PAPERS

DATA DESCRIPTION FOR DATA INDEPENDENCE*

W.C. McGee

My interest in this problem lies in the general area of what has been referred to as data independence. I would like to qualify that what I mean, and what most people mean when they use this phrase, is independence of the application programmer from the manner in which data are physically represented or stored. This sort of independence is useful for two reasons. First, it permits programmers to develop programs more quickly and accurately than they could if they had to be concerned with internal representation. This is just the property which is afforded by the so-called *higher-order languages*.

The second reason data independence is useful is that it permits applications programs, at least in source language form, to be insensitive to changes which are made to the physical data. For example, data can be moved from one type of storage to another or be represented in a completely different way without impacting the application programs. In the so-called *data-base application area*, this type of independence will become increasingly important, since application programmers are all sharing a common set of data and there is a rather large investment in application programs. For various reasons, having mostly to do with efficiency, it will be necessary from time to time in a typical data base application to change the manner in which data are represented. The usage patterns of these systems will change over time. Some individual, whom we refer to these days as the data base manager or the data base administrator, will have to change the manner in which data are represented — for example, by adding new levels of indexing. When he does this, it is very important that the application programs remain insensitive to changes.

Every programming language and every data base system is characterized by some degree of data independence; I think it is intuitively clear that some languages have more than others. For example, Fortran probably has more data independence than a typical assembly language. I have not been concerned so much with absolute measures of data independence, or what it is that absolutely characterizes a source language or what characterizes an object language, as I have been with the difference or distinction between a source language on one hand and an object language on the other. Now, every programming language is also characterized by a class of so-called *logical data structures*. This is the type of structure that Bill Olle referred to as the second level of data description. For example, the Fortran language includes scalars of different types. It includes one- and two-dimensional arrays of scalars; it includes a certain type of direct access files, sequential files, etc. For a particular job, the programmer selects from this class of structures the particular structures required for his job.

---

*Presented to Special Interest Committee on File Description and Translation (SICFIDET) at SJCC, Boston, Massachusetts, 14 May 1969.

Before his program can be executed, it must be translated; in other words, references to the logical data structure must be transformed or translated into references to some corresponding physical structure. In conventional processors the correspondence between logical and physical structures is fixed in the design of the processor. Thus in a Fortran processor, scalars of type REAL are represented by floating point words and arrays are represented by blocks of consecutive words. Generally, in such processors, no provision is made for changing these mapping conventions. For example, to represent a real variable to arbitrary precision or to provide for automatic overflow of arrays onto secondary storage would require major revisions to be made to the compiler.

A certain amount of this mapping flexibility can be built into the design of a processor. To take a very straightforward example: In OS/360 the user is permitted to specify certain of the mapping conventions at the time the system is generated and permitted to specify certain other parameters at the time that he makes a run – for example, he can specify that a certain data set is either on tape or on disk. In general, however, it is difficult to anticipate all variations that might be useful in a given situation and this has led me to the idea that it may be very useful to have what might be called a *representation-independent processor*.

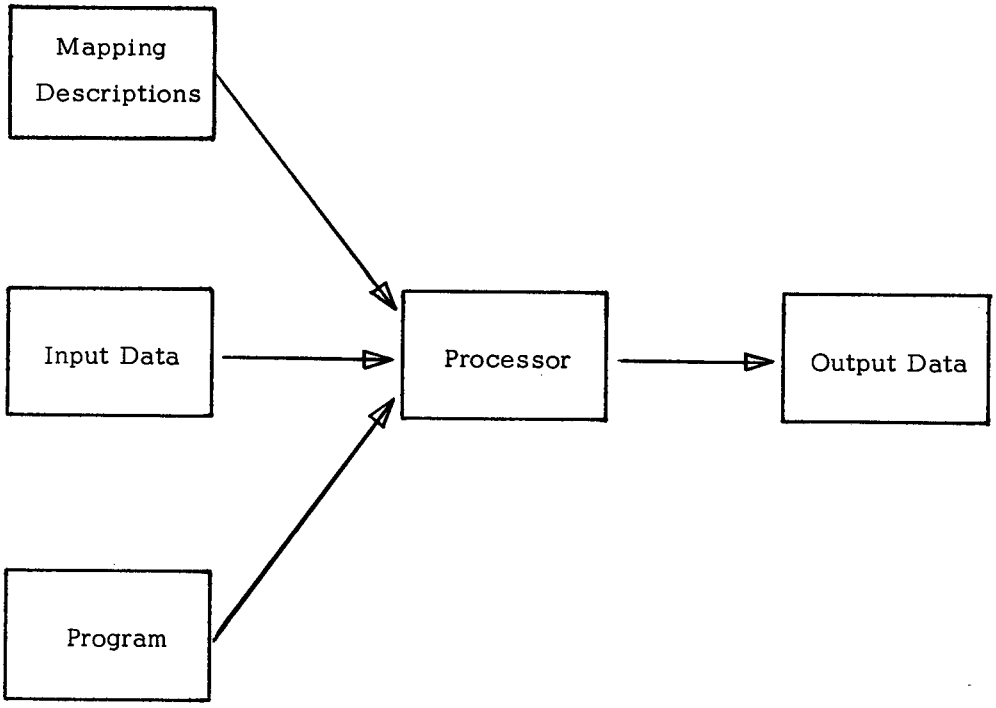


FIG. 1. Data-Independent Processor Usage.

In conventional processor usage the processor has two inputs, namely, a set of input data and a program which describes certain transformations to be performed on the input data. The output then would simply be output data. A Data Independent Processor would differ in that, in addition to the input data and the program, there would be a third input consisting of mapping descriptions

(see Fig. 1). These would be descriptions of the manner in which logical data in general are mapped or represented in physical storage. It is not necessary in such a scheme that all data be represented in the same way. In general, different sets of data would be represented in different ways. Small amounts of data might be represented as core arrays and larger amounts as disk data sets.

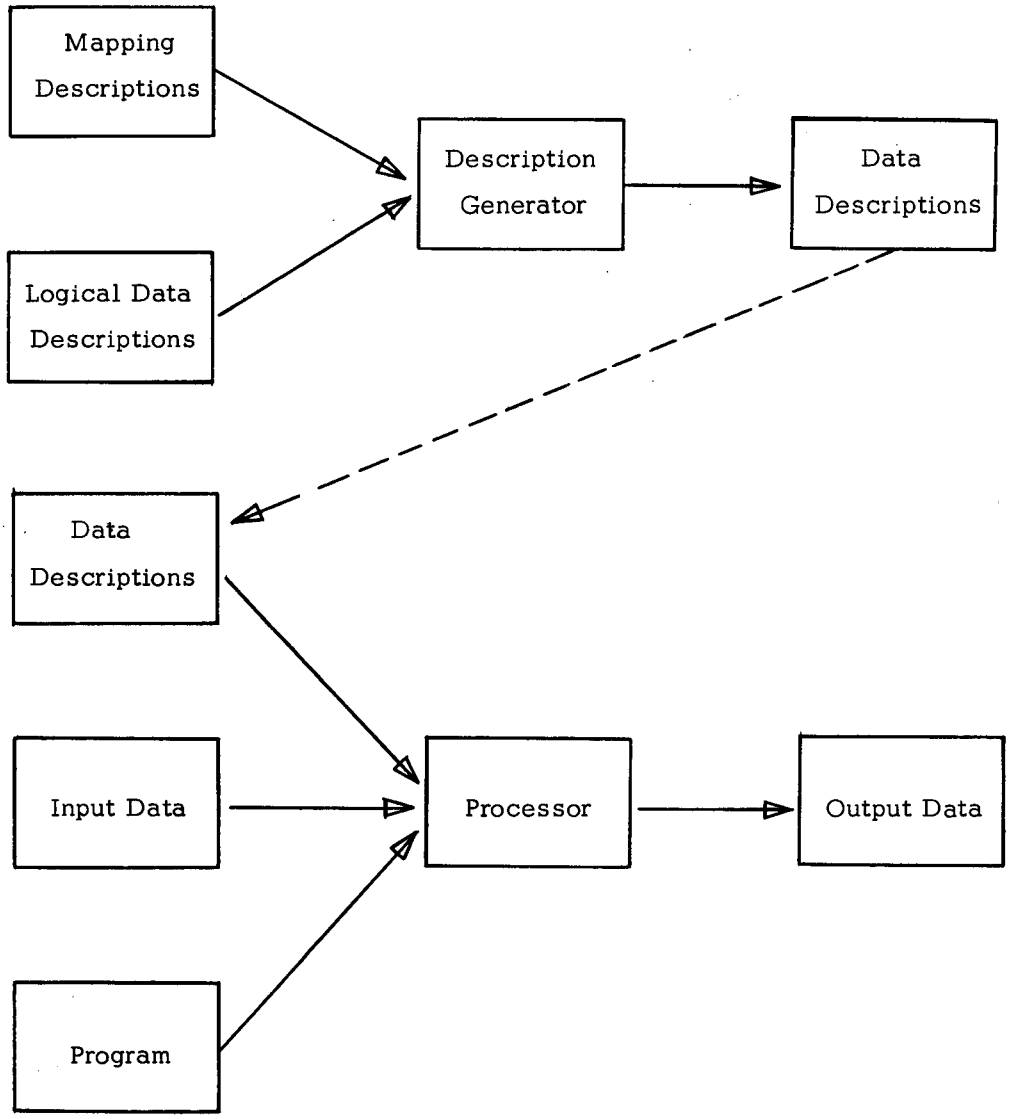


FIG. 2. Data-Independent Processor Usage with Explicit Logical Data Descriptions.

Now, for applications which require the user to make explicit declaration or definition of his logical data, perhaps a more useful arrangement would be to have the processor accept generalized data descriptions instead of mapping descriptions. This is a two-step process (see Fig. 2): A special program called a *description generator* accepts the programmer's logical data descriptions or perhaps the data base manager's logical descriptions. And it also accepts the

mapping descriptions referred to previously. The description generator program then creates a set of generalized data descrptions which could, I suppose, be represented at the level 4 of data description that Bill Olle referred to. These data descriptions then become the third input to the processor. For a typical base-data application the user maintains a file of these generalized data descriptions and when the data base manager decides he wants to change the mapping for one or more of these structures, he proceeds as follows: First, he prepares a new mapping description and inputs this new mapping description and the relevant existing logical data descriptions to the description generator; then he converts existing files whose mapping is to be changed by a series of processor runs. The inputs for each run would consist of a file to be converted, a generalized description of the old file structure and a generalized description of the new file structure, and a simple program whose only function would be to copy the file. I say that the program would be simple because the logical structure of the data would not be changing, only the representation is changing. And it might be noted that this process could be used as one method of solving the problem of data translation, that is, the problem of converting data from one arbitrary physical and logical structure to another.

To study some of these ideas, I have spent some time experimenting with representation of certain types of logical data structures. I referred earlier to the concept of a source machine or source language, if you will, and an object machine. It is really a two-body problem. In describing these mappings there always seems to be one machine which is the machine the programmer works with and which is the one I call the source machine. On the other hand, there's the object machine in which the program is actually carried out. Now these don't have to be machines in the real sense.

SOURCE MACHINE	OBJECT MACHINE	MAPPINGS
FILE PROCESSOR	FORTRAN	ENTRY ⟶ ARRAYS
		ENTRY ⟶ DASD DATA SET RECORD
		ENTRY ⟶ SEQUENTIAL DATA SET RECORD
"TREE" APL	APL	TREE ⟶ DEGREE VECTOR
		TREE ⟶ CONNECTION MATRIX
		TREE ⟶ LINKED LIST

FIG. 3. Examples of Mappings.

For the first source-machine example in Fig. 3, I have taken a typical file processing language such as you might find in a data retrieval system, and I have taken Fortran for the object mahine. The exercise then becomes one of figuring out how one represents logical data structures (which in this particular case consist of elements such as files, records, and repeating groups) and

items with the elements which Fortran supplies; e.g., scalars, one- or two-dimensional arrays, direct-access data sets, and so forth. The third column of Fig. 3 lists three general possibilities one might consider for the mapping: In the first case an entry is mapped into a set of arrays, one array for each item in the entry. The second mapping might be to map an entry into a Fortran direct-access data record and bury right in the record pointers to related entries or groups. The third approach might be the straightforward one of just mapping entries into the records of the sequential data set.

These are three quite different mappings. The point behind this exercise is to see what, if anything, is common and what is different among these representations, in the hope of (1) extracting some useful information that would enable one to devise a language for describing these representations and (2) possibly building a sufficiently general processor that could accept these descriptions and produce the correct object code for handling the specific representations being described.

In the second example shown in Fig. 3, the source machine is Iverson APL which has been extended to the point that it includes some simple tree handling functions. The object machine is the unadorned or standard APL. Some of the possible mappings that I have looked at are mapping a tree into a degree vector (I think Don Hatfield used the term *left list*), into a connection matrix, and into a linked list.

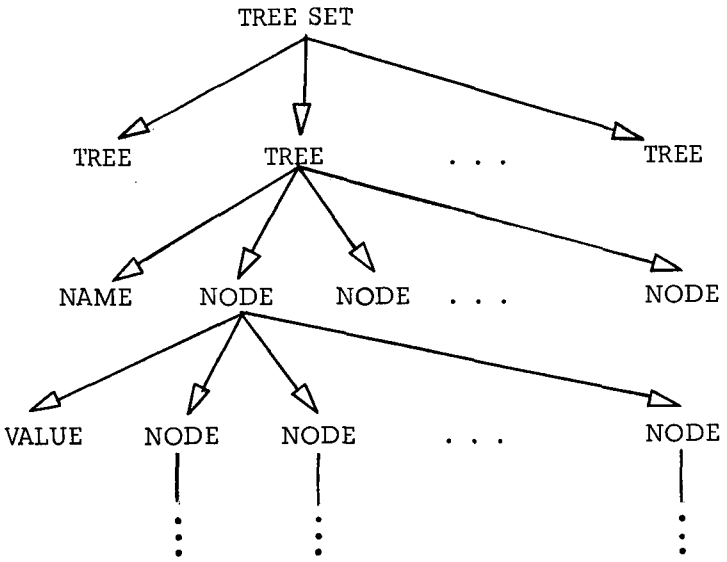


FIG. 4. Tree Set.

Figure 4 is an example of a logical structure for the APL source machine. The programmer works with data consisting of a set of trees. Each tree would have a name associated with it that would be the means by which the programmer would refer to it and it would contain, immediately, a number of nodes – the nodes at that particular first level representing the multiple roots of the tree. Any node, in turn, would have a value associated with it and a set of subordinate nodes, etc.

One way I found useful in describing a logical structure is by means of a *schema* which is essentially a model or template of the logical structure (see Fig. 5). The schema is, itself, a directed graph, essentially, expressing how one generates or how one populates from such a description an actual instance of the structure. Notice, in Fig. 5, that the V in parentheses indicates that in an actual instance of the tree set or the structure being represented there can be a variable number of occurrences for each occurrence of the parent node. This definition is recursive in that just one instance of node is being used to

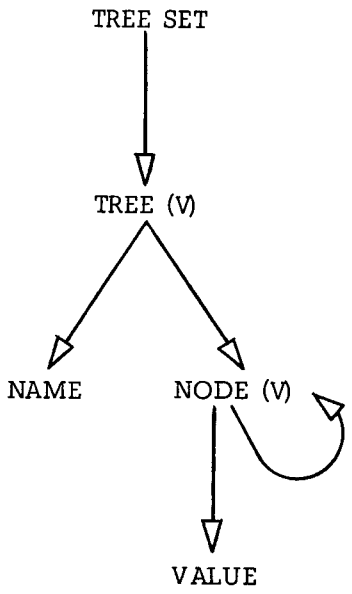
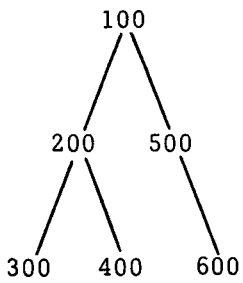
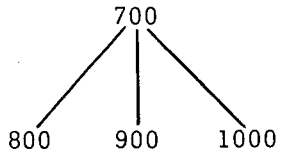


FIG. 5. Tree-Set Schema.

define the roots of the tree as well as the subordinate nodes in the tree. In an actual implementation of this kind of description there would be, presumably, ways of restricting or constraining the possible structures one could generate.

Figure 6 shows some typical tree-handling functions as might be expressed in an APL extension. They are all dyadic with one argument on the left (the index vector which specifies a particular node in the tree) and one on the right (which specifies the tree). This is simply an APL convention. The first function, the degree, gives the number of nodes in the filial set of the selected node. The node function returns the value of the node. The scion function returns the value of the values of the filial set of a given node, and the takenode function is a way of selecting a subset of the tree.






```
1  1  DEGREE   T  =  2
2  3  NODE     T  =  1000
   1  SCION     T  =  200    500
   2  TAKENODE  T  =
```

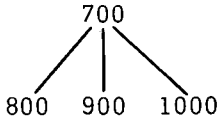
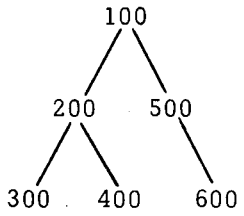


FIG. 6. Typical Tree-Handling Functions.

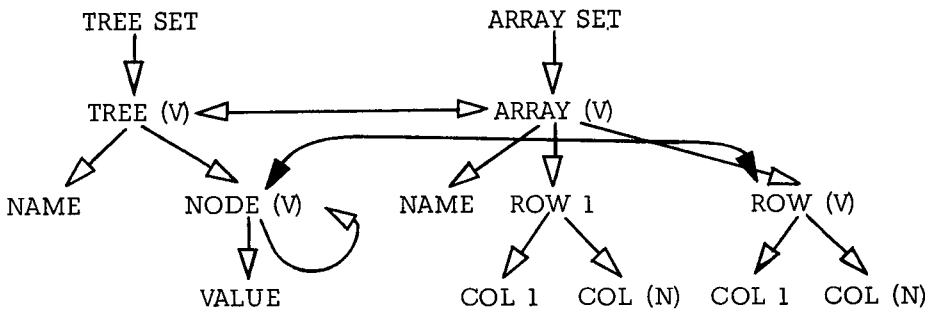


	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	100	0	1	0	0	1	0
3	200	0	0	1	1	0	0
4	300	0	0	0	0	0	0
5	400	0	0	0	0	0	0
6	500	0	0	0	0	0	1
7	600	0	0	0	0	0	0

FIG. 7. Connection Matrix Representation.

Figure 7 shows one of the representations discussed earlier, a tree in the form of a connection matrix. The first column is used to hold the value of the nodes and each row of the matrix corresponds to a different node. The rest of the matrix is filled in with 1's and 0's in the typical incidence matrix type of notation. In APL this would be represented simply as a two-dimensional array.

Figure 8 indicates one possible way of describing the connection matrix representation, again making use of a graph to describe the logical structure of the tree class. Also used is a graph to attempt to describe the physical representation. (The subgraph on the left is the same as in Fig. 5.) The subgraph on the right shows, in general, a way of describing arrays – particularly an array set consisting of a variable set of arrays. An array, in turn, would have a name associated with it; in this particular representation it would have a distinguished first row to hold pointers to the roots of the tree. Then there would be a variable number of rows, each one corresponding to a different node of the logical tree. Each row, in turn, would consist of a distinguished first column and a number of columns whose number is equal to the number of rows. It is possible that with an appropriate language – and I do not maintain that Fig. 8 describes the most lucid way of doing this – one could state explicitly the values that should be associated with every value-bearing node in the right-hand subgraph. For example, the variable N would have a certain expression associated with it. Similarly, the value associated with a particular column would have a specific value associated with it, depending upon the




```
N = count (ROW < ARRAY > *)
val (COL < ROW) = if NODE [index (*)] C
                        NODE < NODE < ROW> *
                  then 1
                  else 0
```

FIG. 8. Description of Connection Matrix Representation.

connectivity expressed in the logical structure. So one can make such a static description. The problem is that it does not appear to be of much use and this is where I am sort of "hung up" for the present. I do not know the reason why it isn't very useful yet, except that it isn't. For these various representations, I have taken a comparative-type look at the code I have generated by hand for performing these individual functions, and the code for each representation is as you might expect. I don't see yet how one could feasibly generate automatically the code which I produced by hand. A more feasible approach might be to describe, in the case of the tree, certain basic primitives that would describe to the processor such techniques as finding the next sibling and finding the first node of a filial set. These primitives could then be synthesized into a sequence which would perform the different logical functions. As an alternative approach I can foresee a problem with this, too: The sequences you get will be, in general, fairly inefficient and a computer programmer doing this would be able to see ways of collapsing these sequences. In particular, in the higher-order language like the APL, operations on arrays can be collapsed into very compact statements.

Essentially, this paper represents my current thoughts on what I am doing in the area under discussion.