * \text{ if } * \in C \\ \lambda \text{ otherwise} \end{cases}$.

2. <u>Retrieval operations</u>

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

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

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

Much that can be said about <u>object</u> is true also for <u>origin</u> with slight variations. The <u>origin</u> of a node must be a graph and the <u>origin</u> of an arc must be a node. The operation <u>relate</u> (in class 3) ties together those nodes and arcs with the same <u>origin</u>. A more complex situation exists for the <u>origin</u> and <u>object</u> of a cursor. When the <u>origin</u> of a cursor is a node, then the <u>object</u> of that cursor must be a node. Similarly, when the <u>origin</u> is an arc, then the <u>object</u> 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 $v_{\underline{body}(*)} \in V$.

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

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

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

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

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

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

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

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

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

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

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

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

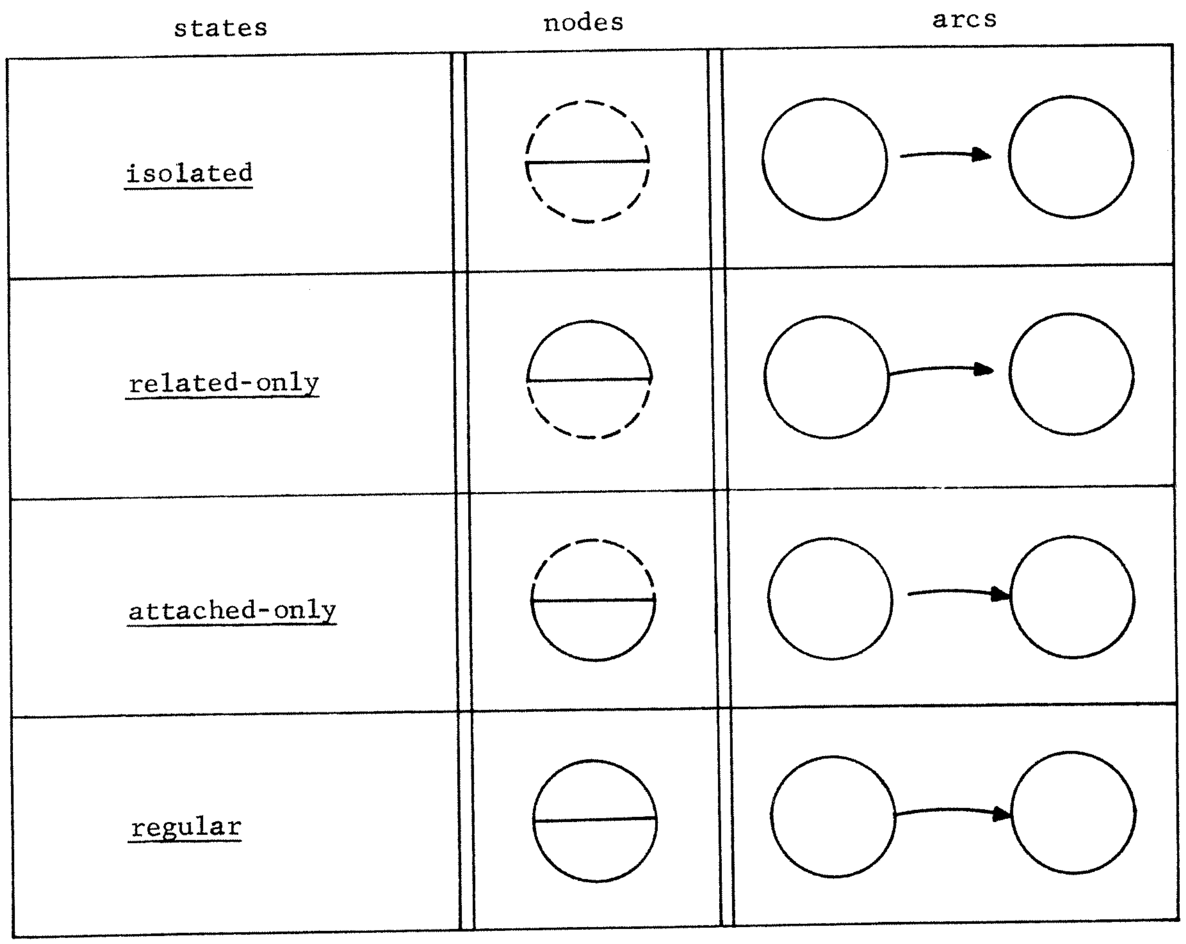


Figure 3.4. The States of Nodes and Arcs

(*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

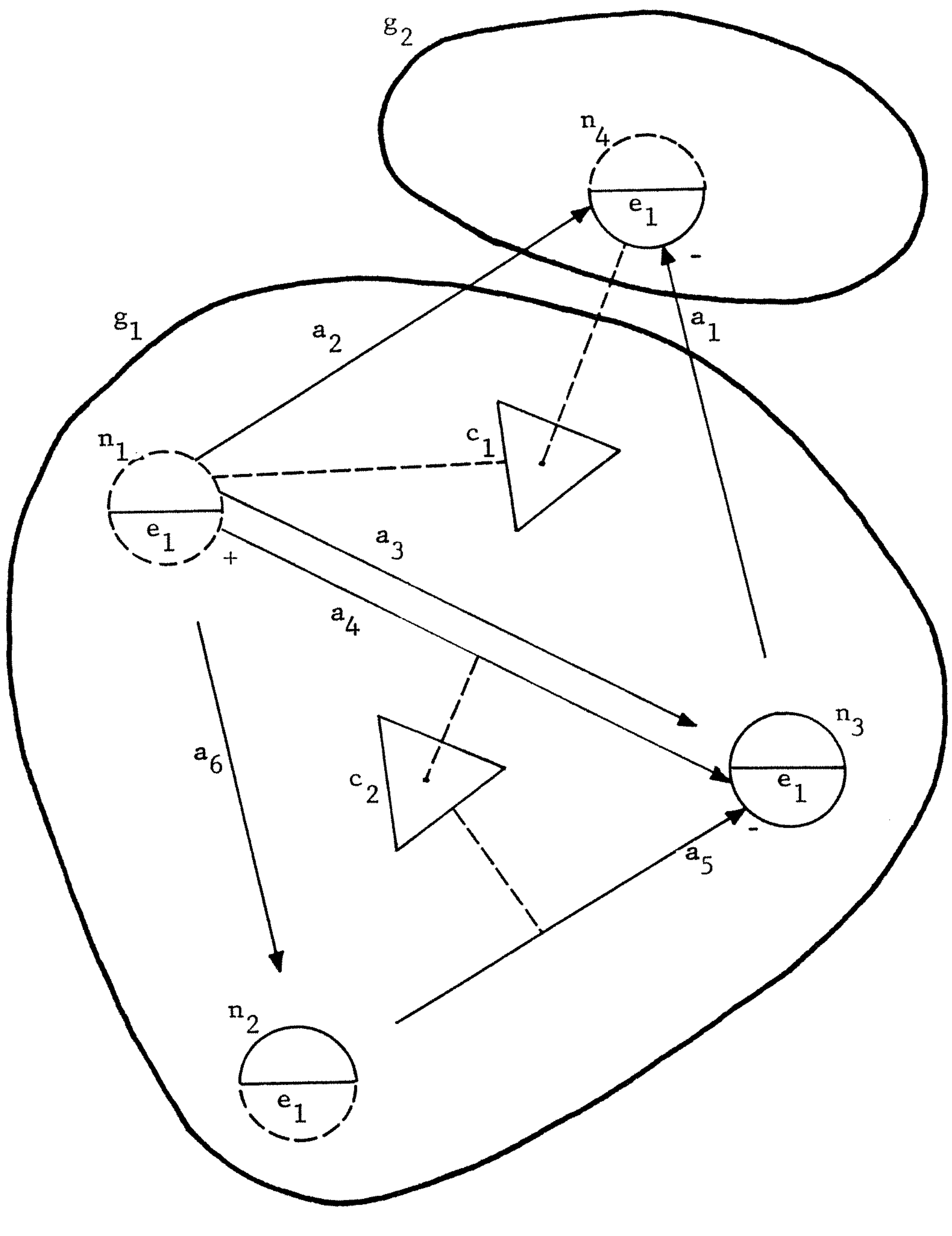


Figure 3.5. Graph Structure for Semantic Examples

$H = (L_0 \cup V, \underline{body})$ 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. $\underset{\sim}{H}$ = $\underline{gds}$ of Graph Structure for Semantic Examples

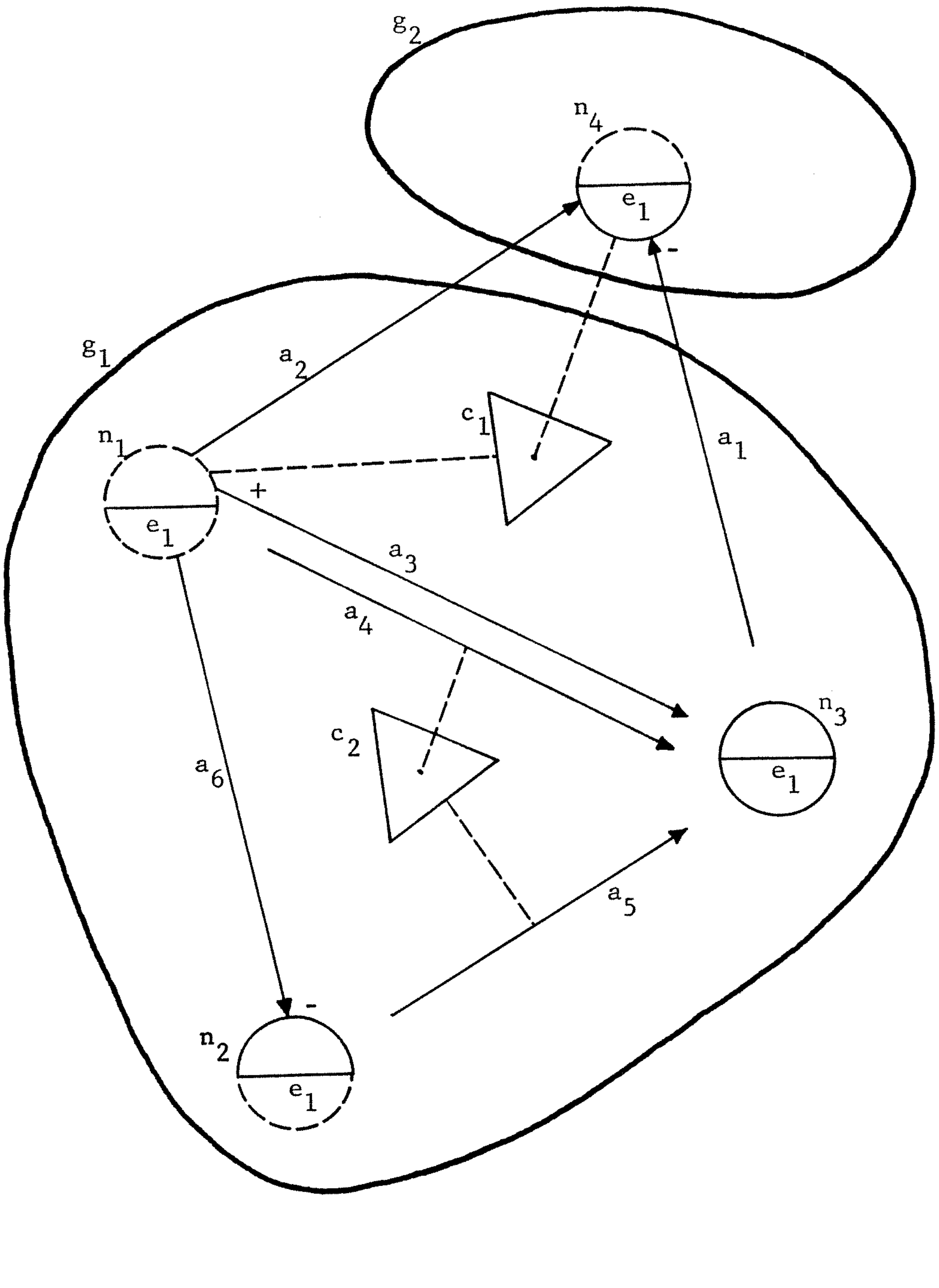


Figure 3.7. gds After State Changing Operations

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. <u>change-current-arc-out</u>$(*) = *$ where $* \in A_u^{+}$. Define

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

c. <u>change-last-of-related-set</u>$(*) = *$ where $* = x_i \in X_u^{+}$. Define

<u>body</u>$(u) = (\dot{v},(\dot{x}_{i \oplus 1},\ldots,\dot{x}_k,\dot{x}_1,\ldots,\mathring{x}_i))$ 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. <u>change-current-arc-in</u>$(*) = *$ where $* \in A_b^{-}$. Define

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

e. <u>change-last-of-attached-set</u>$(*) = *$ where $* = y_i \in Y_b^{-}$. Define

<u>body</u>$(b) = (\dot{v},(\dot{y}_{i \oplus 1},\ldots,\dot{y}_k,\dot{y}_1,\ldots,\dot{y}_i))$ 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. <u>change-origin</u>$(*,**) = *$ where $* \in N \cup A \cup C$ and $** \in G \cup N \cup A$.

Define

<u>body</u>$(*) = (\dot{v},**,\dot{b},\dot{A}^{+},\dot{a}^{+},\mathring{A}^{-},\dot{a}^{-})$ 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 <u>same</u> set or

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

g. <u>change-object</u>$(*,**) = *$ 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) = (λ,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

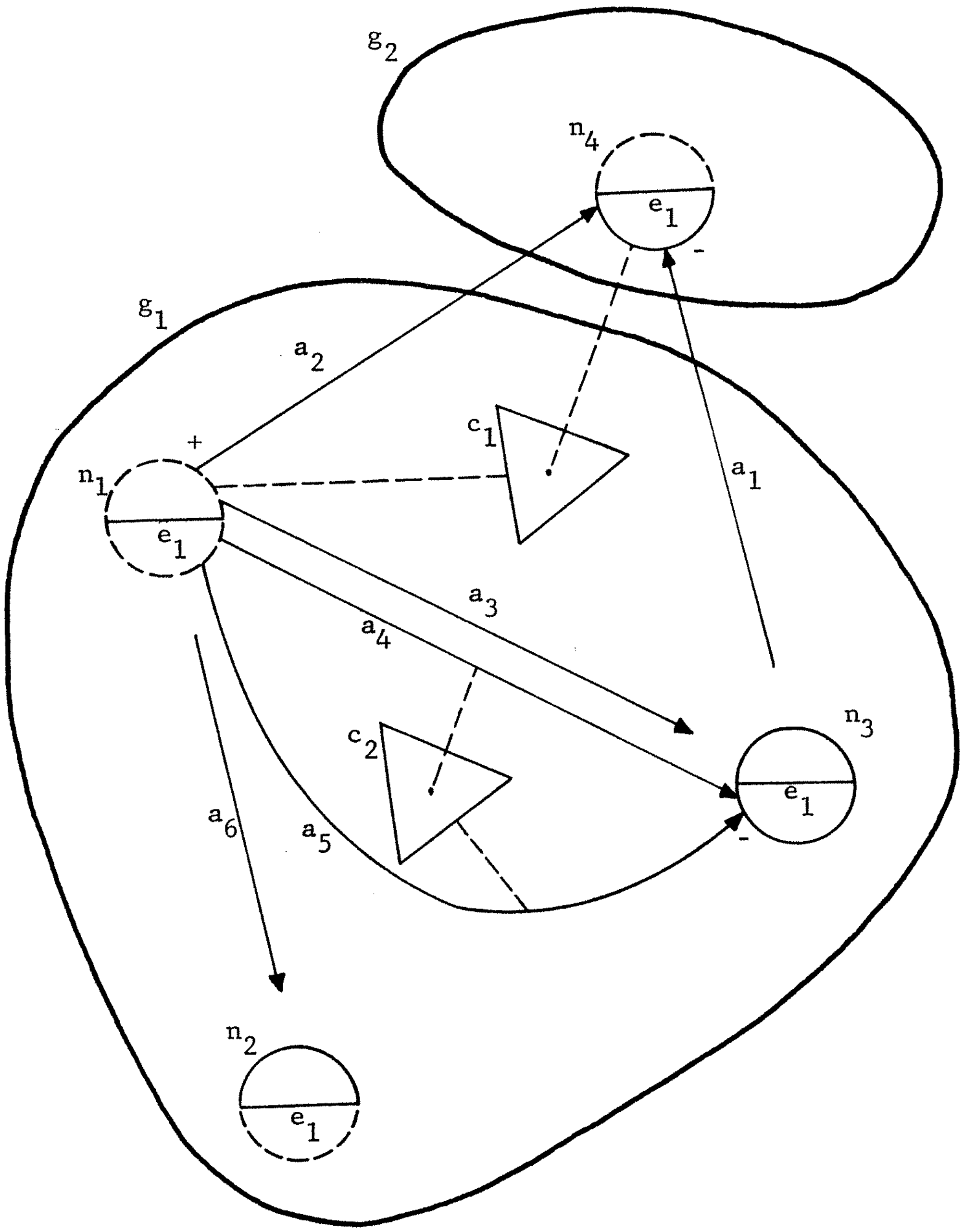


Figure 3.8. <u>gds</u> After State and Structural Changing Operations

Figure 3.9 with a cursor s where <u>object</u>(s) = <u>origin</u>(s) = a. Five executions of <u>traverse-related-successor</u>(s) makes <u>object</u>(s) = f by stopping at b', c', d', and e'. One more execution of <u>traverse-related-successor</u>(s) makes the <u>object</u>(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 = <u>origin</u>(a). Now consider three executions of the operation sequence (<u>traverse-attached-successor</u>(s), <u>traverse-related successor</u>(s)). Here, once again, <u>object</u>(s) becomes a, but this time by stopping at b, c, d, e, and f.

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

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

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

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

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

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

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

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

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

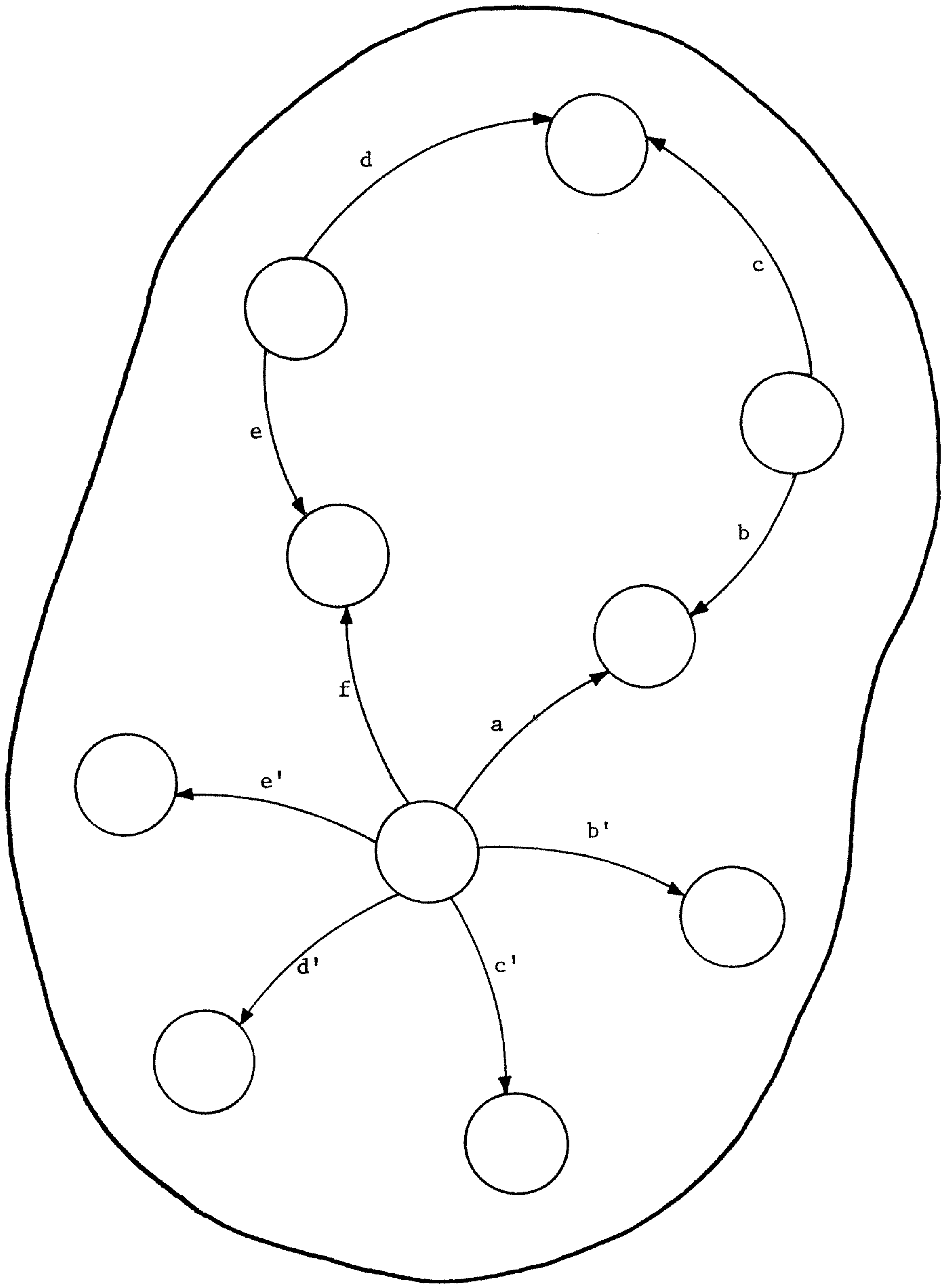


Figure 3.9. Graph for Illustrating "Subtle" Simple Traversal

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.

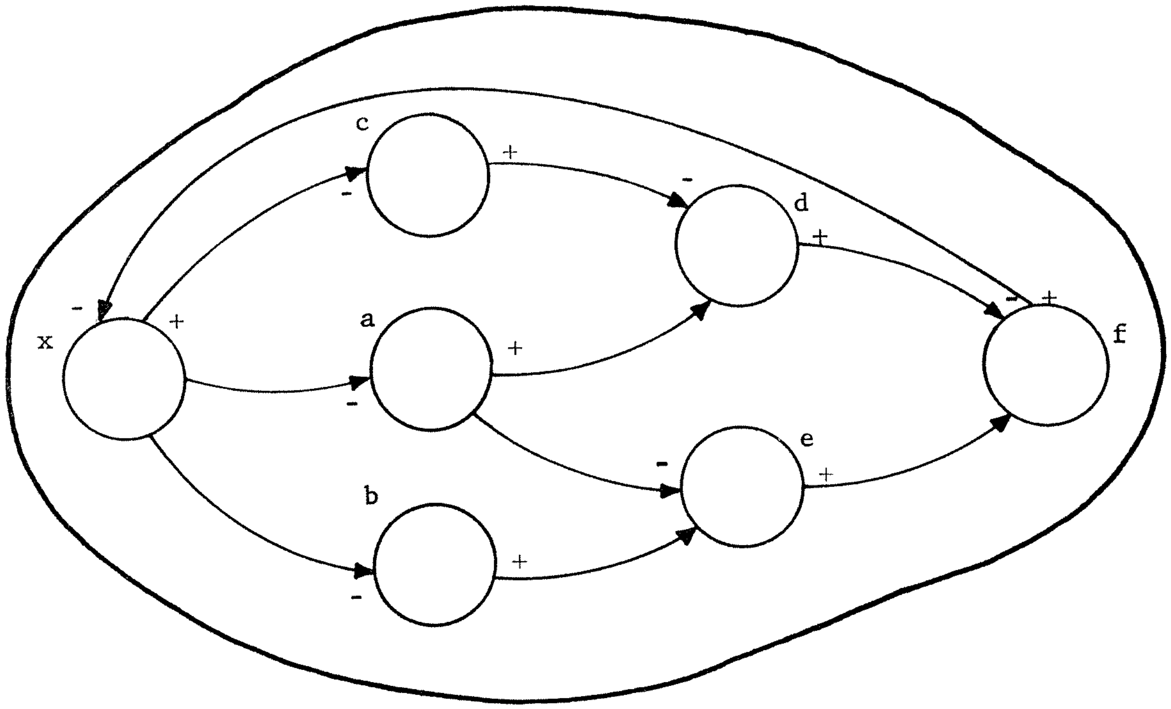
As arcs are crossed they become either the _current-arc-out_ (if crossed in an outward direction) or _current-arc-in_ (if crossed in an inward direction). For notational convenience we place a + (recall Figure 3.2) on the tail of a _current-arc-out_ and a - on the head of a _current-arc-in_.

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.

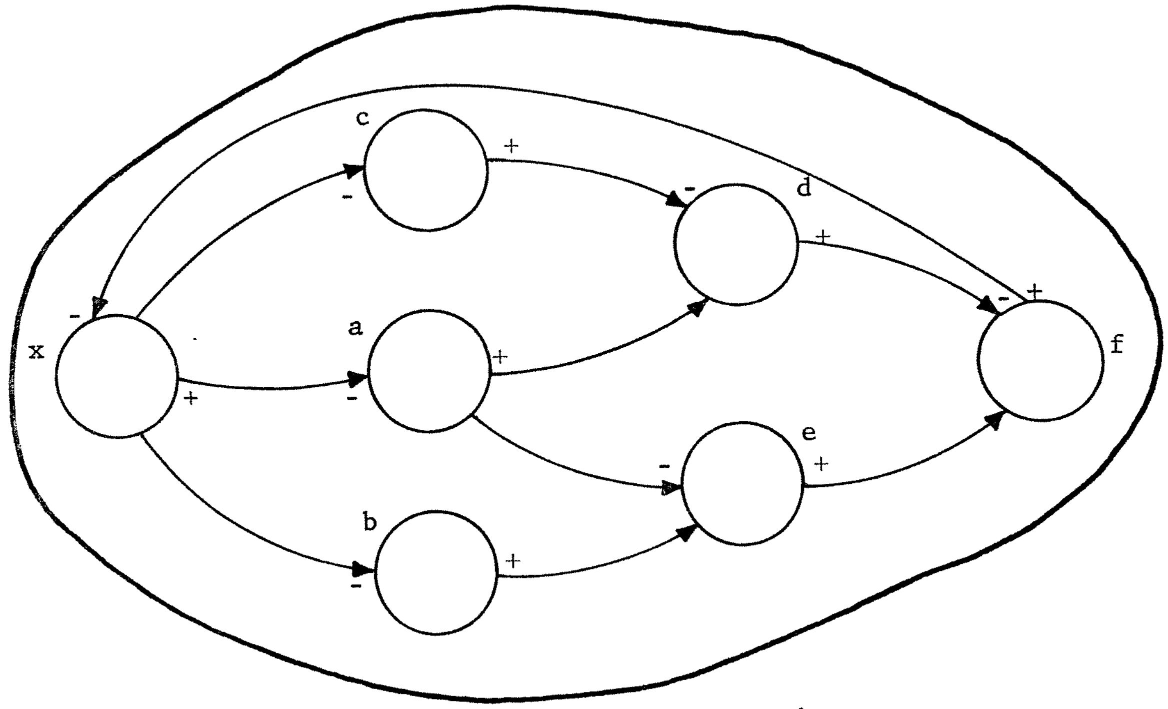
Traversing operations (complex)

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 * ∈ C and b ∈ 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 traverse-node-out operations with the cursor residing on node, x



After 16 traverse-node-out operations

Figure 3.10. The Complex Reader Mechanism

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

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

and define $\underline{\text{body}}(*) = (\dot{v}, \dot{u}, b_{x_{i \oplus j}})$.

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

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

and define $\underline{\text{body}}(*) = (\dot{v}, \dot{u}, u_{y_{i \oplus 1}})$

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

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

and define $\underline{\text{body}}(*) = (v, u, u_{y_{i \oplus j}})$.

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,

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

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.

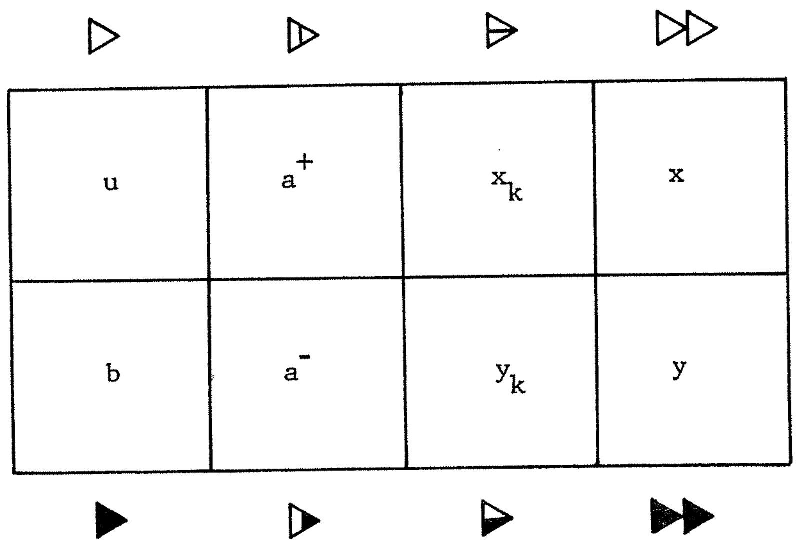


Figure 4.1. Arc Label Conventions

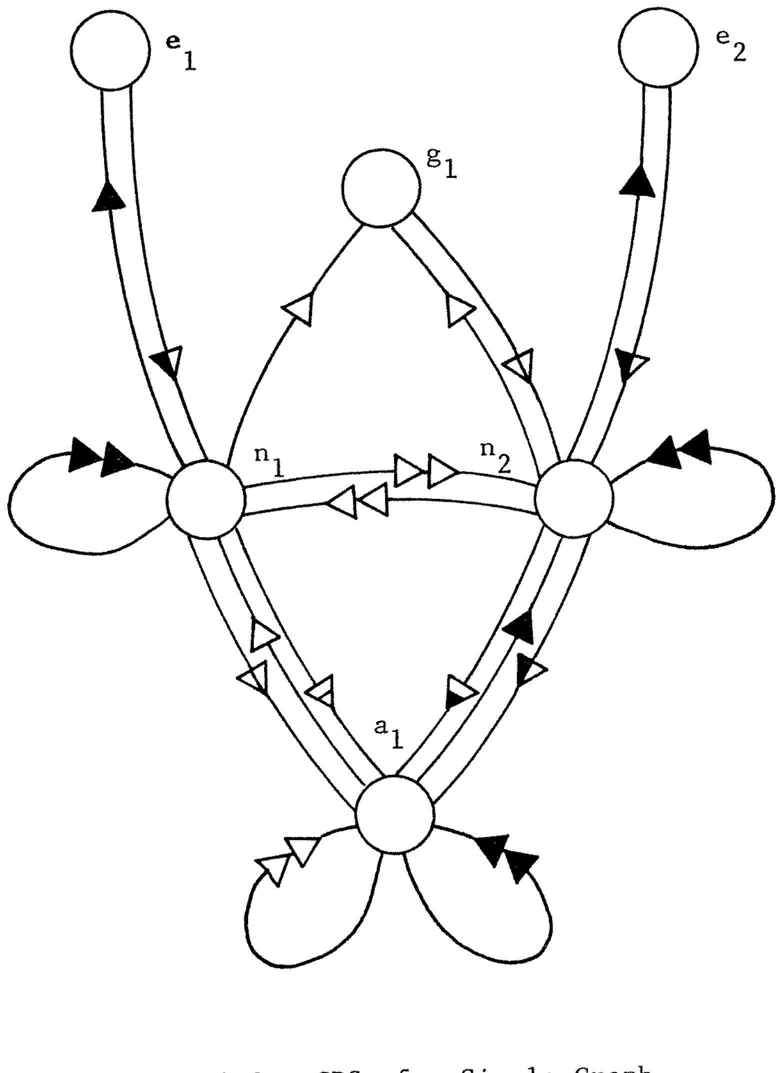


Figure 4.2. GDS of a Simple Graph

(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.

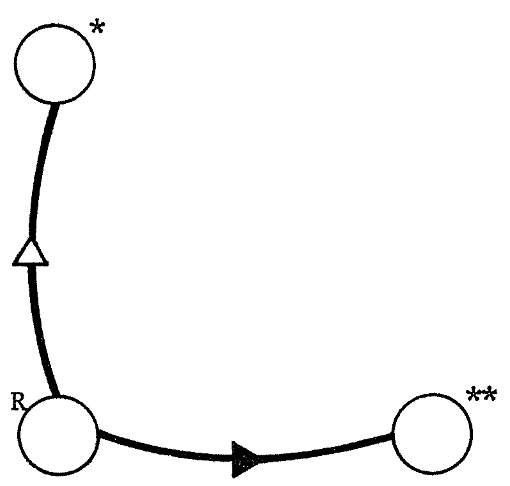


Figure 4.3. Diagram for CREATE-NODE(*,**)

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 ϵ $\bar{L}$. There is a free node R ϵ $\bar{L}_0$, 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 ϵ $\bar{L}$. There is a free node R ϵ $\bar{L}_0$, 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(*) ϵ 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>.

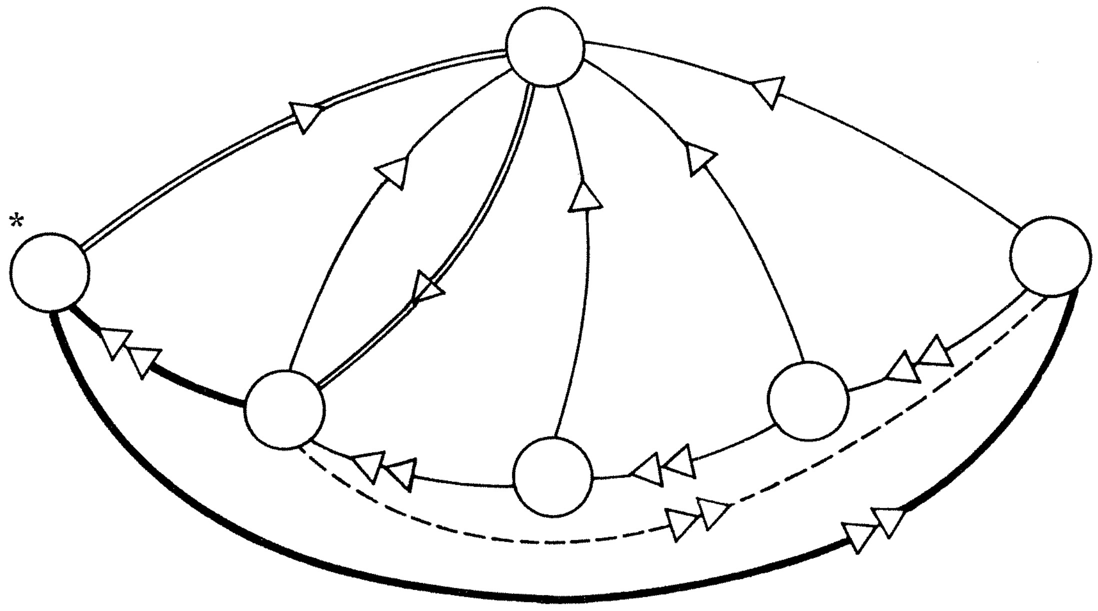
k. <u>ATTACHED-PREDECESSOR</u>(*) = $\begin{cases} y^{-1}(*) \text{ if } y^{-1}(*) \text{ exists} \\ \lambda \text{ otherwise} \end{cases}$,

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

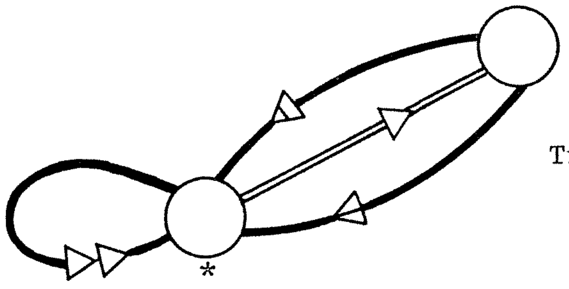
3. <u>State changing operations</u>

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 <u>TYPE</u>(*) is <u>arc</u>

Figure 4.4. Diagrams for <u>RELATE</u>(*)