PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 1/75

USING GRAPH GRAMMARS FOR THE DEFINITION OF SETS OF DIGRAPHS

by

J. Mylopoulos
and
A.L. Furtado

Computer Science Department - Rio Datacenter

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

USING GRAPH GRAMMARS FOR THE DEFINITION OF SETS OF DIGRAPHS

J. Mylopoulos
Assistant Professor
Department of Computer Science
University of Toronto

A. L. Furtado
Associate Professor
Computer Science Department
(Informática) PUC/RJ

Series Editor: Prof. S. M. dos Santos

Jannuary/ 1975

ABSTRACT

A graph grammar formalism is described and used to define several interesting sets of digraphs such as strongly connected graphs, rooted acyclic graphs and lattices. It is argued that the formalism is descriptionally powerful and inexpensive and that it constitutes a constructive method for the precise definition of many structures arising in computer science. A discussion comparing constructive methods for the definition of sets of structures to axiomatic ones is also included.

1. Introduction

Graph grammars have been known since the late sixties and there have been many theoretical studies of their properties. Unfortunately, less effort has been dedicated to the investigation of their applicability and usefulness.

Some of the existing studies concentrate on the introduction of new graph grammar formalisms and an investigation of their properties and their relationship to other formalisms. [1, 2, 3, 4, 5, 6] . A few papers have dealt with the recognition problem for graph grammars, i.e., the problem of deciding for a graph g and a graph grammar G whether g can be generated by G [2, 3, 5, 12] . There are only two papers we know of which study extensively the applicability of graph grammars by defining, in terms of graph grammars, interesting sets of undirected graphs [7, 8] . For a more thorough survey of research on graph grammars the reader is directed to [10] .

This paper explores the application of a formalism developed exclusively for the definition of classes of directed graphs, rather than for the study of properties of graph grammars. We hope to show that by imposing certain reasonable restrictions on the kinds of graphs that can be derived, we can arrive at a formalism which is algorithmically inexpensive and at the same time definitionally powerful and flexible..

Section 2 describes the restrictions we wish to impose on derived graphs and the graph grammar formalism to be used. It also discusses some of the basic properties of the formalism.

In section 3 we consider several well-defined classes of directed graphs from graph theory (e.g., Eulerian and Hamiltonian graphs, acyclic graphs, Hasse diagrams, finite lattices etc.) and present graph grammars for them. It is our intention to demonstrate in this section how one can use the formalism proposed in section 2, but also to contrast the well-known axiomatic definitions for these classes to the more constructive graph grammar ones, and to give the reader at least an idea of the kind of correctness proof he would have to provide to prove his graph grammar correct. The classes of graphs were not chosen randomly. They share many algebraic and topological properties which we have used to "build" graph grammars for the more difficult classes to define constructively (e.g., lattices), in terms of those proposed for simpler classes (e.g., acyclic graphs). This technique also helps in the formulation of correctness proofs. Section 4 discusses potential areas of application of graph grammars, what methods have been used in the past to define classes of graphs and how do these methods compare to that proposed in this paper.

2. Definitions and Notation

It is assumed that the reader is familiar with the basic definitions of graph theory [13] and formal languages [16].

The graphs we will deal with here are rooted, directed with uniquely labelled edges leaving each node. More precisely a <u>rooted labelled digraph</u> (or RLD or simply graph) is a 6-tuple (N, NL, EL, ν, δ, r) , where

N - a non-empty finite set of nodes, denoted by positive integers

NL - is a non-empty finite set of node labels

EL - is a finite set of edge labels

ν - is a total function which assigns a node label to every node of the RLD

$$\nu : N \to NL$$

δ - is a total function which associates an edge label to every edge

$$\delta : N \times EL \to N \cup \{u\}$$

where u stands for "undefined".

r - is a node, $r \in N$, such that for all $n \in N$ there exists a sequence of nodes $(n_1, n_2, \ldots, n_k)$ with $n_1 = r$ and $n_k = n$ such that for $1 \leq i < k$ there exist edge labels $e \in EL$ such that $\delta(n_i, e) = n_{i+1}$.

The node r is called the <u>root</u> of the RLD and it is clear that any other node of the RLD can be reached through a path that begins at its root. Note that since δ is a function, no two edges with the same label can leave the same node. It is quite possible, however, to have an edge label e and a node n such that there is no edge labelled e leaving n .

It follows immediately from the definition of an RLD that

a. A node reference can be defined uniquely in terms of another node reference n and a sequence s of edge labels which defines (uniquely) a path starting at n. We shall call such edge label sequences σ-sequences. Note that sometimes there will be no path which corresponds to a particular σ-sequence for an RLD and one of its nodes.

b. RLDs admit canonical forms that can be obtained through a systematic traversal of an RLD. One possible way to do this would be to start at the root and perform a depth-first search for which the edges from each node are selected according to some lexicographic order of their labels.

Let p be the number of nodes of an RLD, q the number of edges and v the number of edge labels. The complexity of the traversal algorithm described above is completely dominated by the complexity of sorting the edge labels, which is roughly $O(p \cdot v^2)$, since the complexity of the depth-first search is $O(\max(p, q))$ and $q \leq p \cdot v$ in an RLD.

We will use the following conventions in representing RLDs schematically: capital letters will be reserved for node labels, small letters from the beginning of the alphabet, possibly with trailing digits, will be reserved for edge labels. The (integer) node reference for each node will always be located next to the node label. FIG. 1(a) shows an RLD. The root of an RLD will only be mentioned when needed.

An RLD grammar (or graph grammar or simply grammar) is a 5-tuple (V_N, V_T, V_{EL}, P, S) , where

V_N - is a finite non-empty set of non-terminal node labels

V_T - is a finite non-empty set of terminal node labels; unlike phrase structure grammars, the set $V_N \cap V_T$ need not be empty.

V_{EL} - is a non-empty set of edge labels

P - is a finite, non-empty set of production rules (or productions)

S - is the starting non-terminal symbol, $S \in V_N$.

Note that according to this definition, only node labels can play the non-terminal or terminal role symbols play in phrase structure grammars. Moreover, a node label can be both terminal and non-terminal. This makes it possible to consider derived structures of an RLD grammar both as RLDs belonging to the set defined by the RLD grammar, but also as intermediate structures from which other RLDs can be derived.

A production has the form

$$\text{LHS} \Rightarrow \text{RHS}$$

where LHS and RHS are scion patterns, to be defined below.

The concept of a scion is similar to that of a subgraph. More precisely, a scion S of a graph g is a subgraph induced by some subset $N' \subseteq N$, where N is the set of nodes of g , and a labelled cutset containing the edges between N' and N - N' (independently of their direction). FIG. 1(b) shows a scion of the RLD of FIG. 1(a). In representing the scion schematically, we will use negative integers to indicate nodes in N - N' . FIG. 1(c) and 1(d) show two more scions of the RLD of FIG. 1(a).

Every scion when considered independently of any graph is a scion pattern. The scion pattern of FIG. 1(b) will match any graph which contains two nodes labelled A , an edge labelled a linking the first to the second node and two edges labelled a, b respectively leaving the second node and pointing at different nodes. For a given graph g , if $\gamma(1)$, $\gamma(2)$ denote the nodes of g which are associated with nodes 1, 2 of the scion pattern of FIG. 1(b) during the match, the scion induced by $\{\gamma(1), \gamma(2)\}$ is the scion matched by the scion pattern (and looks identical to it).

Thus scions are also scion patterns when considered independently of any particular graph. Every scion pattern, however, is not necessarily a scion.

Scion patterns consist of a finite number of nodes interconnected with edge patterns rather than simple edges. FIG. 2 shows the edge patterns that will be used here. Small letters from the end of the alphabet will serve as variable names. Since node labels and node references are not important here, we represent nodes by small circles.

The edge pattern of FIG. 2(a) will match an edge labelled a . That of FIG. 2(b) will match if there is an edge labelled "a" or there is no edge at all connecting the two nodes under consideration. The specification of an edge label is not necessary. Thus the edge pattern of FIG. 2(c)

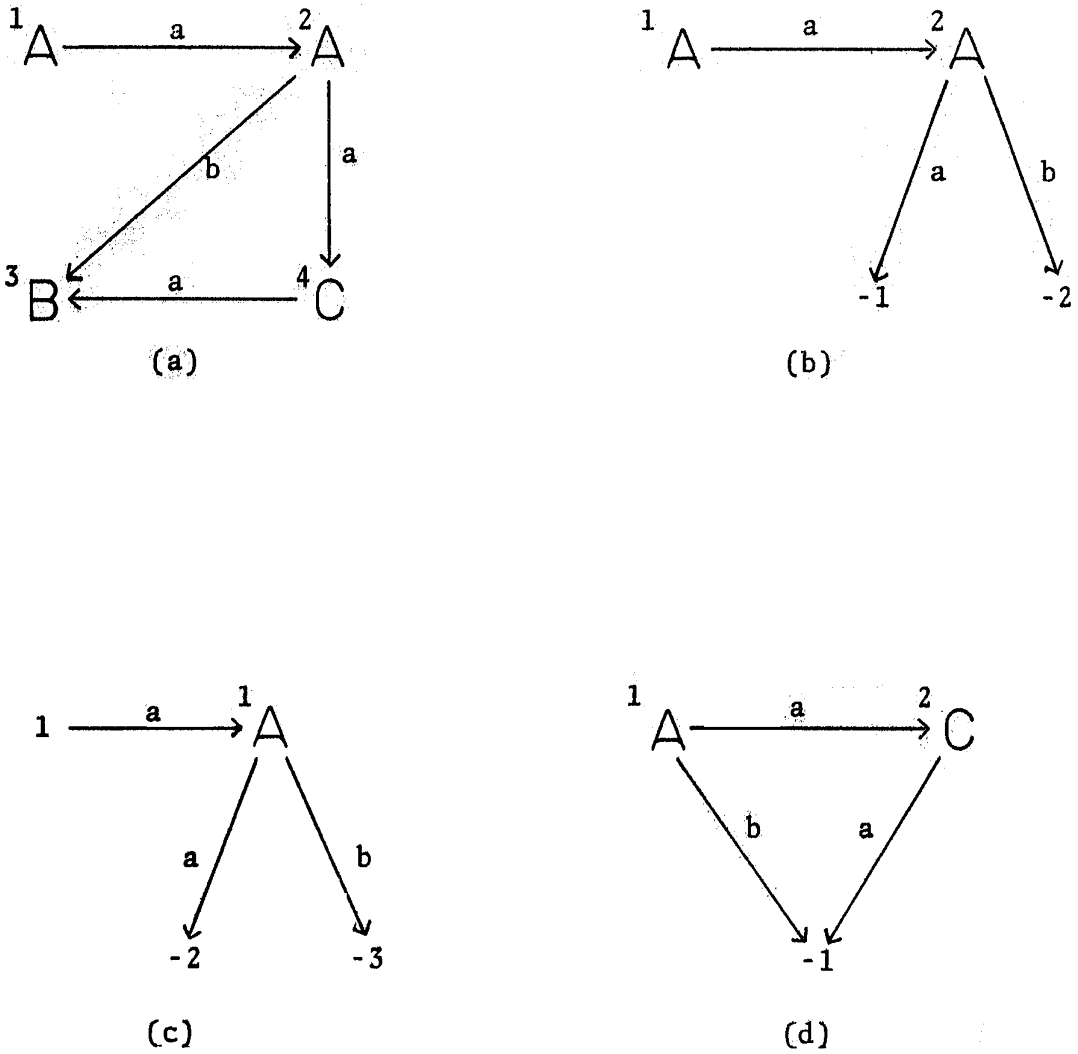


FIG. 1.

will match if there exists an edge (of any label) and that of FIG. 2(d) will match independently of whether there exists an edge.

Arbitrary, directed, rooted graphs will be represented in terms of RLDs by labelling the edges that leave any given node with labels such as d1, d2,... or e1, e2,... which simply serve as indices. The edge pattern of FIG. 2(e) will match any edge whose label is ei , for some positive integer i , and it will assign that label to variable x . Any of those edge patterns can be used as elements of the cutset of a given scion pattern in any combination. This explains why any scion is also a scion pattern, but clearly the configuration of FIG. 2(f) is a scion pattern but not a scion.

Cutset edge patterns may be such that they match a set of outgoing or incoming edges associated with a node. Such set edge patterns will be indicated by an arrow pointing towards or away from an arc. FIG. 3 shows instances of the types of set edge patterns we will allow. The set edge pattern of FIG. 3(~~d~~)c will match a set of ei- and dj-labelled incoming edges with at least one ei and one dj , where i, j are positive integers. This set will be assigned to x . Finally, the set pattern of FIG. 3(d) will match a possibly empty set which does not contain any ei-labelled edges and will assign it to y . The notation of FIG. 3(a), 3(b) will also be allowed for simple (non-set) edge patterns, to specify assignments of edge labels with no restrictions on the type of label that can be assigned.

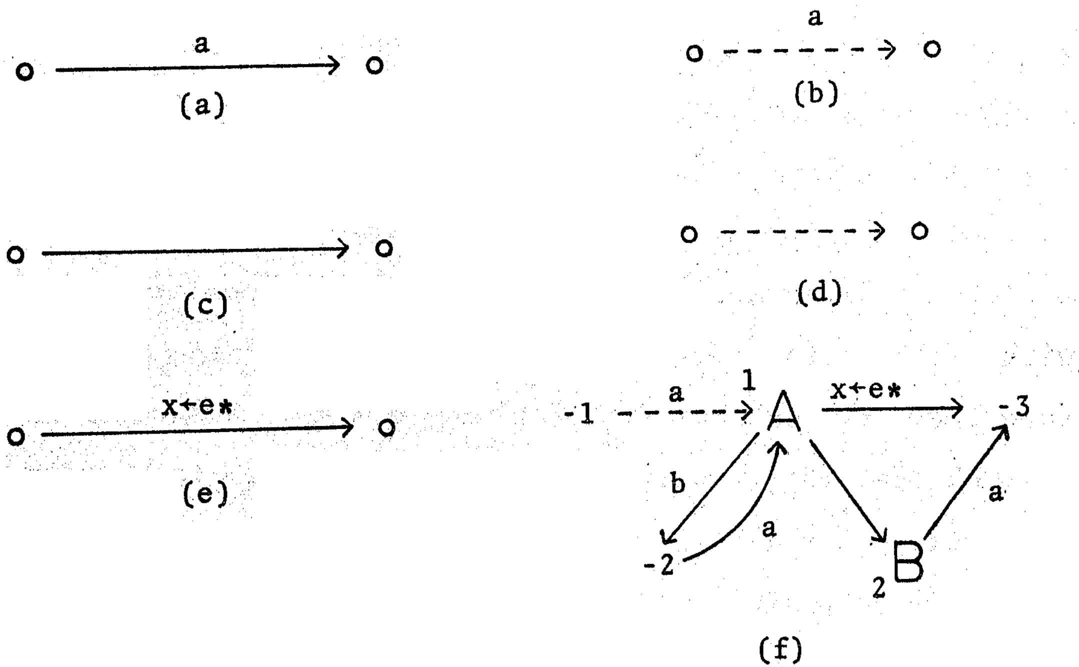


FIG. 2.

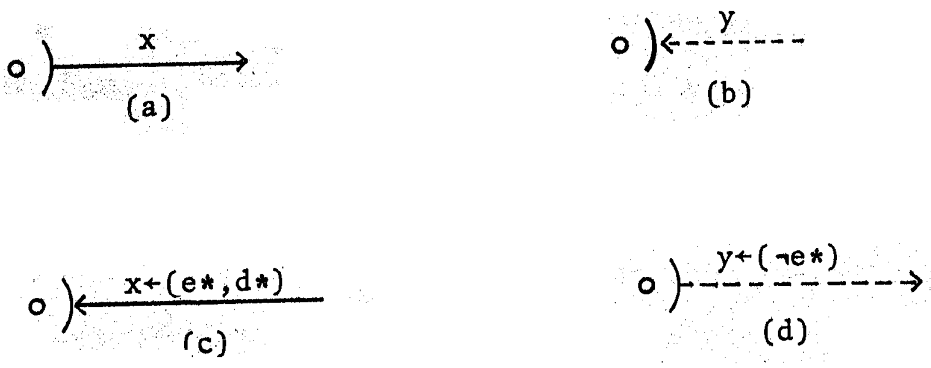


FIG. 3.

Matching a scion pattern with associated node references $1, 2, \ldots, i, -1, -2, \ldots, -j$ against a scion of a graph g with associated node references $\gamma(1), \gamma(2), \ldots, \gamma(i), \gamma(-1), \ldots, \gamma(-j)$ involves considering k and $\gamma(k)$ for $1 \le k \le i$ and performing the following test:

Let $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_m$ be the (possibly set) edge patterns associated with k. Partition the set of edges associated with $\gamma(k)$ into m subsets, say $\varepsilon_1', \varepsilon_2', \ldots, \varepsilon_m'$ and make sure that ε_q matches ε_q' for $1 \le q \le m$. The match must be such that if an edge pattern ε_q links n_1 to n_2 in the scion pattern then ε_q' links $\gamma(n_1)$ to $\gamma(n_2)$.

In applying a production to a graph we must complete two steps:

a. Match the LHS scion pattern to some scion in the graph. If this match fails, the production is not applicable.

b. Use the RHS scion pattern and the information from the match of the LHS scion pattern to modify the graph.

In order for a production to modify a graph, it must be possible to specify node and edge additions and deletions. Node additions and deletions can be determined by comparing the LHS (positive) node references to the RHS ones. If a node reference appears only on the LHS, then the node that is associated to it during the pattern match with the LHS scion pattern, will be deleted during the application of this production. If a node reference only appears on the RHS, a node will be created and will be added to the graph during the application of the production.

RHS scion patterns may use as labels for their edge patterns variables, which are always assumed to have been assigned values during the match of the LHS scion pattern. They may also use labels such as x(e*, d*) , which indicate that the edge set assigned to x is to be changed in the following way:

Let z be the set of ei edges in x , ordered according to the increasing value of their subscripts. Each ei edge is taken in order; if n is its starting node, the edge is renamed dj , where j-1 is the highest subscript of any dk edge running from n .

A new edge may be named e+ (or d+, f+ etc.). This means that it is to be assigned label ei (or di, fi etc.) if (i-1) is the largest integer such that there exists an r(i-1)-labelled edge leaving the source node of the new edge.

FIG. 4(a) shows a production which when applied to the graph of FIG. 4(b) gives the graph of FIG. 4(c). Note that node 2 of the graph of FIG. 4(b) was deleted completely by the production. Moreover, node references have been re-arranged from FIG. 4(b) to FIG. 4(c) to stress the fact that they are to be used only for reference purposes.

Application of a production to an RLD is only allowed if the resulting graph satisfies the conditions described at the beginning of this section. A <u>derivation</u> with respect to an RLD grammar consists of a finite sequence of applications of the RLD grammar productions. A derived RLD all of whose node labels are terminal is an element of the set of RLDs defined by the RLD grammar.

For a more formal definition of RLD grammars the reader is referred to [14, 15].

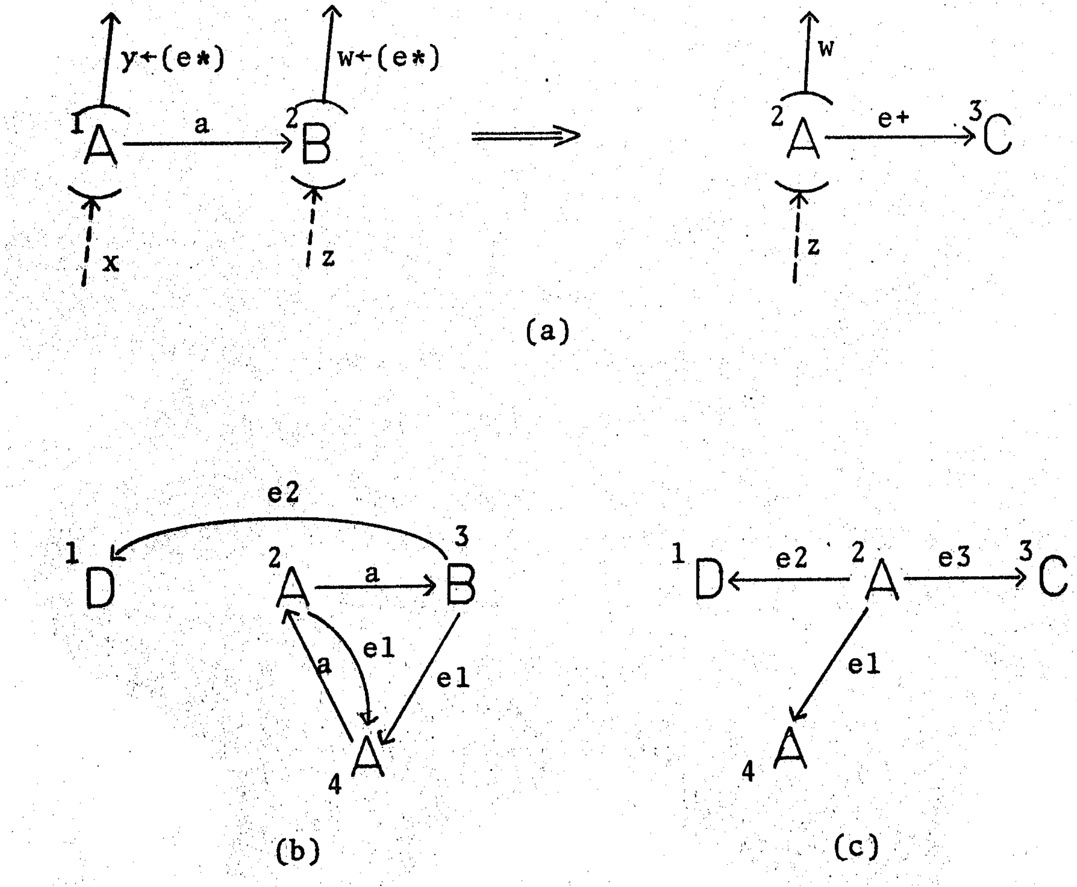
We consider the RLD grammar formalism inexpensive primarily for one reason: it avoids the general subgraph isomorphism problem, a solution to which is essential for a decision on the applicability of a production, and replaces it with the subgraph isomorphism problem for digraphs with uniquely labelled edges leaving each node. The improvement in efficiency is quite dramatic since the complexity of the general algorithm is suspected to be an exponential function of the subgraph to be matched, whereas for the restricted algorithm it is a quadratic function on the number of nodes of the graph.

We end this section with a simple example of an RLD grammar. Consider the grammar G_{chain}

$$G_{chain} = (\{S\}, \{S, B\}, \{e\}, P, S)$$

where P includes the single production named $p1$ of FIG. 5.

This production will add a new node labelled S and will connect it to the node previously labelled S with an edge labelled e . Moreover, it will label with B the node previously labelled S . Thus if we start with a single node labelled S , which is always the starting configuration for a derivation, and apply $p1$ n times, we will obtain the RLDs shown in FIG. 6 all of which belong to the set defined by G_{chain} since both S and B are terminal node labels.



(a)

(b)

(c)

FIG. 4.

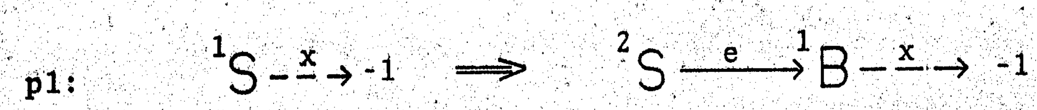


FIG. 5.

This grammar defines the set of all chains with first node labelled S and the rest of the nodes labelled B , and a proof that this is the case is straightforward. Note that the graphs derived by this grammar can be considered as pushdown stacks, since new nodes can be added only at one end of the chain, and the production p1 as an abstract "push" operation. The grammar can be modified by adding to it production p2 , shown in FIG. 7, which amounts to a pushdown stack "pop" operation. The generative power of the new grammar G'_{chain} , is the same as that for G_{chain} . Its productions, however, reflect the kinds of operations that we may wish to perform on elements of the set defined by the grammar. We will return to this point in section 4.

3. Examples of RLD Grammars

In this section we present RLD grammars which define closed 1 chains, 2 ordered trees, 3 unilaterally connected graphs, strongly 4 connected graphs, 5 Eulerian graphs, 6 Hamiltonian graphs, complete acyclic 7 graphs, 8 acyclic graphs, 9 Hasse diagrams and 10 lattices. The grammars we will actually propose will, in some instances, generate graphs which may include more structure than that required for the set under consideration (see [8]). This structure is there to help the actual derivation and may be deleted eventually in terms of node- and edge-deleting productions that will only be given occasionally. The sets of graphs mentioned above were chosen mainly because they can be

S	initial configuration
$S \xrightarrow{e} B$	first application of p1
$S \xrightarrow{e} B \xrightarrow{e} B$	second application of p1
$S \xrightarrow{e} B \xrightarrow{e} \cdots \xrightarrow{e} B$	nth application of p1

FIG. 6.

$${}^{1}S \xrightarrow{e} {}^{2}B \overset{\underline{x}}{\dashrightarrow} -1 \quad \Longrightarrow \quad {}^{2}S \overset{\underline{x}}{\dashrightarrow} -1$$

FIG. 7.

formally defined and are well-known and understood, also because they are interrelated. Thus the reader may view the examples as a sequence of steps which lead to the definition of an RLD grammar for lattices and to an argument we consider convincing that the proposed RLD grammar does in fact define the set of all lattices.

3.1 Closed Chains

By closed chain we mean graphs of the form shown in FIG. 8(a). The grammar G_{cchain} is defined by

$$(\{S\}, \{S, B\}, \{e, a\}, P, S)$$

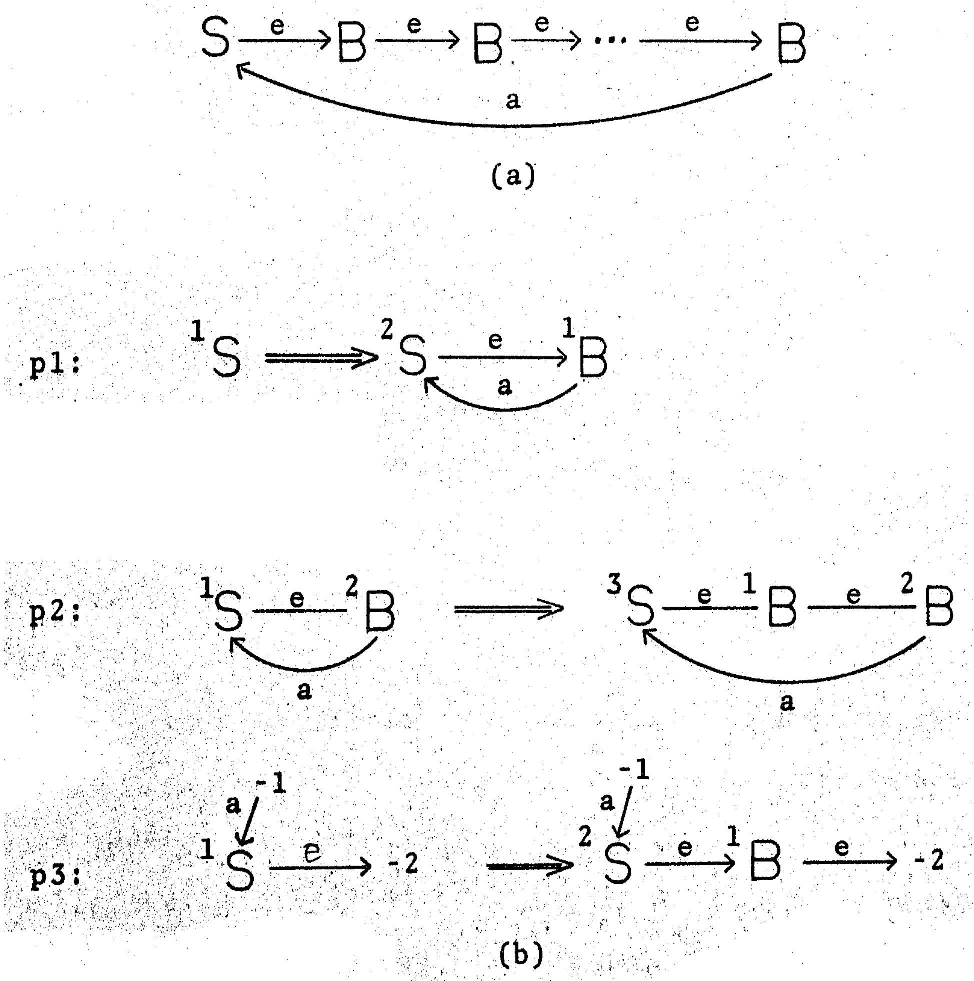
and the productions are given in FIG. 8(b). Production p1 is only applicable during the first step of a derivation, since this is the only time the node labelled S has no incoming or outgoing edges. During every other step of a derivation, production p2 is used and it keeps adding another node to the top of the chain always linking it to the bottom of the chain with the a-labelled edge.

3.2 Ordered Trees

The grammar G_{tree} is defined as follows:

$$(\{S\}, \{S\}, \{ei \mid 1 \le i\}, P, S)$$

where P consists of the single production p1 of FIG. 9. This production simply adds another successor to an already existing node. If the edges that already exist leaving this


S
e
B
e
B
e
...
e
B
a
p1:
1
S
2
S
e
1
B
a
p2:
1
S
e
2
B
a
3
S
e
1
B
e
2
B
a
p3:
a
-1
1
S
e
-2
a
-1
2
S
e
1
B
e
-2

(a)

(b)

FIG. 8.

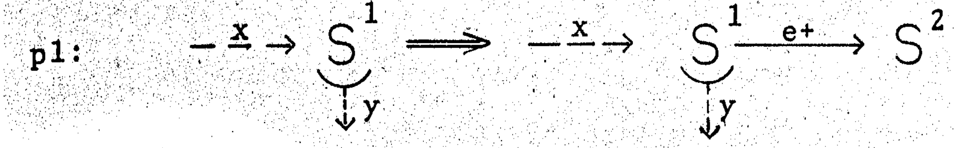

p1:
x
S
1
y
x
S
1
e+
S
2
y

FIG. 9.

node are labelled e1, e2,..., ei , the new edge will be labelled e(i+1) . It is, again, straightforward to show that ordered trees and only ordered trees are generated by this grammar.

3.3 Unilaterally Connected Graphs

A graph g is unilaterally connected if and only if for any two nodes m, n of g it is possible to move from m to n or n to m through a walk, i.e., a path some of whose nodes and edges may be visited more than once.

The definition above is axiomatic rather than constructive in the sense that it tells us how to recognize unilaterally connected graphs but not how to construct them or operate on them so that the unilateral connectedness property is preserved. To solve this problem we will use a simple theorem shown in [13, p. 199] which states that a graph is unilaterally connected if and only if it has a spanning walk, i.e., a walk that visits all nodes.

The grammar $G_{uconnected}$ is defined as follows:

$$(\{S, A\}, \{A\}, \{ei \mid 1 \leq i\}, P, S)$$

and the productions in P are given in FIG. 10. We will now show that this grammar defines precisely the set of all unilaterally connected graphs.

Let us prove that any RLD generated by the grammar is unilaterally connected. First note that at any particular

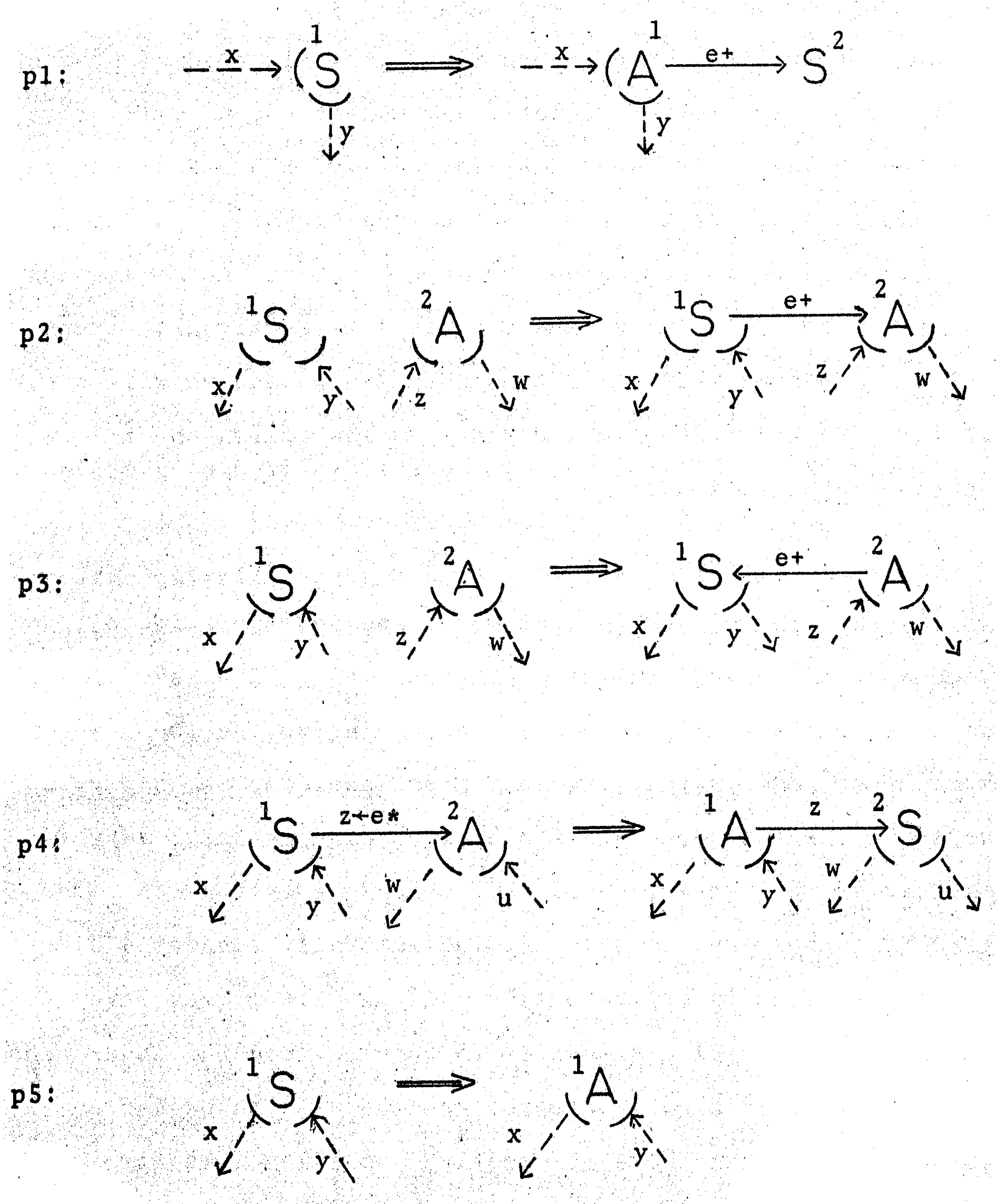

p1:
x
1
S
y
x
1
A
e+
S
2
p2:
1
S
x
y
2
A
z
w
1
S
e+
2
A
x
y
z
w
p3:
1
S
x
y
2
A
z
w
1
S
e+
2
A
x
y
z
w
p4:
1
S
z←e*
2
A
x
y
w
u
1
A
z
2
S
x
y
w
u
p5:
1
S
x
y
1
A
x
y

FIG. 10.

time in the derivation there will be exactly one node labelled S , except after the last step when S is replaced by A through an application of production p5 . Instead of proving that any derived RLD is unilaterally connected, we will prove the (equivalent) statement that any derived RLD has a spanning walk whose last visited node is the node to which p5 is applied. The proof is an induction on the length ℓ of the derivation.

For $\ell = 1$ the above statement is trivially true.

Assume that it is true for $\ell = k$ and we will prove it for $\ell = k+1$. Consider any derivation of length k+1 which involves application of

$$pi_1, pi_2, \ldots, pi_k, p5$$

where pi_j is p1, p2, p3 or p4 . We know that if p5 were applied instead of pi_k , the resulting RLD, say g , would be unilaterally connected, according to the induction hypothesis, and in fact the last node labelled S , say n_1 , would be the last node visited by the spanning walk. Now if instead of p5 we apply p2, p3 or p4 and then p5 , the resulting RLD, say g' , is also unilaterally connected since it differs from g structurally only in that it may have an extra edge. If p1 is applied to node n_1 on the other hand, a new node, say n_2 , is created and there is an edge from n_1 to n_2 . But then there is clearly a spanning walk for g' which visits n_2 last.

We must now show that any unilaterally connected graph can be derived by the grammar, or that any graph which has a spanning walk can be derived by the grammar. The proof is an induction on the length ℓ of the walk.

For $\ell = 0$ the RLD (which consists of a single node labelled A) can clearly be created.

Assume that this is also true for $\ell = k$ and we will show it for $\ell = k+1$. Consider any RLD g which has a spanning walk of length k+1 , and let the last two nodes visited by the walk be n_1 and n_2. Without any loss of generality we will assume that n_2 is visited only once by the spanning walk (otherwise the RLD has a spanning walk of length k and it can therefore be generated by the grammar).

Let the incoming edges of n_2 be $e_1, e_2, \ldots, e_r$ and the outgoing ones $f_1, f_2, \ldots, f_s$ while the other nodes these edges are connected to are respectively $n_{11}, n_{12}, \ldots, n_{1r}, n_{21}, n_{22}, \ldots, n_{2s}$. Consider the RLD g' obtained by deleting n_2 and the edges associated with it. g' has a spanning walk with n_1 as last node visited, therefore there exists a derivation

$$pi_1, pi_2, \ldots, pi_t, p5$$

for it in $G_{uconnected}$. Consider the RLD obtained after the application of pi_t and say that n_3 is the node labelled S at this point. Add then the following applications of productions to the derivation

$$\ldots pi_t, p4, p4, \ldots, p4$$

which move S to n_1 from n_3 (clearly this is possible since there exists a path from n_1 to n_3). Extend now the derivation to construct the new node and create edges $e_1, e_2, \ldots, e_r, f_1, f_2, \ldots, f_s$:

$$\ldots, p4, p1, p2, \ldots, p2, p3, \ldots, p3 .$$

Finally apply p5 to obtain g. This completes the proof that every unilaterally connected graph can be generated by $G_{uconnected}$.

The RLDs derived by $G_{uconnected}$ obviously have labelled edges. If the labelling is important, additional productions can be given which will basically enable the interchange of edge labels for two edges leaving the same node. This problem did not concern us in the definition of $G_{uconnected}$ and will not concern us for the rest of the paper.

3.5 Strongly Connected Graphs

A graph is strongly connected if and only if it contains a closed spanning walk, i.e., a spanning walk for which the first and last node visited are the same.

The grammar $G_{connected}$ we will propose is different from $G_{uconnected}$ only in that it guarantees that there is always a "backpointer" edge labelled a from the last node visited by the spanning walk to the first:

$$(\{S, X, A\}, \{A\}, \{a, ei \mid 1 \leq i\}, P, S)$$

where the productions in P are given in FIG. 11. The proof that this grammar generates a graph if and only if it is strongly connected is similar to that given for $G_{uconnected}$ and will not be given here. Production p1 is applicable only when we wish to derive the one-node RLD. Production p2 constructs a smallest closed chain which is expanded each time p3 is applied. Productions p4, p5, p6 allow the creation of edges which connect any node to that at the end of the walk, or enable the walk to move on along an already existing edge (p6). Production p7 re-labels the node at the end of the walk as A when the derivation is complete.

3.6 Eulerian Graphs

A graph is Eulerian if and only if there exists a closed spanning walk which visits every edge of the graph exactly once [13, p. 204].

It follows from the above statement that the grammar $G_{connected}$ which ensures that there is a closed spanning walk, will do for Eulerian graphs as well after the deletion of production p4 (which allows the creation of an edge which will not be traversed by the walk, and p6 which allows the walk to move along edges that have already been visited) and the replacement of p5 by production p5' (FIG. 12) (which allows the walk to come back to a previously visited node through a new edge).

3.7 Hamiltonian Graphs

A graph is Hamiltonian if and only if it contains a spanning cycle, i.e., a spanning walk which visits each node on the graph exactly once except for the first and last node visited, which are identical.

The grammar we propose simply constructs a closed chain and then proceeds to add edges as needed:

$$(\{S, X, A\}, \{A\}, \{a, ei \mid 1 \le i\}, P, S)$$

where P is given in FIG. 13.

3.8 Complete Acyclic Graphs

A graph is complete acyclic if and only if it is possible to number the nodes so that $i<j$ implies that i is adjacent to j [17].

The grammar $G_{cacyclic}$ is given by

$$(\{S\}, \{A\}, \{ei \mid 1 \le i\}, P, S)$$

where P is given in FIG. 14.

3.9 Acyclic Graphs

A graph is rooted acyclic if it is rooted and contains no cycles. It can easily be shown that the nodes of any such graph can be numbered so that $i<j$ implies that there is no edge from i to j.

The grammar $G_{acyclic}$ is defined by

$$(\{S, R\}, \{A\}, \{a, ei \mid 1 \le i\}, P, S)$$

where P is given in FIG. 15.

Production p1 constructs another node which is placed at the top of a chain of nodes connected with a-labelled edges. Every node of the derived graph appears on this chain. Productions p2, p3 create an ei-labelled edge for $i \ge 1$

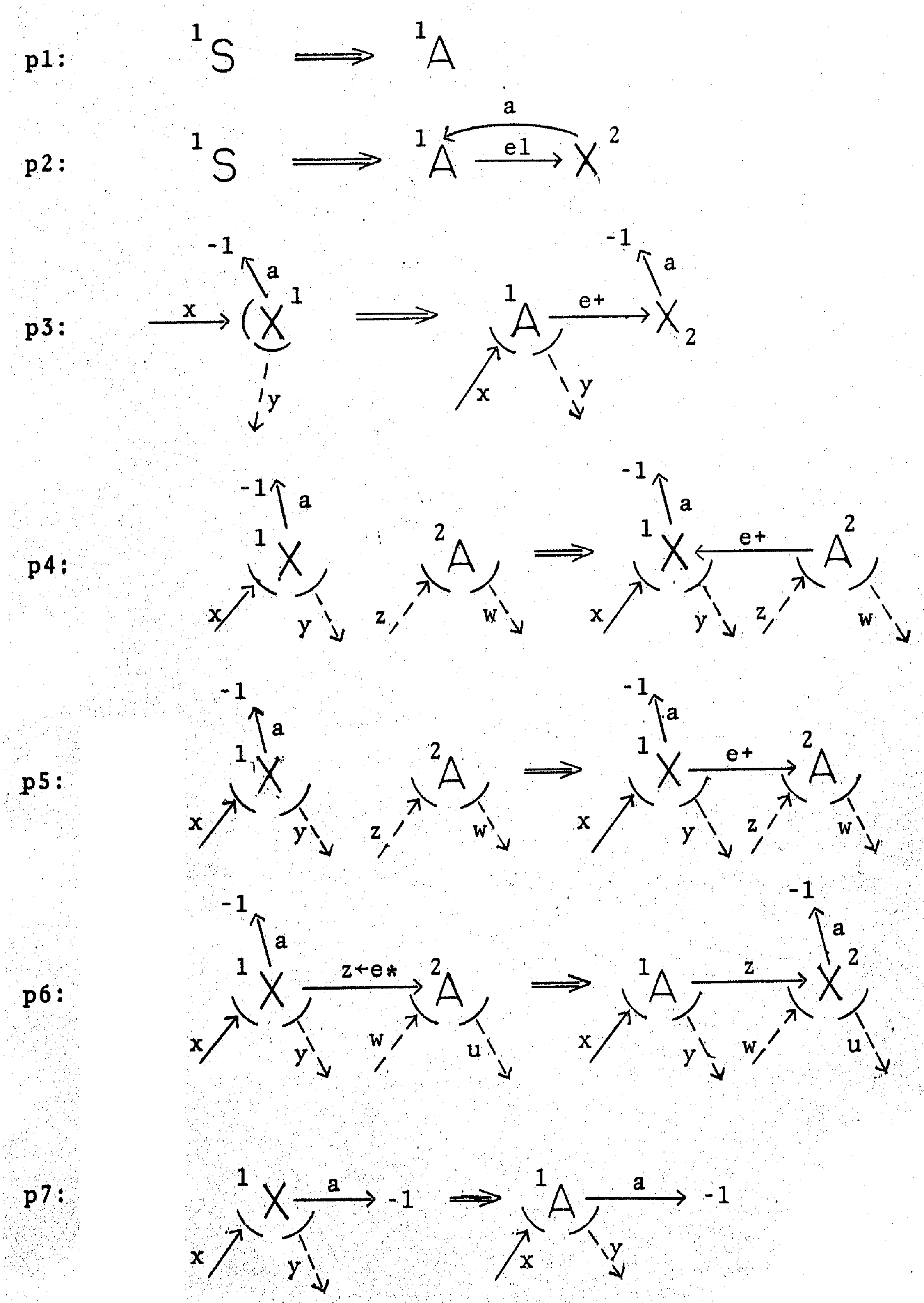

p1:
1 S
1 A
p2:
1 S
1 A
a
e1
X 2
p3:
-1
a
x
X 1
y
1 A
e+
-1
a
X 2
x
y
p4:
-1
a
1 X
x
y
2 A
z
w
-1
a
1 X
e+
2 A
x
y
z
w
p5:
-1
a
1 X
x
y
2 A
z
w
-1
a
1 X
e+
2 A
x
y
z
w
p6:
-1
a
1 X
z←e*
2 A
x
y
w
u
1 A
z
-1
a
X 2
x
y
w
u
p7:
1 X
a
-1
x
y
1 A
a
-1
x
y

FIG. 11.

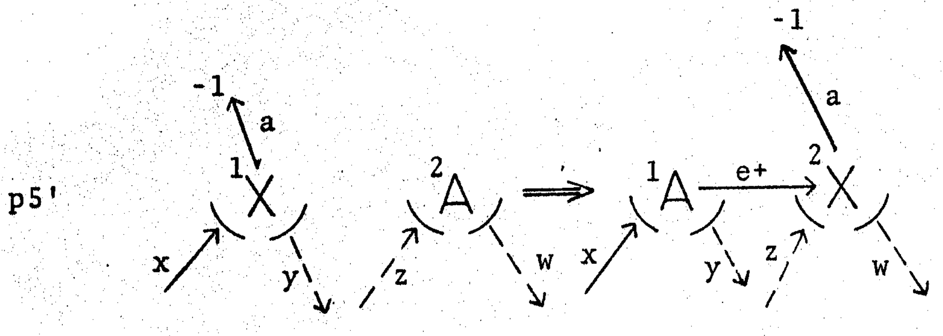

p5'
-1
a
1
X
x
y
2
A
z
w
1
A
e+
2
X
-1
a
x
y
z
w

FIG. 12.

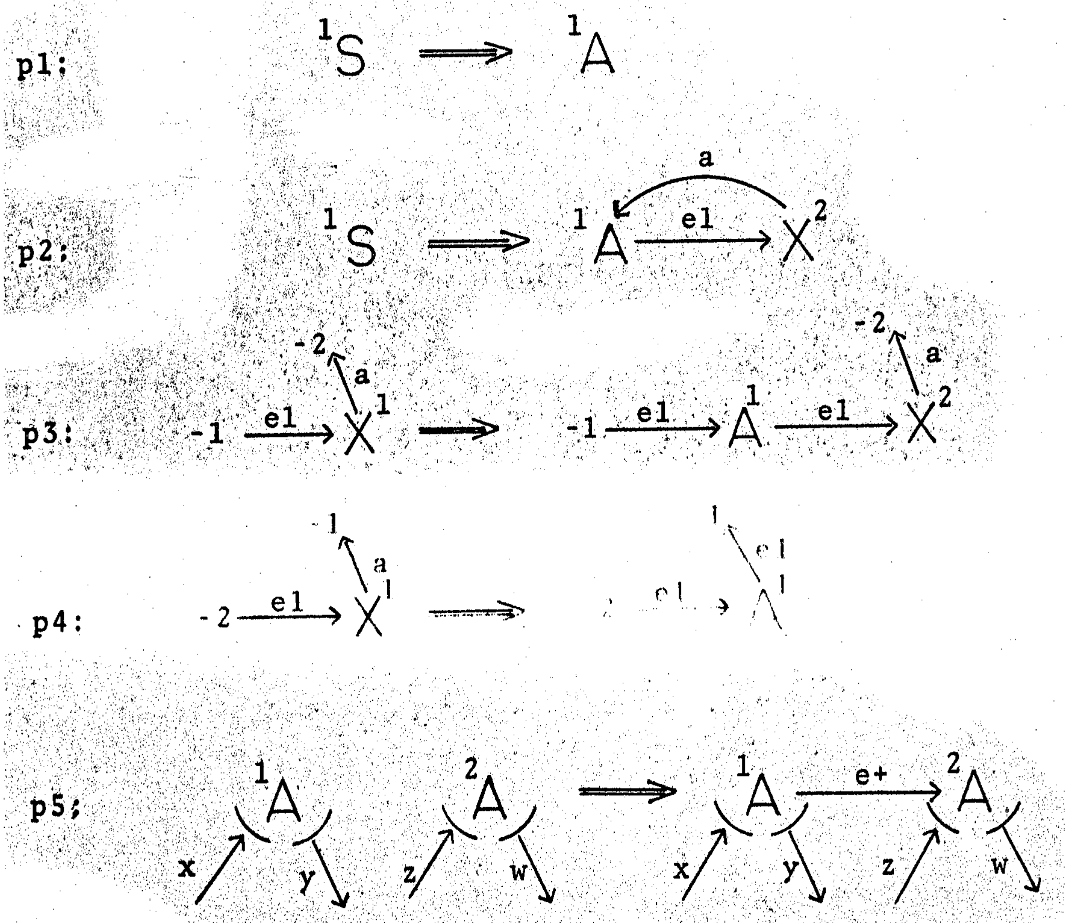

p1:
1
S
1
A
p2:
1
S
a
1
A
el
X
2
p3:
-2
a
-1
el
X
1
-1
el
A
1
el
X
2
-2
a
p4:
-1
a
-2
el
X
1
1
el
el
1
p5;
1
A
x
y
2
A
z
w
1
A
e+
2
A
x
y
z
w

FIG. 13.

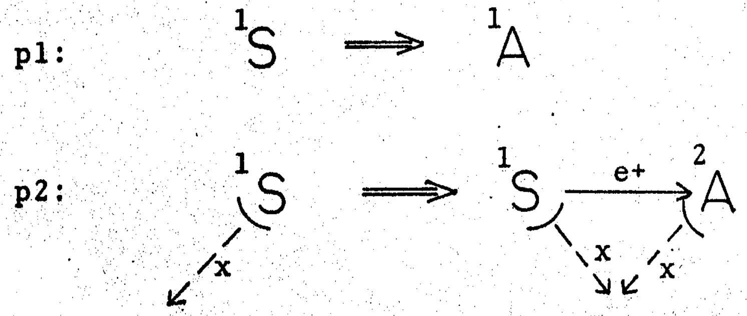

p1:
1
S
1
A
p2:
1
S
x
1
S
e+
2
A
x
x

FIG. 14.

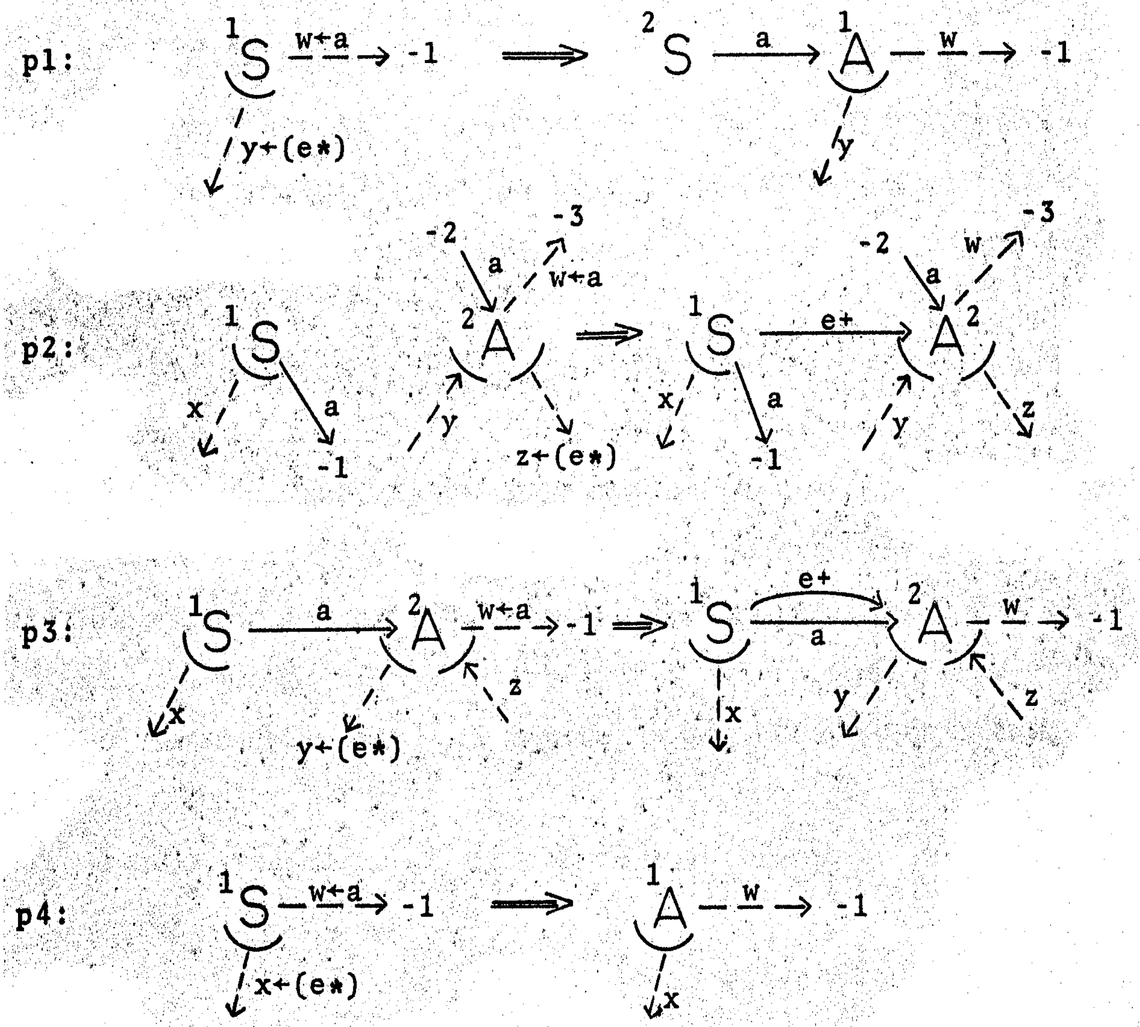

p1:
1
S
w+a
-1
y+(e*)
2
S
a
1
A
w
-1
y
p2:
1
S
x
a
-1
-2
a
-3
w+a
2
A
y
z+(e*)
1
S
x
a
-1
e+
-2
a
w
-3
A
2
y
z
p3:
1
S
a
x
2
A
w+a
-1
y+(e*)
z
1
S
e+
a
x
2
A
w
-1
y
z
p4:
1
S
w+a
-1
x+(e*)
1
A
w
-1
x

FIG. 15.

which connects the node at the top of the chain to some node constructed earlier. The necessary and sufficient condition for acyclic graphs will clearly be satisfied because of this construction.

The chain of a-labelled edges, which can be considered as auxiliary, can be deleted, by adding three simple productions to the grammar. Note however that this can only be done if the rootedness property is not violated by the application of such rules.

3.10 Hasse Diagrams

Every acyclic graph defines a partial order and conversely. Hasse diagrams are acyclic graphs such that if there is a path from node i to node j, then there is no edge from i to j.

The grammar we will propose is similar to that for (rooted) acyclic graphs, it always checks, however, before constructing an edge that there is no path (of ei-labelled edges) already connecting the two nodes associated with the edge. As with the grammar for acyclic graphs, we will use auxiliary edges labelled a to define a linear order on the nodes of the graph which is compatible with the partial order we are attempting to define.

The grammar G_{Hasse} is defined below and its productions are given in FIG. 16.

$$(\{S, T, A, P, B\}, \{A\}, \{a, b, c, ei, di \mid 1 \le i\}, P, S) .$$

We will now show that every graph generated by this grammar is a Hasse diagram and conversely, every Hasse diagram can be generated by this grammar.

First we will prove the first statement by an induction on the following proposition:

The $(m+1)^{st}$ node of a derivation, $m \geq 0$, will be generated through the application of p2 or p3 and at the time of application the partially derived graph must have the form shown in FIG. 17.

The proof of this statement is an induction on m.

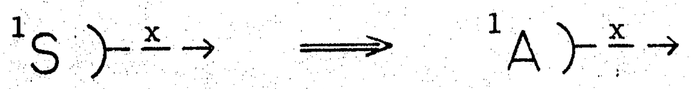
For $m = 0$ the statement is trivially true since the starting configuration is simply S.

Assume that the statement is true for $m \leq n$ and we will prove it for $m = n+1$. Consider a partially derived graph at the time its $(n+1)^{st}$ node is about to be generated. It must have n nodes and be in the form indicated above. The $(n+1)^{st}$ node is generated either by production p2 or p3.

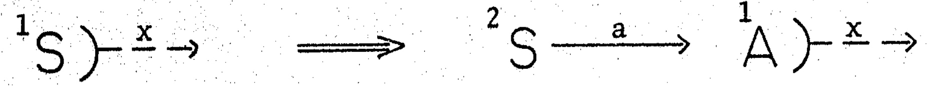
In the first case the new graph has the form described in FIG. 17 and the induction step is true.

In the second case, i.e., when p3 is applied, the new graph has the form shown in FIG. 18(a) and only p5 is applicable resulting in the form shown in FIG. 18(b). Now we can apply either p6 or p8 followed by p6 obtaining the form shown in FIG. 18(c). Production p4 or p5 is now applicable. Application of p4 results in the form of FIG. 18(d), while the application of p5 will not be pursued since it is similar.

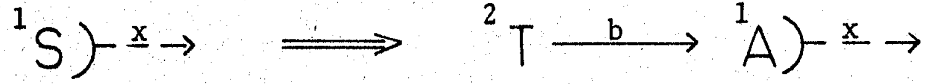
p1:



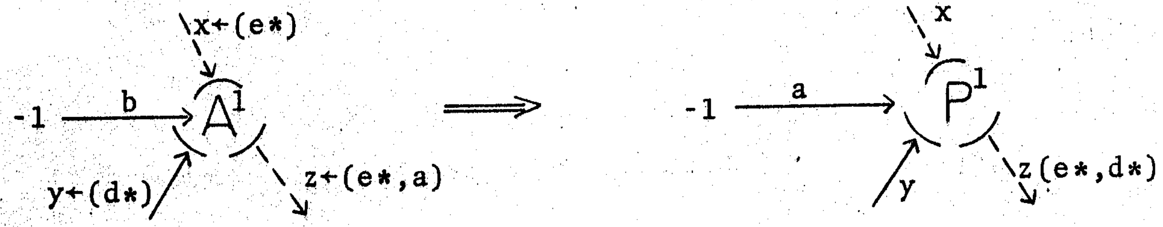
p2:



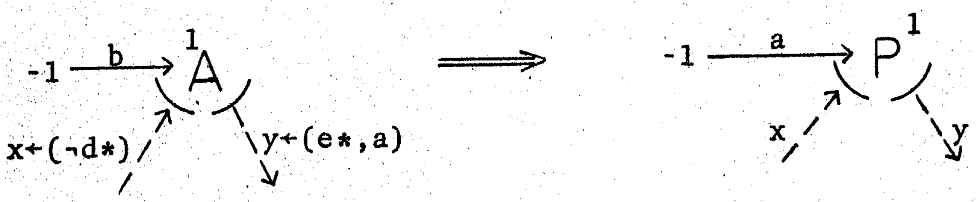
p3:



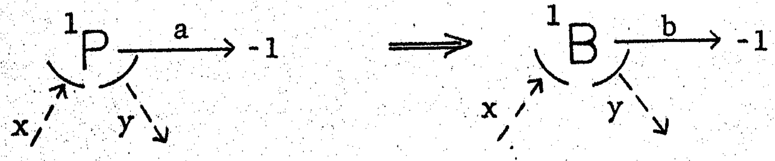
p4:



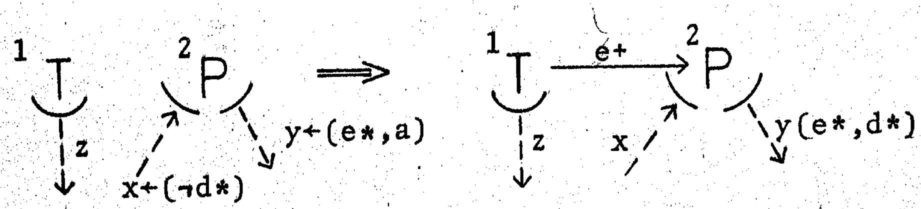
p5:



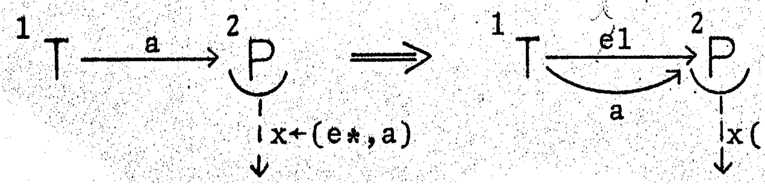
p6:



p7:



p8:



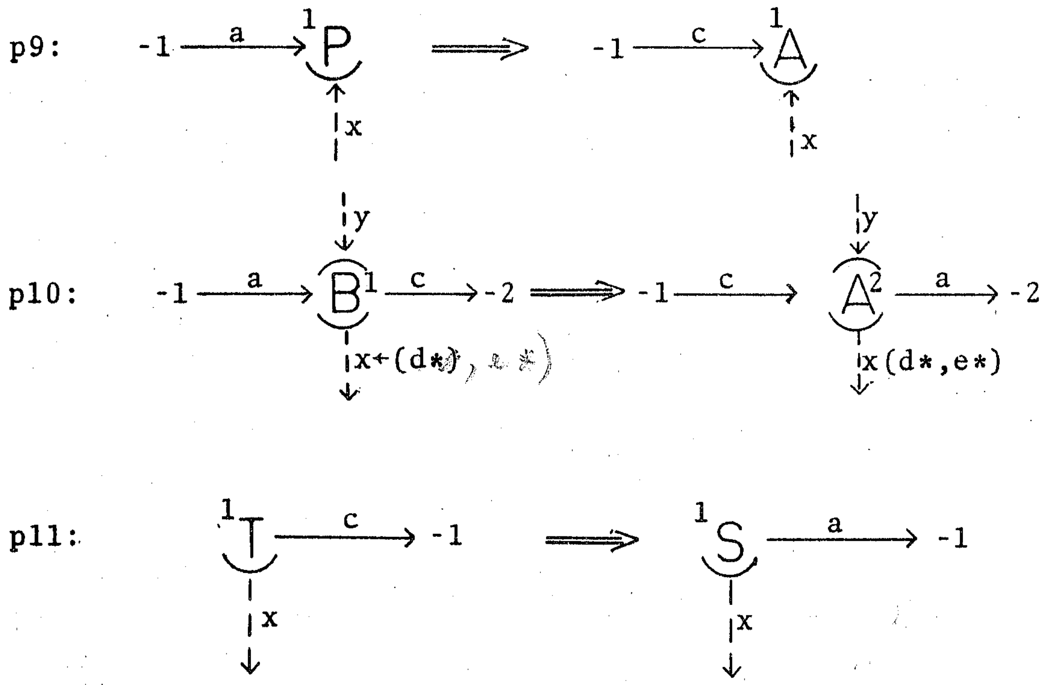

p9:
-1
a
1
P
x
c
1
A
x
p10:
y
-1
a
B
1
c
-2
x+(d*), e*)
-1
c
y
A
2
a
-2
x(d*,e*)
p11:
1
T
c
-1
x
1
S
a
-1
x

FIG. 16.

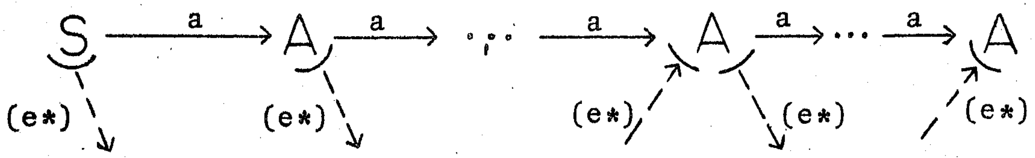

S
a
A
a
a
A
a
a
A
(e*)
(e*)
(e*)
(e*)
(e*)

FIG. 17.

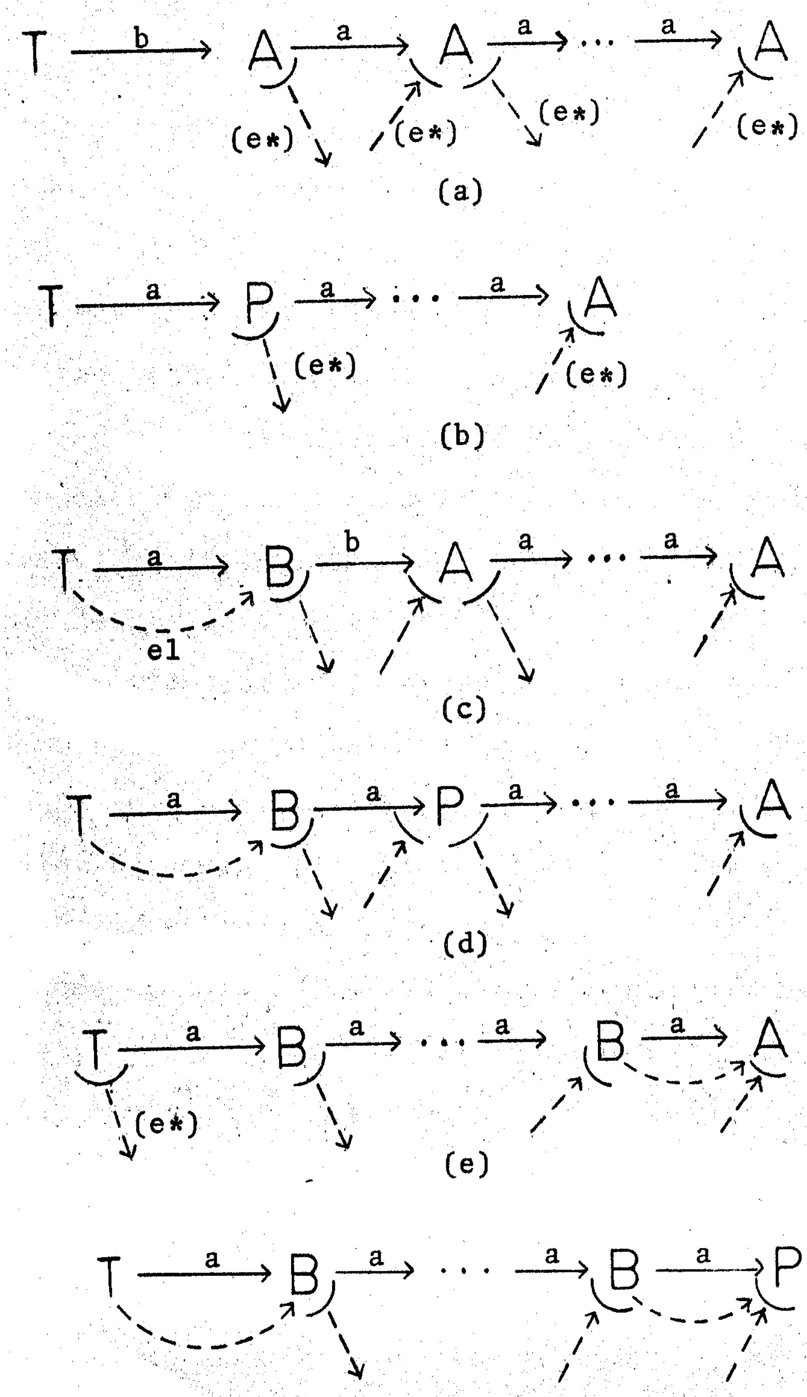

T
b
A
a
A
a
...
a
A
(e*)
(e*)
(e*)
(e*)
(a)
T
a
P
a
...
a
A
(e*)
(e*)
(b)
T
a
B
b
A
a
...
a
A
el
(c)
T
a
B
a
P
a
...
a
A
(d)
T
a
B
a
...
a
B
a
A
(e*)
(e)
T
a
B
a
...
a
B
a
P
(f)

FIG. 18.

We can next apply p6 or p7 followed by p6 to obtain the form of FIG. 18(e) . This process can be continued until eventually the form of FIG. 18(f) is obtained. At this point p9 is applicable followed by a number of applications of p10 and one application of p11 resulting in the form of FIG. 16. Thus the only possible partially derived graph that can be obtained after applying p3 to a graph having the form of FIG. 17, and such that p2 or p3 can be applied again, has itself the form specified by FIG. 17. This proves the statement.

We are in a position now to show that every graph generated by the grammar is a Hasse diagram. First of all, it is easy to show that any generated graph is acyclic since the addition of new edges is always carried out between the new node and already existing ones and it is easy to see that no cycles can be created by such derivations. We will show that the Hasse diagram condition is satisfied for any generated graph by contradiction. If it is not satisfied for some graph, there must be a point in its derivation when p7 is applied creating an edge between T and P when there is already a path of edges (labelled ei's and di's) connecting them. But this is impossible since whenever we create an edge leaving T and pointing to some node n , we also change all ei-labels of outgoing edges for n to di's (through productions p7, p8), and whenever a node is found to have one or more di-labelled incoming edges, its ei-labelled outgoing edges are re-labelled di (production p4). Thus it will never be the case that the Hasse diagram condition will be violated for a derived graph.

This completes the proof that every node generated by the grammar is a Hasse diagram.

We will now show that every Hasse diagram can be generated by this grammar.

The proof is an induction on the number, m , of nodes that are members of the diagram.

For $m = 1$ the graph can be generated trivially.

Assume that any Hasse diagram with $m \leq k$ nodes can be generated by G_{Hasse} , and we will prove the claim for $m = k+1$. Consider any Hasse diagram with $k+1$ nodes and assign an integer to each node so that if $i<j$ then there is no edge from i to j . It is easy to show that such an indexing exists.

Let n be the highest index and remove it from the Hasse diagram. The resulting Hasse diagram has k nodes and it can therefore be generated by G_{Hasse} with the nodes with higher indices coming first in the chain of a-labelled edges. It follows from the statement shown earlier that before the k^{th} node (first on the chain) is re-labelled A from S , the graph has the form of FIG. 17, and all edges that do not involve node n are already on the graph. Instead of applying p1 , we can apply p2 , if n has no ei-labelled edges leaving it, or p3 . If p2 is applied, we can then apply p1 to get the

Hasse diagram with k+1 nodes. If p3 is applied we proceed, as indicated earlier, to construct edges from the newly created node to all the nodes it is adjacent. The resulting graph will be exactly the Hasse diagram under consideration.

3.11 Lattices

The edges labelled ei , $1 \le i$, define a partial order for any Hasse diagram generated by G_{Hasse} . By a lattice we mean here a Hasse diagram such that any two of its nodes have a unique least upper bound (lub) and a unique greatest lower bound (glb) with respect to this partial order.

The grammar we will propose is an extension of G_{Hasse} . It generates graphs the same way G_{Hasse} does, on a chain with the most recently generated node at the top of the chain. Unlike G_{Hasse} , however, it tests for every newly created node i and every other node j with only incoming edge that defining the chain, that i and j have a unique glb . Moreover, at the end of the construction it tests that only the first node on the chain has no incoming ei-labelled edges (clearly the existence of a glb for all pairs of nodes implies that only the last node will have no outgoing ei-labelled edges). Both tests are such that a partially derived graph which does not satisfy the lattice condition will never lead to the derivation of a terminal graph.

The grammar $G_{lattice}$ is defined as follows:

$$(\{S, T, A, P, B, C\}, \{E\}, \{a, a', b, b', c, c', di, d'i, ei, e'i, f, f' \mid 1 \le i\}, P, S)$$

The productions of the grammar include productions p1, p3 - p8 and p10,~~p11~~ of G_{Hasse} . FIG. 19 shows the additional productions needed.

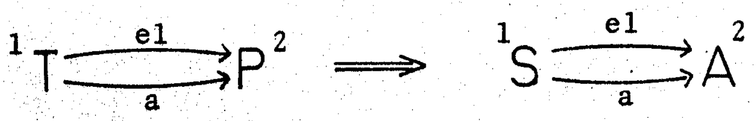
Before we prove this grammar correct, we need two lemmas about lattices and partially constructed lattices. The proofs of these lemmas are given in the appendix.

Lemma 1: A Hasse diagram is a lattice if and only if any two of its nodes have a unique glb and there is only one node with in-degree 0 (see [18, p. 112]).

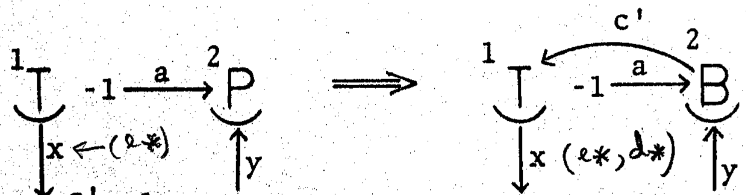
Lemma 2; Suppose that a partial structure has been generated by $G_{lattice}$ (it need not be a lattice) such that any two nodes have a unique glb , and a new node m is added to this structure. Let $n_1, n_2, \ldots, n_k$ be the nodes of the structure with no incoming ei-labelled edges. Then m and any other node n will have a unique glb if and only if m and n_i , $1 \le i \le k$, have a unique glb .

It follows from lemma 2 that once a new node is created with its associated ei-labelled edges, we need only test whether it and already constructed nodes with no incoming ei-labelled edges have unique glbs .

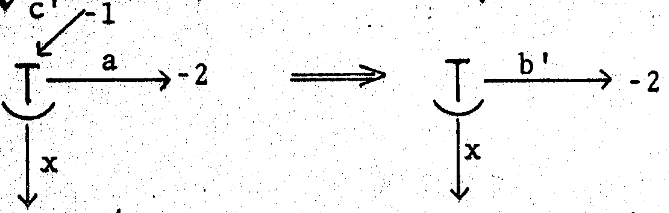
We will now show that a graph generated by $G_{lattice}$ is a lattice. Consider any such graph g . It must be a Hasse diagram since the productions in $G_{lattice}$ which are not in G_{Hasse} never create or destroy nodes and edges, except for a c'-labelled edge created by p9", which is immediately p9':



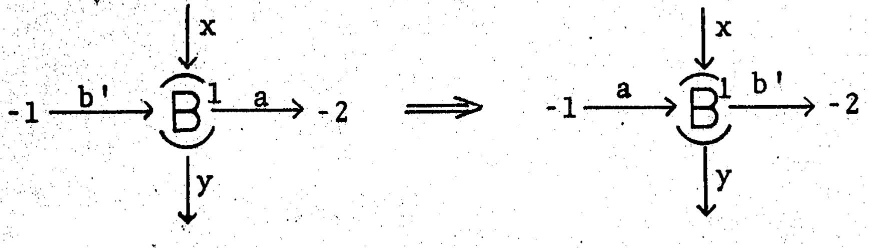
p9":



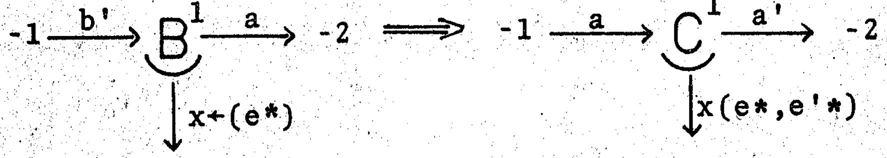
p9.1:



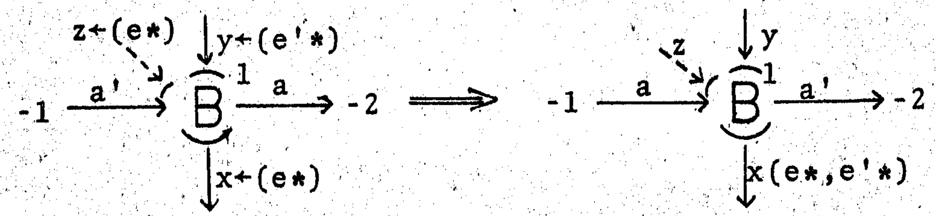
p9.2:



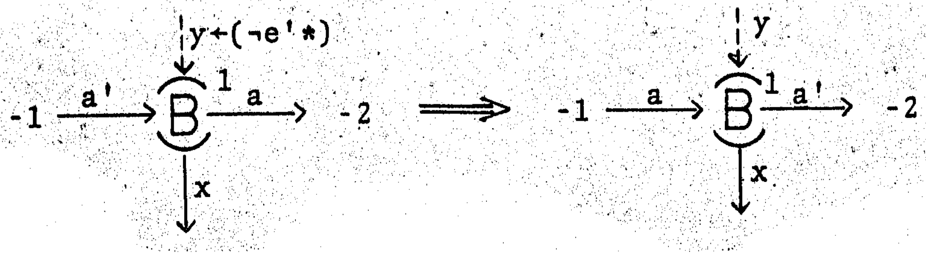
p9.3:



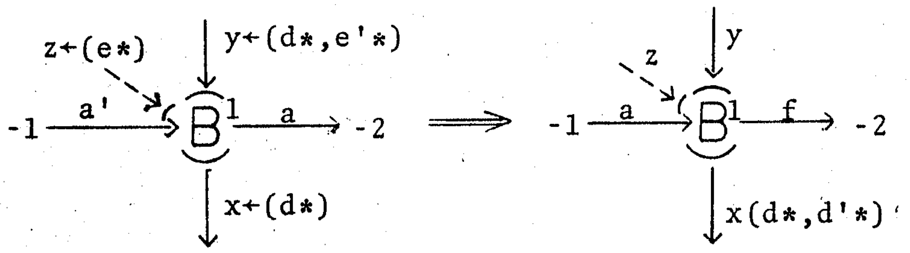
p9.4:



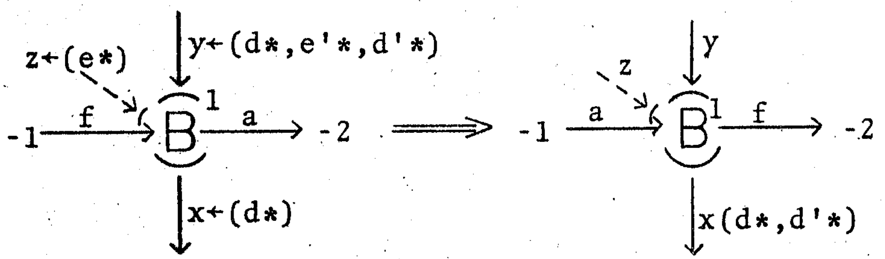
p9.5:



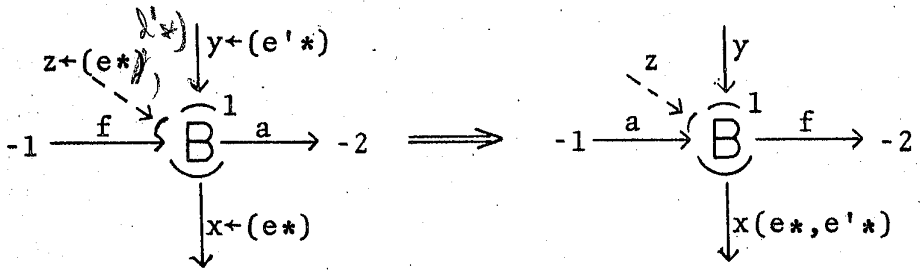
p9.6:



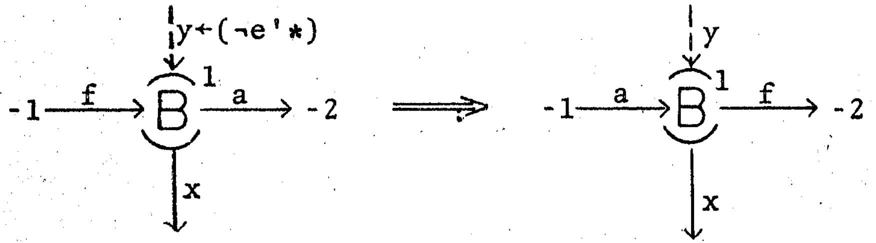
p9.7:



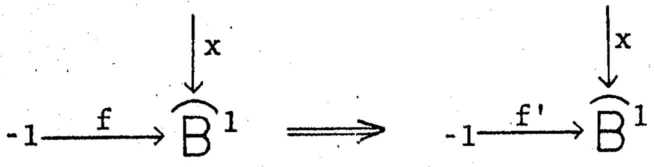
p9.8:



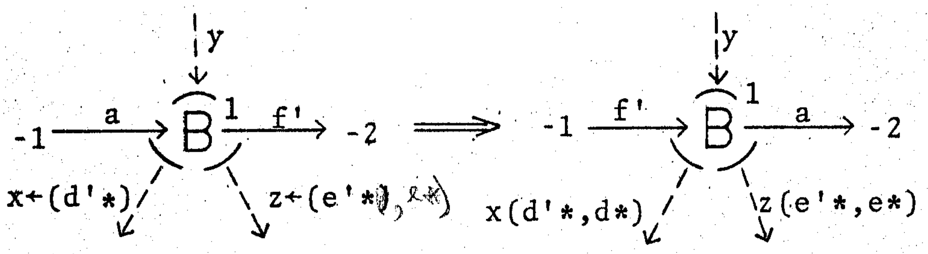
p9.9:



p9.10:



p9.11:



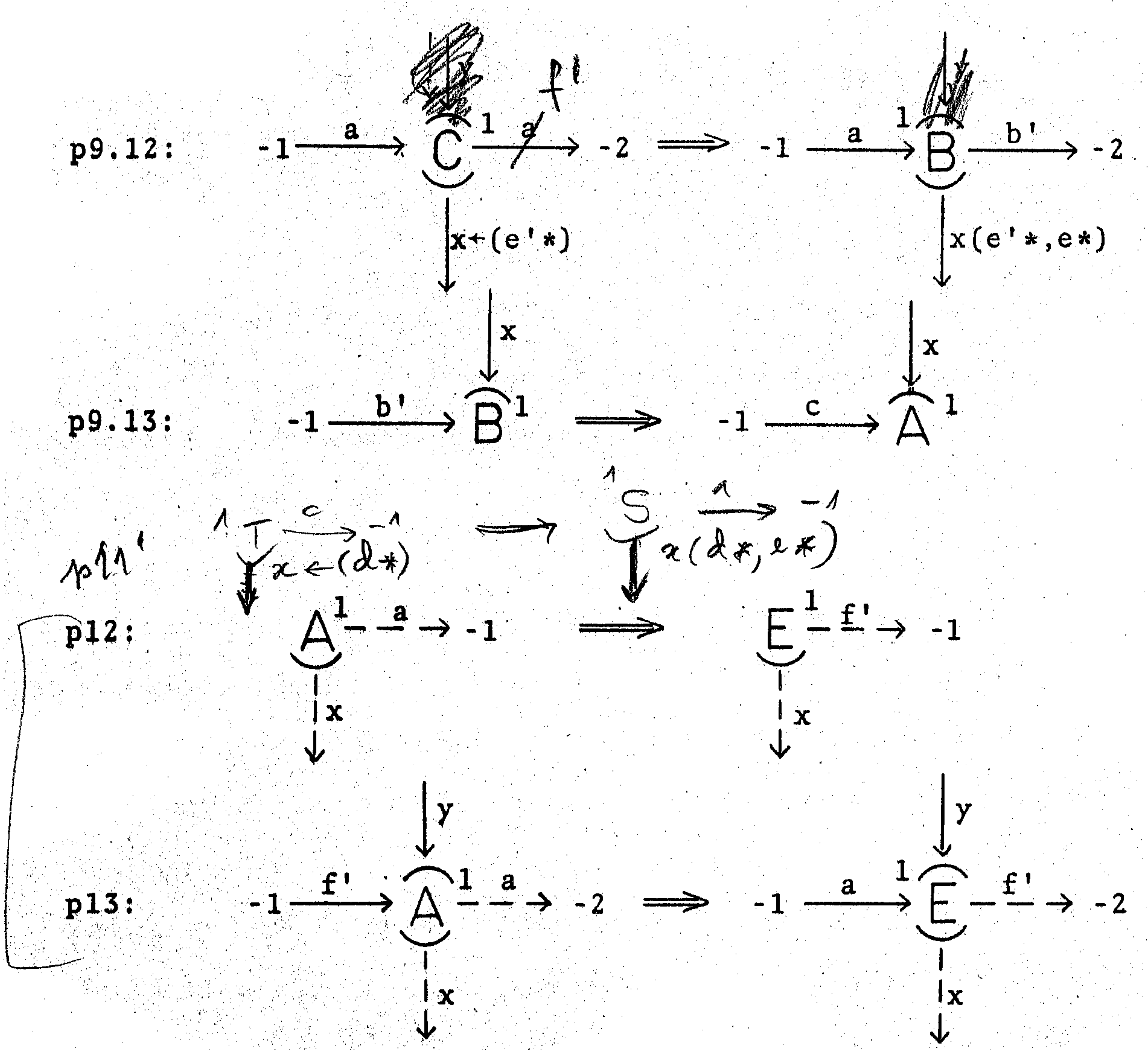

p9.12:
-1
a
C
1
a
f'
-2
-1
a
1
B
b'
-2
x←(e'*)
x(e'*,e*)
x
x
p9.13:
-1
b'
B
1
-1
c
A
1
p11'
1
T
c
-1
x←(d*)
1
S
1
-1
x(d*,e*)
p12:
A
1
a
-1
E
1
f'
-1
x
x
y
y
p13:
-1
f'
A
1
a
-2
-1
a
1
E
f'
-2
x
x

FIG. 19.

deleted when p9.1 is applied. It follows from lemma 1 that it suffices to show that the two conditions specified there are in fact satisfied for g .

Production p9' handles the special case where a lattice with only two nodes has been generated so far. It merely tests whether there is an e1 edge from the first node to the second, in which case the two conditions are obviously satisfied. After the application of p9' , p1 or p3 can be applied.

Productions p9", p9.1 - p9.13 test that the newly created node, labelled T , and any other node with no incoming ei-labelled edges have a unique glb . This is accomplished as follows: Productions p9" , p9.1 - p9.3 search the chain looking for the first (next) node with no incoming ei-labelled edges. When one is found, it is labelled C . Production p9.4 ensures that nodes below C in the partial order defined by ei-labelled edges can be reached through paths of e'i-labelled edges, in the same way nodes below the newly created node can be reached through paths of di-labelled edges. The first node with incoming e'i- and di-labelled edges is the glb of C and the newly created node and production p9.6 renames its di-labelled edges to d'i-labelled ones. Productions p9.7 - p9.9 guarantee that every other lower bound (i.e., every other node which is found to have incoming di- and e'i-labelled edges) is below the glb already found (i.e., has an incoming d'i-labelled edge).

causes a return to m, which is followed by the

Eventually the end of the chain is reached and productions p9.10 - p9.12 search for the next node down the chain with no incoming ei- or di-labelled edges (from p9.2). This search process is complete when production p9.13 is applicable and it opens the door for the application of p10 of GHASSE and p11'. ~~and p11 of G_{Hasse}~~ .

Thus according to lemma 2 and the preceding discussion any two nodes of a derived graph have a unique glb .

Productions p12 - p13 are only applicable once the first node of the chain has been labelled A , and they test that every node of the chain, except for the first one has at least one incoming ei-labelled edge. It follows from proposition 1 that any graph generated by $G_{lattice}$ is a lattice.

The converse is easy. We have already argued that G_{Hasse} will generate any Hasse diagram. If the structure satisfies the two conditions of lemma 1, it will be generated by $G_{lattice}$ as well, since it differs from G_{Hasse} only in that it tests for those two conditions.

4. Discussion and Conclusions

There are many areas of computer science where directed labelled graphs are used. Consider, just to name a few examples, flowcharts [19, 20], program schemata [21], Petri nets [22], VDL objects [23] (which are precisely the ordered trees defined in 3.2), syntax graphs [24], several graph theoretic models of data structures [11, 25, 26] and semantic networks [27].

In dealing with any one of the above examples of labelled graph uses, we often wish to

a. describe the meaning of node and/or edge labels (e.g., the node labels for Petri nets indicate "places" or "transitions").

b. establish the restrictions that should hold for the well-formed instances of the class (e.g., for flowcharts of programs every node should be reachable from at least one entry point and should reach at least one termination point).

c. specify subclasses of interest (e.g., the class of flowcharts of structured programs [28]).

d. enumerate the permissible transformations that may map one well-formed instance into another, within the same class or subclass (e.g., the syntax graph reductions).

The methods that have been used to accomplish these tasks fall roughly into three categories, informal, axiomatic and constructive. Since the emphasis in computer science has been shifting toward formal specifications of algorithms and structures, we expect that axiomatic and constructive methods will become the only real alternatives. Axiomatic methods can be used quite successfully to provide solutions to tasks b and c but not d . An axiomatic definition of lattices is much cleaner and easier to understand than that given in terms of $G_{lattice}$. Unfortunately, however, axiomatic definitions are not constructive in that they do not specify what types of operations preserve the properties of a class of labelled graphs. It is in this respect that constructive definitions, which include graph grammar formalisms, are superior. Moreover, it is not always the case that axiomatic definitions are cleaner and easier to understand. To illustrate this point, consider the definition of threaded binary trees [29] in terms of the RLD grammar $G_{\text{threaded-trees}}$ defined by

$$(\{S, H, A\}, \{H, A\}, \{a, b, c, d\}, P, S)$$

where the productions in P are given in FIG. 20. An element of the set defined by this grammar is given in FIG. 21. c- and d-labelled edges indicate the predecessor and the successor of a node in an in-order traversal (traverse left subtree, visit root, traverse right subtree).

The concepts of "predecessor" and "successor" could certainly be expressed by using some axiomatic approach [30]. However, such an approach would probably fail to convey the "constructive" information that we have incorporated in the simple grammar above, namely that "when a left son is created, it becomes the leftmost node in a subtree and as such it inherits the c-thread from its parent, and points to the parent with a d-thread". An analogous statement holds for the addition of a right son.

It is also important to note that the grammar not only permits the creation of any threaded binary tree but also restricts the valid operations to the addition of left or right sons. Deletions could be permitted by expanding the grammar,

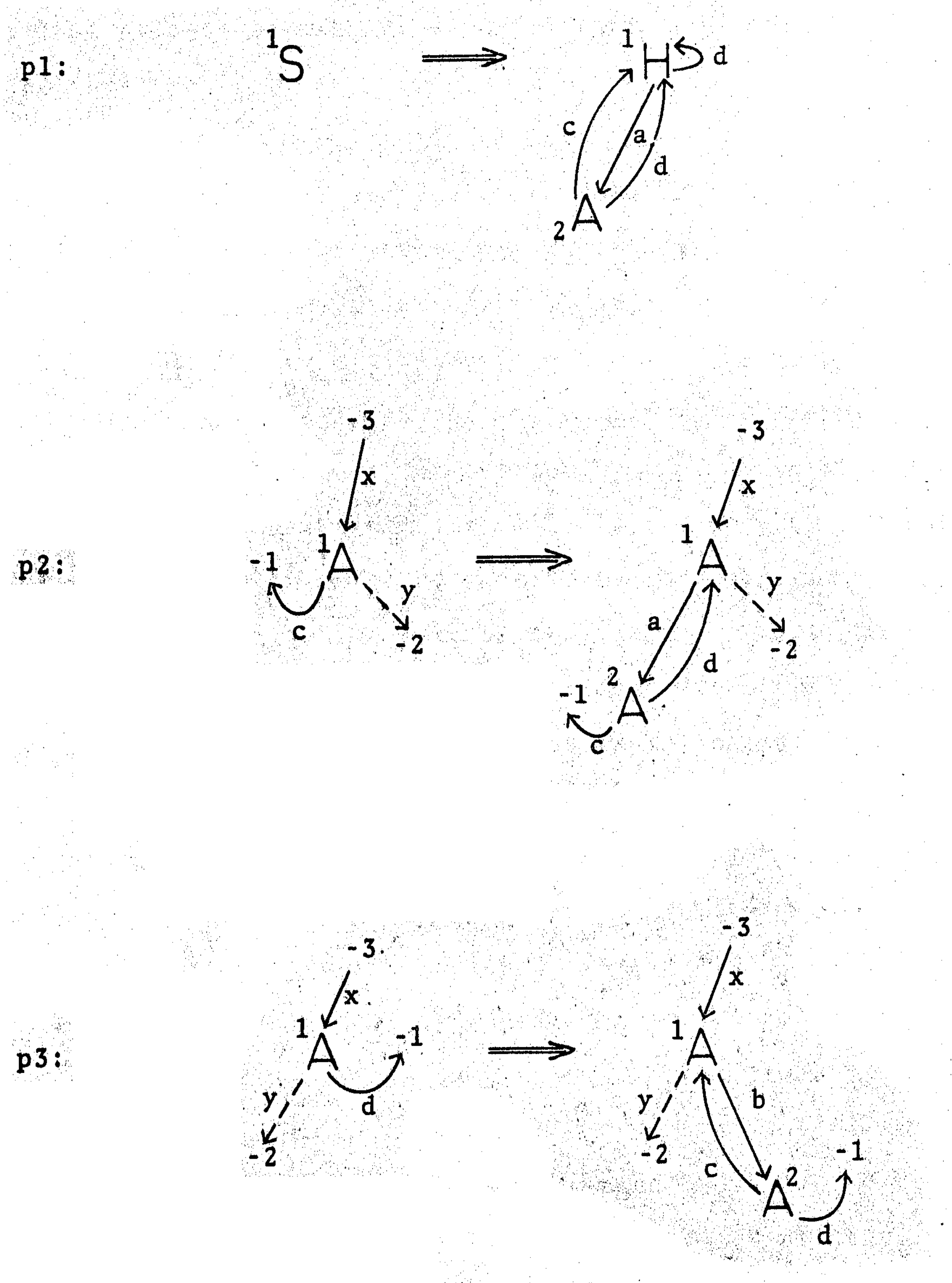

p1:
1
S
1
H
d
c
a
d
2
A
p2:
-3
x
1
A
-1
c
y
-2
-3
x
1
A
y
a
d
-2
2
-1
A
c
p3:
-3
x
1
A
-1
d
y
-2
-3
x
1
A
y
b
-2
c
A
2
-1
d

FIG. 20.

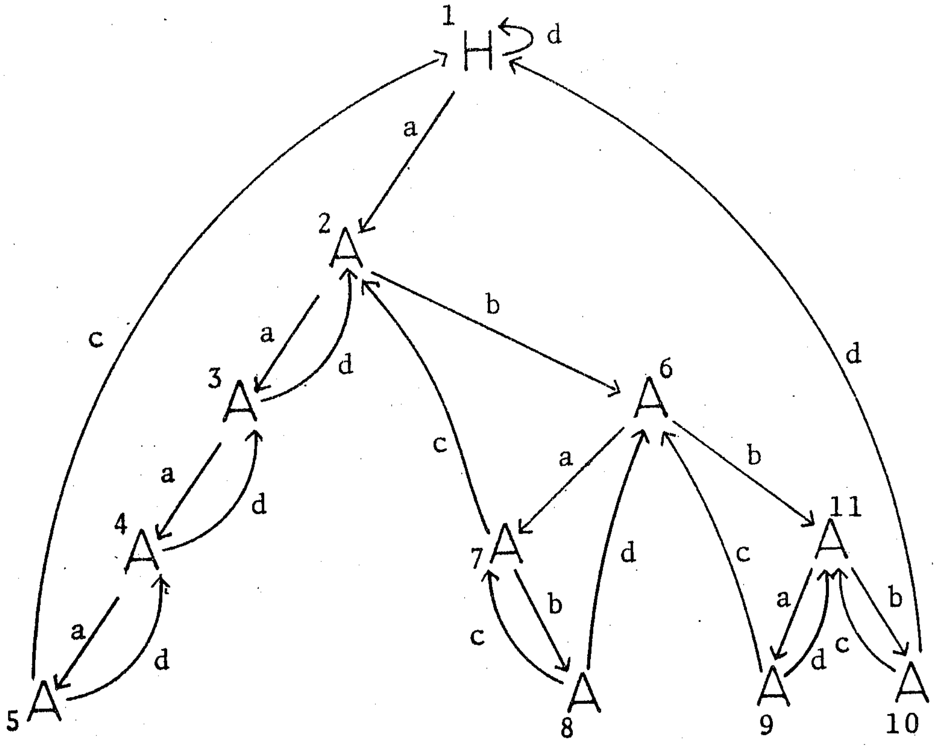


FIG. 21.

as illustrated in section 2, so as to include the inverses of productions p2 and p3 .

Proofs of program correctness can be greatly facilitated by definitions of classes of structures (labelled graphs) which restrict the permissible operations on the structures to those contained in the (constructive) definition of these structures. With this provision it would suffice to prove once and for all that the allowed operations in fact preserve the properties of the class, as understood by the person who defined it. This is not a new claim (see [31, 32]; also, for a treatment of data structures in terms of transformations see [11, 15]). The point we have tried to make in this paper is that graph grammars can serve as formalisms for constructive definitions, provided they are not viewed merely as extensions of phrase structure grammars, but rather as means of expression.

Much of the early interest in graph grammars arose from the belief that they could be used to define classes of real-life objects which are inherently multi-dimensional (the class of all pictorial drawings depicting series-parallel networks, houses, etc.). It is the authors' belief that research on graph grammars should concentrate on applications in areas of computer science such as those mentioned above, which deal with classes of graphs susceptible to a formal (syntactic) definition. This orientation can be much more rewarding than any attempt to use graph grammars for the definition of real-life concepts.

Appendix

We prove here the two lemmas used in section 3.

Lemma 1: A Hasse diagram is a lattice if and only if any two of its nodes have a unique glb and there is only one node with in-degree 0.

Proof: Every lattice clearly satisfies the two conditions. Consider now a Hasse diagram which satisfies the conditions. First of all, the unique node with in-degree 0 must be greater than all other nodes since the nodes are partially ordered. It follows that every two nodes have an upper bound, the unique node with in-degree 0, and therefore at least one lub. Now suppose that i, j have two lubs n, m. Then it is easy to show that n, m have two glbs i', j' thus contradicting one of the two conditions. This completes the proof of the lemma.

Lemma 2: Suppose that a partial structure has been generated by $G_{lattice}$ (it need not be a lattice) such that any two nodes have a unique glb and a new node n is added to this structure. Let $i_1, i_2, \ldots, i_k$ be the nodes of the structure with no incoming ei-labelled edges. Then n and any other node m will have a unique glb if and only if n and i_j, $1 \leq j \leq k$, have a unique glb.

Proof: We assume that n and i_j, $1 \leq j \leq k$, have uniuqe glbs and we will show that the same is true for n and any other node m . The node m must be less than at least one of the nodes i_j , $1 \leq j \leq k$, or be one of them. In the latter case there is nothing to prove, so assume that $m < i_j$. Then clearly $i_j \wedge m = m$, $n \wedge m = n \wedge (i_j \wedge m) = (n \wedge i_j) \wedge m$. Let $n \wedge i_j = p$, so that $n \wedge m = p \wedge m$.

The node p cannot be n since newly created nodes are never below already existing ones (this is true for RLDs generated by G_{lattice} as well as G_{Hasse}). Thus .p must be below n and i_j . Comparing p to m , we can have

(i) $p \geq m$ - implies $n \wedge m = m$ and therefore n and m have a unique glb .

(ii) $p \leq m$ - implies $n \wedge m = p$ and since n and i_j have a unique glb , so does n and m .

(iii) p and m are incomparable - then $n \wedge m = p \wedge m$ and since p was created earlier, we already know that p and m have a unique glb .

This proves lemma 2 in one direction. The converse is trivial.

REFERENCES

[1] Pfaltz, J. and Rosenfeld, A. - "Web Grammars", T.R. 69-84. University of Maryland, 1969.

[2] Shaw, A. - "Parsing of Graph-Representable Pictures", J.A.C.M. 17, 3, 1970, pp. 453-481.

[3] Feder, J. - "Linguistic Specification and Analysis of Classes of Line Patterns", T.R. 403-2. New York University, 1969.

[4] Pavlidis, T. - "Linear and Context-Free Graph Grammars", J.A.C.M. 19, 1, 1972, pp. 11-22.

[5] Mylopoulos, J. - "On the Relation of Graph Grammars and Graph Automata", Proceedings of the 13th SWAT;1972, pp. 108-120.

[6] Ehrig, H., Pfender, M. and Schneider, H. - "Graph Grammars, an Algebraic Approach", Proceedings of the 14th SWAT, 1973, pp. 167-180.

[7] Abe, N., Mizumoto, M., Toyoda, J., and Tanaka, K. - "Web Grammars and Several Graphs", JCSS 7, 1973, pp. 37-65.

[8] Montanari, U. - "Separable Graphs and Web Grammars", Information and Control 16, 1970, pp. 243-267.

[9] Milgram, D. - "Web Automata", T.R. 271. University of Maryland, 1973.

[10] Rosenfeld, A. - "Progress in Picture Processing: 1969 - 1971", ACM Computing Surveys. 5. 2. 1973, pp. 81-108.

[11] Earley, J. - "Toward an Understanding of Data Structures", CACM, 14, 10, 1971, pp. 617-627.

[12] Mercer, A. - "An Array Grammar Programming System", T.R. 180. University of Maryland, 1972.

[13] Harary, F. - "Graph Theory", Addison Wesley, 1969.

[14] Furtado, A. - Ph.D. thesis, University of Toronto, 1974.

[15] Gotlieb, C.C. and Furtado, A. - "Data Schemata based on Directed Graphs", to appear.

[16] Hopcroft, J. and Ullman, J. - "Formal Languages and their Relation to Automata", Addison-Wesley, 1969.

[17] Moon, J. - "Topics on Tournaments", Holt, Rinehart and Winston, 1968.

[18] Birkhoff, G. - "Lattice Theory", Am. Math. Soc., 1968.

[19] Chapin, N. - "Flowcharting with the ANSI Standard: a Tutorial", ACM Computing Surveys, 2, 2, 1970, pp. 119-146.

[20] Berztiss, A. - "Data Structures: Theory and Practice", Academic Press, 1971.

[21] Manna, Z. - "Program Schemas" in "Currents in the Theory of Computing", Aho (Ed.), Prentice-Hall, 1973.

[22] Petri, C.A. - "Concepts of Net Theory", Proceedings of the Symposium on Mathematical Foundations of Computer Science, 1973, pp. 137-146.

[23] Wegner, P. - "The Vienna Definition Language", ACM Computing Surveys, 4, 1, 1972, pp. 5-63.

[24] Cohen, D., and Gotlieb, C.C. - "A List Structure Form of Grammars for Syntactic Analysis", ACM Computing Surveys, 2, 1, 1970, pp. 65-82.

[25] Rosenberg, A. - "Data Graphs and Addressing Schemes", JCSS, V, 6, 1971, pp. 193-238.

[26] Bachman, C. - "Data Structure Diagrams", Data Base - ACM/SIGBDP 1, 2, 1969.

[27] Simmons, R.F. - "Semantic Networks: Their Computation and Use for Understanding English Sentences", in "Computer Models of Thought and Language", Schank and Colby (Eds.), W.H. Freeman and Company, 1973.

[28] Mills, H. - "Mathematical Foundations for Structured Programming", IBM doc. FSC 72-6012 (1972).

[29] Knuth, D. - "The Art of Computer Programming", vol. 1, Addison-Wesley, 1973.

[30] Hoare, C. - "Notes on Data Structuring" in "Structured Programming", Dahl and Dijkstra (co-authors), Academic Press, 1972.

[31] Dahl, O. - "Hierarchical Program Structures" in "Structured Programming", Hoare and Dijkstra (co-authors), Academic Press, 1972.

[32] Liskov, B. and Zilles, S. - "Programming with Abstract Data Types", Proceedings of a Symposium on Very High Level Languages, ACM/SIGPLAN, 1974, pp. 50-59.

[33] Standish, T. - "A Data Definition Facility for Programming Languages", Ph.D. thesis, Carnegie Institute of Technology, 1971.