PUC

Série: Monografias em Ciência da Computação
Nº 15/77

DATA SCHEMATA BASED ON DIRECTED GRAPHS

by

C.C. Gotlieb
A.L. Furtado

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225 — ZC 19
Rio de Janeiro — Brasil

Série: Monografias em Ciência da Computação
Nº 15/77

DATA SCHEMATA BASED ON DIRECTED GRAPHS*

by

C.C. Gotlieb**
A.L. Furtado

Editor: Michael F. Challis September, 1977

* This report has been accepted for publication elsewhere. As a courtesy to the publisher it should not be widely distributed until after the date of outside publication.

** Department of Computer Science, University of Toronto.

For copies contact:

Rosane T.L. Castilho
Head, Setor de Documentação e Informação
Depto. de Informática - PUC/RJ
Rua Marques de São Vicente, 209 - Gávea
20.000 - Rio de Janeiro - RJ - Brasil

RESUMO

Grafos dirigidos, rotulados com raiz (RLD's) são tomados como base para descrever estruturas de dados. Um formalismo construtivo é estabelecido para descrever RLD's, gerá-los por meio de gramáticas, e executar operações tais como o acesso a nodos , inserção e deleção de itens de dados e reconhecimento de padrões em grafos. Mostra-se como o formalismo pode ser traduzido em linguagens para definição e manipulação de dados, do tipo associado a sistemas de banco de dados. O formalismo possibilita um desenvolvimento sistemático de tais linguagens e provê um método para incorporar precondições de consistencia às operações estruturais e para provar a correção das estruturas de dados que se apresentam ao serem usadas tais linguagens.

PALAVRAS CHAVES:

Estruturas de dados, grafos, digrafos, grafos dirigidos , gramáticas de grafos, sistemas de bancos de dados, linguagens de manipulação de dados, linguagens de definição de dados, esquema.

ABSTRACT:

Rooted, labelled, directed graphs (RLD's) are taken as the basis for describing data structures. A constructive formalism is set up to describe RLD's, generate them by means of grammars, and carry out such operations as accessing nodes, inserting and deleting data items, and recognizing graph patterns. It is shown how the formalism can be translated into languages for data definition and data manipulation, of the type associated with data base systems. The availability of the formalism allows a systematic development of such languages, and provides a method of incorporating consistency preconditions to structural operations and proving correctness of the data structures which arise in using such languages.

KEY WORDS:

Data structures, graphs, digraphs, directed graphs, graph grammars, data base systems, data manipulation languages, data definition languages, schema.

CONTENTS

1. Introduction

The concept of *schema* was proposed in a CODASYL report [1], where it was defined as "a description or definition of a set of structures of a given type in terms of a certain subset of the attributes for that type".

The report also employed the phrase *instance of a schema*, to signify a particular member of the set of structures denoted by the schema.

A more formal view of these concepts is a worthwhile goal for theoretical research in the closely related areas of data structures and data management. In fact a great deal of work has been done on defining models where there is a precise characterization of sets of data structures. These models have mainly been based on graph theoretic [2, 3], axiomatic [4, 5], and relational [6] approaches.

Such models are especially valuable if they are constructive, in the sense of containing a mechanism for generating instances of the defined sets. The same mechanism would be used for transforming a valid instance of a set into another valid instance of the same set, essentially through insertions and deletions, which are fundamental operations on data structures. Models with this constructive feature have immediate interest, in that they lead to a data definition facility, DDF [7], or data definition language, DDL [8, 9], which is proposed as a desirable component of a general data base management system, DBMS [1].

In this paper we present a constructive formalism, where data structures are viewed as directed graphs.

There are three distinct parts to the work. In the first (§2) the formalism is presented. A set of data structures is denoted by a graph grammar [10, 11, 12, 13], and the generation of instances of a set and their transformation into other instances are formally described by some appropriate application of production rules from the grammar. In the second part of the paper (§3) examples of the formalism are presented. These examples, which are of increasing complexity, illustrate that graph grammars can be regarded as a generic device for producing data types. Although no characterization of the data types produced by graph grammar has been attempted, it has been possible to produce grammars for all the commonly accepted types, as well as for useful types for which no other generating device has been reported. In the third part of the paper (§4) it is shown how a programming language which allows one to represent and use graph grammars might be defined , and some of the implementation features of such a language are discussed.

2. The Formalism

2.1 Definition of Rooted Labelled Digraphs

Familiarity with the basic definitions of graph theory [14] and formal languages is assumed [15].

The graphs of interest for this work are *rooted, labelled*

digraphs (RLDs, or simply graphs), defined by the 6-tuple $(N, NL, EL, \nu, \delta, r)$:

- N - non-empty finite set of nodes, denoted by positive integers; an element of N is called a node reference;
- NL - non-empty finite set of node labels;
- EL - finite set of edge labels;
- ν - total function which assigns a node label to every node of the RLD

$$\nu : N \rightarrow NL ;$$

- δ - partial function which describes the adjacency structure of the RLD and associates an edge label to every edge

$$\delta : N \times EL \rightarrow N$$

- r - 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 each i, $1 \leq i < k$, there exists an edge label $e \in EL$ such that $\delta(n_i, e) = n_{i+1}$.

The node r is called the *root* of the RLD and it is clear that any other node can be reached through a path that begins at the root†. Since δ is a function, no two edges

† The definition of RLDs is similar to that of V-graphs [2], but a major difference is that rootedness is not required for V-graphs.

with the same label can leave the same node.

An RLD can be reduced to a canonical form by choosing some systematic traversal scheme to be initiated from the root. One possible scheme is depth – first search. Initially, the root is *visited*, which means that it is assigned node reference 1 and its outgoing edges are sorted according to the lexicographic order of their labels. In general, after visiting a node – let i be its new node reference – the traversal algorithm takes iteratively its sorted outgoing edges; the first unvisited node (if any) thus obtained causes the traversal process to be re-entered recursively whereby the node is visited (being assigned node reference $i+1$), the first of its sorted edges is considered, and so on. When the possibly several levels of recursive calls terminate the iterative exploration of edges pointing from i is resumed. The process continues until the entire RLD has been traversed and all its nodes have been renumbered.

The complexity of the reduction algorithm described above is dominated by the complexity of sorting the edge labels. Let p be the number of nodes, q the number of edges, and v the number of edge labels; the time for sorting the edge labels is roughly $O(p \cdot v \log v)$, whereas the time for performing the depth-first search is $O(\max(p, q))$, and $q \leq p \cdot v$ in an RLD. The ability to reduce an RLD to a canonical form in time proportional to a very small polynomial is the justification for the claim that the process of generating

instances of sets of RLDs, and of performing a very general search process (points to be discussed later in this section) can be executed efficiently.

2.2 Domains and Functions

While node references are unique denotations for the nodes, the same label can be given to several nodes. A node label represents a *domain*, here a set of elements of one of the following kinds:

a. Simple elements — These are indecomposable elements with which *values* may be associated. A simple element is a basic item of information, e.g. a supplier number. The computer representation of the associated values demands that their characteristics, such as precision, or length, etc., be indicated when describing a domain of simple elements.

b. Composite elements — These are elements of some Cartesian product of other domains. The elements of such component domains are represented as nodes in the same RLD, and may be regarded as a decomposition "in breadth" of the composite element. The role of composite elements will be explained when discussing the representation of functions.

c. Structured elements — These are elements with

which a *lower level description* may be associated. The lower level description, a decomposition "in depth", takes the form of another (internal) RLD whose root is referenced by the structured node. Structured nodes provide a convenient representation for hierarchies of data structures, such as CODASYL's generic structure types [1] — item, group, group relation, entry, file, data base —, and the several levels of description in artificial intelligence languages.†

The edge labels of RLDs denote *functions* (as might be expected from the requirement that outgoing edge labels be distinct).

The representation of a function with simple domain and range is obvious. If the domain, or the range, or both are composite, the use of projection functions provide a suitable representation. An example is shown in Fig.1a, taking a function $F : K \rightarrow L$, where $K = A \times B \times C$ and $L = D \times E$ (fig. 1).

An n-ary relation R, among whose domains or

---

† Also, at each level, one may have the same or a different class of data structures, which implies, in the terminology to be introduced in section 2.4, that the same or different grammars will be used at each level. Thus, one may have trees of arrays of stacks, recursive structures such as lists of lists ... of lists, etc.

combinations of domains no function has been defined† can be conveniently represented in RLD form by its characteristic function F_R . Thus if $R \subseteq A \times B \times C$, then R induces $F_R : A \times B \times C \rightarrow (T, F)$, where the image of an element from $A \times B \times C$ is T ("true") iff it stands in relation R . An example is given where the same element from domain A is related with two different elements from $B \times C$; a composite domain $K = A \times B \times C$ is employed, and the "true" range element is implied (Fig. 1b).

Unary relations are represented by loops.

Another important class of functions are successor functions, whose purpose is to order elements belonging to the same domain according to some specified criterion. If more than one criterion is used several different successor functions can be defined on the same domain.

From the definition of RLDs one can see that cycles, and multiple edges between pairs of nodes are permitted. Cycles are present even in some rather simple data structures, such as rings and threaded trees*, and the multiplicity of edges between two nodes will reflect the fact that two elements may be related in more than one way.

---

† In the terminology of relational data bases, R is an n-ary relation whose key involves all the n domains.

* In [31] a formal characterization of threaded binary trees is provided, together with a corresponding grammar; see also [29].

2.3 Access Mechanisms

Besides providing a representation for functions, the labelled edges of RLDs represent access mechanisms to data, as will now be shown.

To access a node means that the node has been effectively located and all information related to it has become available. Such information may include:

a. the node reference;

b. the node label;

c. the sets of pairs (edge label, node reference) running into or from the node; following [16] we call SIP the set of inpointing pairs, and SOP the set of outpointing pairs;

d. the value or values associated with the node;

e. the root of the internal RLD in the case of structured nodes.

In some cases the node reference is known and we can have a "direct" access to the node by providing a directory.

The uniqueness of outgoing edge labels provides another access mechanism: given the reference of some node (in general the root) and a sequence of possibly repeated edge labels, a node is unambiguously determined. If the RLD is no more complex than a chain or a tree there will be exactly one such sequence for each node, and some models have taken

advantage of this fact. A sequence of edge labels can be viewed as a functional composition.

For simple nodes another kind of access may be needed, where a value is given, and the node (or nodes) possessing the value must be located. This is tantamount to taking the inverse of the function which maps a simple node into its value. Systems that provide this kind of access are usually said to be "associative" [17, 18]. Associativity may be implemented by software, for example by inverted lists [19], or by special hardware [20]

A fourth and more complex kind of access will now be discussed, under the denomination of pattern-directed search [21, 22]. For certain applications it may be the case that, instead of a single node, a number of *related* nodes should be located, and that for some of these nodes the values are given. In effect what is sought is a subgraph of the RLD; of special interest is the situation where the nodes lie in some arbitrary *rooted* sub-graph (sub-RLD) for in these cases the search will be particularly efficient. In addition to the edges belonging to the sub-RLD, the incoming and outgoing cutset edges may be of interest, which together with the sub-RLD, will be termed a *scion*. A scion considered independently of any RLD will be called a *scion pattern*. A scion is said to *match* a scion pattern if they are identical up to the assignment of node

references†. A search for scions in an RLD matching a scion pattern is conducted as follows:

(1) A depth-first search is started at the root of the RLD; as each node, $\underline{n}$, is visited its node label is compared with that of the root of the scion.

(2) If this phase of the match succeeds, an attempt is made to traverse the sub-RLD part of the scion pattern "in parallel" with a corresponding traversal along the RLD, starting from $\underline{n}$. Note that, due to the RLD properties, either there exists one matching sub-RLD rooted at $\underline{n}$ or there is none.

(3) If this second phase of the match succeeds, the cutset edges are considered by inspecting the SIPs or SOPs of the located sub-RLD nodes.

(4) If this third phase of the match also succeeds, the values of those sub-RLD nodes that should possess some indicated value are retrieved and compared with the sought values. If this comparison again succeeds the process terminates (or is resumed at step (1) if one wishes to locate

---

† cf. the concept of scion isomorphism in [23]. Also, note that, as to be described at section 3, item d, a simple extension permits the characterization of certain sub-sets of cutset (edge label, node reference) pairs in scion patterns.

other possible matching scions rooted elsewhere in the RLD); otherwise the process is resumed at step (1) until all nodes have been visited.

An example of the matching process for phases (1), (2) and (3) is given with the RLD and the scion pattern of Fig. 2. The node references in the scion pattern should be understood as parameters; they are denoted by Greek letters, and prefixed with a minus sign in case of outside cutset nodes.

Node 5 will eventually be reached at phase (1), and since it has the same label as node $\underline{\alpha}$ the process continues.

At phase (2) the following node correspondence is established for the sub-RLD nodes:

$$\alpha \text{ ——— } 5$$

$$\beta \text{ ——— } 2$$

$$\gamma \text{ ——— } 38$$

At phase (3) one outside cutset node is immediately located, because there are cutset edges running into it:

$$-\alpha \text{ ——— } 21$$

The other outside cutset nodes are then ordered lexicographically with respect to their ordered sets of outgoing edge labels. The same is done for the scion pattern. The two resulting tables below give, for each node, their corresponding ordered sets of outgoing (edge label, node reference) cutset pairs:

in the RLD			in the scion pattern		
15	———	((a , 2))	-β	———	((a , β)
19	———	((a , 2))	-γ	———	((a , β))
40	———	((a , 2) , (b , 38))	-δ	———	((a , β) , (b , γ))
78	———	((a , 38))	-ε	———	((a , γ))

Given the node correspondence already established for α , β , and , the "lists" for nodes in corresponding rows of the two tables become identical. Thus it is possible to establish the correspondence:

-δ ——— 40

-ε ——— 78

Two possible choices are acceptable for the remaining nodes:

-β ——— 15 or -β ——— 19

-γ ——— 9 -γ ——— 15

which means that the two nodes are regarded as equivalent, and an arbitrary decision can be made.

The complexity of testing for a sub-RLD rooted at an indicated node is clearly the same as the complexity of reducing an RLD to a canonical form. Thus it is easy to see that the overall complexity of the pattern-directed search restricted to scions is $O(p^2 \cdot v \log v)$.

2.4. Grammars

A matching process is also involved in the graph grammar formalism that is proposed here for generating and transforming RLDs. Formally, an RLD - grammar (graph grammar, or simply grammar) is a 5 - tuple (V_N, V_T, V_{EL}, P, S) where:

- V_N - finite non-empty set of non-terminal node labels;
- V_T - 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} - denumerable non-empty set of edge labels; edge labels may have positive subscripts;
- P - finite non-empty set of production rules (or productions);
- S - starting non-terminal symbol, $S \in V_N$.

According to this definition, only node labels can play the non-terminal or terminal role that symbols play in phrase structure grammars. Moreover, a node label can be both terminal and non-terminal, which makes it possible to consider derived structures of an RLD - grammar both as RLDs belonging

to the defined set, and as intermediate structures from which other RLDs from the same set can be derived.

The set P consists of ordered pairs (LHS, RHS), to be represented here as LHS $\Longrightarrow$ RHS , where LHS and RHS are scion patterns.

If R and R' are RLDs, R' is said to be *directly derivable* from R if

a. there exists in R a scion $\widehat{\text{LHS}}$ that matches a scion pattern LHS ;

b. there exists in R' a scion $\widehat{\text{RHS}}$ that matches a scion pattern RHS ;

c. R' is identical to R up to the substitution of $\widehat{\text{RHS}}$ for $\widehat{\text{LHS}}$ in R ;

d. LHS $\Longrightarrow$ RHS $\in$ P .

An RLD R' is said to be *derivable* (or can be generated) from an RLD R if there is a sequence of RLDs $R_1 , R_2 , \ldots , R_k$, such that:

a. $R_1 = R$;

b. $R_k = R'$;

c. R_{i+1} is directly derivable from R_i for $1 \leq i < k$.

There may be more than one scion in R matching an

LHS scion pattern. In the practical situation of handling data structures, one might want to specify which scion would be compared to the LHS. This can be accomplished by indicating the position in R of the root n of the intended scion; thus, to apply a production p from a grammar G to an RLD R at a node n of R means:

(a) To check whether there exists in R, rooted at n, a scion which matches the LHS scion pattern in p.

(b) If so, to replace the matching scion by a scion created as indicated by the RHS scion pattern in p.

Note that the matching phase will again be executable in a time proportional to $O(p \cdot v \log v)$.

When performing the replacement, the correspondence of nodes and edges between the scion pattern and the matching scion, established in step (a), will be used. In particular, the identification of the cutset part of the LHS is used for detaching the matching scion and embedding the newly created one.

Besides being used for defining the embedding, the cutset may impose additional positive or negative conditions for the applicability of a production [24]. The formalism allows the specification of edge labels of cutset edges and a reference to the endpoints not in the scion (outside nodes);

it does not allow the specification of labels of outside nodes.

It is also possible to require that there be *no* incoming or outgoing cutset edges incident to specified scion nodes. The absence of such edges, the prohibition of edges with indicated labels, and the implicit requirement that there be no cutset elements which are not specified constitute *negative conditions* related to the cutset.

Although it is possible to define Chomsky-like hierarchies of graph grammars (cf. [12] for example), this will not be done here, since this is of interest mainly when parsing is being considered, rather than generation which is the emphasis here. However it is interesting to note that by using context-sensitive productions, where more than one node appears on the LHS and cutset conditions may be imposed, transformations on elements of data structures become dependent upon the *context* surrounding the elements, i.e. upon *local conditions*.

The consideration of *global* conditions, i.e. requirements that should be satisfied by the entire structure, is usually harder to incorporate in a grammar. Sometimes one can take advantage of some convenient "constructive" property. For example, if the intended structures should have the (global) requirement of unilateral connectedness, the fact that a graph is unilaterally connected iff it possesses a spanning walk [14, page 199] could be used when designing the grammar. Also it may be the case that a structure will have to be

generated in a certain order to ensure that the global requirement be satisfied.

An expedient way of testing for global requirements lies in the use of auxiliary nodes and edges, which do not strictly belong to the intended structure and in some cases may be deleted if no further transformations are to be performed (see example D in the next section). Another way is to construct the grammar so as to simulate the behaviour of some algorithm, cf. the grammar for acyclic graphs that simulates a depth-first search algorithm in [23]. Still another way is to simulate some graph automaton [13], a device which is especially useful when it can be constructed to traverse part or perhaps all of a structure, but the traversal fails if the global requirement ceases to hold.

As indicated before, a production p should be applied *at* a certain node n, in the sense that n should correspond to the root of the LHS of p. But n should first be located, conceivably by one of the access mechanisms described earlier. As a consequence, further non-grammatical conditions for the application of a production may be imposed, particularly those related to the values associated with certain nodes. Also, the formalism allows the specification of "breadth" and "depth" requirements, i.e. bounds to the indegree or outdegree of certain nodes and to the length of certain paths (for an example of the former requirement see section 3; the latter is discussed in [23]).

The ability to impose all such conditions provides a mechanism for ensuring *consistency*, a matter of primary concern in handling data bases (for a simple illustration, see example E in section 3).

As to the grammatical formalism itself, it is important to recall its double purpose of not only being able to generate valid instances of the intended set but also of making it possible to specify and hence restrict the permissible transformations among these. It will often be the case that fewer productions than those in a given grammar will suffice for generating all the instances of the set; the others only permit other operations needed for the practical use of the data structure. As an example, consider a grammar for stacks; all stacks could be generated by productions that would correspond to a push-down operation, and the inclusion of productions for a pop-up operation does not increase the generative power of the grammar.

The proof that a grammar generates exactly the intended set of data structures, and is in this sense *correct*, can be done by induction

a. on the stage of the derivation, to show that only valid instances are generated;

b. on the size of the instances (usually number of nodes), to show that all valid instances are generated.

If a grammar contains erasing rules, which are often needed for implementing deletion operations and for eliminating auxiliary nodes and edges, it may not be possible to determine whether or not a given instance can be generated. However, in many (perhaps most) cases of interest erasing rules do not increase the generative power of the grammar, which often happens when the erasing rules are the inverses of other rules in the grammar (as with the pop-up operation for stacks); as to the elimination of auxiliary nodes and edges see the concept of "indirect" generation in [24] (the same reference contains good examples of proofs by induction that a graph grammar generates a given set of graphs, taking advantage of certain properties of such graphs).

3. Examples

In this section five examples are presented to illustrate the grammatical formalism.

The first example deals with linear structures — stacks and queues — and is introduced to illustrate the formalism and to show how two grammars can generate the same set (of chains) and yet provide different operations. The second example introduces headed rings, as a case of cyclic structures. The third example introduces square arrays, as a case of non-linear structures. The fourth example introduces a class of complete d-ary trees and illustrates the determination of bounds to the outdegree of a node and the imposition of a global

condition (the chosen balancement criterion). The fifth example introduces an inventory data base and illustrates the imposition of value conditions and how certain consistency requirements can be enforced.

The conventions employed in the diagrams representing the production rules of a grammar are as follows:

a. Node references are denoted by
   - lower case Greek letters, for nodes belonging to the sub-RLD part of scion patterns;
   - lower case Greek letters preceded by a minus sign, for outside cutset nodes·
   - an asterisk, for the root of the RLD to which the production is to be applied, in the special case where this node (whose position is always known) is regarded as an outside cutset node.

   It should be remarked that node references in a scion pattern are parameters, in the sense that when the LHS is matched against an RLD R they will be bound to actual node references in R .

b. Node labels are denoted by capital Latin letters or strings thereof.

c. Edge labels are denoted by lower case Latin letters from the beginning of the alphabet or strings thereof; an edge label may be subscripted, as in e_3 ; the convention e+ on an RHS edge

means that an edge labelled $e_{\underline{i}+1}$ is to be created from some node $\underline{n}$ which has $\underline{i}$ as the highest subscript of any e-labelled edge coming from it.

d. Sets of pairs (edge label, node reference) belonging to the cutset are denoted by lower case Latin letters from the end of the alphabet. The convention $x \leftarrow (e_{\underline{i}}, f)$ on the LHS means that x represents a set involving edges with subscripted labels e and edges with non-subscripted labels f ; $x \leftarrow (\sim e_{\underline{i}}, \sim f)$ means that x represents a set involving edges with any label except those indicated. On the RHS the convention $x[f, g]$ (or $x[e_{\underline{i}}, d_{\underline{j}}]$) means that in the identified set x all occurrences of f will be replaced by g ; in $x[e_{\underline{i}}, d_{\underline{j}}]$ the new $d_{\underline{j}}$ labels will be such that the uniqueness condition of outgoing edge labels will not be violated. The convention #x means the cardinality of the set x .

Some graphical conventions are introduced in Fig. 3.

Example A (Fig. 4)

Grammar G_{STACK} generates stacks. The productions correspond respectively to the push-down and pop-up operations.

Grammar G_{QUEUE} generates queues. Additions are allowed only at the "end" and deletions only at the "front" of the chains.

Example B (Fig. 5)

Grammar G_{RING} generates Headed Rings. The Header is labelled S and the other nodes B . The edges are labelled e except for the edge running into the Header which is labelled a .

Production p1 generates a ring with a single element. Productions p2 and p3 extend the ring by adding one element; p2 is needed for the special case of a ring with only one B - labelled node (in this case p3 could not be applied because $-\alpha$ and $-\beta$ would not correspond to distinct outside cutset nodes).

Example C (Fig. 6)

Grammar G_{SARRAY} generates square arrays.† Production p1 creates a one node square array.

After a square array of order k has been generated, a square array of order k + 1 can be generated by a "bordering" technique: p2 initiates the creation of a column at the upper right corner and p3 continues the process; similarly, p4 initiates the creation of a row at the lower left corner, which is continued by p5 . The only way to eliminate the two non-terminals X and Y is to make them meet at the lower right corner, whereby the bordering process is completed.

The proof of the correctness of the method is not difficult, although it should be noted that at intermediate

† Other kinds of arrays can be similarly handled, the first step being their rendering into an RLD representation; for sparse arrays, for example, see [31].

stages of the generation there may be several Xs and Ys in the structure (but note that only the X at the $\underline{i}^{\underline{th}}$ column can eliminate the Y at the $\underline{i}^{\underline{th}}$ row).

An interesting property of these structures is that, at any stage, if a node can be reached from the root through a sequence of $\underline{i}$ $\underline{a}$ - labelled edges followed by $\underline{j}$ b - labelled edges, then it can also be reached through a sequence of $\underline{j}$ b - labelled edges followed by $\underline{i}$ $\underline{a}$ - labelled edges. This corresponds to the alternative accessing modes for arrays. A "ragged" intermediate configuration is also shown.

Example D (Fig. 7)

Grammar G_{DBTREE} generates $\underline{d}$ - ary trees ($\underline{d}$ being an arbitrary positive integer) which are complete in the sense that a new level $\underline{\ell}$ can be added to the tree only after all nodes at level $\underline{\ell} - 2$ have outdegree $\underline{d}$. Clearly the tree grows in breadth - first order.

Production p1 creates an auxiliary node X, with an (again auxiliary) incoming $\underline{a}$ - labelled edge; p1 will be the first production to be applied and it can be applied only once.

Production p2 creates sons to a node at level $\underline{\ell} - 2$, provided that the node has still fewer than $\underline{d}$ sons; the links to the sons are labelled b_i, and the sons are linked to the auxiliary node through auxiliary edges labelled e.

Production p3 is only applicable to nodes that possess $\underline{d}$ sons. It relabels the auxiliary $\underline{a}$ edge from the node under

consideration to e .

After applying p oductions p2 and p3 a suffic:ent number of tim s a stage ill be reached where:

a. a l nodes at a level less or equal to $\underline{\ell} - 2$ have $\underline{d}$ sons, and the nodes at level $\underline{\ell} - 1$ have nc sons;

b. as a way to ascertain that the above global s tuation obtains, all $\underline{a}$ - labelled edges running into he X - labelled auxiliary node have been relabelled $\underline{e}$.

At this stage p2 and p3 are no longer applicable (no sons can be added), but production p4 which uses the second fact above may be applied to change into $\underline{a}$ all the $\underline{c}$ - labelled edges coming into the X - labelled node from he nodes at level $\underline{\ell} - 1$†. After doing this it again becomes pos ible to apply productions p2 and p to add nodes at lev l $\underline{\ell}$.

Example E (Fig. 8)

Grammar G_I ge erates an nventory d ta base, similar to the one described in [25].

The following domains may e identified in the data base:

S# — supplier number
SN — supplier name
ST — status
CI — city

† Note also the deletion of the $\underline{e}$ edges.

P# – part number

PN – part name

CO – colour

W – weight

I – inventory on hand

O# – order number

QO – quantity ordered

on which the following functions are defined:

fs : S# → X where X = SN × ST × CI

fp : P# → Y where Y = PN × CO × W × I

fsp : Z → {T F} , where Z = S# × P#

fO : Z → O#

fq : O# → QO

In addition, the data base has a Header labelled S that is linked to all elements from domains S# , P# , and Z by edges with subscripted $\underline{a}$, $\underline{b}$ and $\underline{c}$ labels respectively (i.e. S = S#* × P#* × Z* , where, for any domain D , D* means D × D × ••• × D). Here, the subscripted edge labels provide an "array" - like representation for successor functions, which could also be represented by chains; in the present example the ordering criterion is the order of insertion in the data base. In particular, each $\underline{a}_i$ should be regarded as denoting a function

a_i : S → S#

indicating the i^{th} supplier to be added to the data base.

Production p1 adds a supplier with its attendant elements, and production p2 does the same for a part.

The combined effect of productions p3 , p4 , and p5 is to state that a certain supplier is able to supply a certain part. Productions p3 and p4 mark the desired supplier and part (S# and P# nodes with the intended values), by further attaching them to the Header through edges labelled e and g , respectively.† Production p5 then completes the operation, and removes the auxiliary edges.

Production p6 states that an order has been placed for a part to be obtained from one of its suppliers. A value condition for the application of p6 is that the quantity in inventory for the given part must be below some pre-established level; another value condition is that the supplier and the part be the intended ones.

Production p7 states that an order has been filled. In this case there is no value condition on the inventory but after the production is applied the inventory is increased by the quantity ordered.

It is assumed that the inventory is being reduced as parts are used. If this does not involve issuing a formal order, or if one does not wish to record such an order in the

---

† The a_i and b_j edges also linking the Header to these nodes are included in the cutset.

data base, then it is not necessary to add productions for this purpose. Only an operation to change the value of the inventory is needed.

One would expect that a supplier or part should not be represented more than once in the data base. This requirement is not enforceable when productions p1 or p2 are being applied, because they create S# or P# nodes which still do not possess any values. However, before one such value is assigned it should be possible to verify whether some other node already possesses the value, in which case the assignment would not take place.†

The choice of which elements in a data base should be represented uniquely (a choice which in effect determines what the keys are, or on which elements inverted lists are constructed) depends on the application, and in particular on the expected frequency of queries involving the elements. If, for example, queries on the suppliers located in a given city are frequent, then it might be convenient to require that cities should be represented uniquely. Also, other requirements might be imposed, involving changes to G_I, as for example the presence of an alphabetic ordering on the supplier names, etc.

---

† Also, it is to be expected that production p5 would not create more than one Z node adjacent to the same pair (supplier, part), although a supplier may supply parts with different number and a part may be supplied by different suppliers. It is not difficult to define grammatical processes which enforce this requirement.

It is even more important to note that p6 enforces the following consistency requirement: that there is no way to place an order for a part which is not effectively supplied by the indicated supplier (it is ensured by incorporating the *fsp* characteristic function in the LHS). Other consistency requirements might be similarly taken into account in this and in the other operations, either involving additions or deletions to the data base.

Again using p6 as an example, suppose that, by using suitable additional productions, certain P# nodes may be linked through different chains, but except if one of these chains is formed by h - labelled edges (which could be interpreted as a "lock") p6 should still be applicable. This requirement could be introduced by modifying the specification of the P# node in p6 as indicated in Fig. 9.

Further remarks on data base structures

One of the most interesting data base structures is the CODASYL *set* [8]. Consider a record type K with data items of types A,B,C, and a record type L with data items of types D and E. Let F be a CODASYL set with L records as *owners* and K records as *members*.

Since to a member record there corresponds at most one owner record, a function with (composite) domain K and (composite) range L can be defined. This functional interpretation of CODASYL sets, which has been proposed in [37], leads to the representation already shown in Fig. 1a. In a set *occurrence* there can be several K nodes pointing to the same L node, i.e., several members with the same owner.†

A number of rules govern the manipulation of CODASYL sets. As a brief illustration one variety of delete operation will be discussed within the present formalism.

One wishes to delete a node which is an owner in a set F where membership is *mandatory*, and in a set G where membership is *optional*. According to the representation adopted here this means that there are edges labelled f and g running into the node.

The following requirements are imposed for the deletion: (a) mandatory members, and optional members not participating in other set occurrences must also be deleted; (b) other optional

† CODASYL set occurrences are often represented, at a "lower" level, by headed rings; a grammar for these has been presented before.

members must simply be removed from the set (cf. the *selective* format of the delete command in [8]).

There are many ways to implement these requirements using graph grammars. One particularly simple way is to impose the negative applicability condition that there must be no incoming f or g - labelled edges to the node to be deleted. It is assumed that before one tries to apply the delete production rule, other production rules are applied to delete the nodes in requirement (a), which implies the removal of the attached f and g - labelled edges.

According to (b), optional members participating in other set occurrences need not be deleted. In order to handle this situation a remove production rule is added for such nodes; it destroys their outgoing g edge, on the condition that there are other outgoing edges denoting set - membership.

Thus the delete operation involves preliminary delete and remove operations. Item a in the Conclusions mentions the related notion of "macro" productions.

Trees are other common data base structures, being used with the hierarchical model. As seeen, they are easily handled by graph grammars.

4. Outline of a DDF

Data Definition Facilities, DDFs, are a characteristic feature of extensible languages [26]. Also a number of features are present in some widely known languages, such as PL/I and ALGOL 68, to facilitate data definition.

In the following outline of a DDF, based on the formalism just introduced, it is assumed that some such language would act as a *host language*, to which the DDF is to be added. In particular, the existence of primitive data types, such as integer, real, Boolean, character, and pointer, is assumed, together with the ability to declare in the host language, certain basic attributes pertaining to them (as, for example, the precision of an integer).

With this understanding it is possible to define a new data type by a *schema* statement, which specifies:

a. The domains, denoted by node labels, together with

- the basic attributes in the case of a simple domain;

- the term "composite" where this applies, followed by the corresponding list of component domains;

- the type of the internal structure, in the case of a structured domain.

b. The functions, denoted by edge labels, and the definition of their domain and range, the special domain " Φ " being included in the range whenever the function is a partial one.

c. The grammar that defines the intended structures and the permissible operations on them.

Before exemplifying the *schema* statement, the conventions used for representing an RLD in a linearized notation are illustrated (Fig. 10). The version used with the current implementation is illustrated in [29].

In scion patterns the node references are denoted by lower case Greek letters, and sets of pairs (edge label, node reference) may appear, as already mentioned.

The *schema* statement for example E of section 3 is partly shcwn below.

schema inventory:

domains S# character (5) ,
S# character (20) ,
.....
Y composite (PN × CO × W × I),
.....

functions fs : S# → X ,
fp : P# → Y ,
.....
fo : Z → O# ∪ Φ
.....

grammar p1 <α( )S(x)> => <α( )S(x,a+.β)>
<β(a+.α)S#(fs.γ)>
<γ(fs.β)X(fs1.δ,fs2.ε,fs3.φ)>
<δ(fs1.γ)SN( )><ε(fs2.γ)ST( )>
<φ(fs3.γ)CI( )> ,
.....
p3 <α(x)S#(y)> => <α(x,e.*)S#(y)> ,
.....
..... ;

The inclusion of the grammar conforms with the present trend of describing the set of permissible structural transformations when creating a new data type [27, 28].

Once a new data type, T , has been defined, it becomes possible to *declare* variables of type T . In the *declare* statement the lower and upper bounds on the cardinality of sets appearing in certain productions of the grammar may be specified. Taking example D, an instance t of type ternary tree could be specified by

declare t dbtree (...,0≤#x[p2]<3,...) ;

A declared variable stands for the root of the RLD representing an instance. Immediately after the declaration the RLD consists simply of the root labelled with the start symbol of the grammar. The application of production rules, to be discussed later, causes other elements to be added (and subsequently deleted, in some cases). A representation in memory is needed for each created instance; one possible representation would use the following tables:

a. Node table – there is an entry here for each node in an RLD; an entry contains:

   – the node label;

   – the subscript in the SIP/SOP table of the first SIP entry corresponding to the node;

   – the same for the first SOP entry;

- either a pointer to the location in storage where the value corresponding to the node is kept (in the case of simple nodes), or the subscript in the Dictionary table corresponding to an internal RLD (in the case of structured nodes);
- one bit to indicate whether a node has been visited in a traversal;
- a link to an available-space-list, if the entry is not currently in use.

A node reference is merely the subscript of the entry for the node in the table.

b. SIP/SOP table – there is an entry in this table for each inpointing and outpointing (edge label, node reference) pair; an entry contains:

- the edge label;
- a subscript corresponding to the node pointed from or to in the Node table;
- a subscript in the SIP/SOP table of another (if any) (edge label, node reference) pair belonging to the SIP or SOP of the same node;
- an available-space-list link.

c. Dictionary table – there is an entry in this table for each RLD; an entry contains:

- a subscript in the Type table of the "layer" in the three-dimensional Type table where the production rules of the grammar associated with the RLD are kept;
- a pointer to the associated domain and function descriptions;†
- a subscript of the entry in the Node table corresponding to the root of the RLD;
- a subscript of another "active" RLD (if any) in the Dictionary table;
- an available-space-list link.

d. Type table – this table can best be visualized as a three-dimensional array:

- to each defined type there corresponds a "layer";
- to each production in the grammar there corresponds a row within the "layer";
- finally, as each production has an LHS and an RHS, each row has two elements ("layers" have two columns).

By considering the two-element rows as the basic entries in this table, we may say that the table has an entry for each production, and that an

† To be stored in some compact form not discussed here.

entry contains:

- a subscript of the entry in the Node table corresponding to the root of the LHS scion;
- a subscript of the entry in the Node table corresponding to the root of the RHS scion.

Thus the LHS and RHS scions, like the RLDs, are represented by means of the Node and the SIP/SOP tables.

The representation for an RLD is shown in a simplified form in Fig. 11.

The storage requirements for RLDs comprise, besides the tables, the storage cells where values will be kept. The storage mapping with respect to these cells takes into consideration the attributes attached to the domains in the *schema* statement.

Allocation and de-allocation for entries in the tables and for the value cells are matters of crucial importance to the practicality of models such as the one presented here. For the first three tables, which are the ones for which deletions may occur, available space lists provide a convenient mechanism. For the value cells, the host language allocation, de-allocation and garbage collection mechanisms would be used by the DDF.

The purpose of a DDF is to be able to define a new data type, viewed here as a set of RLDs, and to provide a storage representation for the instances of it. For practical purposes, however, another addition must be made to the host

language, namely a set of *operators* for the *manipulation* of the instances. In data base terminology such operators constitute a Data Manipulation Language, DML, which is a necessary complement of a DDL.

Three basic classes of operators will be discussed here, since the way they are formulated is imposed by the definitional formalism.

a. *Selection operators* — A selection operator would implement some or all of the access techniques previously discussed. Noting that declarative information conveyed by the *schema* statement is referred to from the root, it is clear that an access technique which first visits the root before proceeding to a node for the execution of some operation (which must conform with the schema) is necessary.

For RLDs an obvious way of unambiguously locating a node from the root is to take a sequence of edge labels starting at the root. Edge labels, and sequences of edge labels can be conveniently represented and manipulated as character strings. As an example the statements

```
sequence  S ← 'a' , 'b' ;
.....................
sequence  T ← S , 'c' ;
```

yield the sequence (a , b , c) , which may be used later, given the location of the root of some RLD, to locate an internal node (or no node at all if the sequence is not present in the RLD

under consideration).

The access by scion pattern† may also be used for constructing a sequence of edge labels from the root of an RLD. As described, this technique starts from the root and tries each node as a possible root for a scion matching the given pattern; as the process advances and backtracks the sequence would be constructed. The statement

sequence S : I ← *pattern* <1(c_i.-α)Z(x,fsp1.β,fsp2.γ)>
<3(y,fsp1.α)S#(z)>
<4(w,fsp2.α)P#(v)>
values β = 'S135'
γ = 'P403' ;

would try to match the scion pattern shown in Fig. 12 against an instance I which has been previously declared to be type inventory; if in I the supplier S135 supplies part P403, then a sequence of edge labels from the root of S to the Z-labelled node that corresponds to the root of the scion pattern will be placed in S .

One may add to the statement some information that would restrict the search, thereby making it more efficient. In the example, the root of the matching scion (if any) could be reached only through a sequence consisting of a single c_i edge label, for some i , and this information could be made part of the statement (by writing ... *along* 'c_i').

Now that the determination of edge label sequences has

† This extends to selection part of the power that the grammatical formalism gives to insertions and deletions.

been discussed the selection operation can be simply denoted by

I @ S

where I is an instance and S a sequence of edge labels.

For accessing nodes of internal RLDs associated with structured nodes an appropriate number of sequences of edge labels should be used, and brackets would indicate different levels of hierarchy. Thus in

(I @ S) @ T

the sequence of edge labels in S determines a path from the root of I to a structured node, from whose root a path to the desired node is determined by the sequence of edge labels in T .

b. *Structural transformation operators* — The only permissible structural transformations are those which consist of the application of some production rule from the grammar associated with the RLD.

The statement

apply p4 : I @ S ;

means that a production p4 is to be applied in such a way that the root of its LHS scion pattern corresponds to the selected node I @ S . Since the root of I is being visited the grammar whose production p4 is to be applied, is known.

If I and J are instances of the same data type, then the statement

apply p2 : I @ S/J ;

will use J at the replacement phase of the application of production p2 , i.e. instead of embedding in I a copy of the RHS of p2 , it will embed J , provided that J matches the RHS. After this J loses its individuality, becoming part of I . In this second form the *apply* statement may be viewed as a generalized concatenation operator.

The indiscriminate concatenation of structures belonging to different classes is not allowed, since it would ruin the discipline imposed by the grammatical formalism. Even so, "hybrid" structures, such as lists of trees, are possible when intentionally indicated by:

- having in the list nonterminal nodes which may be unravelled into trees by productions from the same grammar (a one-level structure), or
- having in the list structured nodes with a grammar for trees associated with them (a hierarchical structure).

In an implementation, the part of the *schema* where the functions are defined can be used for a consistency check when a production is being applied. In the modified RLD the domain and range of the functions, represented by their node labels

must be as defined, and there must be present outgoing edges with the indicated labels, unless they denote partial functions.

c. *Assignment operator* — This operator places a value in a memory cell pointed to by the appropriate entry corresponding to a simple node in the Node table.

The statement

I @ S ← <host language expression> ;

performs a component-wise assignment (assignments to entire RLDs might be done by procedures, using the statement repeatedly). The *schema* information on the attributes of the domains would indicate what conversions, if any, are needed before the value is stored in the cell corresponding to node I @ S .

An implementation using PL/I as the host language is described in [29] .Among several convenient features, PL/I provides a pre-processor which permits some syntactical extension of the language, and the inclusion of text brought from a named file; the latter capability is used to insert the procedures of the DDF interpreter.

5. Conclusions

Taking a purely structural point of view as in [3], the RLD model can be described as a *single* algebra consisting of the set of RLDs and the operation of applying a production rule. The distinction between the several classes of data structures

is established by defining the grammar from which the production rules will be taken. Finally, an instance of a given class can be specified unambiguously by any of the sequences of pairs (production rule number, position of scion root) which characterize its derivation from the starting symbol of the grammar.

The model clearly encompasses all structures and multi-structures in [30]. The well-known techniques for determining what can be generated by a graph grammar [24] could be used to prove that unintentional tangles will not result from the intended operations, and thus to permit the enlargement of the number of classes that can be safely handled when adopting the approach suggested in [30].

In fact, it seems fair to conjecture that the ability to impose local and global structural conditions, breadth conditions (on nodes) and depth conditions (on edge sequences), and value conditions allows the specification of most classes of data structures of practical interest.

The representation of transformations by diagrams, particularly in a graphics setting, appears to be far more natural and intuitive than either a procedural or an axiomatic description. In practice, the axiomatic approach has been used more frequently for describing the properties (invariants) of a class. In [31] an axiomatic notation for describing invariants is presented, and it is shown how a transformation expressed as a production rule can be translated into the same notation

(augmented only with the concept of state), whereby the proof that given transformations and invariants are compatible is facilitated.

In principle all elements of a structure generated by the grammatical formalism are accessible, given the general selection operators. If for some reason one wants to impose restrictions one could incorporate the allowable selections into procedures [28] and use only these. For example, if only the topmost element in a stack is to be visible (which characterizes what is sometimes called a push-down list) the corresponding procedure would only produce one-edge sequences.

Further research would investigate efficient implementations of the proposed DDF. At least five directions for such work can be identified:

a. The desirability of assembling libraries of certain sets of elementary transformations that constitute solutions to frequently occurring problems, and of permitting "macro" productions to be expanded by using such libraries. This would spare the user of going into certain less obvious details†, and should permit in most cases to specify each meaningful operation by a single production.

---

† Even though graph transformations constitute a very high level of specification they still require a certain expertise for their formulation.

b. The comparative study of interpretation and compilation strategies; compilation strategies are usually more efficient, whereas interpretation strategies favour generality.

c. The introduction of an inferential capability [33], whereby some functions would not be explicitly represented by labelled edges, but would instead be defined in terms of others; while storage space is critical an "intelligent" system would resort to as much implicit representation as possible, proceeding to a more explicit form when more space is available and run time becomes of primary concern.

d. The design of restricted DDFs, that would take advantage of the characteristics of the problems arising in a specific application area, in order to achieve more efficiency.

e. The investigation of special hardware, such as associative memories [20] or special secondary storage devices [34], as also of visual displays used in interactive mode, as in [35], given that a graph-theoretical model is used.

One further point should be mentioned. All the discussion in the present work is related to the *logical* design of data structures, and therefore it represents only one stage of

the entire problem of data structure design and implementation. The very general storage representation presented here must be understood as a provisional or, alternatively, a default solution, until at a later stage a *refinement* is undertaken by choosing the most appropriate *storage mapping techniques* [36]. This is particularly true with respect to the physical implementation of different sets of edges which, in the same instances of a class of data structures, can be differently represented, in view of efficiency considerations, by contiguity, pointers, hashing functions, or other mechanisms.

Acknowledgments

We would like to acknowledge the support given to us by the National Research Council of Canada (to CCG) and the National Research Council of Brazil (to ALF) in carrying out the research described in this paper.

References

1. CODASYL - "Feature Analysis of Generalized Data Base Management Systems" - (1971).
2. Earley, J. - "Toward an Understanding of Data Structures" - Comm. ACM 14, 10 (1971) 617-627.
3. Rosenberg, A. - "Data Graphs and Addressable Schemes" - J. of Computer and Systems Sc..ences V, 6 (1971) 193-238.
4. Hoare, C. - "Notes on Data Structuring" - *in* "Structured Programming" - Dahl and Dijkstra (co-authors) - Academic Press (1972).
5. Standish, T.A. - "Data Structures - an Axiomatic Approach" - TR2639 - Bolt, Beranek and Newman (1973).
6. Codd, E.F. - "A Relational Model for Large Shared Data Banks" - Comm. ACM, 13, 6 (1970) 377-387.
7. Standish, T.A. - "A Data Definition Facility for Programming Languages" - Ph.D. thesis - Carnegie Institute of Technology (1971).
8. CODASYL - "Data Base Task Group Report" - (1971).
9. Codd, E.F. - "A Data Base Sublanguage Founded on the Relational Calculus" - Proceedings of the ACM/SIGFIDET Workshop on Data Description, Access and Control (1971) 35-68.
10. Pfaltz, J. and Rosenfeld, A. - "Web Grammars" - TR 69 - 84 - University of Maryland (1969).
11. Shaw, A. - "Parsing of Graph-Representable Pictures" - JACM 17, 3 (1970) 453-481.

12. Mylopoulos, J. - "On the Relation of Graph Grammars and Graph Automata" - Proceedings of the 13th SWAT (1972) 108-120.

13. Milgram, D.L. - "Web Automata" - TR 271 - University of Maryland (1973).

14. Harary, F. - "Graph Theory" - Addison-Wesley (1969).

15. Hopcroft, J. and Ullman, J. - "Formal Languages and Their Relation to Automata" - Addison-Wesley (1969).

16. Pratt, W. and Friedman, D. - "A Language Extension for Graph Processing and Its Formal Semantics" - Comm. ACM 14, 7 (1971) 460-467.

17. Feldman, J.A. and Rovner, P.D. - "An ALGOL Based Associative Language" - Comm. ACM 12, 8 (1969) 439-449.

18. Crick, M.F.C. and Symonds, A.J. - "A Software Associative Memory for Complex Data Structures" - IBM document G320-2060 (1970).

19. Datapro Research Corporation - "ADABAS - Software AG" - Datapro 70 (1973).

20. Love, H. and Savitt, D. - "An Iterative Cell Processor for the ASP Language" - *in* "Associative Information Techniques" - Jacks (ed.) - Elsevier (1971).

21. Griswold, R.E., Poage, J.E., Polonsky, I.P. - "The SNOBOL 4 Programming Language" - Prentice-Hall (1968).

22. Mylopoulos, J., Badler, N., Melli, L. and Roussopoulos, N. - "1.pak: A SNOBOL-Based Programming Language for Artificial Intelligence Applications" - Proceedings of the 3rd International Joint Conference on Artificial Intelligence (1973) 691-696.

23. Gotlieb, C.C. and Furtado, A.L. - "Data Schemata Based on Directed Graphs" - T.R. 70 - University of Toronto (1974).

24. Montanari, U.G. - "Separable Graphs, Planar Graphs and Web Grammars" - Information and Control, 16 (1970) 243-267.

25. Date, C.J. and Codd, E.F. - "The Relational and Network Approaches: Comparison of the Application Programming Interfaces" - IBM document RJ 1401 (1974).

26. Wegbreit, B. - "The Treatment of Data Types in EL1" - CACM 17, 5 (1974) 251-264.

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

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

29. Barroso, P.B. and Furtado, A.L. - "Implementing a Data Definition Facility Driven by Graph Grammars" - Journal of Computer Languages (to appear).

30. Shneiderman, B. and Scheuermann, P. - "Structured Data Structures - CACM 17, 10 (1974) 566-574.

31. Furtado, A.L. - "Characterizing Data Structures by the Connectivity Relation" - IJCIS 5,2(1976) 89-109.

32. Codd, E.F. - "Further Normalization of the Data Base Relational Model" - *in* "Data Base Systems" - Rustin (ed.) - Prentice-Hall (1972).

33. Brown, J.S. - "Steps Toward Automatic Theory Formation" - Proceedings of the 3rd International Joint Conference on Artificial Intelligence (1973) 121-129.

34. Minsky, N. - "Toward 'Intelligent' Rotating Storage Devices" - TR 73 - 2 - University of Minnesota (1973).

35. Christensen, C. - "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language" - *in* "Interactive Systems for Experimental Applied Mathematics" - Klerer and Reinfelds (eds.) - Academic Press (1968).

36. Gotlieb, C.C. and Tompa, F.W. - "Choosing a Storage Schema" - Acta Informatica 3 (1974) 297-319.

37. Nijssen, G.M. - Set and CODASYL Set or Coset" - *in* "Data Base Description" - Douqué and Nijssen (eds.) - North-Holland/ American Elsevier (1975).

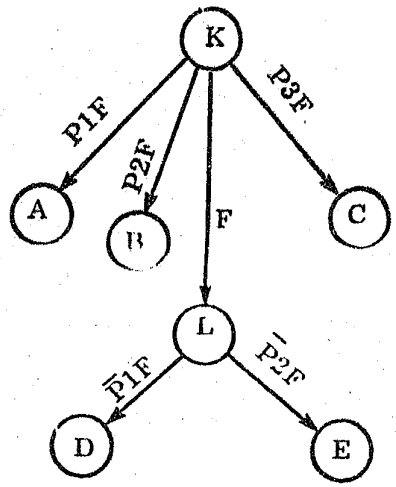


Fig. 1a : Representation of a function with composite domain and range

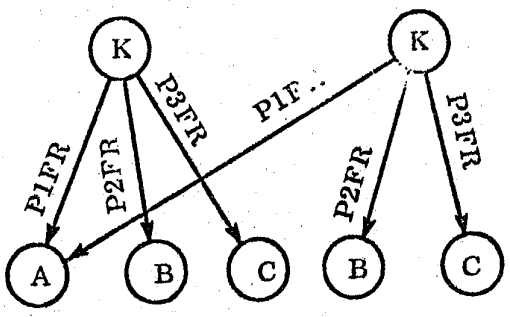


Fig. 1b : Representation of a proper 3-ary relation

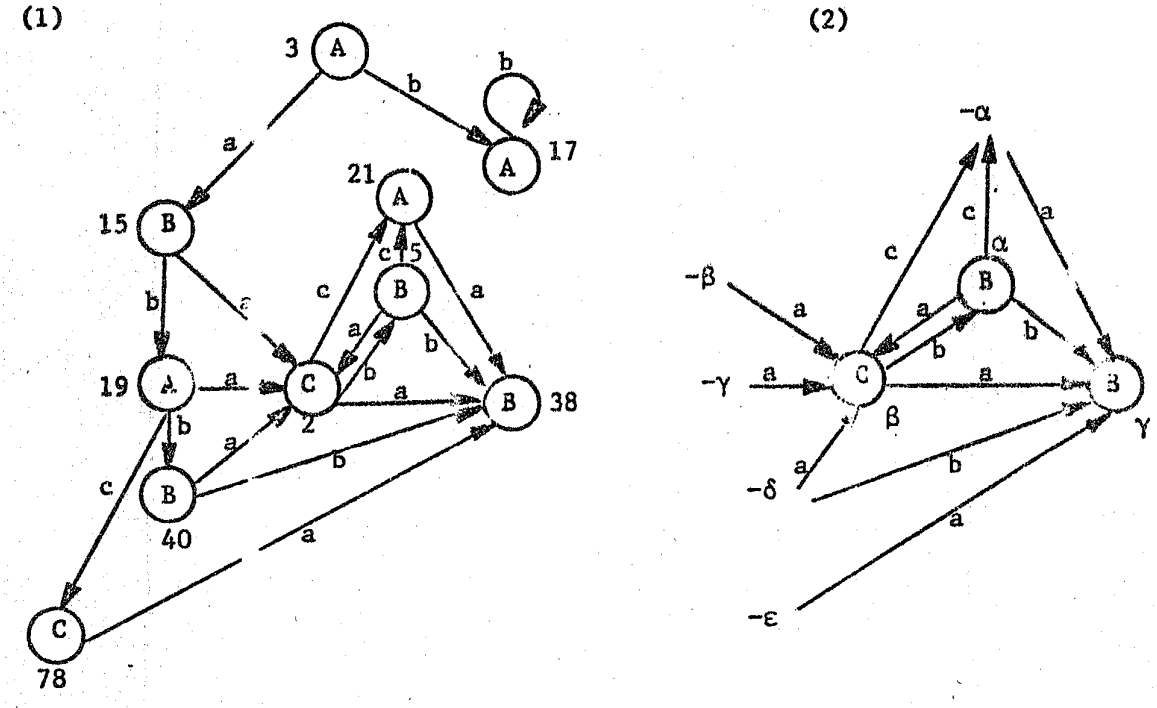
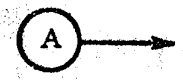
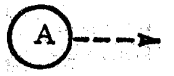


Fig. 2: (1) an RLD rooted at node 3

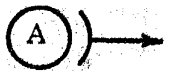
(2) a scion pattern rooted at node α



- required edge

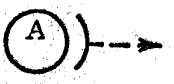


- edge that may be present or not



- non-empty set of pairs (edge label, node reference);

if x ⟵ (e_i, d) at least one occurrence of an edge label e_i and of an edge labelled d should be involved



- possibly empty set

Fig 3: Graphical conventions for cutset edges and sets of cutset pairs

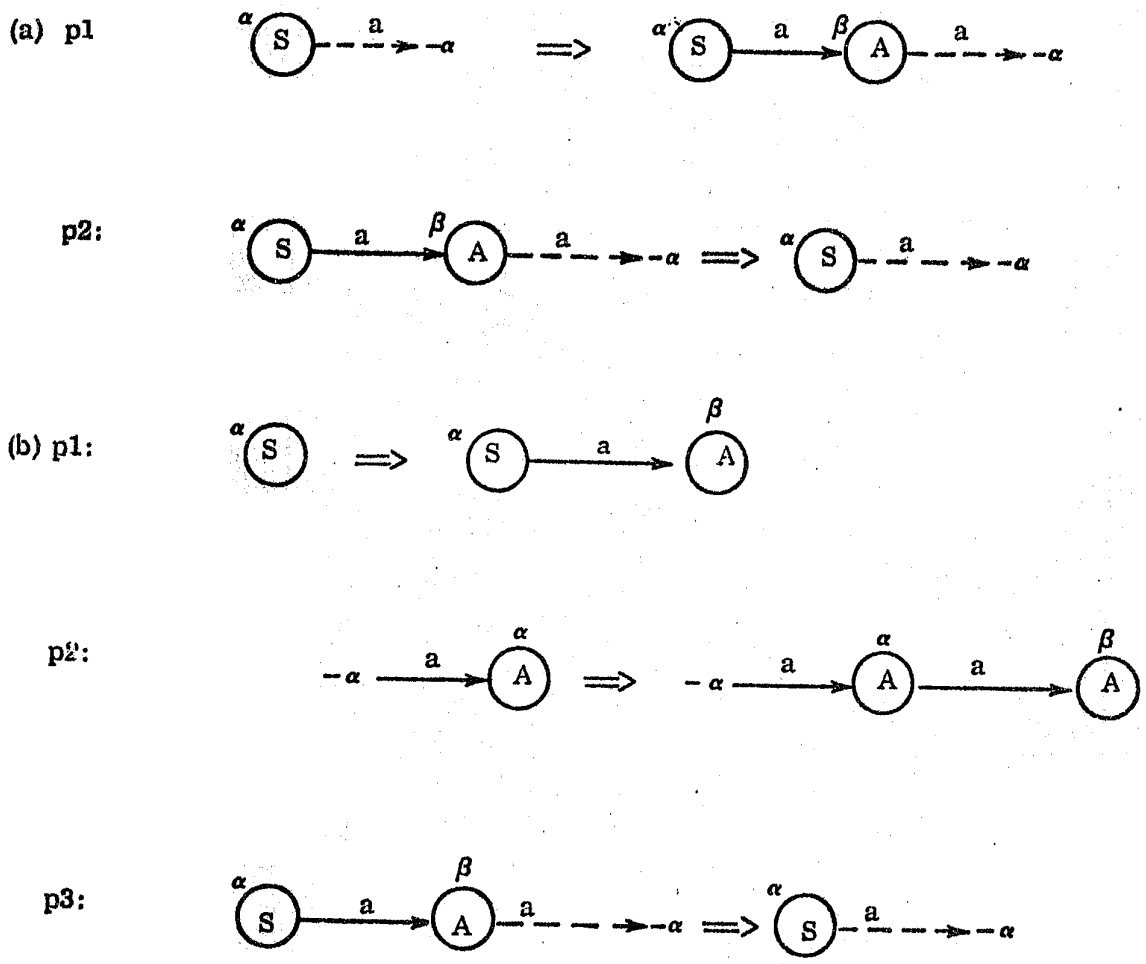


Fig. 4: (a) Grammar $G_{STACK} = (\{S,A\}, \{S,A\}, \{a\}, P, S)$

(b) Grammar $G_{QUEUE} = (\{S,A\}, \{S,A\}, \{a\}, P, S)$

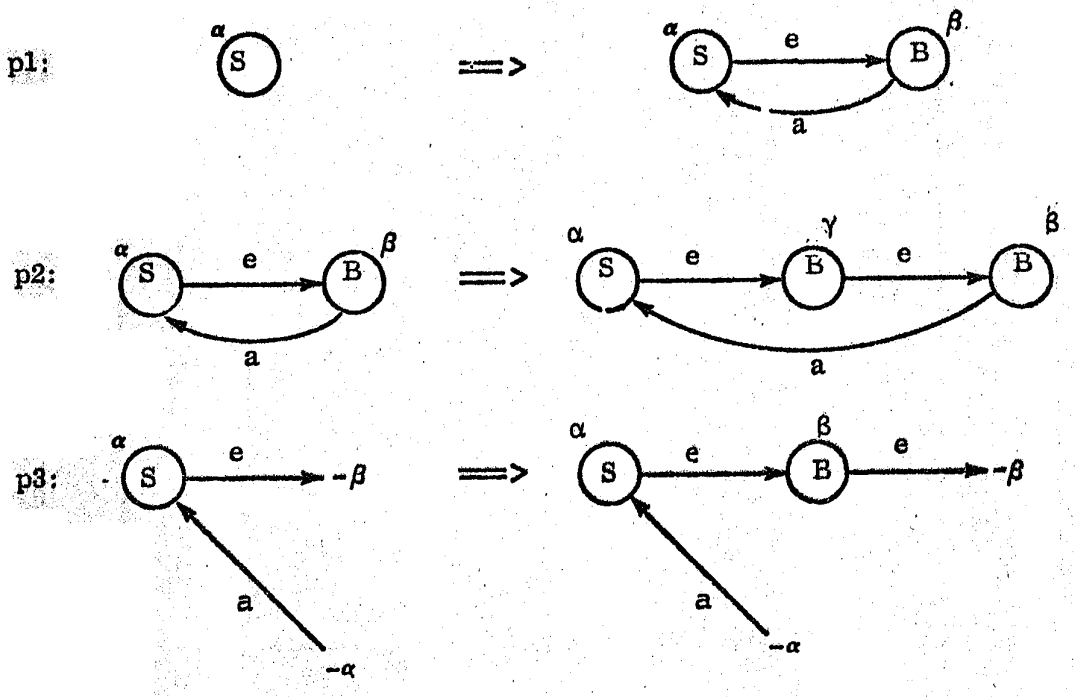
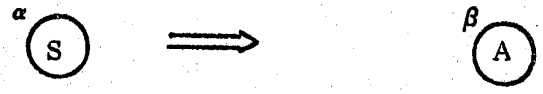
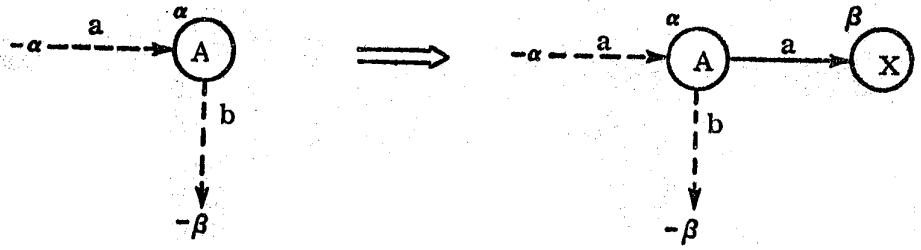


Fig. 5: Grammar $G_{RING} = (\{S\}, \{S,B\}, \{e,a\}, P, S)$

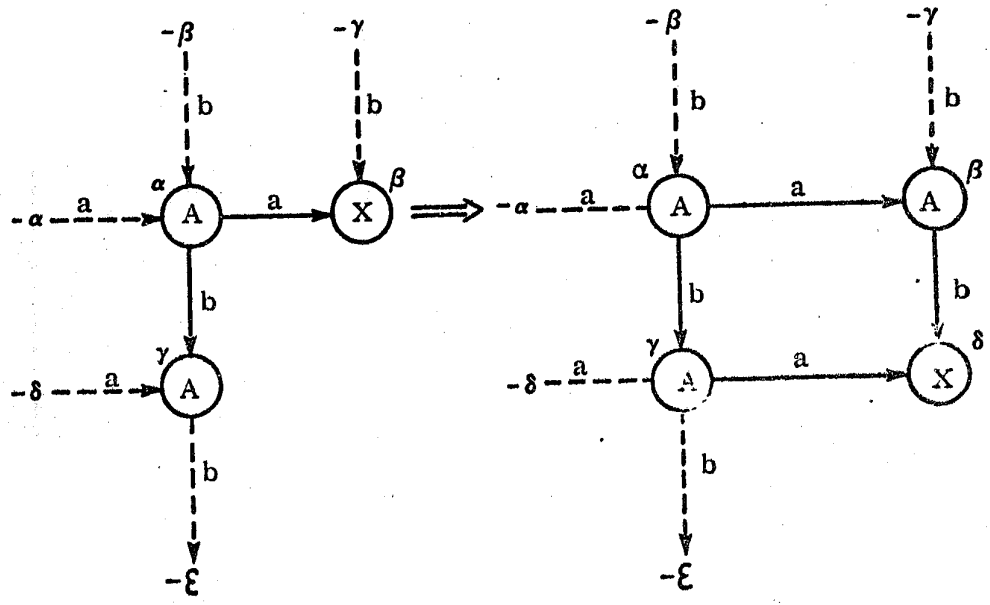
(a) p1:



p2:



p3:



p4:

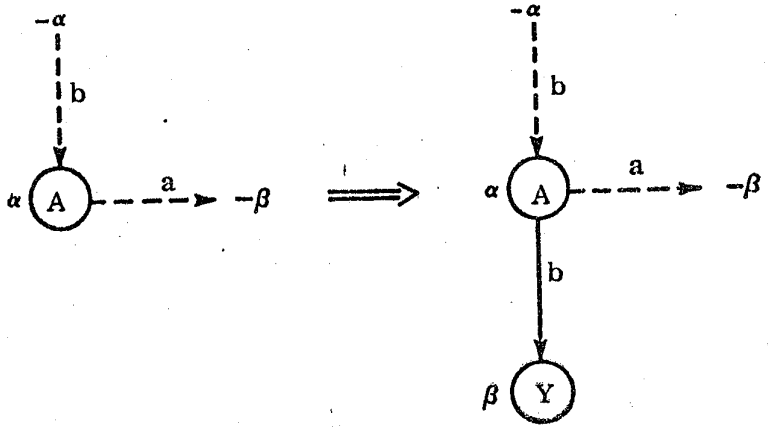
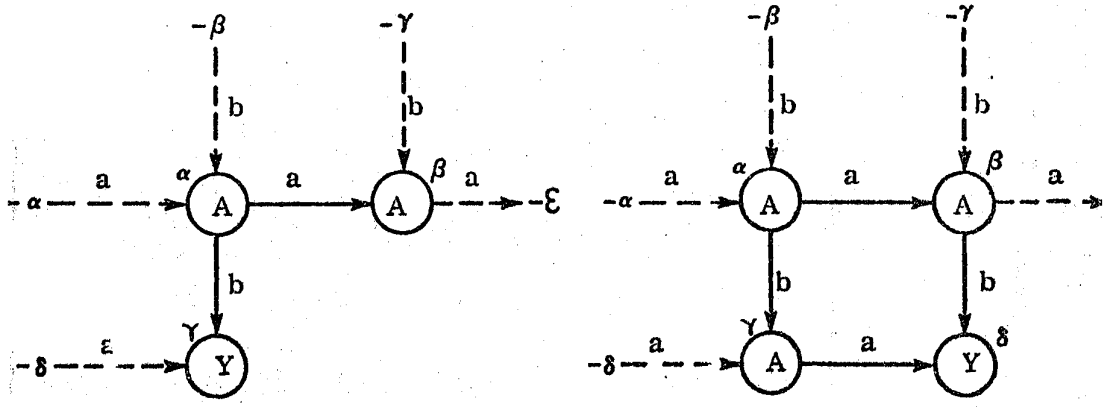
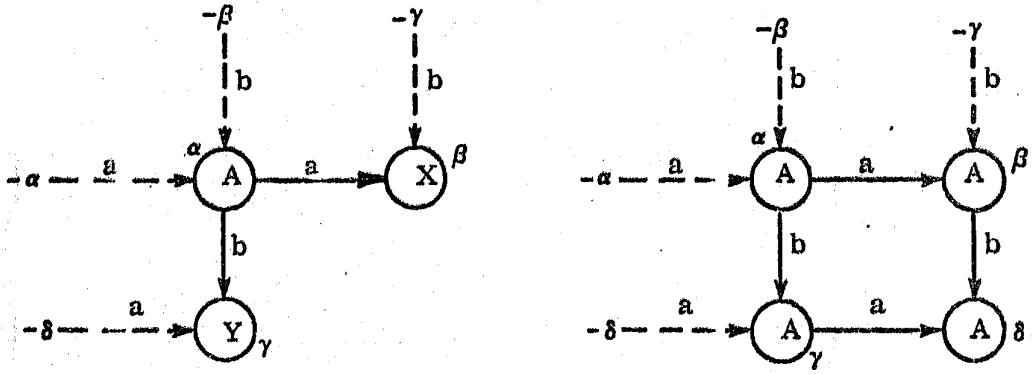


Fig 6(a): Grammar $G_{SARRAY} = (\{S,A,X,Y\}, \{A\}, \{a,b\}, P, S)$

p5:



p6:



(b)

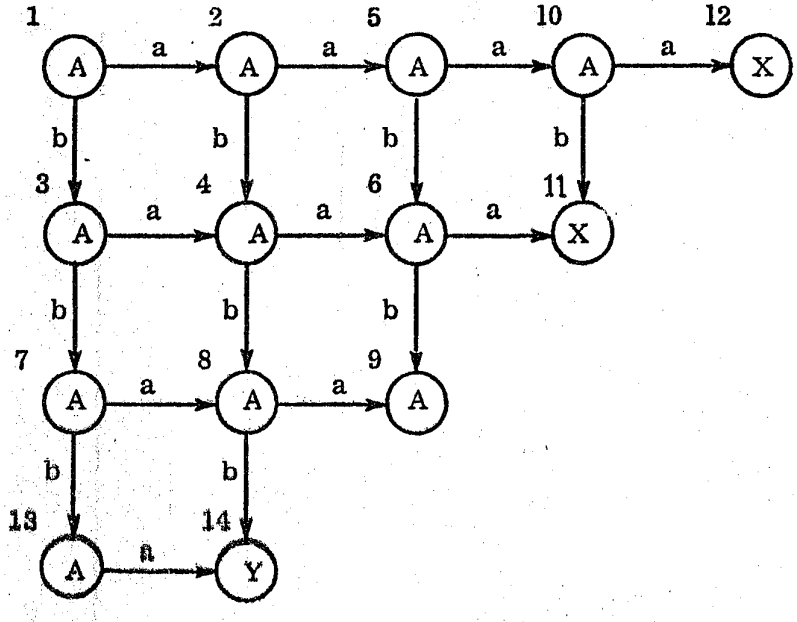


Fig. 6: (b) An intermediate configuration in the generation of a square array.

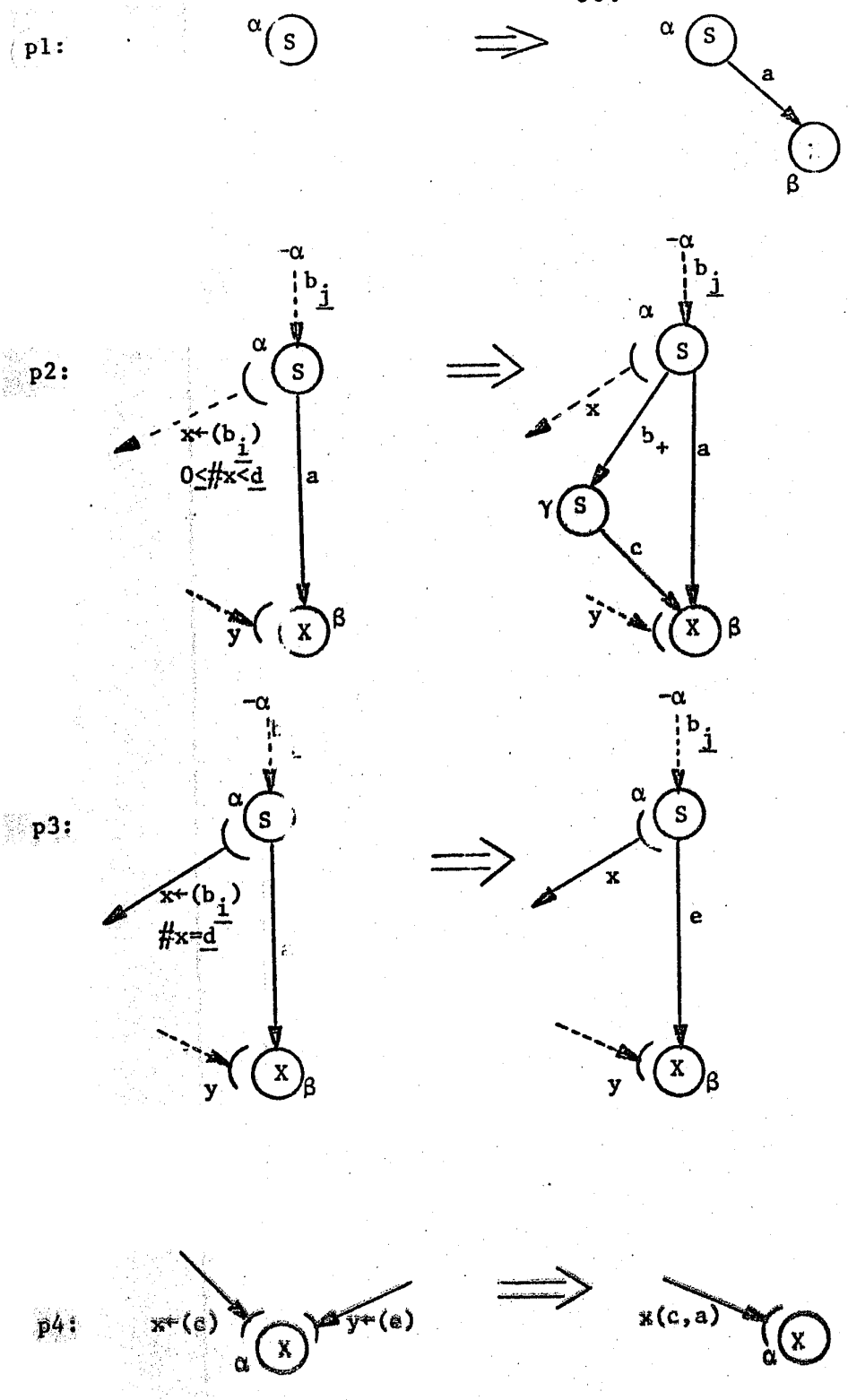


Fig. 7: (a) Grammar $G_{DBTREE} = (\{S,X\},\{S,X\},\{a,b_i \mid 1 \leq i,c,e\},P,S)$

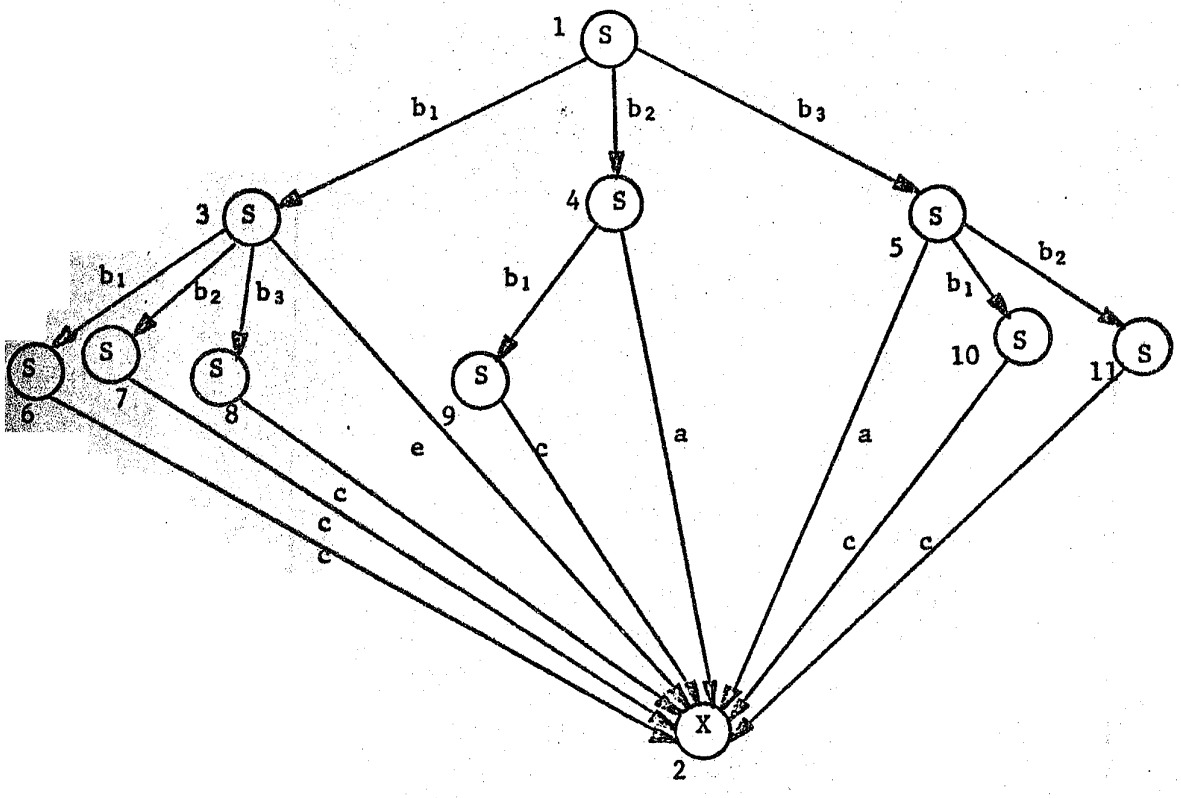


Fig. 7: (b) An intermediate configuration in the generation of a d-ary tree.

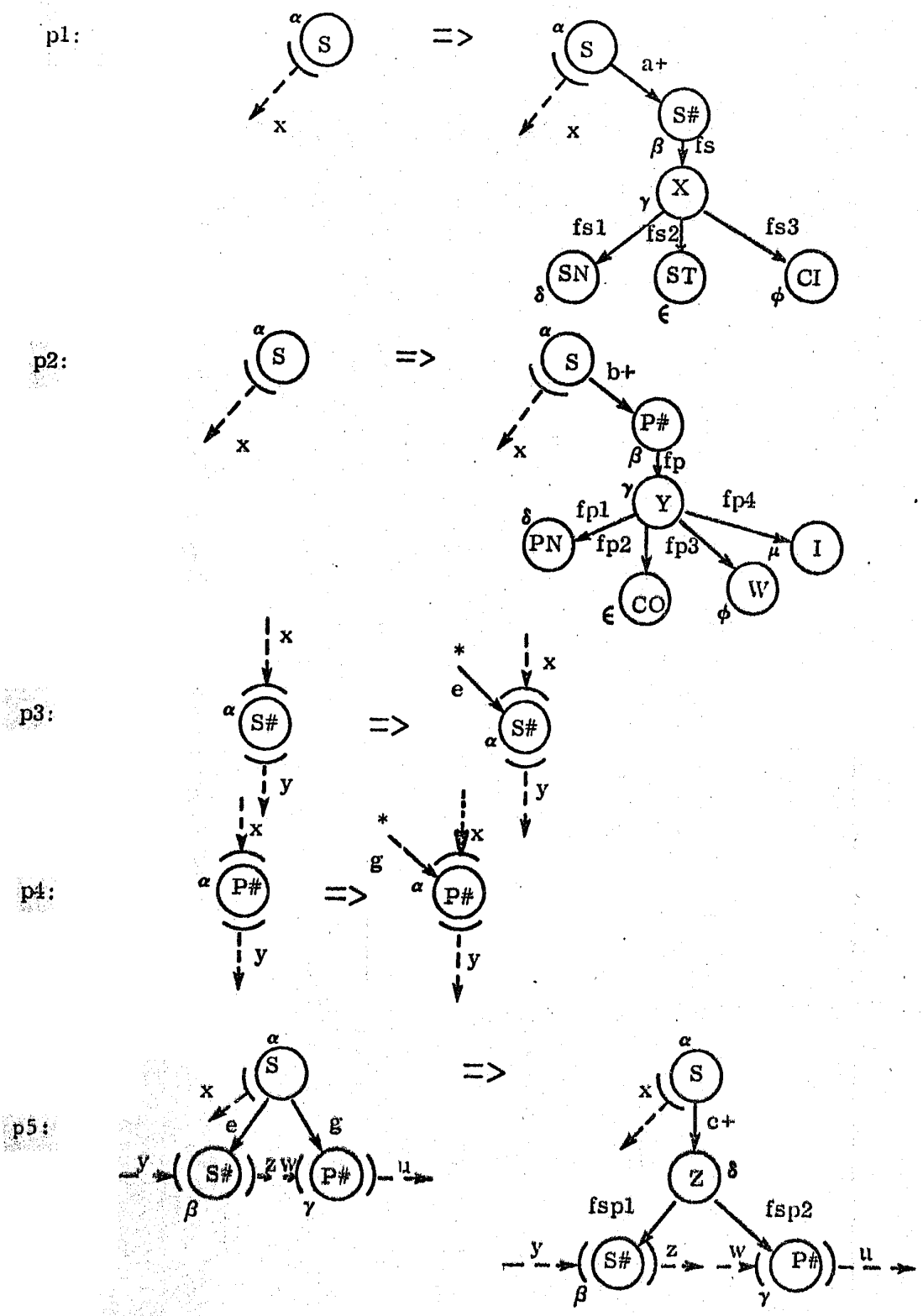

p1:
S
x
=>
S
a+
S#
x
β
fs
X
γ
fs1
fs2
fs3
SN
ST
CI
δ
ϵ
ϕ
p2:
S
x
=>
S
b+
P#
x
β
fp
Y
γ
fp1
fp2
fp3
fp4
PN
CO
W
I
δ
ϵ
ϕ
μ
p3:
x
S#
y
=>
*
e
x
S#
y
p4:
x
P#
y
=>
*
g
x
P#
y
p5:
S
x
e
g
y
S#
z
W
P#
u
β
γ
=>
S
x
c+
Z
δ
fsp1
fsp2
y
S#
z
W
P#
u
β
γ

Figure 8

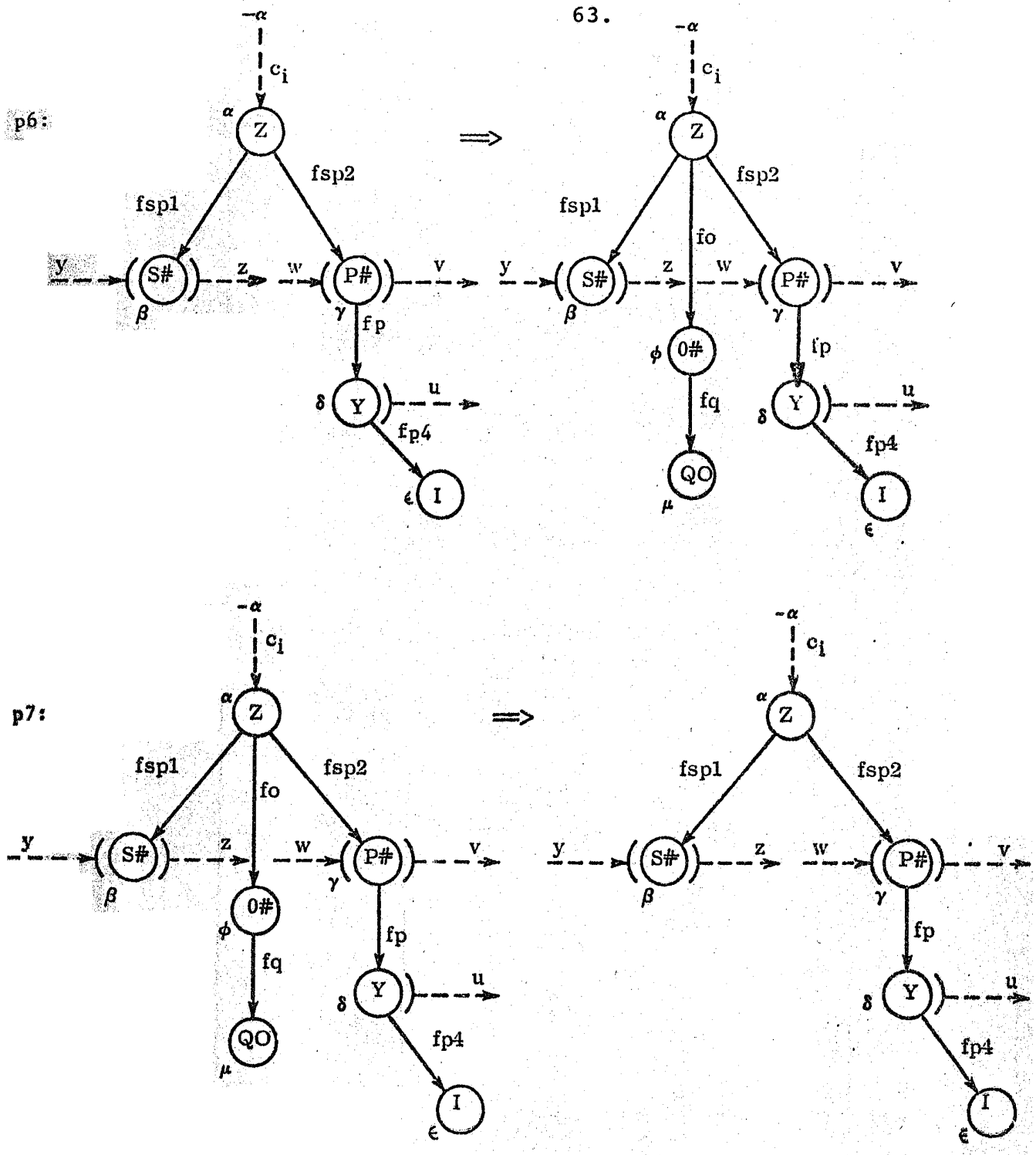


Fig. 8: Grammar $G_I = (\{S, S\#, P\#, Z\}, \{S, S\#, P\#, Z, SN, ST, CI, PN, CO, W, I, O\#, QO, X, Y\}, \{a_i, b_i, c_i, | 1 \leq i, e, g, fs, fs1, fs2, fs3, fp, fp1, fp2, fp3, fp4, fsp1, fsp2, fo, fq\}, P, S)$

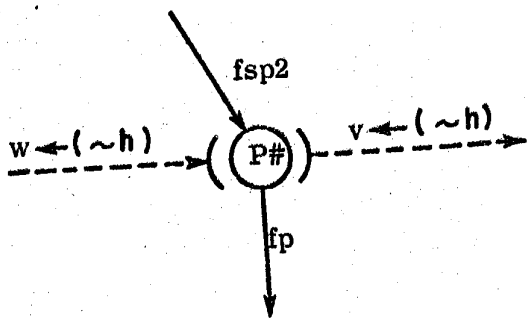
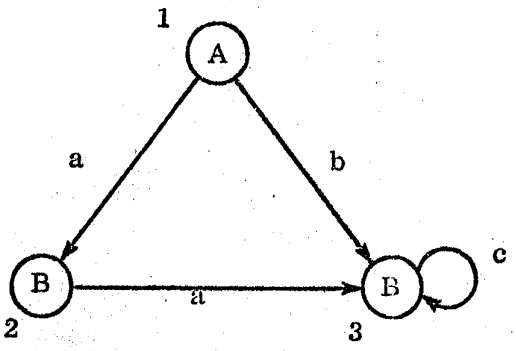


Fig. 9: Example of a negative applicability condition



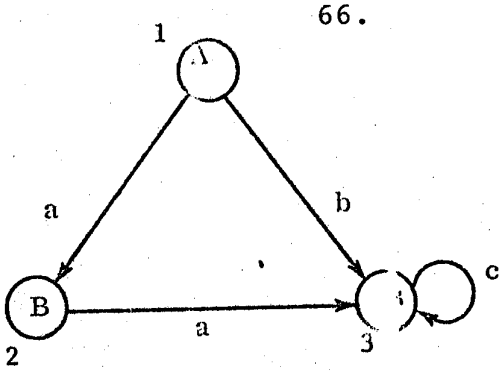
node format: <node-reference SIP node-label SOP>

<1 ( )A (a.2,b.3)>

<2 (a.1) B (a.3)>

<3 (b.1,a.2,c.3) B (c.3)>

Fig. 10: Example of the liearized representation of an RLD



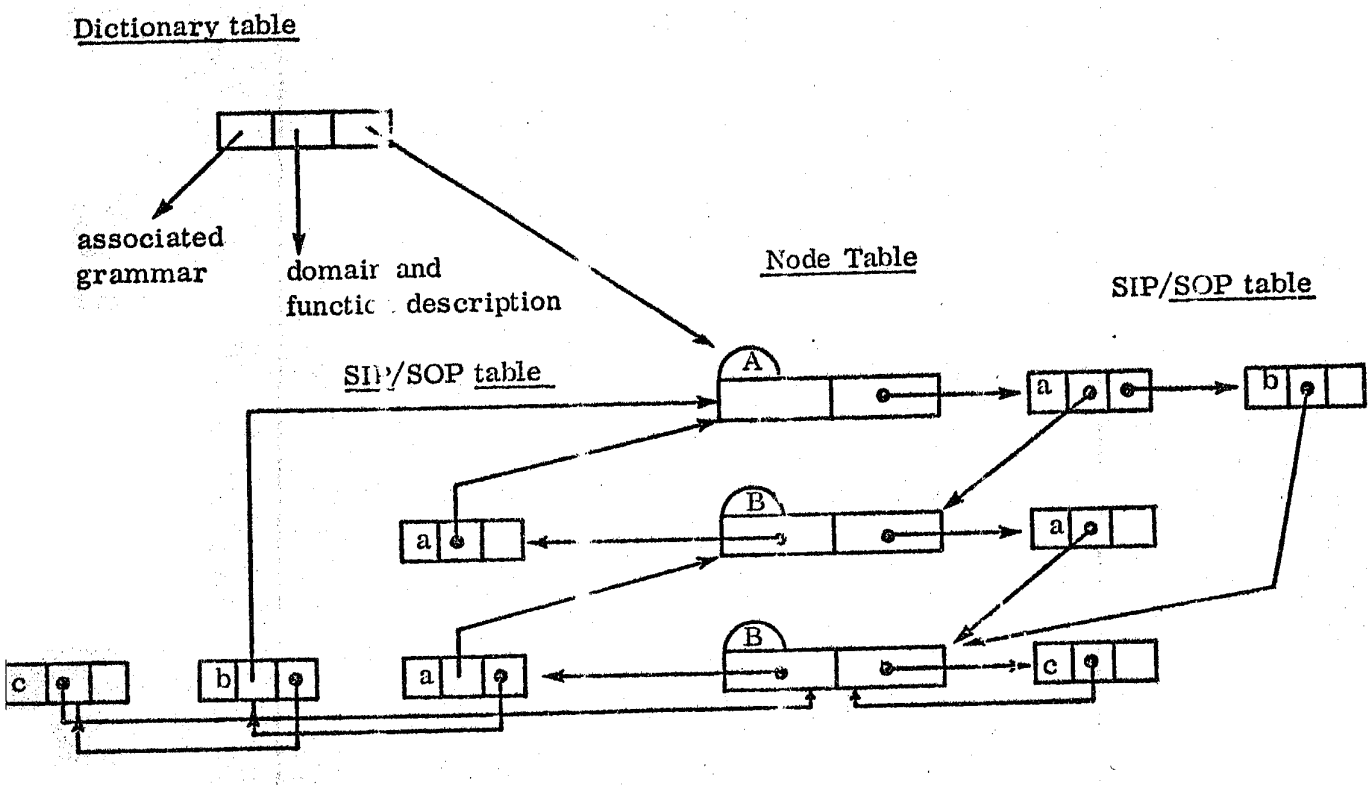


Fig. 11: Example of a simplified storag representation of an RLD

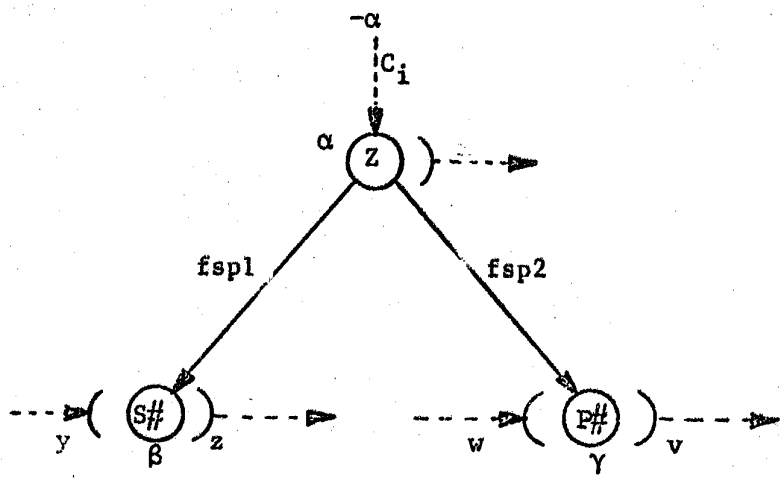


Fig. 12: A scion pattern to be used for determining an edge label sequence.