MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Computation Structures Group Memo No. 52

On the Exchange of Information*

Jack B. Dennis

October 1970

*Prepared for the 1970 ACM-SICFIDET Workshop on Data Description and Access, Rice University, Houston, November 15-16, 1970.

On the Exchange of Information

Jack B. Dennis

Project MAC
M.I.T., Cambridge, Massachusetts 02139

October 16, 1970

There is widespread and increasing interest in moving information between computer installations which may differ in their hardware, software, or their program libraries and data files [1,2,3]. There is general agreement that the task of making a major program written at one installation available for use at another installation is difficult, and is not seriously undertaken without thorough analysis. Yet there seems to be considerable optimism that the problem of accessing a data base at a foreign installation can be solved through development of a "data description language" that would serve to characterize any class of data objects that might be communicated among computer installations [4]. The purpose of this paper is to argue that the general problem of data exchange is no less difficult than the problem of program exchange, and that the concept of a "data description language" is not a solution to the problem.

The Problem

To express the general problem of data base transfer, consider two contexts within which procedures may be executed so as to accomplish computations on behalf of users. Let these be known as context A and

This research was done at Project MAC, M.I.T., and was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01), and in part by the National Science Foundation under grant GJ-432.

context B. By the term "context" we mean the collection of all those factors that determine the detailled course of execution of computations performed by a computer installation. The factors defining a context for execution of a procedure P are normally these:

1. The programming language in which P is expressed for presentation to the computer installation.
2. The compiler for the programming language in which P is expressed.
3. The computer hardware in which the compiled form of P is run.
4. The file manipulation and communications services provided by the operating system of the installation.
5. The set of rules used by the installation for binding external references contained in P.
6. The status of catalogs or directories of data and procedures; the files themselves; the user account under which P is executed.

A difference in any of these factors may cause the procedure P to have different effect when performed at two installations.

Now let us formulate the problem. A data base D has been created by procedures operating in context A. As shown in Figure 1a, let Q be representative of these procedures. It is desired to make use of D in a distinct context B. We assume that "make use of D" means that a user wishes to carry out computations in context B in which certain procedures either retrieve information from D or alter the content of D. Let P be representative of these procedures (Figure 1b).

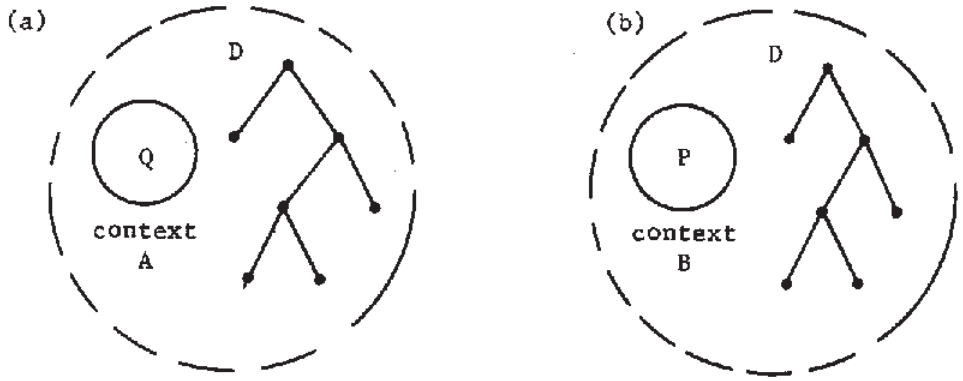


Figure 1.

Now in the users mind P, Q, and D are objects that have existence independent of any computer system: P and Q are abstract procedures; D is an abstract data object. Procedure Q must be expressed in a programming language and compiled before it is ready for execution in context A. Let the compiled form of Q be Q_A some representation of Q in context A. Operation of Q_A and related procedures in context A produces D_A a representation of D according to the data types and compiling conventions of context A.

To apply procedure P to structure D in context B requires that P be expressed in a programming language and compiled into P_B, and that D be translated into D_B, where P_B and D_B are representations of P and D in context B.

The problem is to ensure that the effect of executing P with D in context B will be "consistent" with the effect of executing Q to generate D in context A. To be more specific we assume that the effect desired by the user of P is exactly the effect that would result from applying P to a copy of D in context A: The desired effect is the result that would be

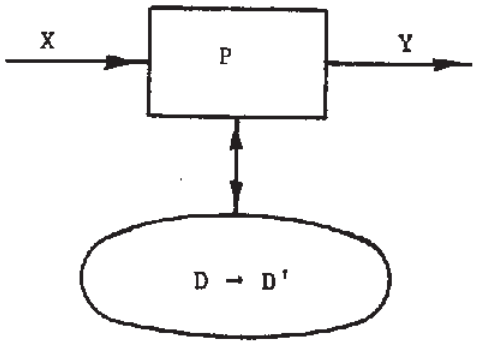


Figure 2.

obtained by expressing P in the same language as Q, translating it with the same compiler, and running it in the same operating environment as P except for use of a copy of D. Thus we adopt the following as our criterion for successful achievement of program and data transfer between two contexts:

> Application of P to D in context B must have the same effect as though P were applied to a copy of D in context A. If this property holds for all procedures P and all structures D we say that contexts A and B are consistent.

For investigating the conditions necessary for two contexts to be consistent, we offer the following interpretation of the phrase "have the same effect." Suppose the diagram in Figure 2 is an adequate model for procedure P:

> D represents the data base prior to execution of procedure P.
>
> D' represents the data base after execution of P.
>
> X denotes data (other than D) accessible to P but not altered by P (input data).
>
> Y denotes data (other than D') that execution of P generates (output data).

Example: The procedure P might be an information retrieval program that responds to a command message X by producing an augmented file D' from D and generating an output message Y.

By considering the data objects X, Y, and D or D' to be components of a single compound data structure, we may think of procedure P performing the transformation on this structure illustrated by Figure 3. Thus the computation of Figure 2 is a special case of the computation in Figure 4 where D and D' may be arbitrary data structures. We shall use the model of Figure 4 for our discussion of consistency.

Let $\mathcal{D}$ be the (infinite) class of abstract data structures from which D is chosen. Similarly, let $\mathcal{P}$ be the (also infinite) class of abstract procedures from which P is chosen. Each abstract structure $D \in \mathcal{D}$ and each program $P \in \mathcal{P}$ will have one or more representations in each context. We define several relations that give the correspondence

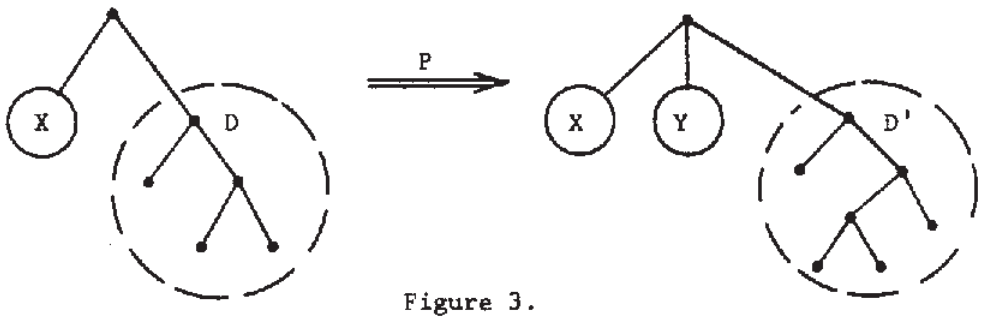


Figure 3.

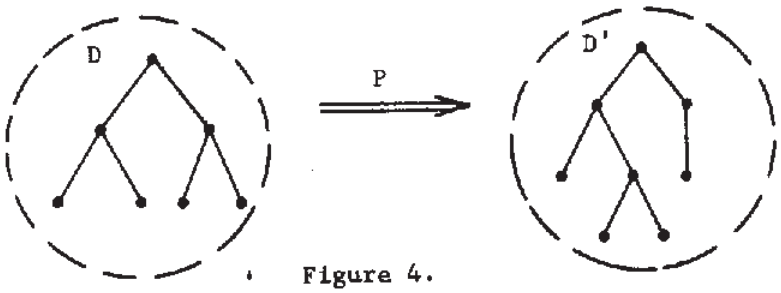


Figure 4.

of representations to abstract objects:

1. Let the relation

$$C_A \subseteq \mathcal{D} \times \mathcal{D}_A$$

contain each pair (D, D_A) such that D_A is a representation of D in context A. We assume that the range of C_A includes all of $\mathcal{D}_A$; that is, each element of $\mathcal{D}_A$ is a representation of some object in $\mathcal{D}$.

Let

$$C_B \subseteq \mathcal{D} \times \mathcal{D}_B$$

be similarly defined for context B. We denote by C_A^* and C_B^* the converse relations to C_A and C_B.

2. Let the relations

$$t_A \subseteq \mathcal{P} \times \mathcal{P}_A$$

$$t_B \subseteq \mathcal{P} \times \mathcal{P}_B$$

give the correspondence of each abstract program $P \in \mathcal{P}$ to its representations in $\mathcal{P}_A$ and $\mathcal{P}_B$. Again we suppose, for example, that the range of t_A is all of $\mathcal{P}_A$.

3. Each abstract program $P \in \mathcal{P}$ is an operation on abstract structures:

$$P: \mathcal{A} \rightarrow \mathcal{A}$$

We assume the domain of any program to be all of $\mathcal{A}$; the result of applying P to invalid data may be indicated by a "success" or "failure" bit in the output data structure. We ignore the possibility that P may loop or contain bugs. Each representation of a program is an operation on representation of data structures:

$$P_A: \mathcal{A}_A \rightarrow \mathcal{A}_A$$

$$P_B: \mathcal{A}_B \rightarrow \mathcal{A}_B$$

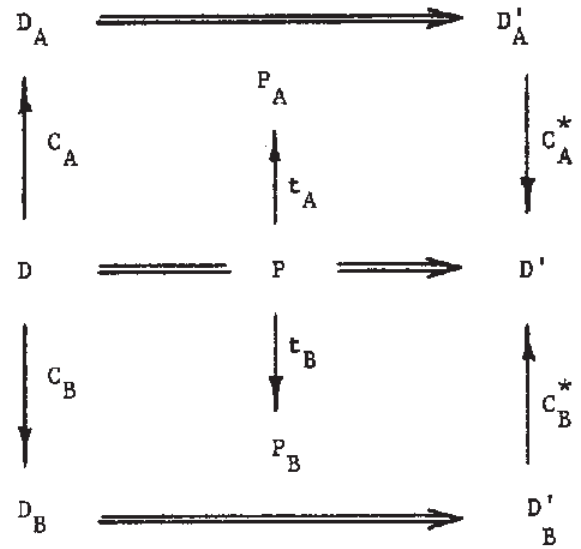


Figure 5.

The consistency of contexts A and B is characterized in terms of these definitions by Figure 5, where P is an arbitrary procedure and D is an arbitrary abstract data object. The diagram expresses the requirement that the result of applying P to D in either context A or B must result in some representation of the unique data structure $D' = P(D)$. For consistency, this condition must hold for any choice of $P \in \mathcal{P}$ and $D \in \mathcal{D}$. In symbols, consistency requires that the two compositions of relations

$$F_A = C_A \circ P_A \circ C_A^*$$

and

$$F_B = C_B \circ P_B \circ C_B^*$$

be identical functions where

$$(P, P_A) \in t_A \qquad \text{and} \qquad (P, P_B) \in t_B$$

and P is any member of $\mathcal{P}$.

For F_A to be a function it is necessary that the relation C_A^* be a function. To show this we must assume that each representation $D_A \in \mathcal{D}_A$ can be reached by computation in context A for some choice of P and D.

Consider any $D_A \in \mathcal{D}_A$ and suppose

$$(D_1, D_A) \in C_A, \ (D_2, D_A) \in C_A, \ D_1 \neq D_2$$

Because D_A is reachable we may choose P and D such that

$$(D, D_A) \in C_A \circ P_A \text{ where } P_A = t_A(P)$$

But then

$$(D, D_1) \in F_A, \ (D, D_2) \in F_A$$

contradicting the assumption that $F_A = C_A \circ P_A \circ C_A^*$

is a function.

Similarly, C_B^* must be a function. Thus C_A and C_B partition $\mathcal{A}_A$ and $\mathcal{A}_B$ into equivalence classes -- each equivalence class consisting of all representations of some data object in context A or context B, respectively.

Now let us consider the requirements that consistency imposes on the representation of a procedure (Figure 6). Suppose procedure P transforms D into D'. Then the implementation of P in context A may be applied to any representation in the set

$$\alpha = \{D_A \mid (D, D_A) \in C_A\}$$

and, in each case, must produce a representation in the set

$$\beta = \{D_A' \mid (D', D_A') \in C_A\}$$

if the consistency condition is to be satisfied. The conclusion is thus:

> For two implementations of a procedure to be consistent, they must perform the identical transformation of the corresponding data equivalence classes of their contexts.

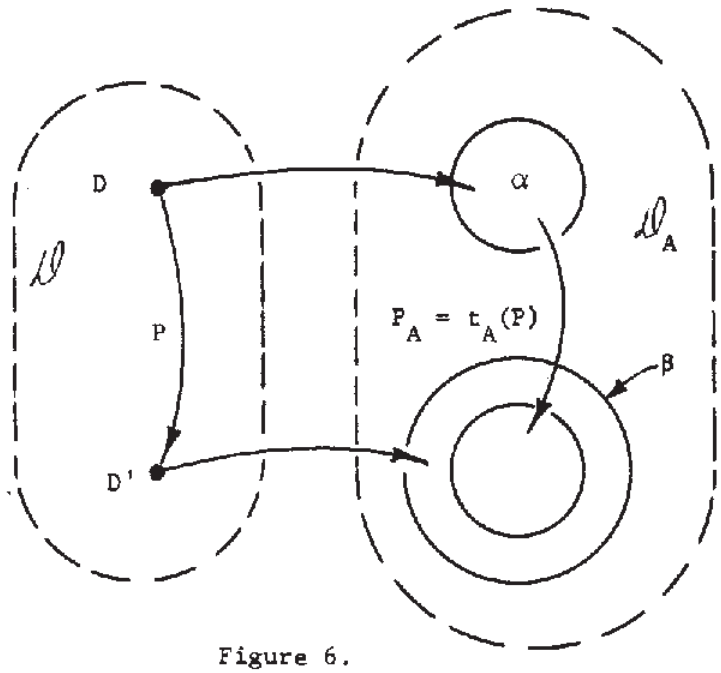


Figure 6.

Computations on Structured Data

It is interesting to examine the consequences for consistency of further assumptions regarding the structure of programs. In this section we consider the consequences of adopting a specific formal model for structured information in computer systems. According to our model [5,6], a data structure has two parts — first, a collection of elementary objects; and second, a directed graph that specifies the manner in which elementary objects are selected and accessed.

The important subclasses of elementary objects (primitive data types) are fairly well established:

data type	range
truth values	{true, false}
integers	{0, ±1, ±2, ...}
reals	power set ({integers})
bit strings	$\{0, 1\}^*$
character strings on some alphabet V	V^*

Since in practice the set of data representations $\mathcal{D}_A$ in context A is always a countable set, the abstract class of "reals" can be at most countable if our condition for consistency is to be met.

Since each elementary object must be considered as a possible data structure, the conditions developed above must apply to the primitive transformations of elementary objects used in constructing programs in the class $\mathcal{F}$.

Example: A simple example of the concepts just discussed is the treatment of zero by computer hardware. In certain computers zero has two representations, +0 and -0. Moreover, the rules of arithmetic are not uniformly interpreted where both operands are zero. For instance, the rules

$$(-0) + (-0) = -0 \quad \text{and} \quad (-0) + (-0) = (+0)$$

have both been used in different contexts. Nevertheless, the two contexts may be consistent provided +0 and -0 belong to the same equivalence class in each context -- that is, they are both representations of a unique abstract object zero. If the class of abstract programs allows a test that distinguishes +0 from -0, the contexts are not consistent.

The structural part of a data structure provides a logical means of making reference to a particular elementary object within the structure. Operationally, computer systems implement primitive operations such as indexing and table searching that permit these references to be coded in procedures. Two types of primitive operations are provided -- those that permit access to and alteration of elementary objects without changing their number or arrangement, and those that alter the structure (by adding, deleting, or rearranging elementary objects). To be more specific, it is useful (and conventional) to represent a data structure by a directed graph. Each elementary object of the data structure is associated with some node of the graph; these nodes are drawn as circles with the elementary object written inside. Each branch of the graph of a data structure is labelled with a selector that distinguishes the branch from other branches originating from the same node.

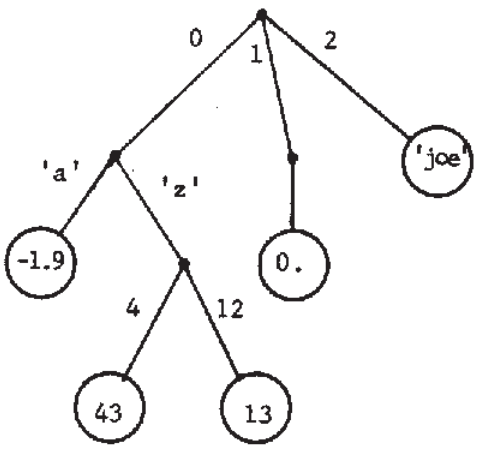


Figure 7.

Many workers have restricted the graphs of data atructures to be trees, while others have argued for general graphs including graphs having arbitrary cyclic structure. My preference is for directed acyclic graphs having unique roots. The reasons for this choice concern access control, concurrent processing, and the sharing of access to substructures. Figure 7 is an example of a data structure.

A pointer is a variable whose value may range over the nodes of all data structures in a computer system. A pointer value is thought of as a token that denotes the data structure having the node identified by the pointer as its root node, and consisting of all nodes and branches reachable over directed paths from the root. We assume (as in practice) that access to a data structure by a procedure is made possible by a pointer argument to the procedure. The procedure may obtain a pointer to any node of the data structure by repeated execution of a primitive operation called select:

> select p, n → q: If a branch labelled n emanates from the node designated by pointer p, then pointer q is given as value the

node on which this branch terminates. Otherwise a new branch labelled n that terminates on a new node is created, and this node becomes the value of q.

Two other primitives are delete and link.

delete p, n: The branch labelled n emanating from node p is erased. Any substructure that becomes detached evaporates.

link p, n, q: A new branch labelled n replaces a previous branch labelled n emanating from node p (if any), and the new branch terminates on node q.

Computation is performed by assignment statements like

$$w \times (x + y) \rightarrow z$$

in which the variable letters denote pointers to nodes having associated elementary objects.

Different implementations for computation on structured data make use of various machine-level representations, and different choices of primitive operations. Examples are: LISP where address-linked cells are used to represent binary trees and the basic operations are car, cdr, cons, atom, equal; "structures" in PL/1 where selection is limited to integer subscripting and table look-up is used; COBOL indexed files where directories and hash-coded searching are usual.

A procedure that operates on structured-data is a logical arrangement that specifies how execution of a set of primitive operators is to be sequenced. We now consider the relation of such an abstract procedure P to P_A, any representation of P in some context A. We claim that each

primitive operator in P must be translated by t_A into a representation that effects a consistent transformation of data objects in context A. To justify this claim, consider the following example: An abstract procedure P accepts as input a data structure x_0 designated by pointer P_0. Under direction of an input message P performs a series of <u>select</u> operations to obtain a pointer p_k to a substracture x_k of x_0. This structure is given to procedure Q which examines the structure x_k and generates an output message. The computation performed is illustrated by Figure 8.

According to our earlier argument, to the abstract structures $x_0, \ldots, x_k$ there correspond equivalence classes of representations $\alpha_0, \ldots, \alpha_k$ in context A as shown in Figure 9. Our claim requires that

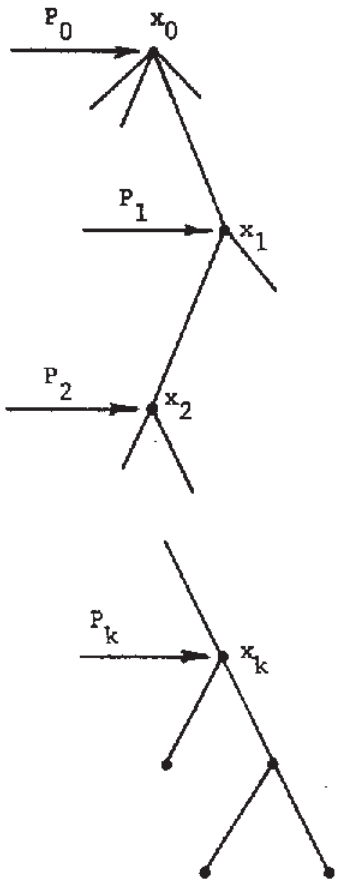


Figure 8.

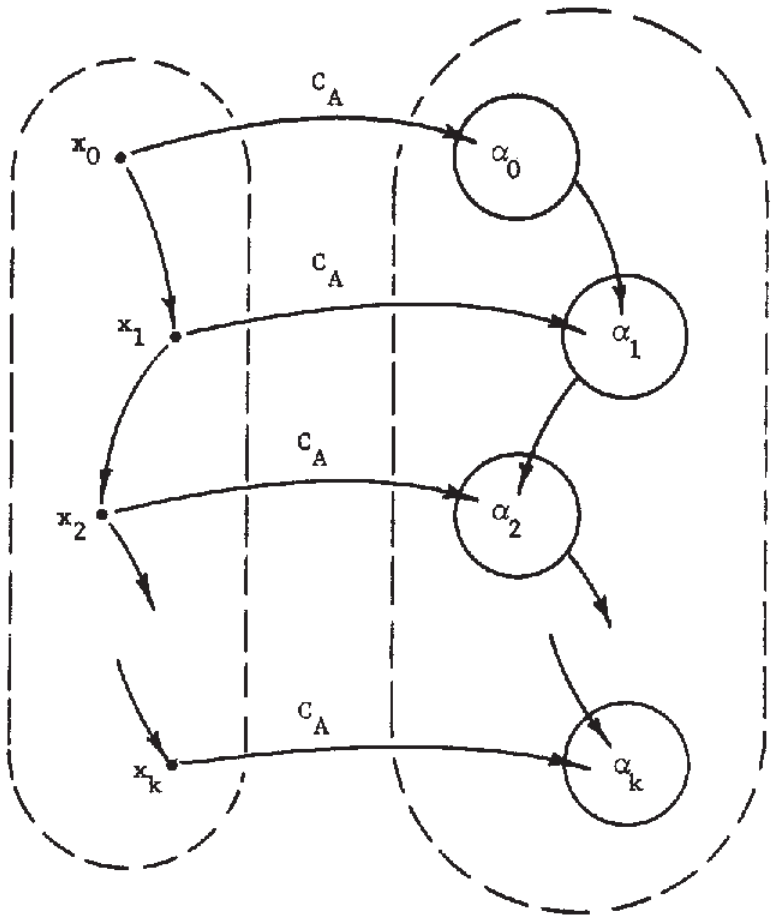


Figure 9.

the representations of the select operations in context A must map between the successive equivalence classes as shown in the figure. This must be true, for otherwise we could determine a sequence of selectors to present to P such that P's output could not be guaranteed to be the correct substructure of x_0.

Data Description

We regard a data description language as a way of setting down in a finite number of symbols, a specification of a class of abstract data structures. Implicit in all discussions of data description we have seen is the concept of a (usually infinite) universe U of all possible data structures. If L is a data description language then each sentence of L is a formal specification of some (finite or infinite) subset of U. As an illustration (and possibly as a useful proposal) we outline a data description language L based on the universe of tree-like data structures introduced earlier and some "definition schemata" used by the staff of the IBM Vienna Laboratory in their work on formal definition of programming languages [7].

Each sentence in the language L will specify some subclass of data structures in the universe U consisting of all data structures that can be formed from elementary objects in the class E and selectors in the class S. For our examples of the use of L we will let E consist of the types of elementary objects introduced above:

$$E = \underline{\text{integer}} \cup \underline{\text{real}} \cup \underline{\text{truth-value}} \cup \underline{\text{string}}$$

Elementary objects that are integers or strings may occur as selectors in objects in the universe.

$$S = \underline{\text{integer}} \cup \underline{\text{string}}$$

The elementary objects will be represented as follows:

integers	by their Arabic numerals
reals	numbers with radix print
truth values	{true, false}
strings	strings in V^* enclosed by single quotes. $V = \{a, b, \ldots, z, 0, \ldots, 9, +, -, \times, /\}$

A data description in L consists of a finite list of declarations, each of which defines a class of data structures in terms of simpler classes of structures. We denote the classes of structures involved in a data description by underlined words. Each definition starts from the classes of elementary objects and the class of selectors.

Each declaration has one of five forms characterized by the five "definition schemata" introduced below. For each scheme of definition, we give the general form of its syntactic construction, and then a statement of its meaning. The letters W, W', W_1, W_2, ... stand for the names of classes of structures, the letters s, s_1, s_2, ... stand for selectors (elements of S); Q denotes some subset of S; e, e_1, e_2, ... denote elementary objects.

The first definition schema allows the specification of restricted classes of elementary objects, including finite classes.

1a. $W \equiv \{e_1, e_2, \ldots, e_k\}$

Each elementary object e_i, $1 \leq i \leq k$, is a member of the class W

1b. $W \equiv \{e \in W' \mid p(e)\}$

Each elementary object in the class W' that satisfies the predicate p is a member of W.

1c. $W \equiv [i, j]$ where $i, j \in$ integer

A shorthand notation for $W \equiv \{i, i + 1, \ldots, j - 1, j\}$.

Examples:

day ≡ {'monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday', 'sunday'}

range ≡ {e ∈ real | 0 ≤ e ≤ 1.}

precedence ≡ [1, 10]

The second schema provides a means of stating that each structure in a class W must have exactly k components identified by specified selectors, where each component is in a specified class.

2. $W \equiv (\langle S_1; W_1\rangle, \langle S_2; W_2\rangle, \ldots, \langle S_k, W_k\rangle)$

The class W contains all data structures of the form shown in Figure 10.

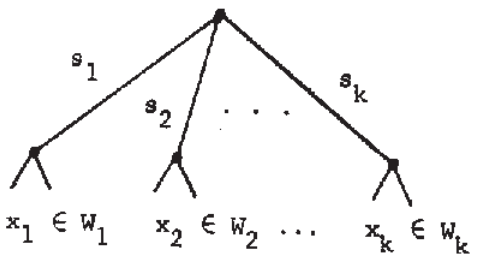


Figure 10.

Example:

record ≡ (⟨'name'; string⟩, ⟨'age'; integer⟩, ⟨'sex'; sex⟩, ⟨'salary'; integer⟩)

sex ≡ {'male', 'female'}

A data structure in the class record is shown in Figure 11

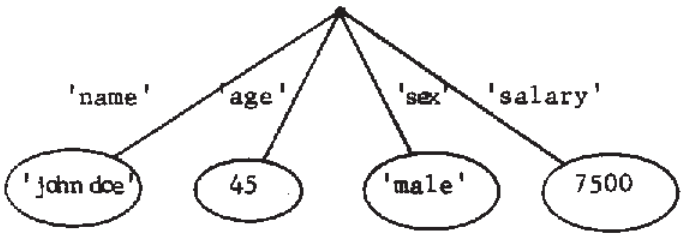


Figure 11.

The union symbol of logic is used to specify a class of structures formed by joining several classes.

3. $W \equiv W_1 \cup W_2 \cup \ldots \cup W_k$

A structure x is in W if and only if x is in some W_i, $1 \leq i \leq k$.

Example: By using definition schemata 2 and 3 together, we can specify a class of recursively formed structures having unbounded depth.

stack ≡ rest ∪ last

rest ≡ (⟨'value'; real⟩, ⟨'next'; stack⟩)

last ≡ (⟨'value'; real⟩)

Figure 12 shows a data structure in the class stack.

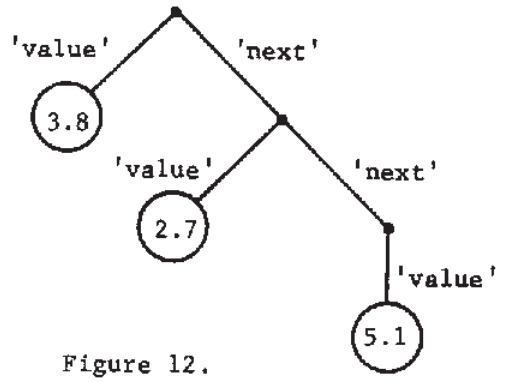
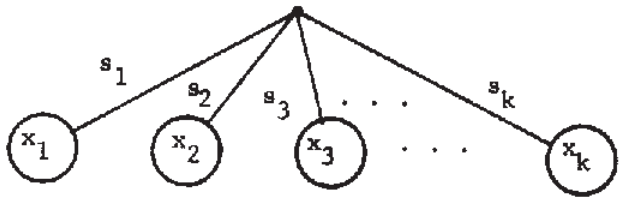


Figure 12.

Schema 2 only permits definition of structure classes in which the number of components under a node of a structure is bounded. Also, each selector that appears in a structure defined by schemata 1, 2, 3 must appear in the definition. The fourth definition schema overcomes these restrictions by specifying the construction of data structures from stated sets of selectors and structures.

4. $W \equiv \{\langle S;\ W' \rangle \mid S \in Q\}$

   Each member of W is a structure having the form shown in Figure 13 for some integer k.



$\{s_1, s_2, \ldots s_k\} \subseteq Q,\ s_i \neq s_j$ if $i \neq j$

$\{x_1, x_2, \ldots x_k\} \subseteq W'$ (not necessarily distinct)

Figure 13.

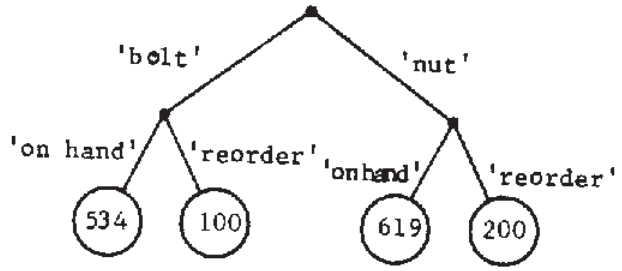


Figure 14.

Example:

inventory ≡ {⟨part-name; part record⟩ | part-name ∈ string}

part record ≡ {⟨'on hand'; integer⟩ | ⟨'reorder'; integer⟩}

Figure 14 shows a data structure in the class inventory.

Finally, schemata 1 through 3 do not give us the ability to specify that the components of a structure are selected by arbitrarily many consecutive integers. This is essential for specifying vectors and arrays of unspecified dimension. In the following J, J_1, and J_2 stand for arbitrary sets of integers.

5a. $W \equiv [W']$

5b. $W \equiv \langle [J_1, J_2]; W' \rangle$

5c. $W \equiv \langle [J_1, *]; W' \rangle$

5d. $W \equiv \langle [*, J_2]; W' \rangle$

5e. $W \equiv \langle J; W' \rangle$

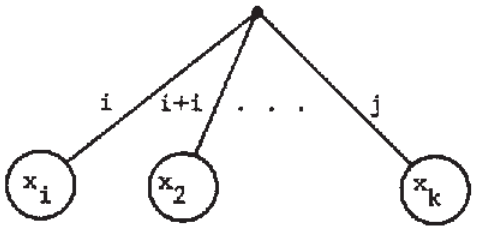


Figure 15.

Schemata 5a through 5d define classes of structures as shown in Figure 15, where $i \leq j$ and

5a: $i, j \in \underline{\text{integer}}$

5b: $i \in J_1, j \in J_2$

5c: $i \in J_1, j \in \underline{\text{integer}}$

5d: $i \in \underline{\text{integer}}, j \in J_2$

Schema 5e is shorthand for

$$W \equiv (\langle j_1; x_1\rangle, \langle j_2; x_2\rangle, \ldots, \langle j_k; x_k\rangle)$$

where

$$\{j_1, j_2, \ldots, j_k\} = J$$

$$\{x_1, x_2, \ldots, x_k\} \subseteq W'$$

Example (symbol table):

$$\underline{\text{symbol}} \equiv \{x \in \underline{\text{string}} \mid 1 \leq |x| \leq 6\}$$

$$\underline{\text{value}} \equiv \underline{\text{integer}} \cup \underline{\text{string}} \cup \underline{\text{real}}$$

$$\underline{\text{entry}} \equiv (\langle 0; \underline{\text{symbol}}\rangle, \langle 1; \underline{\text{value}}\rangle)$$

$$\underline{\text{table}} \equiv \langle [1, *], \underline{\text{entry}}\rangle$$

A data structure in the class table is shown in Figure 16.

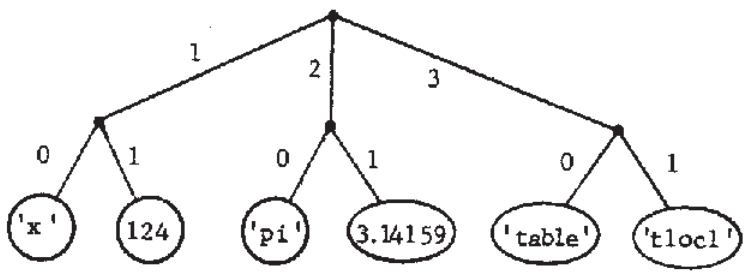


Figure 16.

The following example illustrates a more elaborate recursive definition using schemata 1, 3, and 5a.

Example (abstract arithmetic expressions):

expr	≡	sum ∪ difference ∪ product ∪ quotient ∪ real
sum	≡	(⟨'ratr', '+'⟩, ⟨'rand', term-list⟩)
term-list	≡	⟨[1, *]; term⟩
term	≡	difference ∪ product ∪ quotient ∪ real
difference	≡	(⟨'ratr', '-'⟩, ⟨'rd1', expr⟩, ⟨'rd2', expr⟩)
product	≡	(⟨'ratr', 'x'⟩, ⟨'rand', factor-list⟩)
factor-list	≡	[factor] ⟨[1, *]; factor⟩
factor	≡	sum ∪ difference ∪ quotient ∪ real

This definition is illustrated by Figure 17.

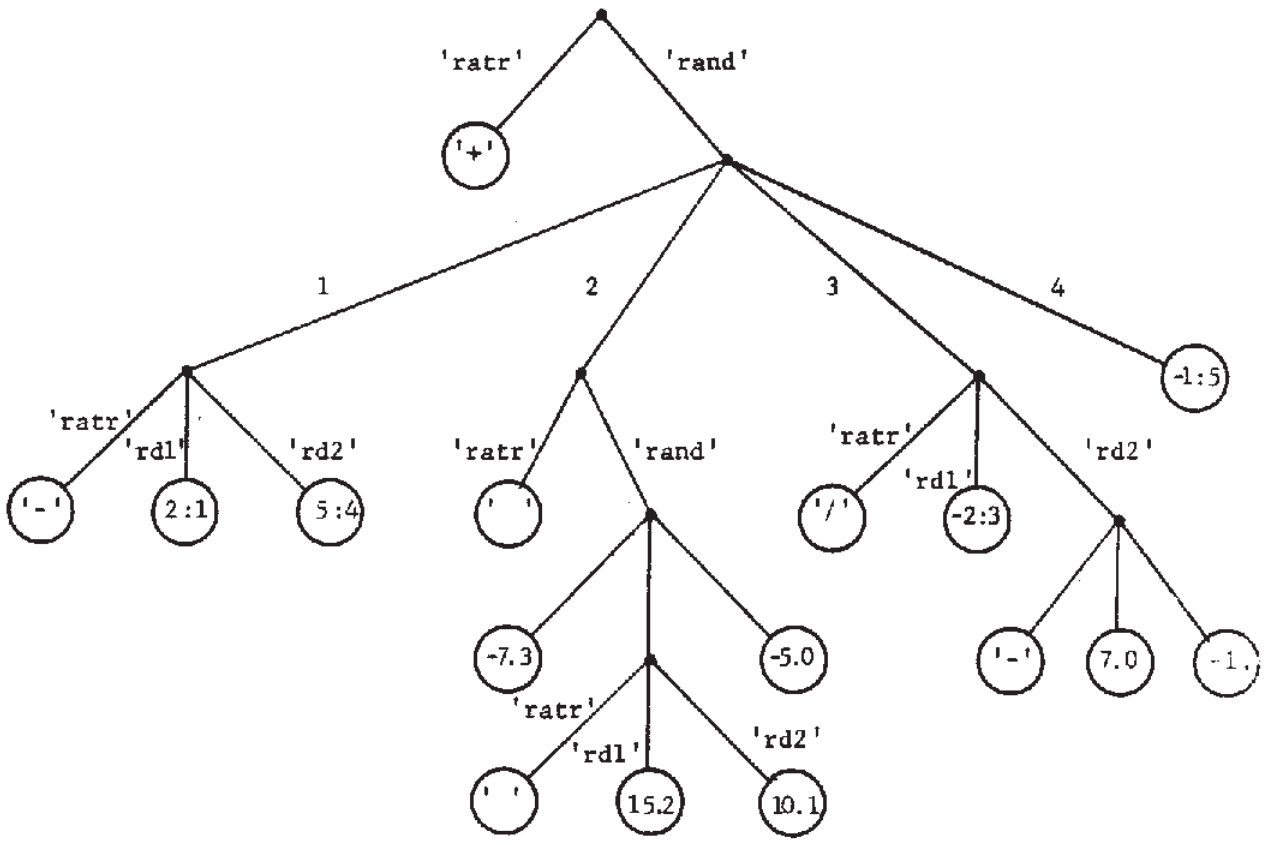


Figure 17.

The classes of data structures that may be defined using schemata 1 through 5 are limited in that it is not possible to specify that two substructures must have arbitrary but equal numbers of components. A convenient way to remedy this limitation is to associate integer-valued parameters with class names and permit these parameters or simple arithmetic expressions to appear in place of integers anywhere in the right parts of class declarations.

Example (triangular matrix):

$$\underline{\text{trimatrix}}(m) \equiv [\langle i;\ \underline{\text{row}}(m-i+1)\rangle \mid i \in [1, m]\}$$

$$\underline{\text{row}}(j) \equiv \langle [1, j];\ \underline{\text{real}}\rangle$$

This definition is illustrated by Figure 18.

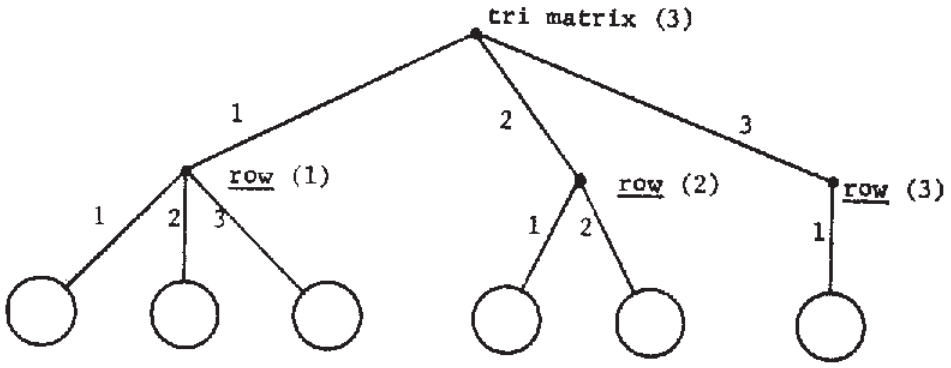


Figure 18.

All structure classes so far defined have graphs that are strictly trees: No method has been provided by which classes of structures that contain shared substructures may be specified. It seems that the most convenient way of providing this missing capability is by permitting chains of selectors to appear in the right parts of schemata. For instance, the declaration

$$\text{matrix}\ (m, n) \equiv \{\langle \text{'row'} \cdot [1, m] \cdot [1, n];\ e\rangle,\ \langle \text{'col'} \cdot [1, n] \cdot [1, m];\ e\rangle \mid e \in \underline{\text{element}}\}$$

specifies a class of rectangular matrices whose elements may be accessed either as elements of rows or as elements of columns. Our final example shows how the full class of directed graphs might be represented by a class of data structures convenient for implementing the common procedures used with graphs.

Example (directed graphs):

$$\underline{\text{graph}}(n) \equiv \{\langle \text{'orig'} \cdot i \cdot j;\ b\rangle, \langle \text{'dest'} \cdot j \cdot i;\ b\rangle \mid b \in \underline{\text{branch}}(i, j),\ i \in [1, n],\ j \in [1, n]\}$$

$$\underline{\text{branch}}(i, j) \equiv (\langle \text{'or'};\ i\rangle, \langle \text{'ds'};\ j\rangle, \langle \text{'da'};\ \underline{\text{branch-data}}\rangle)$$

A structure in the class graph (3) representing a graph of three nodes is shown in Figure 19.

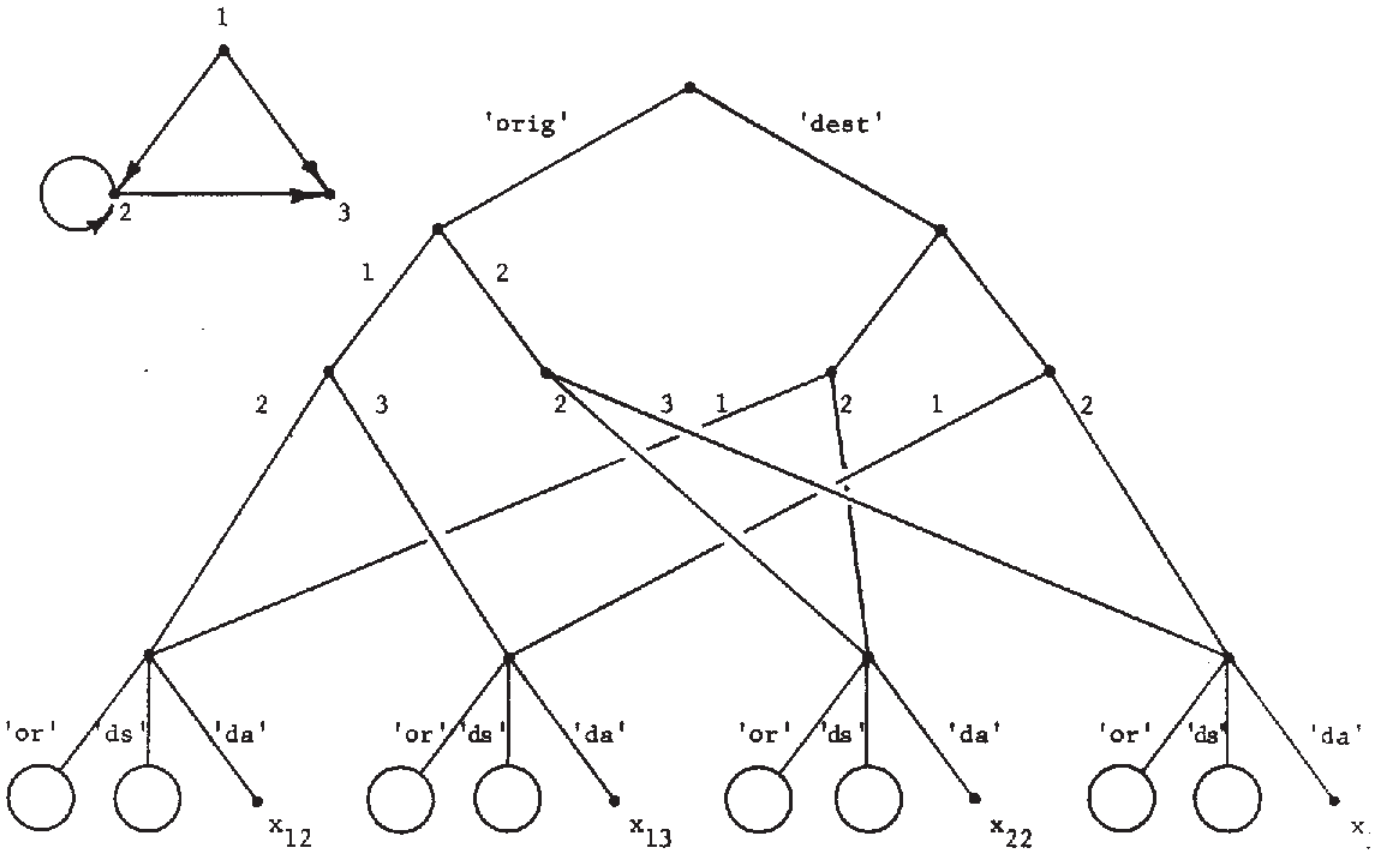


Figure 19.

Conclusion

For the purpose of moving data structures among computer installations, it will be necessary to define a "concrete representation" of a suitable class of abstract data structures for use as a common language for data interchange via communications networks. The examples in this paper have been chosen to demonstrate the merits of the class of abstract data structures.

I see two principal uses for a data description language of the kind suggested above. One of these uses is in translating data structures in one context into the common representation adopted for interchange. If the representations in the context do not contain type information for the elementary objects, or if the representation is ambiguous without additional information, then a formal specification of the class of data structurers may make it possible for a standard program to perform the translations. If complete type information is included as part of representations of data structures, then they may be converted into the common form without need for a data description.

The second use for a data description is in the context to which the data is moved. A description is not needed to translate from the common form to representations in the new context, because the common form should have complete type information. However, if the new context does not retain complete type information, a description may be useful to a general purpose program for retrieving information about transferred data structures.

Neither of these uses of a data description language waives the requirement of consistency of contexts between which data is moved.

Thus the concept of data description language is not a substitute for a universal representation for data structures and the procedures that operate on them. The eventual solution will be a common intermediate language used as a standard semantic base for assigning meaning to programs and information structures. We believe this is not an unreasonable goal: After all, System 360 machine language is currently serving this purpose for a large class of programs even though its qualifications for the role are far from ideal.

References

1. Naur, P., and Randell, B., Eds. Software Engineering, Scientific Affairs Division, NATO, Brussels 1969.

2. Proceedings of National Symposium on Modular Programming. Information and Systems Press, Cambridge, Mass., June 1968.

3. Dennis, J.B. A position paper on computing and communications. Communications of the ACM, Vol. 11, No. 5 (May 1968) pp. 370-377.

4. Cheatham, T.E., Jr. Data description in the CL-II programming system. Digest of Technical Papers, ACM National Conference, Assoc. for Computing Machinery, New York, September 1962.

5. Dennis, J.B., and Van Horn, E.C. Programming semantics for multiprogrammed computations. Communications of the ACM, Vol. 9, No. 3 (March 1966) pp. 143-155.

6. Dennis, J.B. Programming generality, parallelism and computer architecture. Information Processing 68, North Holland, 1969, pp. 484-492.

7. Lucas, P. and Walk, K. On the formal description of PL/I, Annual Review in Automatic Programming, Vol. 6, Part 3, Pergamon Press 1969.