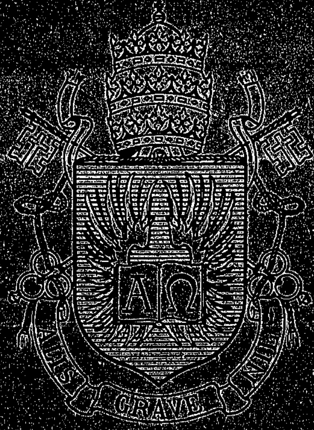


Series: Monographs in Computer Science and Computer Applications

Nº 7/72

ALGEBRAIC CONCEPTS IN DATA STRUCTURES

by

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

ALGEBRAIC CONCEPTS IN DATA STRUCTURES

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

Series Editor: Prof. A. L. Furtado

October / 1972

ABSTRACT

A survey of some aspects of data structures to which algebraic formalisms apply is attempted.

In each section one or a few references provide most of the material; they are marked with an asterisk in the list of references.

We believe that the concepts reviewed here are strongly related and that they should be considered together for any further work in the area of data structures.

Two valid research directions can be pursued:

a). the construction of a comprehensive formalism that would contribute to a better understanding of data structures, and hence to the study of program correctness and low-level complexity;

b). the elaboration of a reasonably powerful data definition facility capable of supporting the more complex data structures currently used in data base systems and non-numerical computation in general.

ACKNOWLEDGMENTS

The author is grateful to Prof. C. C. Gotlieb, who suggested this work and indicated a considerable part of the references, and to Prof. J. Lipson for a helpful discussion.

Any errors are of course the sole responsibility of the author.

TABLES OF CONTENTS

1. Atomic Data, Data Structures, Data Space 1

2. Properties of Data Types and Data Structures 8

3. Sets and Graphs as Data Structures 20

4. Algebraic Formalization of Data Structures 32

5. Generating Instances of a Schema 38

6. Conclusions 52

References 59

1. ATOMIC DATA, DATA STRUCTURES, DATA SPACE

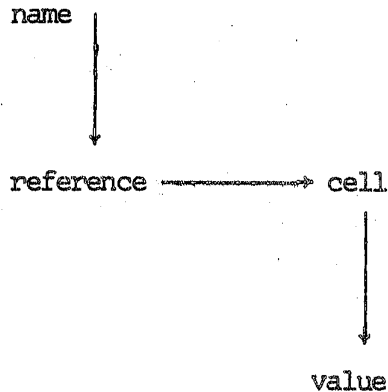
Before considering data in connection with computers, one has to consider data as mathematical objects.

Some usual data types are [2]:

- string – free monoid on a finite number of tokens; the concatenation operation is defined, and the identity element is the null string;
- Boolean – Boolean algebra (or, equivalently, ring);
- integer – integral domain;
- real – field;
- pointer – directed linear graph.

The mathematical nature of these objects imposes certain requirements in terms of convenient representation (scale, base, precision, etc. for numeric data) and the ability to perform mathematical operations on them.

When we come to computer implementation other objects arise. They should be viewed as auxiliary objects, whose only purpose is to allow the representation and processing of mathematical objects. A diagram will show the relations between them [15]:



We say that a name possesses a reference, a reference refers to a cell, and a cell contains a value.

This model is specially adapted to describe data in some very flexible languages as ALGOL 68, PL/I, or PASCAL.

Obviously the mathematical objects are the values. Also the names could be included in this category; they are comparable to variables in the mathematical sense.

Until now we have talked about atomic data. The word atomic may be understood in two different senses: as elementary objects from a mathematical point of view, or as values occupying a single cell of computer storage .

Atomic data can be put together, as elements are assembled to form sets. This is our informal notion of data structures.

An early but very useful notion of data structures [1] follows. Set-theoretic concepts are employed:

1. The Cartesian product $A \times B$ of two sets A and B is the set of all ordered pairs

   $(a \,.\, b)$ where $a \in A$, $b \in B$.
   Associated with the pair of sets

   (A, B) are two canonical mappings:

   $\Pi_{A,B} : A \times B \rightarrow A$ defined by $\Pi_{A,B}((a \,.\, b)) = a$

   $\rho_{A,B} : A \times B \rightarrow B$ defined by $\rho_{A,B}((a \,.\, b)) = b$

   Standard algebra texts call these mappings projections ([18] , page 12).

   A canonical function γ in two variables is also used:

   $\gamma_{A,B} : A, B \rightarrow A \times B$ defined by $\gamma_{A,B}(a, b) = (a.\, b)$.

2. The direct union $A \oplus B$ of the sets A and B is the union of two non-intersecting sets one of which is in 1 - 1 correspondence with A and the other with B. The elements of $A \oplus B$ may be written as elements of A or B subscripted with the set from which they come, i.e. a_A or b_B.

   The canonical mappings associated with the direct union $A \oplus B$ are:

   $i_{A,B} : A \rightarrow A \oplus B$ defined by $i_{A,B}(a) = a_A$

$j_{A,B}$: $B \to A \oplus B$ defined by $j_{A,B}(b) = b_B$

$p_{A,B}$: $A \oplus B \to \Pi$ defined by $p_{A,B}(x) = T$ iff $x \in A$

$q_{A,B}$: $A \oplus B \to \Pi$ defined by $q_{A,B}(x) = T$ iff $x \in B$

where $\Pi = \{ T, F \}$, the space of truth values. A function whose range is Π is called a predicate.

Two canonical partial functions are also defined:

$r_{A,B}$: $A \oplus B \to A$ defined for elements coming from A, by $r_{A,B}(i_{A,B}(a)) = a$

$s_{A,B}$: $A \oplus B \to B$ defined for elements coming from B, by $s_{A,B}(j_{A,B}(b)) = b$

3. The powerset A^B is the set of all mappings $f : B \to A$. A canonical mapping is given:

$\alpha_{A,B}$: $A^B \times B \to A$ defined by $\alpha_{A,B}(f,b) = f(b)$

Here $\alpha_{A,B}$ is a functional, that is a function having a function as argument. It applies the function to its second argument.

Isomorphism of sets is considered; the two sets $A \times B$ and $B \times A$ are not taken as equal, but:

$t_{A,B}$: $A \times B \to B, A$ defined by $t(u) = \gamma_{B,A}(\rho_{A,B}(u), \Pi_{A,B}(u))$

shows them to be isomorphic.

Now we shall see a useful application of the direct union operation, combined with cartesian product, yielding a definition of the set of sequences of elements of A:

$$S = \{ \Lambda \} \oplus A \times S$$

where Λ is the null sequence.

Note that this is notationally equivalent to the BNF recursive definition:

$$\langle S \rangle ::= \Lambda \mid \langle A \rangle \ \langle S \rangle$$

Two examples of a sequence generated by S would be:

$$a_1 \circ (a_2 \circ \Lambda)) \quad , \quad a_1 \circ (a_2 \cdot (a_3 \cdot \Lambda)))$$

In general such sequences can be represented either by the expansion

$$S = \{ \Lambda \} \oplus A \times \{ \Lambda \} \oplus A \times A \times \{ \Lambda \} \oplus \ldots,$$

or equivalently by

$$S = 1 \oplus A \oplus A^2 \oplus A^3 \oplus \ldots$$

where 1 represents $\{ \Lambda \}$, or even more simply by

$$\text{seq} \ (A)$$

Another recursive construction, which obviously corresponds to LISP 1.5 dot notation is

$$S = A \oplus S \times S$$

giving rise to the so-called S-expressions, (sexp (A)), such as

$$(a_1 . a_2) \quad , \quad ((a_1 . a_2) . a_3), \quad (a_1 . \quad (a_2 . a_3))$$

The concepts so far expounded allow the definition of the five LISP 1.5 primitives:

$$\text{atom } (x) \equiv p_{A,SxS}\,(x)$$

$$\text{eq } (x,y) \equiv (i_{A,SxS}\,(x) \equiv i_{A,SxS}\,(y))$$

$$\text{car } (x) \equiv \Pi_{S,S}\,(s_{A,SxS}\,(x))$$

$$\text{cdr } (x) \equiv \rho_{S,S}\,(s_{A,SxS}\,(x))$$

$$\text{cons } (x,y) \equiv j_{A,SxS}\,(\gamma_{S,S}\,(x,y))$$

We have taken a considerable space reproducing this formalism because it still plays a fundamental role in the theory of data structures (see also [34]).

Several authors (e.g. [8]) have agreed that the basic <u>structural</u> mappings (a concept suggested in [2]), are:

constructors - such as cons

selectors - such as car, cdr

and that to examine the structural properties of data, one needs

predicates - such as atom, eq

A higher generalization would take us to consider the data space, that is the entire collection of atomic and structured data, present in computer storage at a given time.

A convenient formalism [15] describes a digital computer as the triple:

$$(I , I^0 , F)$$

where:

I - set of information structures which may occur in information - carrying components of the computer (memory and auxiliary storage);

I^0 - set of initial configurations at the beginning of a computation ;

F - set of primitive instructions.

During the computation the sets I_j , $j = 0, 1, \dots, n$ are called information configurations, instantaneous descriptions, snapshots or states. I_n is the final state.

2. PROPERTIES OF DATA TYPES AND DATA STRUCTURES

Usually the first remark about data types as stored in a computer, as opposed to the abstract entities that they represent, refers to the finite nature of computer storage.

This applies specially to numeric data types. Although there is some variation among programming languages for a same machine, there are always limits to the maximum and minimum integers and reals allowed. So these domains become "sub-sets" of the abstract integer and real domains.

Moreover a given value may be just an approximation to the abstract "perfect" value. Since one could have several approximations to the same abstract value, and also, unrelated to these, several approximations to other abstract values, the concept of a partial ordering imposes itself quite naturally.

The following axioms have been proposed [3]:

Axiom 1 - A data type is a partially ordered set. The ordering is established by means of the relation $\sqsubseteq$, where $x \sqsubseteq y$ means that y is consistent with x and is possibly more accurate than x. The relation is clearly reflexive, transitive, and antisymmetric as required.

Axiom 2 - Mappings between data types are monotonic, that is $x \sqsubseteq y \implies f(x) \sqsubseteq' f(y)$. Intuitively this means that more accurate arguments given to a function, yield more accurate results from their evaluation.

Axiom 3 - A data type is a complete lattice under its partial ordering. So every subset of a data type has a least upper bound and a greatest lower bound. The whole of a data type has a l.u.b. called T. Whenever $x \sqcup y = T$, x and y are said to be inconsistent (T characterizes "over-determination").

On the other hand the element $\perp$ is at the bottom of the lattice. If $x \sqcap y = \perp$, x and y are said to be unconnected ($\perp$ represents the most "under determined" element).

Axiom 4 - Mappings between data types are continuous. A function $f: D \rightarrow D'$, between data types D and D' is continous iff for all directed subsets $X \subseteq D$ (X is directed if every finite subset of X has at least one upper bound) we have:

$$f(\bigsqcup X) = \bigsqcup \{ f(x) \mid x \in X \}.$$

By saying that mappings between data types are continuous one means that such mappings preserve limits (the l.u.b.'s of the directed subsets).

Axiom 5 - A data type has an effectively given basis. A subset $E \subseteq D$ is a basis if it satisfies:

- whenever $e, e' \in E$ then $e \sqcup e' \in E$;

- for all $x \in D$ we have $x = \bigsqcup \{ e \in E \mid e \leq x \}$.

The relation $\leq$ is defined in terms of $\sqsubseteq$ by:

$x \leq y$ iff $y \in \{ x' \in D \mid x \sqsubseteq x' \}$.

The phrase "effectively given" as applied to a basis means that the basis must be known.

As a consequence of the assumption of an effectively given basis it becomes possible to define what it means for an element to be computable: if $x \in D$ and E is a basis then x is computable relative to E iff there is an effectively given subsequence

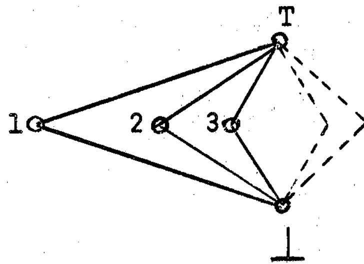
$$\{ e'_0 , e'_1, \ldots , e'_n , \ldots \} \subseteq E$$

such that $e'_n \sqsubseteq e'_{n+1}$ for each n and $x = \bigsqcup_{n=0}^{\infty} e'_n$

the last expression meaning that successively better approximations would allow convergence to x in the limit.

We shall now exemplify the lattice concept for data types with the positive integers and the reals.

In the case of the positive integers we have:

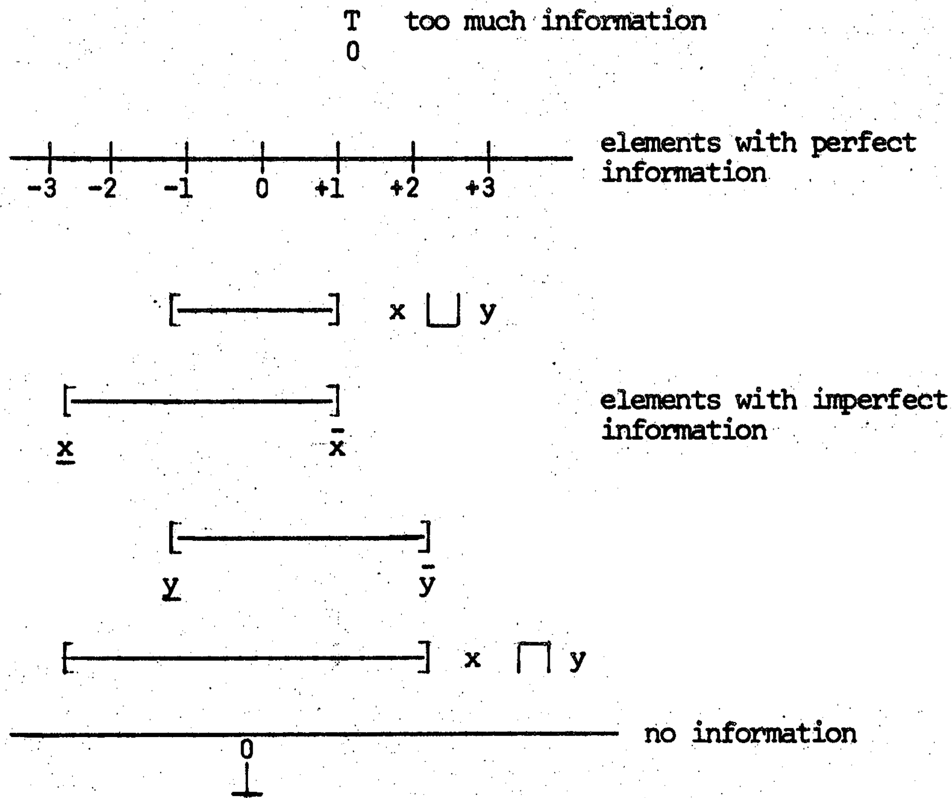


and clearly the concept does not bring anything new. From the totally under-determined element we jump to the exact integers and from them to the overdetermined element.

However the concept becomes useful in the case of the reals, which comes from the well-known fact that precision in computer storage is restricted by the maximum number of digits allowed; this in turn brings the idea of intervals, and we say that

$$[\underline{x}, \bar{x}] \sqsubseteq [\underline{y}, \bar{y}] \text{ iff } \underline{x} \leqslant \underline{y} \leqslant \bar{y} \leqslant \bar{x}$$

i.e. narrower intervals correspond to better approximations. "Perfect" reals have $\underline{x} = \bar{x}$ and for approximate reals $\underline{x} < \bar{x}$. The basis consists of the intevals with rational end points plus the element $\perp$. The situation is depicted in the diagram below:



The lattice concept is generalized to data structures, which are constructed in several ways (e.g. by forming cartesian products, disjoint unions, by taking the set of continuous mappings (function space $D \to D'$), etc.). In any case the study of a partial ordering is defined in terms of the partial ordering of the constituent elements; for example:

$$(x, x') \varepsilon \; D \times D' \quad , \quad (y, y') \; \varepsilon \; D \times D'$$

$$(x, x') \sqsubseteq (y, y') \text{ iff } x \sqsubseteq y \text{ and } x' \sqsubseteq' y'$$

Another class of properties, pertaining to data structures, shall now be considered [5].

First we shall introduce some notation:

α - the domain of data structures
number - the domain of numeric values
A, B - instances of data structures
$\Uparrow$ - adds an element to a data structure
& - logical operator and
$\otimes$ - selects first element
ε - selects element whose position is indicated
N() - cardinality of a data structure

Now consider the following assertions:

A1 $(a \; \varepsilon \; A) \quad A \Uparrow a = A$ - each distinct element is represented only once

A2 $a \; \varepsilon \; A$ evaluates as a pseudo-primitive - an element can be accessed regardless of its position

A3 $A = B \to \otimes A = \otimes B$ - ordinality

A4 $A = B :: \rightarrow :: (\forall_i)\ (0 \leq i \leq N(A)$ - circular ordinality

$: \rightarrow: (\exists_j)\ (0 \leq j \leq N(A)\ \&$

$(\forall_k)\ (0 \leq k \leq N(A) \rightarrow$

$\varepsilon(A, i + k \bmod N(A))$

$= \varepsilon(B, j + k \bmod N(B)))))$

A5 $A \Uparrow a \neq A \rightarrow \otimes A \Uparrow a = a$ - LIFO

A6 $A \Uparrow a \neq A. \rightarrow .\ a \equiv \varepsilon\ (B \Uparrow a,$ - FIFO

$N(A) + 1)$ where $B = A$

A7 $\otimes\ A$ = the $a \varepsilon A$ with - sorted list

max $f(a)$ where f is
a mapping $f: \alpha \rightarrow$ number

These assertions allow the characterization of usual data structures, such as stacks and queues. It seems an attractive idea to allow the user to specify his needs and leave to the "system" the burden of defining and creating suitable data structures; the notation above may well be a step in this direction.

According to whether (V) or not (X) each assertion holds, we have the following table of data structures:

	A1	A2	A3	A4	A5	A6	A7
bag - np	X	X	X	X	X	X	X
bag	X	V	X	X	X	X	X
set - np	V	X	X	X	X	X	X
set	V	V	X	X	X	X	X
list - np	X	X	V	X	X	X	X
list	X	V	V	X	X	X	X
rf - list - np	V	X	V	X	X	X	X
rf - list	V	V	V	X	X	X	X
ring - np	X	X	X	V	X	X	X
ring	X	V	X	V	X	X	X
rf - ring - np	V	X	X	V	X	X	X
rf - ring	V	V	X	V	X	X	X
pushdown	X	X	V	V	V	X	X
stack	X	V	V	V	V	X	X
rf - pushdown	V	X	V	V	V	X	X
rf - stack	V	V	V	V	V	X	X
queue - np	X	X	V	V	X	V	X
queue	V	X	V	V	X	V	X
rf - queue - np	X	V	V	V	X	V	X
rf - queue	V	V	V	V	X	V	X
sorted - list - np	X	X	V	V	X	X	V
sorted - list	X	V	V	V	X	X	V
rf - sorted - list-np	V	X	V	V	X	X	V
rf - sorted - list	V	V	V	V	X	X	V

where:

rf - repetition - free (see A1)
np - no-peeping (see A2)
bag- unordered data structure with repetition

Also note that a pushdown is distinguished from a stack by allowing access to any element (A2) in the latter. Thus the add /remove strategy is not confounded with accessibility.

Still another class of properties of data structures can be studied by "masking" the data items which compose a data structure, and concentrating on the linkages between the items. This gives rise to the concept of a data graph [12] .

Formally a data graph Γ is defined by

$$\Gamma = (C , \Lambda)$$

where:

C - countable set of data cells
Λ - finite set of partial transformations of C, denominated atomic link transformations, such that $\forall c, d \in C \ \exists \xi \in \Lambda^{\tau}$ such that $c \xi = d$.

An element of Λ is a path of length one (an edge), whereas Λ^{τ} contains paths of arbitrary length, including 0, since the identity transformation 1_C belongs to Λ^{τ}.

The existence of a ξ as postulated above implies that data graphs are strongly connected; in some cases this implies a reset transformation, going from the current cell c to a special cell c_0 (the root, to be defined further) and from there to the destination cell d.

We need some preliminary concepts:

Given the cartesian product (S x T) of sets S and T, and a set Ω of relations on S and T, we define:

$$\Delta(\omega) = \{s \in S \mid \exists\, t \in T, \langle s, t \rangle \in \omega\}$$

$$\nabla(s) = \{\omega \in \Omega \mid \exists\, t \in T, \langle s, t \rangle \in \omega\}$$

Now, let $\Gamma = (c, \Lambda)$ be a data graph, and let A be a set with $\# C \leq \# A$. A <u>realization</u> of Γ in A is a pair of mappings $\langle r, \rho \rangle$, where

1. r maps C one-one into A;
2. ρ is a monoid homomorphism $\rho: \Lambda^{\tau} \to A^{(A)}$, $A^{(A)}$ being the set of partial transformations of A. Thus, $(1_C)\,\rho = 1_A$ and for $\xi, \eta \in \Lambda^{\tau}$, $(\xi\,\eta)\rho = (\xi\,\rho)\,(\eta\,\rho)$

The pair $\langle r, \rho \rangle$ must satisfy two conditions for all $c \in C$ and $\lambda \in \Lambda$:

a. $c\,\lambda \in C$ iff $(cr)\,(\lambda\,\rho) \in C_r$;

b. if $\lambda \in \nabla(c)$, then $(c\,\lambda)\,r = (c\,r)\,(\lambda\,\rho)$.

Clearly we are considering a mapping for the data graph "nodes" (that is, r) and a mapping for the data graph "edges" (ρ). Conditions a and b show that the incidence relation must be preserved, whence if $\langle r, \rho \rangle$ realizes $\Gamma = (C, \Lambda)$, then Γ and $(Cr, \Lambda\rho)$ are isomorphic.

Given these preliminary concepts let us consider some important properties that we wish a data graph to have:

<u>Relocatability</u> - $\Gamma = (C, \Lambda)$ is relocatable in the set A, with $\neq\neq C = \neq\neq A$ if there is a cell $c_0 \in C$ and an injection $\rho_0 : \Lambda^{\tau} \to A^{(A)}$ such that for each $a \in A$, $\exists\, r_\alpha : C \to A$ satisfying:

a. $c_0 r_a = a$

b. $\langle r_a, \rho_0 \rangle$ is a realization of Γ in A.

This means that Γ is relocatable in A if there is a mapping of the link transformations of Γ which is insensitive to relocations within A of some designated cell of Γ. In other words, we can specify the address of the root c_0 at will, without any change to ρ_0; we point out that not all ρ enjoy of this characteristic.

<u>Adressability</u> - $\Gamma = (C, \Lambda)$ is addressable if it admits an addressing scheme, which is a total function $\alpha: C \to \Lambda^{\tau}$ such that:

a. $\exists c_0 \; \varepsilon \; C, \;\; c_0 \, \alpha = 1_c$

b. $\forall c \; \varepsilon \; C, \; \forall \lambda \; \varepsilon \; \Lambda$, if $c \; \varepsilon \; \Delta \, (\lambda)$ then

$(c \, \lambda) \, \alpha = (c \, \alpha) \, \lambda$.

An addressing scheme allows one to refer to data cells by the names of the transformations (that is, to refer to the nodes by the paths giving access to them). So it exposes the accessing structure of the data graph.

A concept which is in a sense associated with relocatability is the concept of realization by relative addressing:

Let $<r, \rho>$ be a realization of $\Gamma = (C, \Lambda)$ in a set A. We say $<r, \rho>$ is a realization by relative addressing if there is an element $a_0 \; \varepsilon \; C_r$ and a bijection $\delta: c \to \Omega = \{\omega \; \varepsilon \; \Lambda^{\tau} \mid a_0 \, \omega \; \varepsilon \; Cr\}$ such that, $\forall \, c \; \varepsilon \; C, \; cr = a_0(c \, \delta)$.

a_0 is called the base address of $<r, \rho>$ and δ the displacement function.

Rootedness - $\Gamma = (c, \Lambda)$ is rooted if there is a cell $c_0 \varepsilon C$ such that, for all $\xi, \eta \varepsilon \nabla (c_0)$, $\xi = \eta$ whenever $c_0 \xi = c_0 \eta$, c_0 being the root of Γ.

Thus in a rooted graph there is a unique transformation from the root to any other node.

Two kinds of rootedness play an important part in the study of data graphs:

- $\Gamma = (c, \Lambda)$ is freely rooted if there is a cell $c_0 \varepsilon C$ such that, for all $\xi, \eta \varepsilon \nabla (c_0)$ and $\lambda \varepsilon \Lambda$, either $\xi = \eta \lambda$ or $\xi \cap \eta \lambda = \phi$ (as binary relations), c_0 being a free root of Γ.

- $\Gamma = (C, \Lambda)$ is universally rooted if, $\forall c \varepsilon C$ and $\forall \xi, \eta \varepsilon \nabla (c)$, $\xi = \eta$ whenever $c \xi = c\eta$.

It turns out that:

1. addressable data graphs are precisely those which can be realized by relative addressing;

2. freely - rooted data graphs are precisely those which can be realized relocatably.

3. universal - rooted data graphs are precisely those which can be realized relocatably *in situ* .

These important results have been formally demonstrated in [12], where further research along the same lines is suggested.

3. SETS AND GRAPHS AS DATA STRUCTURES

We have seen that data structures are formed by putting together instances of one or more data types. Several kinds of data structures can be defined by checking whether or not they have certain properties (as those presented in the previous section).

However every data structure, being a composite entity, can be viewed as a set, regardless of its "computational" properties.

The realization of this fact has brought the language designers to include sets as a (very general) data structure in some programming languages.

One motivation for this is the usefulness of set-theoretic operations, very specially in such areas as information retrieval. Mathematicians are of course interested, having in mind their own kind of research.

But another point is that, as suggested before, a set can model any data structure, and therefore if efficient implementations are available sets can be put to general use. We must anticipate at this point that graphs (to be discussed in the next section) can also be represented by means of sets, since relations (arcs) between objects (nodes) can be represented by sets of ordered pairs.

A set-theoretic data structure [4] - STDS - is a storage representation of sets and set operations such that: given any family of sets η and any collection S of set operations an STDS is any storage representation which is isomorphic to η with S (see the concept of realization in the previous section). More formally, an STDS comprises:

1. S - collection of set operations;
2. β - set of datum names;
3. data - a collection of datum definitions, one for each datum name;
4. η - collection of set names.
5. a collection of set representations, each with a name in η.

Efficiency when handling sets with the set operations in S requires that the sets involved must be isomorphic to a unique representation of their elements, where uniqueness refers to some predefined well-ordering relation.

This well-ordering must be preserved under union.

Another class of problems arise from the implementation of n-tuples:

1. An ordered set is usually defined through nesting and repetition of the elements of the set; in Kuratowski's definition: $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$. We could use the order imposed by machine representation instead of resorting to this uneconomical redundancy, but then we would have trouble distinguishing n-tuples from sets, where unique representation and no ordering of its elements (taking ordering in the mathematical sense, rather than in the computational sense as in well-ordered representations in the paragraphs above) is required.

2. There is no consensus in set theory as to a canonical interpretation of n-tuples with more than two elements. Thus $\langle a, b, c, d\rangle$ could be any of $\langle\langle a, b\rangle, \langle c, d\rangle\rangle$; $\langle a, \langle b, \langle c, d\rangle\rangle\rangle$; $\langle a, \langle\langle b, c\rangle, d\rangle\rangle$; $\langle\langle\langle a, b\rangle, c\rangle, d\rangle$; $\langle\langle a, \langle b, c\rangle\rangle, d\rangle$; or $\{\langle 1, a\rangle, \langle 2, b\rangle, \langle 3, c\rangle, \langle 4, d\rangle\}$.

3. n-tuples, though defined as sets, are excluded from meaningful set operations. Thus $\langle a, b, c, d\rangle \cap \langle x, b, c, d\rangle = \phi$ unless $a = x$, and even this depends on which assignment is used.

As an attempt to solve these problems the concept of a **complex** is presented [11, 4].

Any two sets A and B form a complex (A; B) iff:

$$(\exists X)(\exists Y)(X \varepsilon \{A, B\})(Y \varepsilon \{A, B\}) \left[(\forall x \varepsilon X)(\exists i \varepsilon N) (\{\{x\}, i\} \varepsilon Y) \;\&\; (\forall y \varepsilon Y)(\exists j \varepsilon N)(\exists x \varepsilon X) (\{\{x\}, j\} = y)\right].$$

To see what the definition means, consider the sets:

$$A = \{a, b, c\}$$
$$B = \{\{\{a\}, 1\}, \{\{b\}, 3\}, \{\{c\}, 963\}, \{\{b\}, 6\}$$
$$C = \{a, b, \{\{b\}, 3\}, \{\{a\}, 1\} \;\; \{\{d\}, 6\}\}$$

According to the definition (A; B) and (B; A) are complexes. Note that no ordering of A before B is imposed; one of the two sets is an ordinary set while the other is a set of sets each of them composed of a singleton set (containing an element from the ordinary set) and a natural number; (A; B) and (B; A) are the same complex.

Also $(A \cap C;\ B \cap C)$ can be seen to be a complex, but $(A;\ A)$, $(A;\ C)$, $(A;\ B \cap C)$, $(A \cap C;\ B)$ are not.

Let us now look at the set-theoretic representation and at the equivalent representation as complexes for some examples. Here $x\ \varepsilon_i\ A$ will be understood to mean that x is in the ith position of the complex A; with this in mind the following notation can be defined:

$$\{x^i \mid \Psi(x, i)\} = A \quad \text{iff} \quad \left[(\forall x)(\forall i \in N)(x\ \varepsilon_i\ A \rightarrow \Psi(x, i)) \ \&\ A \text{ is a complex}\right]$$

Now for the examples:

set	$\{a, b, c\} = \{a^1, b^1, c^1\}$
ordered pair	$\langle a, b\rangle = \{a^1, b^2\}$
ordered triple	$\langle a, b, c\rangle = \{a^1, b^2, c^3\}$
ordered quadruple	$\langle a, b, c, d\rangle = \{a^1, b^2, c^3, d^4\}$

ordered pairs of ordered pairs

$$\langle a, \langle\langle b, c\rangle, d\rangle\rangle = \{a^1, \{\{b^1, c^2\}^1, d^2\}^2\}$$

$$\{\langle 1, a\rangle, \langle 2, b\rangle, \langle 3, c\rangle, \langle 4, d\rangle\} = \{\{1^1, a^2\}, \{2^1, b^2\}, \{3^1, c^2\}, \{4^1, d^2\}\}$$

Some operations for handling an STDS are:

1. union

a - binary union: $A \cup B = \{x \mid (x \in A) \vee (x \in B)\}$

b - unary union of a family of sets: $\bigcup G = \{x \mid (\exists A \in G)(x \in A)\}$

c - indexed union of a set f (A) over the family G:

$$\bigcup_{A \in G} f(A) = \{ x \mid (\exists A \in G)(x \in f(A))\}$$

2. <u>intersection</u>

a - binary intersection: $A \cap B = \{x \mid (x \in A)(x \in B)\}$

b - unary intersection : $\bigcap G = \{x \mid (\forall A \in G)(x \in A)\}$

c - indexed intersection: $\bigcap_{A \in G} f(A) = \{x \mid (\forall A \in G)(x \in f(A))\}$

3. <u>symmetric difference</u>

a - binary symmetric difference: $A \Delta \beta = \{ x \mid (x \in A) \Delta' (x \in B)\}$

b - unary symmetric difference: $\Delta G = \{ x \mid$ (for an odd number of $A \in G)$ $(x \in A)\}$

c - indexed symmetric difference:

$$\Delta_{A \in G} f(A) = \{ x \mid (\text{for an odd number of } A \in G)\ (x \in f(A))\}$$

note: Δ' means exclusive or.

4. relative complement: $A \sim B = \{ x \mid (x \in A)(x \notin B)\}$

5. exactly n! : (set of elements common to exactly n elements of a given set G): $E_n\, G = \{x \mid (E\ (n)\ !\ A \in G)(x \in A)\}$

note: E(n)! means exact number quantifier

6. domain of a set A: $D(A) = \{x \mid (\exists\, y)(\langle x, y\rangle \in A)\}$

7. range: $R(A) = \{y \mid (\exists x)(\langle x, y\rangle \in A)\}$

8. image of B under A: $A[B] = \{y \mid (\exists x \in B)(\langle x, y\rangle \in A)\}$

9. converse image of B under A: $[B]A = \{x \mid (\exists y \in B)(\langle x, y\rangle \in A)\}$

10. converse of A: $\bar{A} = \{\langle y, x\rangle \mid \langle x, y\rangle \in A\}$

11. restriction: $A|B = \{\langle x, y\rangle \mid (\langle x, y\rangle \in A)(x \in B)\}$

12. relative product of A and B: $A/B = \{\langle x, y\rangle \mid (\exists z)(\langle x, z\rangle \in A)(\langle z, y\rangle \in B)\}$

13. cartesian product: $A \times B = \{\langle x, y\rangle \mid (x \in A)(y \in B)\}$

14. domain concurrence of X relative to A: $D(X: A) = \{B \mid (B \in A)(X \subset D(B))\}$

15. range concurrence of X relative to A: $\mathfrak{a}(X: A) = \{B \mid (B \in A)(X \subset R(B))\}$

16. set concurrence of X relative to A: $\mathfrak{g}(X: A) = \{B \mid (B \in A)(X \subset B)\}$

17. cardinality of A: $\# A = n$ iff there are exactly n elements in A

18. A is a subset of B: $A \subset B \longleftrightarrow (\forall x)(x \in A \rightarrow x \in B)$

19. A is equal to B: $A = B \longleftrightarrow (A \subset B \wedge B \subset A)$

20. A and B are disjoint: $A \cap B = \phi$

21. A is equivalent to B: $\# A = \# B$

Some programming languages have adopted sets as one of their data structures (e.g. Pascal [15], which has a "powerset" type). Others use sets as the main data structures, such as SETL [16].

SETL includes basic set operators for membership test (ε), extracting an arbitrary element ($\exists$), cardinality (#), comparison (eq, ne), inclusion (incs), addition and deletion of an element (with, less), ordered pair deletion (lesf), defining the set of all subsets of a (pow (a)), or of all subsets of a having exactly k elements (npow (k, a)). The language also provides facilities for set-definition (by enumeration, range restriction, and by specifying predicates), functional application, quantified boolean expressions (using $\exists$ and $\forall$), searching, extraction and replacement, etc.

In order to get some idea of the practical problems involved in the computer representation of sets, consider sets $S_1, S_2, \ldots, S_n$ consisting of atomic data only (that is, sets of sets are not considered).

Suppose that when an atomic datum is created a natural number is associated to it; as an example assume that α has been created and the number i was associated with it.

Also, each set S_j is represented by means of a bit-pattern.

The membership test $\alpha \varepsilon S_j$? becomes simply $S_j^i = 1$?, where the superscript refers to the corresponding position in the bit-pattern.

If S_j, S_k, S_ℓ are sets of the same cardinality (or, in case they are not, the representative bit-patterns are padded to the right with a sufficient number of 0's) we can write $S_j := S_k \cup S_\ell$ as an OR bit operation between the bit-patterns S_k and S_ℓ, the result being put into bit-pattern S_j. Note that $\alpha \varepsilon S_j$? is still $S_j^i = 1$?, whence we

conclude that the ordering imposed by assigning natural numbers to data is preserved under the union of sets, as required.

Some authors, notably [6] having a great concern for efficiency in the implementation of sets and relations, and also for other considerations such as "data independence", have tried to avoid pointers, hash addressing schemes, indices, and ordering lists. Clearly the overhead involved in managing such access devices is considerable.

It is also clear that an application program should not be affected by the physical implementation of data and much less by any changes that such implementation may undergo. This is the issue of data independence.

One solution is the relational approach:

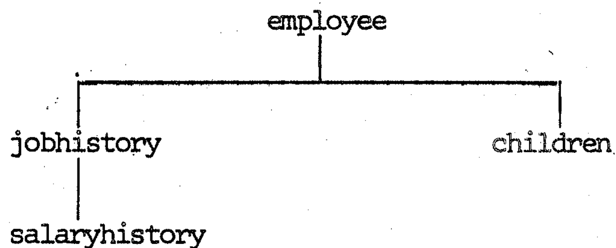
Let $S_1, S_2, \ldots, S_n$ be sets not necessarily distinct and let R be an n-ary relation among them. Then R will be represented by an array with the following properties.

1. each row represents an n-tuple of R.
2. the ordering of rows is immaterial
3. all rows are distinct
4. the ordering of columns is significant - it corresponds to the ordering $S_1, S_2, \ldots, S_n$ of the domains on which R is defined.
5. the significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

Property 4 can be relaxed, either by requiring that the domains must be uniquely identifiable, or by making them unique by means of "role names".

Since the ordering of columns becomes immaterial we are no longer dealing with relations (ordered n-tuples). The term "relationship" is used, and a remark is made in addition that they will be time-varying.

Sometimes the relationships are not all simple, as in the example:



employee (man # , name birthdate, jobhistory, children)
jobhistory (jobdate, title, salaryhistory)
salaryhistory (childname, birthyear)

where the underlined words are "primary keys".

If the relationships satisfy the requirements below they can be normalized, that is converted into a form that can be represented by the two-dimensional column homogeneous array already described:

1. the graph of interrelationships of the nonsimple domains is a collection of trees (forest).

2. no primary key has a component domain which is nonsimple.

The normalization process consists essentially of expanding the keys. In the example the normalization would give:

employee' (man ≠≠, name, birthdate)

jobhistory' (man ≠≠, jobdate, title)

salaryhistory' (name ≠≠, jobdate, salarydate, salary)

children' (man ≠≠ , childname, birthyear).

Operations on relationships are defined:

1. permutation - interchanges the columns of R; since the domains in a relationship are unordered this operation is only relevant to the consideration of stored repretions of R.

2. projection - let $L = i_1, i_2, \ldots, i_k$ be a list of indices and R an n-ary relationship ($n \geq k$); then Π_L (R) is the k-ary relation whose j^{th} column is column i_j of R ($j = 1, 2, \ldots, k$) except that duplication in resulting rows is removed.

3. join - let R and S be binary relationships; R is joinable with S if there is a ternary relationship U such that $\Pi_{12}(U)=R$, $\Pi_{23}(U) = S$. The operation can be generalized to non-binary relationships.

4. composition- given two relationships R and S, T is a composition of R with S if there exists a join U of R with S such that $T = \Pi_{13}(U)$.

5. restriction - let L, M be equal-length lists of indices such that $L = i_1, i_2, \ldots, i_k$, $M = j_1, j_2, \ldots, j_k$ where $k \leq$ arity of R and $k \leq$ arity of S; then the L, M restriction of R by S denoted by $R_L \mid_M S$ is the maximal subset R' of R such that

$$\Pi_L (R') = \Pi_M (S),$$

the operation being defined only if equality is applicable between elements of Π_{ih} (R) on the one hand and Π_{jh} (S) on the other for all $h = 1, 2, \ldots, k$.

Sets represent a very general kind of data structure, and their manipulation can be defined in easily understood mathematical terms. These two characteristics are also found in directed graphs [17].

An abstract directed graph G consists of a non-empty set V, a possibly empty set E with $V \cap E = \phi$, and a mapping $\phi: E \rightarrow V \times V$. Thus a directed graph can describe completely a set of objects and any binary relations among them; n-ary relations can also be represented in several ways (e.g. by linking n nodes to the same "dummy" node).

There are several languages for processing graphs having in mind problems in graph theory (GEA, GRASPE, GRAAL, etc.). At least one language (AMBIT/G [32]) uses graphs as the sole data-structure; the programs themselves are structured as graphs.

It is interesting to point out that since graphs are defined in terms of an unordered set (of nodes) and an unordered set of ordered pairs (arcs), the implementation of graphs as a data-structure may well use some of the ideas already discussed for the implementation of sets.

Thus, GRAAL adapts Childs' ideas (all atomic objects, are singleton sets, having a natural number associated to them when they are created; GRASPE defines the primitive operations of referencing, creating and deleting nodes and edges in set-theoretic notation, an example being [20]:

$$\text{son}\ (n, \ell) = \{m \mid \exists a \ni (n, a, m) \in f(\ell)\}$$

which reads:

"the set of outpointing nodes going from node n in graph ℓ is the set of nodes m such that there are labeled arcs a linking node n to nodes m"

Also operationg having graphs as operands are expressible as set-theoretic operations:

Let $G_1 = (V_1, E_1)$

$G_2 = (V_2, E_2)$

$\text{union}\ (G_1, G_2) = (V_1 \cup V_2,\ E_1 \cup E_2)$

Trees and networks (allowing nodes of indegree greater than one) are considered in present day data systems [21, 22].

A very desirable feature in the representation of graphs would be the definition of a canonical form. Although, for general graphs, a canonical form is perhaps unfeasible, canonical forms for the most usual data structures are already helpful.

It is conceivable that the reordered graph [29], and to a minor extent the ordering in [31] and the "palm-tree" [30] turn out to be steps toward a canonical form for graphs.

4. ALGEBRAIC FORMALIZATION OF DATA STRUCTURES

A data structure for a basic type variable is formally defined [14] by the 5-tuple:

$$[\, B, \mathcal{M}, V, DEC^{B}(v), \Phi^{B} \,]$$

where:

B - set of basic elements (characters, integers, reals, or pointers);

$\mathcal{M}$ - set of storage cells capable of storing a basic element (bytes, words, address words);

V - set of variable names (strings constructed according to a given grammar);

$DEC^{B}(v)$ - bijective mapping $V \to M$, defined by $v \mapsto \sigma_v$, $v \in V, M \subseteq \mathcal{M}$, $\sigma_v \in M$; this mapping may be thought of as a symbol table in - sertion, σ_v being interpreted as the "origin" assigned to variable v in computer storage;

Φ^B - operators and functions having elements of B as operands or arguments; $(B; \Phi^B)$ is an algebra in the mathematical sense.

The above definition is generalized to allow composite data, such as rational and complex numbers, character strings, etc.:

$$[\mathfrak{D}, \mathfrak{M}, V, DEC^{\mathfrak{D}}(v), SMF^{\mathfrak{D}}(v, D), \Phi^{\mathfrak{D}}]$$

Where:

$\mathfrak{D}$ - set of possibly composite data; $D \subseteq \mathfrak{D}$ is called an item and $d \in D$ is a component. We write $D = \{d_1, d_2, \ldots, d_{\tilde{d}}\}$ or, if the order matters, $D = \langle d_1, d_2, \ldots, d_{\tilde{d}} \rangle$.

$SMF^{\mathfrak{D}}(v,D)$ - storage mapping function $V \times D \rightarrow M$, defined by $v(\langle d_1, d_2, \ldots, d_{\tilde{d}} \rangle) \mapsto \langle m_1, m_2, \ldots, m_{\tilde{m}} \rangle$ where $\tilde{m}$ is not necessarily equal to $\tilde{d}$, and $m_1 = f(\sigma_v)$; when $\tilde{d} = 1$, that is when we are dealing with a basic type, SMF becomes simply $b \mapsto \sigma_v$;

$\mathfrak{M}$, V, $DEC^{\mathfrak{D}}(v)$, $\Phi^{\mathfrak{D}}$ - same as before;

To be more rigorous we should distinguish two kinds of operators and functions in $\Phi^{\mathfrak{D}}$: we have already mentioned those that are specific to a certain data type and that characterize the mathematical nature of the particular data type; another kind is composed by operators and functions that are common to all data types or to all composite structures:

assignment — $v \leftarrow D$

$v \leftarrow \varepsilon$ (where ε is the neutral element of the data type)

delete variable — $\sigma_v \leftarrow \Lambda$

add component — $v \leftarrow v \oplus d$

delete component — $v \leftarrow v \ominus d$ (note: $(v \oplus d) \ominus d = v$)

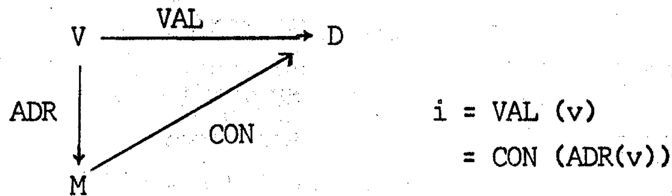
locate variable — $Loc^{\mathfrak{D}}(v, p)$, defined by:

$p \leftarrow$ <u>if</u> $v \in$ domain $(DEC^{\mathfrak{D}})$ <u>then</u> σ_v <u>else</u> Λ

locate component — $Loc^{\mathfrak{D}}(v, d, p)$ defined by:

$p \leftarrow$ <u>if</u> $d \in VAL(v)$ <u>then</u> $ADR(d)$ <u>else</u> Λ

We can define the three mappings VAL (value), ADR (address), and CON (contents) by the following diagram; the fact that it commutes shows VAL to be a composition ADR o CON:



DEC and SMF provide the representation of the abstract data structures in physical computer storage.

They can be executed at different times, depending in general on the language that is being used:

- statement read time
- program read time
- compile time
- link edit time
- block entry time
- instruction execution time

The decision on the most convenient "binding" time should be an attempt to find a compromise between binding earlier (more efficiency) or later (more flexibility).

DEC occurs first and requires the availability of good table look-up techniques.

SMF execution is distributed at different times. Using ALGOL 68 terminology we might say that DEC assigns a reference to a variable, while SMF allocates storage cells and updates the reference (see the reference as a symbol table entry) at each allocation. Relative addressing and relocation are important considerations here.

There are essentially two strategies for mapping a composite structure, noting that its leading element, as we saw, is allocated by $m_1 = f(\sigma_v)$; proceeding by induction, we consider the allocation of the $(i + 1)^{th}$ component assuming that the ith component is already allocated:

1. linear list - $d_{i+1} \rightarrow SUC\ (m_i)$
2. linked list - $d_{i+1} \rightarrow SUC\ (d_i)$

where SUC means successor.

When we consider a data structure $\mathfrak{D}$ from the mathematical point of view, we see that $SMF^{\mathfrak{D}}$ has to behave as a morphism of similar algebras, that is $\forall f_{\alpha} \varepsilon \Phi^{\mathfrak{D}}$, $\forall D_1 , D_2, \ldots , D_{n(\alpha)} \varepsilon \mathfrak{D}$ we must have, in order to make it possible to do any machine computation:

$$f_{\alpha}^{M} (SMF^{\mathfrak{D}} (D_1), \ldots , SMF^{\mathfrak{D}} (D_{n(\alpha)})) = SMF^{\mathfrak{D}} (f_{\alpha} (D_1 ,\ldots, D_{n(\alpha)}))$$

noting that for the sake of simplicity we are omitting the V component of the domain of $SMF^{\mathfrak{D}}$.

Until now we have only considered basic data types B and composite data types $\mathfrak{D}$. This could be generalized by specifying each of the usual more structured data such as arrays, structures, records, files, etc.

A much more useful generalization, which avoids such particularization lies on defining data structures as directed graphs:

$$\mathfrak{D}^* = (N , A)$$

where:

N - the node set - $N \subseteq \bigcup_i \mathfrak{D}_i$, the set union of the simple and composite data types allowed in a given programming language;

A - the arc set - $A \subseteq N \times N$, when $N \times N$ is the cartesian product of the node set by itself; as usually arcs represent relations between the nodes (data).

Two remarks have to be made about such graphs:

1. One node must be chosen (arbitrarily or not) as the root; assigning a tag to it makes it simple to discover the type of data structure to which it belongs (see [8,19]).

2. Nodes and arcs should be labelled, specially the arcs. If arcs of the same name leaving one node are not allowed it becomes possible to define an access path to a node n, unambiguously by:

$$(a_{i_1}, a_{i_2}, \ldots, a_{i_k})$$

where:

$$a_{i_\ell} \in A \quad \text{for } \ell = 1, \ldots, k \quad , \quad k \leq \# A$$

$$a_{i_1} = (\text{root}, n_p) \text{ for some } n_p \in N$$

$$a_{i_k} = (n_q, n) \text{ for some } n_q \in N$$

As before $SMF^{\mathfrak{D}^*}$ must preserve $\Phi^{\mathfrak{D}^*}$. It must also preserve incidence (see the concept of data graphs in section 2; note also that the rootedness property of data graphs is adopted here, as well as the strong connectivity requirement through the root).

The mathematical operations in $\Phi^{\mathfrak{D}^*}$ include the operations on basic data types executed componentwise and new operations defined for the data structure. As an example, suppose that A, B and C are square matrices of the same order n; in PL/I C = A * B means $c_{ij} = a_{ij} * b_{ij}$; $i = 1, \ldots, n$; $j = 1, \ldots, n$. On the other hand,

by using languages like ALGOL 68 where new operators can be defined, we could have a matrix multiplication operator: $C = A \otimes B$, meaning

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj} \quad ; \quad i = 1, \ldots, n; \quad j = 1, \ldots, n.$$

A rather confusing point is the concept of generic operators. Operators like + can be used for the addition of integers or of reals, and sometimes even mixed type operands are allowed.

However the use of the same symbol for different operations is standard practice among mathematicians, and if mixed operands are encountered the "system" will convert the lower type operands to the highest type in the expression (e.g. when adding reals and integers the integers will be transformed into reals). In certain cases mixed operands are legal, as when raising a real to an integer power, or when concatenating a string to itself n times, where n is a natural number.

The idea of describing a data structure by means of directed graphs is adopted by other authors (see [13]).

Still other formalisms can be found in [2, 7, 9, 10].

5. GENERATING INSTANCES OF A SCHEMA

When describing a data structure $\mathcal{D}^*$ it is very often the case that the description applies to several D that diverge in certain characteristics that are not part of the description. The description of binary trees for example does not specify the number of nodes, and so:

are both valid <u>instances</u> of the data structure binary tree.

In [14] it is suggested that a $\mathcal{D}^*$ can be described by:

- a data schema - the description proper; the CODASYL report [22] defines a schema as the description or definition of a set of structures of a given type in terms of a certain subset of the attributes for that type; since we are viewing data structures as directed graphs we could characterize such attributes as restrictions to the general directed graphs (e.g. each node except the root must have indegree 1, etc.).

- a grammar - for producing other directed graphs with the properties indicated in the schema; such derived graphs are the instances of the schema; note that a particular instance can change during program execution time, and therefore the grammar will be used at each time to ensure that the change (adding or deleting nodes or arcs, etc.) conforms to the schema.

We shall now review some concepts and formalisms that have been built around this idea.

In [13] a V-graph is a 7-tuple $< N, NT, nt, A, L, F, C >$ where:

- N is a set of nodes
- NT is a set of node - types
- nt is a total function nt: $N \rightarrow NT$, which associates with each node a type

- A is a set of atoms
- L is a set of links
- F is a partial function $F: Q \times N \rightarrow L$ (where $Q = A \cup N \cup L$), which represents the links in the graph; F(a, n) is defined if there is a link named a from node n and its value is the link.
- C is a function $C: L \rightarrow Q$, which defines what each link points to.

The definitions below show the purpose in defining V-graphs:

- A V-graph V_1 matches a V-graph V_2 if there exists a one-to-one total function $M: N_1 \rightarrow N_2$ such that $\forall_{n1} \in N_1$:

  1. $nt(n_1) = nt_2\ (M(n_1))$
  2. $\forall_{q1} \in Q_1$ if $F_1(q_1, n_1)$ is defined, then

     $M(C_1\ (F_1(q_1, n_1))) = C_2\ (F_2\ (q_2, M(n_1)))$

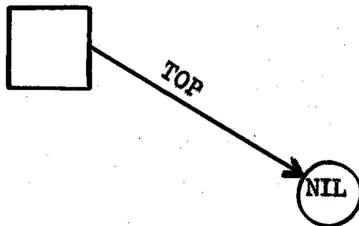
- A transformation is a triple $< V_1, V_2, P>$ where V_1, V_2 are V-graphs and P is a one-to-one partial function which establishes a correspondence between some of the nodes of V_1 and V_2, $P: N_1 \rightarrow N_2$. The nodes of N_1 not in the domain are deleted by the transformation and those of N_2 not in the range are added, and the links are also changed according to the difference between V_1 and V_2.

-A data structure is a pair consisting of a set of initial V-graphs (a concept not unlike the concept of schema) and a set of transformations (defined in [14] as a grammar).

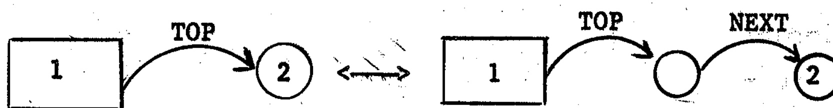
A careful distinction is established between the semantics of data structures (the nature of the basic data types and what are the relations between them, how each part of the data structure is accessed, what transformations are allowed) - which can be expressed in the V-graph notation - and the implementation (realization in a machine, or in the terminology of [14], the SMF$^{\mathfrak{D}^*}$ function).

The transformations are described in an entirely pictorial way. A stack with the operations of pushing - down and poping - up is represented by:

initial configuration (the empty stack):



tranformations (two, represented in a single picture)



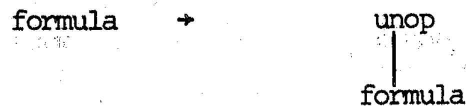
Note that the LIFO strategy is clearly characterized. As in AMBIT/G different types of atoms are represented by different symbols; □ represents a header and ○ an ordinary entry in the stack.

The idea of representing the transformations as productions in a grammar is already present in [8]. We shall restrict ourselves to an example, that deals with formula trees with binary and unary operators:

- non terminals:

< atom > :: = a | b | c |... |z
< binop> :: = + | - | x | ÷
< unop > :: = sin| cos| tan| ln |exp| -

- grammar:



The following rules are used in order to recognize a valid formula. A rule succeeds whenever its left part matches the given graph or a sub-graph of the given graph.

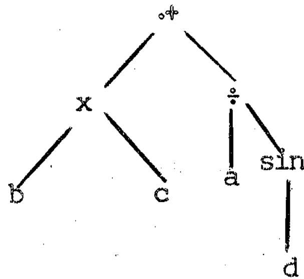
rule label	left part		right part	go-to
f	.binop formula formula	→	.formula	f
+1	.unop formula	→	.formula	f
+2	.binop formula σ	→	binop formula .σ	f
+3	.unop σ	→	unop .σ	f
+4	.binop σ	→	binop .σ	f
+5	σ .atom	→	σ .formula	f
+6	σ .formula	→	.σ formula	f
+7	σ .atom	→	σ .formula	f
+8	σ .formula	→	.σ formula	f
+9	σ .atom	→	σ .formula	f
+10	σ .formula	→	.σ formula	f
+11	.formula	→	.formula	exit

In the rules the dot "." is used as a marker. Several rules just change the position of the dot.

The symbol "σ" stands for any terminal or non terminal.

Control passes to the next sequential rule in case the current rule does not apply. If the current rule applies then the corresponding transformation is executed and control passes to the rule whose label is indicated in the go-to field. Thus this set of rules may be viewed as a Markov algorithm.

To see how it works consider the expression $b \times c + \frac{a}{\sin(d)}$, or in tree form, with the marker at the root:



The successfully applied rules form the sequence:

f+4, f+4, f+5, f+6, f+2, f+9, f+10, f, f+6, f+2, f+4, f+5, f+6, f+2, f+3, f+7, f+8, f+1, f+10, f, f+10, f, f+11, exit.

More recent efforts, in the area of pattern recognition, have led to the definition of graph grammars [24], which generate labeled graphs.

In a graph grammar the terminals consist of a label a from the set V of vertex values and a vertex x; a terminal is denoted by a(x).

A nonterminal consists of a nonterminal label A and K entities $w_1, w_2, \dots, w_K$, each of which can be a vertex or a set of vertices; a nonterminal is denoted by A $(w_1, w_2, \dots, w_K)$ or $A(\underline{W})$.

Assume that D is the maximum degree of the vertices and $1 \leq k \leq D$ is a natural number used to label the arcs having a node x as origin. Then x.k indicates the vertex adjacent to x along direction k; since no two edges coming out from the same node can have the same label, x.k is unique.

Using small letters with bars to denote sets of vertices, $\bar{x}.k$ will indicate the set of vertices adjacent to vertices in $\bar{x}$ along direction k. Both x.k and $\bar{x}.k$ will be referred to as the derivations of x, $\bar{x}$.

A constraint on a set of entities $\{w_1, w_2, \dots, w_S\}$ is a simple set theoretic relation between two elements or an element and the derivative of another element of the set. The relations are = , ε, $\subseteq$ or the negation of any one of these; testing for empty or non-empty set is also a constraint. Eg : $x_1 = x_2$, $x_1.k \ \varepsilon \ \bar{x}_2$, $x_1 \ \varepsilon \ \bar{x}_2.k$, $\bar{x} \subseteq \bar{y}$, $\bar{x} = \phi$, $\bar{x} \neq \phi$, etc.

A context-free graph grammar (C F G G) is a triple (N, S, P), where:

N - set of nonterminal labels

S - starting label, S ε N

P - set of productions of the form:

$$A(\underline{w}^0) \rightarrow B_1(\underline{w}^1) \ B_2 \ (\underline{w}^2) \ \dots \ B_r(\underline{w}^r) \quad \text{(con)}$$

where $\underline{w}^i = (w_1^i, w_2^i, \ldots, w_K^j)$ and (con) is a finite set of constraints on the set $\{w_1^0, w_2^0, \ldots, w_K^r\}$; the starting nonterminal of the grammar is $S(x, \phi, \phi, \ldots, \phi)$.

Instead of a complete description of the formalism we shall use the same example in [24] to show the two possible ways of generating a new structure from a given one by applying a production.

The production is

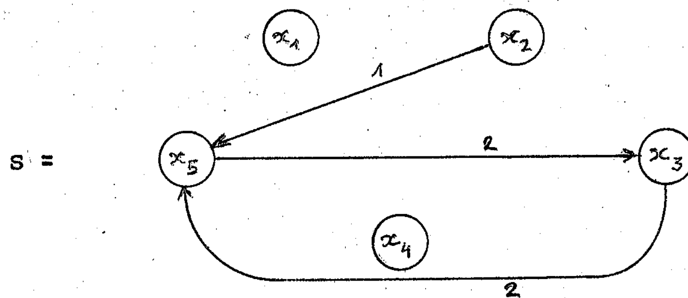
$$A(x_1, \{x_2, x_3\}) \rightarrow B_1(x_1, x_2)\ B_2(x_3, x_4)\ (x_2 = x_1.1,\ x_3 = x_4.2)$$

Now consider a structure s with a nonterminal $A(\underline{w})$ matching the left side of the production:

$$s = s_0\ A(x_1, \{x_4, x_5\})$$

The structure (graph) being:

s =



The correspondence below is established between the left side of the production, namely $A(x_1, \{x_2, x_3\})$ and the labeled set of nodes and node sets $A(x_1, \{x_4, x_5\})$ from the given graph s:

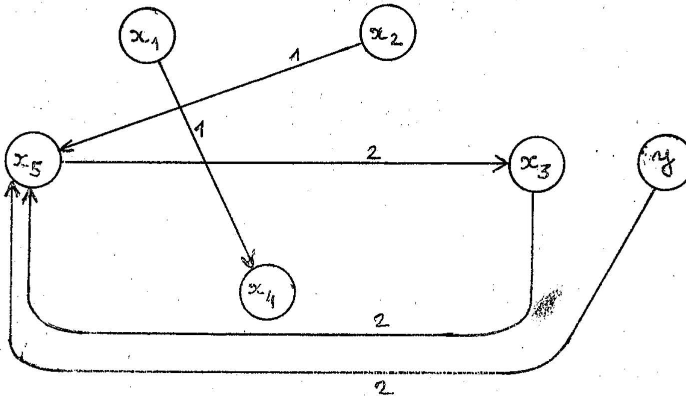
left side	graph
x_1	x_1
x_2	x_4
x_3	x_5

Now the transformation indicated in the right side is performed.

The fact taht node x_4 is in the right side but not in the left side means that a new node - call it y - must be added to the graph. The constraints must be satisfied, which means that new arcs are added:

right side	graph
$x_2 = x_1.1$	$x_4 = x_1.1$
$x_3 = x_4.2$	$x_5 = y.2$

The new (derived) graph is:



The other possible strategy - weak derivation - allows to take nodes already in the graph to correspond to nodes in the right but not in the left side of the production. In the example, instead of creating y the correspondence

right side	graph
x_4	x_3

could be established, and since we already have in the graph $x_5 = x_3.2$ the constraint $x_3 = x_4.2$ is already satisfied (does not require the addition of a new arc).

Other relevant works in this area are [26, 27, 28].

Since it is possible to represent data structures as graphs and since there are grammars to handle graphs the next question is to see if there are efficient algorithms to deal with graphs representing data structures, or perhaps some of the most useful classes of data structures. Here the concepts from the well-known string grammars are applicable.

It has been shown, for example, that the list structures of the usual list-processing languages correspond to context-free languages [25].

The following formalism describes this class of data structures:

1. A simple list $L(\Sigma, N)$ over the alphabet Σ and name set N, Σ and N being finite and nonempty, is a finite collection of triples $L(\Sigma, N) = \{(n_i, \sigma_i, \ell_i)\}\ 1 \leq i \leq m$ such that:

a. There is exactly one triple (n_i, σ_i, ℓ_i) so that $n_i = L$ (and $\sigma_i = \Lambda$), the head of the list; Λ is the termination symbol.

b. There is exactly one triple (n_i, σ_i, ℓ_i) so that $\ell_i = \Lambda$, the end of the list.

c. For every triple (n_i, σ_i, ℓ_i) other than the end, there is exactly one triple (n_j, σ_j, ℓ_j) of $L(\Sigma, N)$ so that $\ell_i = n_j$;

d. For every triple (n_i, σ_i, ℓ_i) other than the head there is a triple (n_j, σ_j, ℓ_j) so that $\ell_j = n_i$.

2. A list set $L(\Sigma, N)$ over the alphabet Σ and name set N is a finite set of simple lists $L(\Sigma, N) = \{L_1(\Sigma, N_1), L_2(\Sigma, N_2), \ldots, L_m(\Sigma, N_m)\}$, so that $N_i \subseteq N$ $(1 \le i \le m)$, $N_i \cap N_j = \phi$ if $i \ne j$ and $L = L_1 = L_2 = \ldots = L_m$. L is the name of the list set.

3. A list structure $L(\Sigma, N)$ over the alphabet Σ and name set N is a finite set of list sets, $L(\Sigma, N) = \{L_1(\Sigma', N_1), L_2(\Sigma', N_2), \ldots, L_k(\Sigma', N_k)\}$ so that:

a. For a unique i, $L_i = L$

b. $N_i \subseteq N$ for all $1 \le i \le k$.

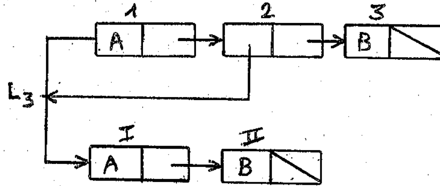
c. $N_i \cap N_j = \phi$, $i \ne j$

d. $\Sigma' = \Sigma \cup \{L_1, \ldots, L_k\}$

L is the name of the list structure.

From the definition of list structures it is easy to see that the head cell is a multi-link element, and that loops are allowed. An example given in [25] displays a list structure called L_3:

triple notation	LISP-like notation
$\{(L_3, \Lambda, 1)\ (L_3, \Lambda, I)$	
$(1, A, 2)\ (I, A, II)$	
$(2, L_3, 3)\ (II, B, \Lambda)$	
$(3, B, \Lambda)\}$	



We could also represent L_3 by the BNF definition:

$$< L_3 > ::= A < L_3 > B \mid A B$$

which shows that L_3 generates:

$$\{A^n B^n \mid n = 1, 2, \dots \}$$

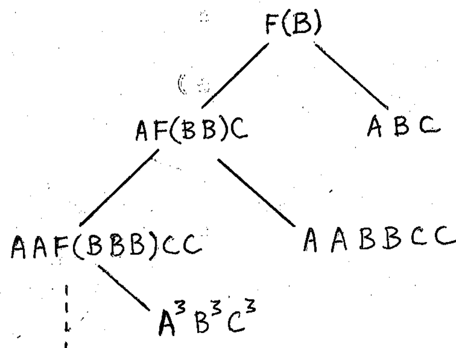
a well - known context-free language.

Another interesting result in [25] is that patterns, as in SNOBOL, under certain restrictions represent only context-sensitive languages, although relaxing the restrictions noncontext-sensitive languages can occur.

One of the features of SNOBOL patterns that extend their power beyond BNF patterns is the possibility of using functions inside the pattern. The context-sensitive (noncontext-free)language $A^n B^n C^n$, that is used commonly in the demonstration that ALGOL 60 is not context-free, can be generated by:

```
          &ANCHOR = 1
          DEFINE  ('F(X)')                       :(FEND)
F         Y = 'B' X
          F = 'A' X 'C' | 'A' * F(Y) 'C'         :(RETURN)
FEND
          ....................
          TESTPAT =  (*F('B') RPOS(0))
```

whose evaluation may be shown in the tree



This is an idication that the choice of SNOBOL to write algorithms implementing graph grammars is a sensible one; it was adopted in [24], where some polynomial algorithms are presented.

6. CONCLUSIONS

We have seen that several properties of data types and data structures can be described by means of algebraic formalisms. These properties can be classified tentatively as:

a. mathematical
b. structural, from a logical point of view
c. structural, in the sense of physical machine implementation

The mathematical properties have to do with the operations that can be done with values. They are affected by the physical representation available in a given machine, and concepts like the lattice approach to reals, and others not discussed here such as multi-precision, rational, and sample values representation are some of the important topics to consider.

The logical structural properties refer to the relations or relationships among data and to some very special aspects such as rules to add or to delete elements (as in stacks, queues, dequeues, etc.). It should be stressed that considering these properties without regard to physical implementation is a sound approach for certain purposes:

- it is immaterial from a logical point of view to determine that "links" in a list are to be understood as pointers, or subscripts (if an array of available space is used) or any other possible access mechanism; this is a very much machine dependent consideration that should be studied independently of the algorithm design phase;
- it is not immaterial however the decision on whether or not a structure should be rooted; although the physical storage mapping is one of the reasons to insist in rootedness, this

seems to affect any implementation in any of the present-day machines, and the algorithm-designer should be aware of that.

The physical structural properties are related to actual machine implementation. Here the concept of efficiency means speed for the operations of adding and deleting, searching, ordering, etc., and minimization of the required space; the two objectives usually conflict and a trade-off must be found in each case. The key concepts, as we have seen, are addressability and relocatability.

Clearly the physical implementation is affected by differences in hardware design and in the system's architecture (consider memory hierarchies, availability of associative memory, hardware stack, microprogramming, etc). It is also affected by the existing software(e.g. it may be difficult to implement complex data structures using FORTRAN, without resorting to machine language routines), in which category the interaction with the machine via an operating system must be included.

It has been noted that the appropriate choice of a data structure may be essential for ensuring a desired complexity measure in an algorithm; this was the case, for instance, in Tarjan's planarity algorithm where list processing is cleverly used. However the algorithm designer may find that his chosen logical data structure will result in oversized constants in the complexity formula when a given machine with a given software is used.

Coming back to logical data structures, we have seen that the most general data structures are sets (of atomic objects or of atomic objects and sets, recursively) with relations defined among them. More precisely they are rooted strongly connected directed graphs with labelled

nodes and arcs and tagged root.

Some modern programming languages already include sets and graphs, the most important examples being SETL for sets and AMBIT/G for graphs.

Here the problems are the non-ordered nature of sets as opposed to the ordering imposed by machine implementation, the representation of tuples, efficient ways to determine whether an element belongs to a set, how to represent links, access paths, etc.

It is interesting to note that set - theoretic languages permit the representation of graphs, the arcs being shown as ordered pairs. The relational approach goes even further, allowing the representation of n-ary unordered relations (relationships); certain restricted classes of graphs can in fact be normalized (linearized) and brought into the relationship representation.

In any case it seems important to distinguish between the storage mapping of a graph taking the root and the links into consideration, and run-time access to a given node. Should we go all the way from the root to the node at each time the node is to be accessed? This has to do with several devices such as symmetric lists, hash code, access functions, etc.; sometimes compile-time strategies are applicable.

After characterizing directed graphs as the most general data structure (and learning that they can be implemented in computers), we presented this concept in a formal algebraic notation. A nice aspect of the notation is that besides being descriptive it also provides a practical basis for devising a data definition facility - namely schemas and grammars.

Schemas are defined by imposing restrictions to general directed graphs. It is usual to define a tree by any of the restrictions below imposed to a graph G:

1. G is connected and contains no cycles.
2. G contains no cycles and it has n nodes and n-1 edges.
3. G is connected and it has n nodes and n-1 edges.
4. G contains no cycles and by adding and edge between two non-adjacent nodes exactly one cycle is formed.
5. G is connected and by suppressing any edge G is disconnected.
6. Every pair of nodes is connected through exactly one chain.

In a tree entry (since we are always dealing with directed graphs we should prefer the word arborescence; for the sake of simplicity such distinctions shall not be stressed) as in a CODASYL report [22] all elements except the root have exactly one parent. In the same report a plex entry extends tree entries by allowing any number of parents if exactly one edge corresponds to a "hierarchic group relation" and the other edges to "non-hierarchic group relations". Incidentally this shows how important is the ability to specify the labels of the edges (different labels meaning different relations over the same objects) when writing the restrictions.

Among the kinds of restrictions in the CODASYL report are those limitating the degree of certain nodes, forbidding loops, forbidding cycles, etc. Thus schema definitions should incorporate a great deal of terminology from graph theory.

The purpose of grammars is twofold:

1. to generate valid sentences of a language (in our case: to generate instances of a schema or to "populate" the schema [23]);

2. to recognize sentences as belonging to a language (by parsing an instance, finding if it corresponds to a schema). The CODASYL report is really describing a grammar when it refers to " a) rules for composing a schema; and b) rules for generating instances of a schema".

The existence of very serious research in graph grammars (in the area of pattern recognition) and the possibility of expressing restrictions in algebraic and graph - theoretic terms suggest that a data definition facility (in CODASYL terms a data definition language = DDL) could be devised in a fairly formal notation.

We shall not argue here on the "naturalness" of algebraic notation as opposed to say COBOL - like notation. A possible compromise would be to build a rigorous formal notation and then "sugar - coating" it in some natural language or programming language style.

By reasoning in formal terms when building the facility one could more easily and rigorously discuss meaning and correctedness; then, contrarywise to what happened with PL/I, only what can be justified in formal terms will be supported. However the other direction - all that can be justified is supported - is not a realistic goal.

Indeed in [25] mention is made to data structures that do not correspond to even context sensitive languages.

The realistic approach consists of taking only those data structures that are in fact useful. Even then the availability of several parsing techniques and the identification, before initiating the process, of helpful restrictions on the data structure would allow the selection of the "lowest" and hence most efficient parser in each case.

The formalism in section 4 allows us to define the data definition facility as an SMF (a storage mapping function). It could be written, in the style of a Floyd / Evans/Feldman Production Language [33] where the syntactic part would implement an adequate graph grammar and the semantic part would effectuate the storage allocation activities.

Since binding uses to occur at different times in modern programming languages, notably at compile and at execution time, the data definition facility should be continuously available to be invoked. It would be used not only to allocate storage to a given valid instance of a schema but also to allocate and free space as the instance undergoes valid additions and deletions.

Some preliminary work, hopefully interesting both from a theoretical and a practical viewpoint would consist of:

1. Adopting and possibly extending some graph grammar, as adequate to describe the usual data structures.

2. Expressing such data structures in the chosen grammar.

3. Designing the syntax of a data definition facility.

4. Comparing parsing algorithms for the data grammar.
5. Comparing storage allocation strategics.

In items 4 and 5 the declared restrictions (or constraints [24], or attributes [22], or assertions [5]) would be taken into account, in order to select the most efficient parser - in item 4, and also the best machine implementation - in item 5. We repeat here that the word best refers to a trade - off between speed and space and that it is a machine dependent consideration.

It is probable that at first only very restricted data structures will be conveniently treated, the higher types still being handled by more empirical methods. Hopefully, the work that is being done in the area will produce formalisms with more descriptive power and adaptable to practical usage.

REFERENCES

* [1] - Mc Carthy, J. - "A Basis for a Mathematical Theory of Computation" - in "Computer Programming and Formal Systems" edited by Brafford, P. and Hirshberg, D. - North Holland, 1963.

* [2] - Mealy, G. - "Another Look at Data" - Proceedings of the Fall Joint Computer Conference, 1967.

* [3] - Scott, Dana. - "Outline of a Mathematical Theory of Computation" - Proceedings of the Princeton Conference on Information Sciences, and Systems, 4th, 1970.

* [4] - Childs, D. - "Description of a set-theoretic data structure" Proceedings of the Fall Joint Computer Conference, 1968.

* [5] - Laski, F. - "The Morphology of prex-an Essay in Meta-Algorithms"- in Machine Intelligence 3 - edited by Michie, D. - American Elsevier, 1968.

* [6] - Codd, E. - "A Relational Model of Data for Large Shared Data Banks" - CACM 13/6 June 1970.

[7] - Kapps, C. - "Sprint - A Programming Language with General Structure" - Moore School Report nº 71- 78, 1970.

* [8] - Standish, T. - "A Data Definition Facility for Programming Languages" - Carnegie Institute of Technology, 1967.

[9] - Sibley, E. and Taylor, R. - "Logical Structure to Physical Storage Mapping in Data Base Management Systems" - University of Michigan, 1971.

[10] - Nelson, E. - "A Mathematical Theory of Data Structures" - private communication.

* [11] - Childs, D. - "Feasibility of a Set Theoretic Data Structure based on a Reconstituted Definition of Relation" Proceedings of IFIP Congress 68 - North Holland, 1969.

* [12] - Rosenberg, A. - "Data Graphs and Addressing Schemes"- Journal of Computer and System Sciences - vol. V, June 1971.

* [13] - Earley, J. - "Toward an Understanding of Data Structures" - CACM 14/10/ October 1971.

* [14] - Gotlieb, C. - "Data Structure Definition" - University of Toronto, 1972.

* [15] - Wegner, P. - "Data Structure Models for Programming Languages" Proceedings of a Symposium on Data Structures in Programming Languages - University of Florida, 1971.

[16] - Schwartz, J. - "Set Theory as a Language for Program Specification and Programming" - New York University, 1970.

[17] - Busacker, G. and Saaty, L. - "Finite Graphs and Networks" - McGraw-Hill, 1965.

[18] - Mc Lane, S. and Birkhoff, G. - "Algebra" - Mc Millan, 1968.

[19] - Wirth, N. - "The Programming Language Pascal" - Eidgenössische Technische Hochschule, 1970.

[20] - Pratt, W. and Friedman, D. - "A Language Extension for Graph-Processing and its Formal Semantics" - Comm. ACM 14. 460-467, 1971.

[21] - Data Base Task Group - April 71 - Report - CODASYL - 1971.

*[22] - Feature Analysis of Generalized Database Management Systems - CODASYL, 1971.

[23] - Schwebel, J.-"Graph Transformations for Composite Formation" University of Illinois, 1969.

*[24] - Mylopoulos, J. - "On the Relation of Graph Automata and Graph Grammars" - University of Toronto, 1971.

*[25] - Fleck, A. - "Towards a Theory of Data Structures"- Journal of Computer and System Sciences - 5, October, 1971.

[26] - Pfaltz, J. and Rosenfeld - "Web Grammars" - Technical Report 69 - 84 - University of Maryland, 1969.

[27] - Shaw, A. - "Parsing of Graph-Representable Pictures" - JACM 17 , 3, July 1970.

[28] - Schwebel, J . - "Graph Transformations for Composite Formation" Report nº 368 - University of Illinois at Urbana-Champaign 1969.

[29] - Corneil, D. - "Graph Isomorphism" - Doctoral thesis - University of Toronto, 1970.

[30] - Tarjan,R.-"Depth - First Search and Linear Graph Algorithms" Stanford University - working paper.

[31] - Earnest, K. et al - "Analysis of Graphs by Ordering of Nodes" JACM - 19 , 1, January 1972.

[32] - Christensen, C. - "An Example of the Manipulation of Directed Graphs in the AMBIT/G programming Language" - in "Interactive Systems for Experimental Applied Mathematics", Academic Press 1968.

[33] - Feldman, F. - "Formal Semantics for Computer Languages and its Application in a Compiler - Compiler" - CACM 9, 1, January 1966.

[34] - Collins, G. - "A Method for Overlapping and Erasure of Lists" CACM 3 , 1960.