Citation: G. H. Mealy, "Another Look at Data," AFIPS, pp. 525-534,
1967 Proceedings of the Fall Joint Computer Conference, 1967.

http://doi.ieeecomputersociety.org/10.1109/AFIPS.1967.112

# Another look at data

*by* GEORGE H. MEALY
*Computer Consultant*
Scituate, Massachusetts

## INTRODUCTION

We do not, it seems, have a very clear and commonly agreed upon set of notions about data—either what they are, how they should be fed and cared for, or their relation to the design of programming languages and operating systems. This paper sketches a theory of data which may serve to clarify these questions. It is based on a number of old ideas and may, as a result, seem obvious. Be that as it may, some of these old ideas are not common currency in our field, either separately or in combination; it is hoped that rehashing them in a somewhat new form may prove to be at least suggestive.

To begin on a philosophical plane, let us note that we usually behave as if there were three realms of interest in data processing: the real world itself, ideas about it existing in the minds of men, and symbols on paper or some other storage medium. The latter realms are, in some sense, held to be models of the former. Thus, we might say that data are fragments of a theory of the real world, and data processing juggles representations of these fragments of theory. No one ever saw or pointed at the integer we call "five"—it is theoretical—but we have all seen various representations of it, such as:

V $(101)_2$ $(5)_8$ 5 0.5E01

and we recognize them as all denoting the same thing, with perhaps different flavors.

We could easily resurrect disputes in medieval philosophy at this point! The issue is ontology, or the question of what exists. While a Platonist would claim that even universal concepts, such as redness, exist independently of whether anyone perceived them properly or at all, a conceptualist might claim that we perceive only ideas and they have no existence until they are perceived. No doubt, both of these gentlemen would quarrel with what I have already said. My friend the nominalist, however, would permit me to entertain such notions, so long as I did not insist upon his treating them as anything but words.* Since it happens that the following does not depend on any particular ontology, we can avoid a quarrel by adopting the nominalist's position.

Our plan of attack is to indicate the nature of the theory of relations, based on the example of genealogical data. This will lead immediately to formulation of our notions about data in general, including rather precise definitions of concepts such as data structure, list processing, and representation. These notions are used in the second part of the paper as the basis for some remarks and suggestions concerning language and system design.

### *Toward a theory of data*

#### Relations

To fix our ideas, consider the following example of genealogical data, taken from Reference 2:

... SNOW ...

4. HENRY (7) [Henry (6), David (5), Anthony (4), John (3-2), Nicholas (1)], b. 18 Sept., 1810; m. 13 Dec., 1840, Susan Stoddard, dau. of John and Betsey (Stoddard) Lincoln. She was b. 21 Aug., 1822, and d. 13 Sept., 1880. He d. 25 April, 1904. "Master mariner." Resided in house which he built on So. Main St. south of his father's.

Ch., b. in Coh.,—

i. SON, 17 April, 1841; d. 6 May, 1841.
ii. JAMES H., 28 June, 1842.
iii. ANN FRANCES, 24 Aug., 1844; d. 5 July, 1869, unm.
iv. SUSAN ELIZABETH, 20 Oct., 1846; m. 1 Jan., 1869, Leonard A. Giles, Troy, N.Y., She d. 25 April, 1827.
v. RUTH NICHOLS, 29 June, 1848; m. 24 Jan., 1892, James H. Nichols.
vi. CHARLOTTE OTIS, 8 Nov., 1850; m. 5 Mar., 1879, George W. Mealy, Troy, N.Y.

*See W. V. Quine's essay "On What There Is" in Reference 1 for an interesting and frequently entertaining discussion of these points of view.

vii. BENJAMIN LINCOLN, 2 Aug., 1853; d. 24 Jan., 1859.

The above, of course, does not record more than a few facts concerning Henry(7) Snow, Jr.; we are already in the realm of symbols. The nominalist is equally prepared to be told that James H. Snow is a misprint or that he is alive today. Officially, the nominalist is under no illusions about data: A data base never records all of the facts about a group of entities; a fact may be recorded with complete or lesser accuracy; and non-facts may be recorded with equal facility.

It was, no doubt, the study of genealogical data that led to the invention of the theory of relations, which will lead in turn to our notions about data in general. Informally, relations are simply a generalization of family relationships, and genealogical data is one of the older instances of recorded data.

We start with a set of individuals (or any other type of entity) and a second set, which may or may not be the same set as the first. A *relation* is a correspondence between members of the two sets. For instance, son-of, children-of, father-of, ancestor-of, sib-of, and the like are all relations, as are birth-date-of, occupation-of, age-of, residence-of, marital-status-of, etc. The children-of relation, in the case of the Snows, is a correspondence between Henry (7) and his children, Henry (6) and his, and so forth. To be somewhat more accurate, there are at least two possible children-of relations; due to the possibility of remarriage, for instance, the set of children of a given pair of spouses is not necessarily the same as the set of children of a given individual of that pair. This is to say that, in general, relations are correspondences between n-tuples from a set and m-tuples from some possibly different set.

Another manner of speaking about the same subject material is also in common use; we speak of some set of things, *attributes* of those things, and *values of attributes*. Attributes are the same as relations, being a correspondence between the things and the values (which may also be things). Still another manner of speaking is to use the term "*map*," as we shall do shortly. A final manner of speaking is to use the term "*function*"; in the theory of relations, this term is reserved for the special case of relations which are single-valued.

The notion of attribute should be distinguished (as PL/I does not) from that of *property*. To say that something has a given property is to say that some attribute of that something has a certain value. Thus, when I say that a house is red, I mean that the value of its color attribute is red, not that I intend to identify the house with the universal concept of redness. Properties may be combined using the usual logical connectives to form new properties, unlike values. Thus, the tall, red house has a property not shared by the long, red house, except by accident. Its color attribute has the value red *and* its height attribute has the value tall. PL/I would with more precision state that the variable X has the property FIXED BINARY DATA-TYPE; the data type is the attribute, and FIXED BINARY is its value, or part thereof, as we shall see.

Before proceeding, it is worth noting that the genealogical example illustrates two types of relations which seem to be qualitatively different. On the one hand, we have the family relationships and, on the other hand, we have relations between individuals and elements of other sets, such as dates. There are other relations which are more akin to family relationships, such as the relations between people who live in the same dwelling or who work for the same organization. An organization chart also illustrates this type of relation, which we will call "*structural.*"

**Data maps**

We have called data fragments of a theory of the real world. It is now time to examine the nature of that theory. Along the way, it will be possible to propose precise definitions for many terms which are normally used rather more loosely. We start with the (undefined) notion of a set and introduce the notion of a map more precisely than we did above. We will use standard notation from set theory: Sets will be represented by capital, Roman letters and elements of sets by lower case letters. If a is an element of the set A, we write

$$a \in A \cdot$$

If A is a subset of B (that is, all elements of A are also in B), we write

$$A \subseteq B \cdot$$

Maps will be represented by Greek letters. In talking about a map of the set A into the set B (that is, the map makes values in B correspond to arguments in A), we will write

$$\mu: A \rightarrow B$$

or

$$\mu$$
$$A \rightarrow B \cdot$$

The latter form is useful in diagrams displaying several sets and maps.

We require of a map that it assign to each element of A either nothing or one or more elements of B; some members of B may not be assigned to any element of A. If every element of B is assigned to at least one element of A, we call the map "*onto.*" If a unique element of B is assigned to each element of A,

the map is called "*one-one.*" A one-one onto map pairs each element of A with a unique element of B and *vice versa.*

We will write ordered n-tuples of set elements as a parenthesized list:

$$(a_1,a_2 \ldots ,a_n)$$

In such an n-tuple, if each $a_i$ comes from a corresponding set $A_i$, then the set of all possible such n-tuples is written as:

$$A_1 \times A_2 \times \ldots \times A_n$$

If all of the sets $A_i$ are the same set A, then the set of all n-tuples from A is written as:

$$A^n$$

The set of all subsets of a set A is usually symbolized as

$$2^A$$

owing to the fact that there are exactly $2^n$ possible subsets of a set of n elements. Figure 1 displays a two element set A together with the set of pairs of elements in A and the set of subsets of A.

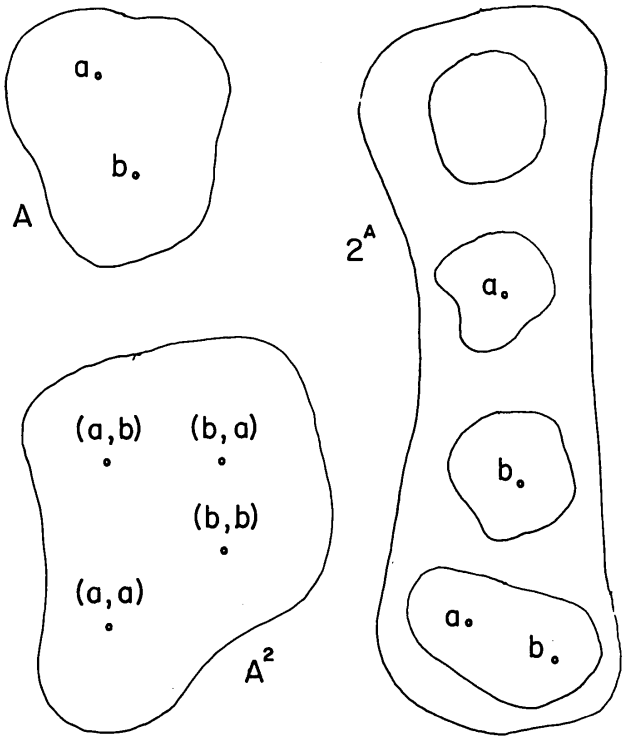



Figure 1 – Sample sets

Now data are supposed to record a set of facts about some set of entities, be they real or abstract. In our present formal manner of speaking, we can contemplate a set of *entities*, E, a set of *values*, V, and a set D whose members are maps of the form:

$$\vartheta : E \rightarrow V$$

We will call D a set of "*data maps*". As we shall see later, D is a set whose elements are subsets of the set $E \times V$. That is,

$$D \subseteq 2^{(E \times V)}.$$

In the case of the Snows, one of the data maps in D assigns to Henry(7) his birth date in the value set of dates. The same data map assigns a birth date to James H. Snow. The date-of-death data map assigns no death date to James (at least, not on the basis of the evidence quoted earlier); its value for him is *undefined.* The father-of map assigns Henry (6) to Henry (7), but its value for James H. Nichols is again undefined. In the case of this map, the value set V must contain E as a subset; we have an instance of a *structural map,* which we define as any map of the form:

$$\sigma : E^n \rightarrow E^m$$

for some non-negative n and m. This is, by our previous definition, a data map only if $n = 1$.

The set of entities, E, might be larger than one might expect. For technical reasons, it is often convenient to introduce certain auxiliary entities. For instance, in the genealogical example, it might be appropriate to introduce entities corresponding to married couples as well as the entities representing individuals. Again, one normally wishes to record information that has no direct bearing on the entities the data are about, such a file name, retention date, etc. In this case, we merely augment the set E with a special element $e_0$ together with data maps defined only for $e_0$.

We have not inquired into the possible structure of the value set V, nor have we admitted all possible structural maps as candidates for the set of all possible data maps. The motivation for this is to simplify matters by considering only the structural data maps

$$\sigma : E \rightarrow E$$

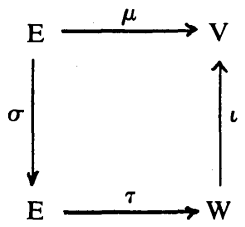
and by restricting the data maps to be functions – that is, to have a single value for each argument. By appropriate adjunction of additional elements to E and data maps to D, this can always be done.

V, itself, might be considered to be constructed from other sets. For instance, in the definition of data maps we have implicitly assumed that E is a subset of V in order to admit structural data maps. The set composed of the elements of V which are *not* elements of E, that is

$$W = V - E,$$

could have structure, however. It might be a set of ordered triples or something more complicated, such as a set of vectors whose elements are vectors, and so forth – in other words, a set of tree structures. *On the contrary,* we will insist upon explaining structure by using structural data maps, and it will then be the case that all data maps can be defined in terms

of a non-structural map $\tau$ applied to a structural map $\sigma$. That is, if $\mu$ is an arbitrary data map, the situation shown in the following diagram can be made to obtain.

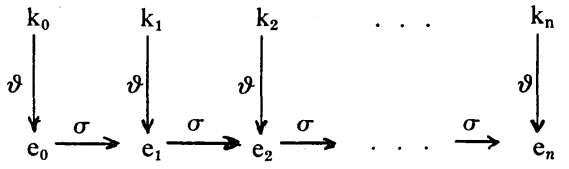



In the diagram, $\iota$ is the identity map of W into V, expressing merely that W is a subset of V. The diagram expresses the fact that, starting with an element of E, one gets the same value by applying the map $\mu$ as by applying $\sigma$, $\tau$, and $\iota$ in that order.

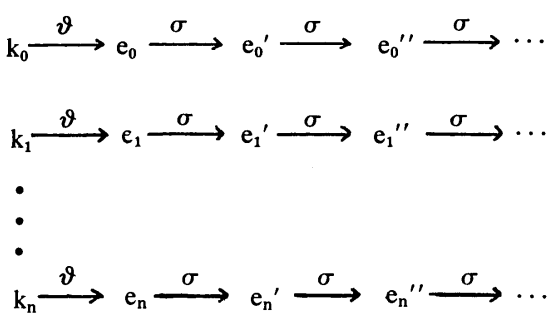
### Access functions

We need a way of travelling around over the entities, or locating them either relative to each other or from something roughly like names or addresses. This mechanism is provided by the notion of an *access function*, which will be any map whose value set is the set of entities or a subset thereof. Access functions are not necessarily structural data maps, as we shall see immediately.

Certain special types of access functions fall out at once. Suppose we have a set K (for "key") and a map from K which is one-one into E; this is a *direct access function*, for we may regard K as a set of keys, addresses, or names of entities in E. On the other hand, suppose that we have a structural data map and some initial element of E, say $e_0$; if each application of the map gives us a new element of E, until the final application gives us nothing, then we say that the map is a *sequential access function.* It mimics the behavior of the successor function of elementary number theory, except that E is finite rather than countably infinite. These two situations can be illustrated by the diagram:




Note that we have not required that the maps be onto —a map into may or may not be onto. A case in which the maps are not onto is given in the following diagram:




This is reminiscent of the indexed sequential access method of Operating System/360.

### Data

We have still not explained which of the objects above we mean to regard as data. The data maps themselves, or rather their elements, will be so considered, but this choice requires further justification.

What happens when data are processed? Our naive notion is that values are used as arguments to the procedure and the result is that values get defined, redefined, or undefined, so that the value set V must be regarded as being the data. But, what if V is the set of integers? It certainly cannot be the case that data processing redefines the integers! Nor are the entities changed, so we are left with the data maps. This notion may be a bit hard to stomach at first, but may be made more palatable by considering ways in which a map may be specified:

- We may specify a test which decides whether or not the map actually assigns a given value to a given argument.
- We may have a rule, or procedure (such as applying several maps in succession) which will designate the value, given the argument.
- We may define the map by actually exhibiting the pairs of corresponding arguments and values.

In the theory of relations, the third approach is usually used to define a relation—it is simply a set of ordered pairs.

In our case, suppose that

$$\mu(e) = v$$

This is the same as saying that the ordered pair (e,v) is a member of the set $\mu$. To redefine the value of the *data item* (e,v) is to redefine $\mu$ by removing that pair from it and adding a new pair (e,v'). This justifies our earlier statement that

$$D \subseteq 2^{E \times V)}$$

Thus, data processing changes neither E nor V, as desired.

We have, incidentally, slipped in a definition of data item, which is an element of a data map. A *data element* will be the set of all data items associated with a given entity. List elements in IPL-V and LISP are data elements, in this sense. The notion of a logical record also corresponds to data element in our sense, and field corresponds roughly to our data item.

This explanation of data processing may seem quite artificial, in view of our Platonistic feeling that the "right" rule for assigning the value of a data item should be independent of how we do our data processing. My friend the nominalist would not be bothered by this scruple—he did not claim that such a thing as a "right" rule existed in the first place; data do not necessarily represent facts with utter accuracy. Data processing, he might say, is data's way of attempting to adjust to the facts, if such there be.

**Procedures**

We have now noted the effect of a procedure—it redefines one or more data maps or, what is the same thing, changes the value part of certain data items. The effect on D is to map it into a new subset of the data maps. In other words, *procedures* are maps of the form

$$\pi : 2^{(E\times V)} \rightarrow 2^{(E\times V)}.$$

Our idea about D is that it is the data at any given moment of time, not the data for all time.

The import of our introduction of the auxiliary entities was to effect a clean separation of structural from other considerations. That is, we have set things up so that any data map can be decomposed into a structural data map followed by a non-structural data map. The structural data maps are maps of E into E, by definition. Our long-standing name for data items in such maps is "*pointers*." This, in turn, suggests an identification of *list processing* with procedures which process structural data. A list processing procedure, hence, is any map of the form

$$\lambda : \quad 2^{E^2} \rightarrow 2^{E^2}$$

This is a precise version of our vague notion that list processing has something to do with pointers and data structures.

**Data storage and representation**

The foregoing model obviously can be taken to apply directly to physical storage media.[3] To entities correspond cells in storage (blocks, words, characters, bits, registers, etc.). Maps specify attributes of the storage cells (more properly, properties) such as content, structure, parity, ability to read and/or write, address, protection key, and the like. The structural maps and access functions clearly correspond to our more usual notions of storage structure and access.

If our data maps are an abstract theory of the real world, we must do data processing with something else; computers are, after all, not abstract objects. However, the abstract theory is just as capable of modeling computation as it is of providing models of the real world—possibly even more so. We are confronted, we might say, with three *systems* in any specific situation. Each such system is composed of a quadruple of entities, values, data maps, and procedures. The first system is, at least from a Platonistic point of view, some part of the real world, the second is our theory of the first, and the third is a machine representation of that theory. A *representation* is, itself, now defined as a map establishing a correspondence between two systems.

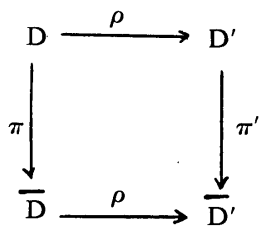
What criteria should a representation satisfy? Well, consider a system in the above sense:

$$S=(E,V,D,P)$$

where P is the set of procedures. Further, let $\pi$ be any procedure in P, mapping D into a new set of data maps D, and let $\rho$ be a representation map which maps S into S′:

$$\rho: \quad (E,V,D,P)\rightarrow(E',V',D',P')$$

For any object in S, we wish the representation to assign a unique object in S′, and *vice versa*. In other words, $\rho$ should be one-one onto. However, we desire more than just this; in order to insure that anything happening in the one system also happens in the other, we require that the following diagram be *commutative:*




or, in other words, that:

$$\rho\pi=\pi'\rho$$

This criterion can fail in two ways: (1) obviously, when the map $\rho$ is not one-one onto, and (2) when the procedure $\pi'$, chosen in the belief that it corresponds to $\pi$, does not in fact so correspond. It might be thought that the second alternative can happen only by mistake, since we could presumably define the

procedure maps in terms of E and V and they are mapped into their primed counterparts in a one-one onto fashion. The rub is that, in practice, the first condition frequently does not obtain, and this gives rise to doubt as to which procedure $\pi'$ *best* represents an abstract computation $\pi$.

The reason $\rho$ fails to be one-one onto, usually, is that the set V′ is of a different size than the set V. The most obvious example is the case of machine representation of the real numbers; only rational numbers may be represented with complete accuracy on a machine (numerically, at least), and our common floating-scale representations are capable of representing only a finite number of rationals at that.* Moreover, the primitive operations out of which we compound procedures are only "best" representations of the abstract operations we have in mind. The size of the literature on floating-scale representations and arithmetic testifies to the amount of disagreement existing on what "best" means!

In the case of a machine representation, we have an abstract system and a representation map mapping it into the machine system. In fact, we frequently employ more than one machine representation for certain data items (e.g., numeric data). On the other hand, there is the physical storage system, and the machine representation of the data and procedure maps must be mapped into the physical storage. The structural part of this mapping—that is, the correspondence between the structure of the data and the structure of storage, we call the *data organization.* While this *enables* data access, it is not access. Access is a feature of the processing of the data, not of the data itself or how it is represented; different procedures will, in general, want to access the same data in different ways and orders. The order in which data items are fetched and stored is (or should be) independent of the data organization; this notion was an important principle in the design of the data management subsystem of Operating System/360.

### Data description

What do we mean in general by the term "data description?" We might be tempted to confine our attention to the theoretical realm and, like ALGOL, talk of real numbers, integers, dynamic own arrays, and the like, considering it the job of any representation to be faithful to our theory. To do so, however, would be to beg the question by ignoring descriptive information which we use every day, and in machine-processable form at that. For example, the COBOL Data Division contains information which we would call data description, such as field lengths, and these certainly describe the machine representation, albeit incompletely. Information describing a data set appears in the volume and data set labels, and still other information is used to compile the procedures. All of this information, and more, is data description, although not all of it is stored explicitly or even in one place, nor is it available at all times. Some of it exists only in the minds of men!

So, I will be dogmatic: Data description describes *machine* data systems, representations, and organizations, rather than abstract data itself. That is, it is a specification of the maps, usually in terms of procedures which will accomplish the mapping, and the salient characteristics of the entity and value sets. To describe a data aggregate—that is, a file or data set—we supply this information together with information concerning the aggregate as a whole.

The term "data type" has been used informally above, but not in any essential manner. Intuitively, we feel that the data type tells us what kind of data we are dealing with. Had we not discovered that data items must be regarded as being elements of data maps rather than elements of value sets, our first false start toward a definition of data type might be to regard it as an attribute of the value set. A moment's reflection, however, is enough to convince one that this is an untenable notion; a given bit pattern may represent an element of any one of a number of value sets (binary integer, floating hexadecimal number, character string, or something else). In fact, it is not unusual to treat the content of a given storage cell as if it were of one data type at one point in a program and of another data type somewhere else.

Our next theory about data type, then, might be that it is related to the kinds of procedure maps used to process the data items. We retreat to the abstract realm, and consider the data type as specifying the mathematical system which governs processing. We might then come out with the following generic data types:

- *String* —free monoid on a finite number of tokens
- *Boolean* —Boolean algebra (or, equivalently, ring)
- *Numeric* —field (or more general system)
- *Pointer* —directed linear graph

This doesn't help very much. We distinguish between fixed binary and double precision complex floating decimal data, and this false start manages to keep us from making distinctions of practical importance by preventing us from talking about representation.

*Matula[4] has studied the maps which convert a floating-scale number from one radix to another. He shows that the map may be one-one or onto, but not both!

Nevertheless, the above are not irrelevant to data type; they are merely incomplete definitions. We are forced to conclude that a data type is a fragment of data description and, as such, describes a portion of a system and its associated representation and organization maps. It is an attribute of entities. We can tie the notion down better by looking at further determinants of data type for each of the generic data types mentioned above:

*String*

- Code—the code for each token (character, bit, etc.). E.g., USASCII, EBCDIC and the like.
- E.g., USASCII, EBCDIC and the like.
- String length—fixed or variable, and value of length attribute.
- Justification—left or right, and padding token.

*Boolean:*

- Code—the code for each truth value (or n-tuple).
- Field length.

*Pointer:*

- Code—machine address, base and displacement, item number in table, etc.
- Code for null pointer.

*Numeric:*

- Code—digit code, radix or weights, excess.
- Sign treatment—unsigned, sign and magnitude, radix or radix minus one complement.
- Scale—fixed or floating, value of scale.
- Rounded or truncated calculation.
- Arithmetic algorithms.
- Numeric limit(s) on value.
- Field length, or precision.
- Aligned or packed (storage mapping restriction).

It is evident that a complete description of a given data type contains information concerning representation of elements of the value set; information stating which representation, procedure, and organization maps are applicable; and data to be used in any mapping (such as value of scale). It is further evident that two entities which are not identical in all of these respects should not be regarded as having the same data type, unless the variable information is stored as data to be used interpretatively in accomplishing any of the mappings. Thus, variables of type real in ALGOL must be regarded as having different data types as represented, say, on the IBM System/360 and the CDC 6600.

Thus far, we have not discussed structural data. We even seem to have done a bit of violence to the notion of separation of structural from other data, but this is easily fixed. For instance, consider a floating-scale entity. Both the mantissa and the characteristic must be available during calculation, and they are treated as one entity during floating-scale arithmetic but as two entities during radix conversion. To be completely consistent, for each floating-scale entity we should introduce two more auxiliary entities; each of these is a fixed scale number and can be described appropriately. Similarly, we can take care of complex and multiple precision entities.

Turning to purely structural data, it is clear that we can invent data types, such as arrays, lists, tables, ring structures, etc., for any sort of structure worth classifying. In practice, of course, we suppress most of the detail involving the auxiliary entities and write down descriptions like:

DECLARE A(3,3) FIXED BINARY (15);

to describe a three-by-three array of 15 bit fixed-scale binary integers.

*Languages and systems*

On the basis of the point of view about data advocated earlier, more light can be shed on several issues of current interest. In some sense, these issues are all related to the possibilities of flexibility in choice of machine (machine independence), choice of data representation (ability to define new data types), or ability to strike a balance between compilation and interpretation (variable binding time).

**Representation independence**

For some years now, one of the more persuasive arguments in the favor of narrative languages such as ALGOL, COBOL, FORTRAN, JOVIAL, NELLIAC, and PL/I has been that they have offered some measure of machine independence. That is, the user is offered a greater or lesser degree of hope that a program written in a particular language together with data processed by the program can be processed on a variety of computing systems, with something like the same results.

Independent of one's personal degree of confidence in fulfillment of such an objective (and most of us believe that the objective is a Good Thing), I would urge adoption of the term "representation independence" as being more appropriate than the term "machine independence." In espousing such an objective, one's philosophical point of view might be that data processing takes place in the abstract realm in which our theory of data is formulated. It is, following this line of thought, more or less an accident (economic issues and questions of accuracy aside)

which computing system and machine representations are chosen in order to do the actual processing, and all such choices should lead to the same results.* In principle, I cannot quarrel with such a view; in practice, I wonder if blind pursuit of the objective does not often result in prejudgment of the economic issue. I hasten to add, however, that most programmers tend to be *too* pragmatic—in the long run, generality often costs less than specificity.

Representation independence is a more stringent objective than is machine independence. Representations are equivalent only when the representation maps are one-one onto and commute with the procedures, as we have seen, and there are cases in which this simply cannot be achieved in practice. Even in cases where one can choose a representation which satisfies our criterion, the cost may be unacceptable and we are forced to make do with a "best" representation. This should be counted as no disaster; it simply means that in practice we must relax the criteria of representation independence sufficiently to stay within optimal bounds of accuracy and economic data processing. Paradoxically, the best way to follow the spirit of our objective is to recognize that we can't live up to the law in all cases.

Our Platonistic tendencies have led us, in the past, to attempt to banish considerations of representation from language design and usage. I would suggest, contrariwise, that any serious attempt to design languages which are significantly more representation independent that at present is doomed to failure unless the notion of representation is made part of the language, ungrudgingly. At that point, we are again faced with the issue of how cleanly we can separate specification of the algorithm from specification of the data representation. Possibly, this introduces a new version of the UNCOL controversy! To overstate the case, the notion behind UNCOL was that one could pump a language description into one end of a compiler, a machine description into the other end, and come out with quality compilations. While many would still like to believe this, the evidence is that the two descriptions cannot be separated that cleanly. What I am suggesting, however, is counter to separatist sentiment—I claim that we should see what happens if we let the programmer talk about *both* procedure *and* representation.

PL/I is the latest language design to attempt suppression of representation. Yet, UNSPEC finally made its appearance. For a purist, the use of UNSPEC is as unacceptable as in-line machine language coding. Why, then, did UNSPEC slip in? The reason, I suspect, was two-fold: (1) to allow use of data types not already defined in the language, and (2) to allow a given storage cell content to be treated according to one data description at one point in a procedure and according to another somewhere else. It seems to me that the attempt to banish representation considerations from language design has, in this case at least, led inevitably to their sneaking in the back door in a quite unpalatable form.

The dilemmas posed by representation independence, UNSPEC, and the CELL construction in PL/I are unfortunate inheritances from past history, earlier forms of which occurred in FORTRAN. They resulted from our inherited belief that symbols are names of storage locations rather than names of entities assigned to them by the data organization. The crux of the situation is this: The programmer wishes to be representation independent to the extent that considerations of accuracy and economics allow. On the other hand, in some cases, he is definitely concerned with specific representations—for instance, in specification of procedures which convert data from one representation to another. To be so doctrinaire in language design as to insist on complete representation independence is to cut off one's nose to spite one's face. The language feature needed in the specific case at hand is the ability to say "I have a value for data item A in storage, and I now wish to use that as the value for data item B, which has a different data type. Furthermore, no matter how complex the structure and data organization for A and B, I *know* that the bit patterns for the two values are identical (I just told you so), so don't bug me about representation independence." In other words, the programmer must be allowed to pay his money and take his choice.

### Language extension

PL/I has been lambasted both for having too many and for having too few data types available. The former point is made by many who have been unfavorably impressed by the sheer size of the code required in PL/I compilers, although this might with more justice be blamed on the amount of automatic data conversion required. The latter point is made by those who wish to handle data of more complex structure with less circumlocution. The suggestion has been made in many quarters that what we really need is fewer data types along with apparatus which will allow the user to define his own. I find myself in this camp, although I don't believe that the problem is by any means as trivial as is sometimes claimed.

Part of the problem, of course, lies in adopting a consistent view of what data are and finding suitable

*This is not my own point of view, needless to say. However, the notion of representation independence makes sense whatever one's philosophical bias.

methods for writing and storing data descriptions (including both representation and organization). The theory of data advanced above seems potentially adequate for these purposes; what is not so clear is how to link the data description apparatus to the code generation and optimization apparatus of the compiler. Galler and Perlis[5] have done some interesting work in this area.

### Variable binding time

How much of the data description is stored explicitly, and where, is partially a matter of taste. It is largely determined by the language, language processor, and operating system one is working with at present. Classically, we have tended to use the data description at compile time and then throw it away. Moreover, much of the data description has been implicit in the compiler's structure; the programmer has had little explicit control. List processing is an intermediate case—processing variable data structures must be done interpretatively, even though the code for other processing can be compiled out. At the other extreme, interpretative operation tends to use the data description at execute time, over and over again. In the former case, the bet is that the program will not be recompiled too often and the data description will not have to change during execution, nor will the data structures. In the latter case, the bet is that less overhead will occur through interpretation than would be incurred by frequent massive compilations. Conversational processing tends in the latter direction.

But, in the case of all systems that have been designed thus far, the choice of binding time is pretty much cast into concrete by the system and language processor design, even though it is hard to believe that the choice can possibly be appropriate for all tasks. We should, I believe, seriously investigate the possibility of system designs that allow the individual user to make his own judgment of the proper tradeoff. This would necessitate explicit storage of data descriptions—a few systems have allowed this, notably the CL-I and CL-II systems.[6]

## SUMMARY

In the first part of the paper, we have proposed a theoretical model for data and data processing. The model is a *system* of sets of *entities, values, data maps,* and *procedure maps*. The entities correspond to the objects in the real world about which data are recorded or computed. The data maps assign values to attributes of the entities; these maps are regarded as being sets of ordered pairs of entities and values, or *data items*. *Structural data* is a special type of data map where the value set is the set of entities itself; structural maps are composed of *pointers*. By introducing structural data maps and auxiliary entities, we can explain any data map as being composed of a structural data map followed by a map whose value is a quantity with as simple structure as we wish (for example, an integer). *Procedures* are operations on the data maps, producing new (or redefined) data maps. *List processing* procedures operate on structural data maps, or sets of pointers.

Data processing occurs in the machine realm, operating on objects which are mapped into the storage facilities of the computing system. We require, ideally, that such a system be a *representation* of the real or abstract system being modelled; this means that the representation map is one-one onto as regards the entities, values, data maps, and procedure maps and, further, that the representation map commutes with the procedure maps.

*Access functions* are maps whose values are entities; they are used by procedures to get access to the entities, and hence to the data. Access functions may involve use of structural data, but are principally a feature of the individual procedure. *Data organization*, on the other hand, is the way the structure of the data is mapped into the structure of the storage media.

*Data description* is a specification of machine data systems and representations; a *data type* is a fragment of data description, describing an entity and its applicable maps.

The first part of the paper, then, was a discussion of the nature of data and the relation between what it is and what we do with it. In the second part of the paper, these notions were used as a framework for discussing several issues of current interest.

The notion of machine independence, apart from its roots in practical needs, proceeds from the point of view that data processing takes place in the abstract realm and, hence, its results should be representation independent. This is an attainable goal only when our criterion for a representation can be satisfied economically. However, language design has tended to suppress the notion of representation to the extent that the programmer frequently cannot talk about it. I regard this as being a mistaken approach toward our practical goals; on the contrary, significant progress toward representation independence of results can only be made by making the notion of representation much more explicit in language designs than it is at present.

A second argument for emphasizing the role of data description in language design is related to the issue of language extension. A trend in language develop-

ment has been to expand the number of data types available to the programmer, at the expense of significant increases in compiler size and complexity, and without satisfying those who have a genuine need to use data types over and above those already available in a given language. The way out of this dilemma appears to lie in the design of languages with a limited number of data types (perhaps only one instance of each of the four generic types mentioned earlier) together with facilities enabling the user to introduce arbitrary data type definitions as needed.

Finally, it was argued that language and system design should make increased use of stored, explicit data descriptions. This will serve two purposes: First, it is a prerequisite for the design of systems which allow the user to strike his own economic balance between compilation and interpretation. Second, the extent to which current system designs achieve the goal of independence of procedure specification from data representation is due to the use of stored descriptive information, together with interpretation of the description, be it at compilation, execution, or some other time. Standardization of methods of data description may ultimately prove to be much more important than standardization of methods of data representation and procedure specification.

## REFERENCES

1 W V QUINE
*From a logical point of view*
Harvard University Press Cambridge 1953

2 G L DAVENPORT E O DAVENPORT
*The genealogies of the families of Cohasset, Massachusetts*
Stanhope Press Boston 1909

3 A W HOLT
*Proceedings of IFIP Congress 1965*
Spartan Books Inc Washington D C 1966

4 D W MATULA
*Base conversion mappings*
AFIPS Conference Proceedings, Spring Joint Computer Conference vol 30 pp 311-318 1967

5 B A GALLER A J PERLIS
*A proposal for definitions in ALGOL*
Comm ACM 10 204 1967

6 T E CHEATHAM, JR.
*Data description in the CL-II programming system*
Digest of Technical Papers National Conference Association for Computing Machinery 1962