# Bringing Data Base Technology To The Programmer

William L. Honig
Bell Labs
Naperville, Illinois
March 1, 1974

ABSTRACT: This paper reports work toward melding some ideas from data base management with currently popular views of structured programming. The basic idea is to allow all data structures used inside programs (instead of just those in an external data base) to be described at multiple conceptual levels by a separate definition. This approach brings to programmers the benefits of data independence and also allows structured programming to have its hitherto unrealized full effect on data.

KEYWORDS: data structures, program design, data independence, structured programming

CR CATAGORIES: 4.34, 4.33, 4.22

# Bringing Data Base Technology To The Programmer

William L. Honig
Bell Labs
Naperville, Illinois
March 1, 1974

## 1. INTRODUCTION

Data structures have been a ubiquitous part of programming since the Beginning. Currently, programming is maturing through a phase of new disciplines and methods often grouped together under the somewhat nebulous term "structured programming." To date, structured programming has stressed program design, control structures, and correctness proofs. Very little has appeared which is apropos to the role of data structures in structured programming. This paper suggests that some relevant aspects of generalized data base systems be taken "inside" the program and melded with the complementing ideas of structured programming. The pertinent data base management concepts are data independence and data definition languages.

This paper is a prolegomenon - the goal of the work whose motivation is reported here is a methodology implementing these ideas in a way which is consistent with contemporary programming languages. The current status of this work is a list of criteria which is reported in Section 4 with an indication of the next steps to be taken. Section 2 discusses briefly the evolution of data structures in programming languages, describes the view of data which structured programming seems to encourage, and points out this view's consistency with the basic themes of modern data base management systems. Section 3 surveys some relevant work by Parnas, Shneiderman, and Earley and appraises its suitability for structured programming.

## 2. THE NEED

Programming languages have provided various types of data structures. However, programmers have often used data structures not directly supported by the language in use. For instance, the FORTRAN programmer often writes a program using tree-like data structures which must be implemented in FORTRAN with arrays and functions of array indexes. Nevertheless, conceptually the programmer thinks and programs with trees. Structured programming draws sharp attention to conceptual steps and levels in programming and would seem to suggest the propriety of such an approach.

### 2.1 The Evolution of Data Structures

The initial and most far reaching change in the programmer's view of data occurred some time ago when high-level languages were first introduced. This step in the evolution can best be illustrated with an example shown in Figures 1 and 2. Assembly languages provide only the ability to allocate storage, usually in fixed length pieces. A programmer using a 4 x 2 array might draw the picture shown in Figure 1, but the closest he/she can come to representing this idea in assembly language is something similar to

```
A      DS      8F      4 x 2 ARRAY
```

Unfortunately, the assembler is unaware of the programmer's intentions; everytime the array is accessed, all indexing must be carried out explicitly by the programmer.

Now, with the advent of FORTRAN, the same programmer has a declaration

```
INTEGER  A(4,2)
```

This statement represents a very basic change in the way programmers think about aggregates of memory: an explicit definition of the data structure which is separate from all uses of the data is required. Now, if a picture is drawn by the programmer, it will probably resemble Figure 2.

Explicit and separate are key words above. These basic ideas have been propagated through most higher level languages

(with the possible exception of some special purpose languages such as SNOBOL or LISP). Likewise, these basic ideas are no strangers to the field of data base management. Data independence is apposite to this task. Separate, explicit definitions of data are the cornerstone of many approaches to generalized data management systems (for example [3]).

## 2.2 The Influence of Structured Programming

Structured programming has become quite popular even though it remains an undefined term [4] for a nebulous collection of programming techniques. In this paper the term 'structured programming' is used to represent this entire collection of methods for designing and writing easier to understand, error-free software. Many of these structured programming techniques have a common underlying philosophy which has been called 'levels of abstraction,' 'top-down programming,' or 'step-wise refinement.' This philosophy relates to the conceptual steps taken in the design and implementation of software and, basically, suggests that thinking and understanding of programs start at a high, general, nonspecific level and then proceed to refine these ideas at more specific, more detailed lower levels.

This technique of refinement seems to be a well accepted part of structured programming; however, it has never been fully applied to data structures. Dijkstra has discussed "abstract data objects" [5] which the programmer should freely assume whenever needed; but no efforts toward formalizing these ideas have appeared. This lack is unfortunate since, in our experience, software designers often have no idea of what data structures are appropriate to a given problem or of how to provide such abstract data objects in terms of the languages they are constrained to use.

At a recent conference [6], Dijkstra further pointed out the lack of such a mechanism as the most important item on a list of programming needs. He spoke of an "intellectual zoom lens" which would enable a programmer to look at a data structure and see only the necessary amount of detail. The programmer should be able to click down the lens, showing more and more detail until the individual bits of the data are seen.

The value of such a zoom lens can be seen by considering some examples. First, let's consider a normal syntax tree as it might be used in a compiler and attempt to identify as many different conceptual levels as could possibly be useful. Such a tree is shown in Figure 3. The conceivable levels of interest and possible uses in a compiler include:

1. The entire tree as an elementary thing; the highest-level compiler driver uses a syntax analyzer to build "a tree" which it passes to code generation. At this level nothing about the internal structure of the tree need be known.

2. A tree is a set of interrelated nodes; there is a relation (in the mathematical sense) between a node and the set of its subnodes. The begin-end code generator uses this level of data description; it does some prologue, calls a processor for each related subnode, then does an epilogue.

3. Each node consists of a statement TYPE and any number of fields which specify the COMPONENT subnodes (i.e., at this level we make more specific the way the relation between the nodes is implemented). The arithmetic expression processor can find out which operator it's doing and its arity. The processor recurses pointing to each related subnode.

4. TYPE is a one-of-N encoding. COMPONENT is a key which must be passed to a hashing routine to access the subnode. This level of detail is used by a routine which branches according to node TYPE.

5. A node is a stream of bits of an arbitrary but specified length which is accessed by some key; used by the storage handler which allocates nodes.

The major conclusion to be drawn from this example is that some parts of the program really need to use abstract data structures such as NODE and RELATION. To require these programs to be aware of the detailed structure is contrary to the structured programming philosophy of continual refinement.

For another example, let's consider part of a manufacturing information data base similar to those which have been used by Codd [2] and others to describe relational data structures. The part of the data base considered describes the quantity of parts used by some manufacturing projects, and the suppliers who provide those parts. At the highest level this data base can be cast in a relational form similar to a mathematical relation as follows.

| supply (supplier, | part, | project, | quantity) |
|---|---|---|---|
| ABCo | 1007 | HELIX7 | 3 |
| XLtd | 2216 | HELIX7 | 27 |
| ABCo | 2011 | MAND | 15 |
| ACME | 556 | HAMP | 7 |
| . | | | |
| . | | | |
| . | | | |

At this level the data base consists of relations (supply), their domains (supplier, part, etc.), and sets of related 4-tuples (each row of the relation above). This characterization of the information is useful for an information retrieval system which processes requests based upon relations. An example request is: "Which suppliers provide parts to any project in quantity greater than 10?" or "Which projects use parts supplied by ABCo?"

Again, let's consider as many conceptual levels as might be useful in programming this information retrieval system. First, suppose the relational retrieval system must be implemented to work from an existing traditional data base consisting of files with keyed access. At this level, the above relation may take on the form of a single file, each record of which contains all the information for a single project, like this:

```
file SUPPLY    project*
               part*      ⎫
               quantity   ⎬  repeating group
               supplier   ⎭
```

where the stars indicate items for which keyed access is possible. Within this characterization there are at least four different levels of detail which might be useful to programmers; they are

1. The SUPPLY file is a set of RECORDS, accessed by specifying either a PROJECT or a PART. With each PROJECT there is an associated set of PART_INFORMATION (the repeating group).

2. The SUPPLY file is implemented as an inverted file structure in which there is a DIRECTORY which associates keys with a set of RECORDS.

3. The directory association between keys and records is given by a set of RECORD ADDRESSES for each key. These addresses must be passed to a retrieval routine to access a RECORD.

4. Each record consists of fields of various types and sizes and in a certain order. One of these fields gives the number of occurrences of the PART_INFORMATION repeating group.

In addition to these four, we could take each field or address all the way down to the bit string level if necessary.

Although the philosophies of data independence were motivations for the idea of separately defining a hierarchical, multiple level description of a programmer's data structures, current data base management systems are not germane to the task. Most data base schemes provide, in effect, a single data structure with many optional parts. It is the user's duty to adapt the information to be stored into this Procrustean structure. The CODASYL DBTG proposal [1] is the prime example of this approach. The technique of Diane Smith and collegues [13] is more apt; however, it is concerned with data translation and not with any higher level purely conceptual data structures such as those in the above examples.

## 3. WORK OF OTHERS

Some other work of relevance to the role of data structures in structured programming has appeared.

Jay Earley has identified specific levels of interest for data structures [7,8,9]. These levels are:

1. Relational level - used to describe the "essential relationships between the data items;" based on relations similar to the manufacturing example above plus mathematical n-tuples, sets, and sequences. In addition to these primitive data structures, operators called "iterators" are provided for programming at the relational level.

2. Access path level - used to make explicit the actual paths or connections between various data elements without making any commitment as to how the paths will be implemented (which could be as addresses, sequential words in memory, hashing, etc.). The access path level has been implemented in the language VERS [10], and is based upon the idea of 'V-graphs,' directed graphs in which nodes represent parts of the data structure and links the access paths.

3. Machine level - used to map the data structures into actual storage by expressing them in terms of the facilities provided by the compiler. At this level the traditional questions such as how big fields are and how they are allocated are answered. Only at this level is the programmer concerned with actual words of memory and what is in them.

This 3-level approach certainly is consistent with the refinement philosophy of structured programming. A data structure can be described first at the relational level and then as more details are needed, at the access path and machine levels. Additionally, Earley has proposed an "implementation facility" which provides default implementations automatically, relieving the programmer from the burden of completely describing the data structure at the lowest levels.

However, more than three levels are clearly useful, as shown by both the examples in Section 2.2 of this paper. Structured programming would seem to require an arbitrary number of arbitrarily high general levels. Additionally, basing the highest levels on purely mathematical objects may not be sufficient to describe the rich class of data structures routinely used by programmers today.

Ben Shneiderman has attacked, quite successfully, a somewhat different problem: to increase the understandability and reliability of programming with linked list structures [12]. To do so, the data structures allowed have been restricted to 1- and 2-way lists, trees, rings, queues, stacks, and dequeues. However, hierarchical compositions of these basic structures are allowed, for instance, a tree of 2-way linked lists.

The basic parts of this approach are a "Data Structure Description and Manipulation Language" which provides an

explicit, separate definition of the data structures, and an execution-time monitor which performs all modifications to the data structures making sure the definition is not violated. For instance, given a data structure which is defined as a tree of queues, the monitor will prevent a stack from being inserted as a node of the tree.

Shneiderman's approach is admirable as far as it goes; however, it is still desirable to allow the programmer to define completely arbitrary data structures not just a restricted set. Additionally this approach seems to be aimed at a single conceptual level at which things like nodes and pointers are manipulated. Most programmers would want to be able to escape these traditional data structures and invent new ones which are not based on the ideas of pointers. The relational data base example in Section 2.2 showed a new data structure of this type - the SUPPLY relation. Other more sophisticated, less pointer oriented data structures will continue to be useful in software design and development, and thus, a scheme for data structures should be able to adapt to handle them.

Parnas has suggested a philosophy for program modularization called "data hiding" [11] which, although directed towards a somewhat different problem, is pertinent to the question of data structures for structured programming. Data hiding is suggested as an alternative to the traditional way of program modularization; it suggests modules be selected so that an entire data structure and its accessing procedures are all part of a single module. This philosophy leads to a design process in which each module hides some design decision from the rest of the system. As noted by Parnas, this approach can be used to break down a system into a hierarchical structure à la structured programming, by seeing which resulting modules use each other.

This author has attempted to apply Parnas' data hiding philosophy, in a slightly different manner, to a top-down design of a complex, ill-understood software system. The approach was more like a true refinement (starting with generalities and adding details) than the approach of Parnas which is carried out after the system has been modularized. The basic method is to define the data structures in a top-down manner according to the philosophy of data hiding, i.e., at each level assume as little structure or detail as possible.

This approach is somewhat similar to the two examples in Section 2.2, but more involved since multiple, interrelated data structures are usually needed. Two conclusions were drawn from this exercise. First, it aided the better definition of the system under development. Considering the data structures first, before designing the functional parts, suggested some major changes in the overall system design. Second, it seemed that more tools were needed to carry out exercises of this type easily and beneficially. This conclusion resulted in the start of new work to identify these tools; this work is described in Section 4.

## 4. WHAT TO DO NOW

The above topics motivated some work toward identifying a more proper way of using data structures in structured programming. This section describes the current status of this work and outlines the steps to be taken in the near future.

So far, the major characteristics of such an approach to programming with data strucutures have been identified; they are:

1. An unlimited number of conceptual levels of data structures must be provided. The lower or more detailed levels need not be strictly refinements of higher levels.

2. A wide range of data structures must be representable. This range cannot be restricted to solely mathematical objects, or to structures constructed with explicit pointers.

3. The description or definition of the data structures must be explicit and separate from all uses of the data.

4. The methods of dealing with data structures must be applicable to the normal spectrum of programming languages.

5. Data independence must be provided to alleviate the problems of program modification.

To determine which kinds of data organizations are desirable (to satisfy item 2 above), the plethora of data structures currently used in programs is being investigated. The

programs which are being examined include compilers, debugging and testing aids, simulators, and other system programs. The purpose of this effort is to distill out of these ad hoc structures, the basic kinds of data structures which programmers invent when left to their own devices (i.e., current languages). These basic data organizations should be a felicitous base upon which a general scheme for data structures can be built. Other capabilities can then be added to this base to incorporate other data organizations of possible future interest to programming. Foremost among this class of new organizations are the various relational schemes which have been quite useful in data base management.

Once the valuable kinds of data structures have been recognized and understood, it only remains to fit them all together into a methodology which is consistent with the levels of abstraction approach to structured programming. In order to satisfy the other items described above, this method must provide multiple levels of data description separately from the programs which use the data. This battleplan is currently being carried out. The results of this endeavor will be appropriate for an automatic system organized similarly to that shown in Figure 4. This system could interface with programs either during compilation or via a monitor during program execution. However, the stored base of data structure definition is also a useful tool during program design and is an aid to understanding the structure of large software systems.

## REFERENCES


1. CODASYL, Data Base Task Group Report (April 71).

2. E. F. Codd, A Relational Model of Data for Large Shared Data Banks, Comm. of the ACM 13, 6 (June 70) 377-387.

3. C. J. Date and P. Hopewell, File Definition and Logical Data Independence, Proc. 1971 ACM SIGFIDET Workshop, (Nov. 11-12, 1971) 117-138.

4. P. J. Denning, Letter to the Editor, SIGPLAN Notices 8, 10 (Oct. 73) 5-6.

5. E. W. Dijkstra, Notes on Structured Programming, Structured Programming, Academic Press (1972) 1-82.

6. E. W. Dijkstra, Lectures at the Regional Conference on Programming Methodology, U. of New Mexico (Jan. 7-11, 1974).

7. J. Earley, Toward an Understanding of Data Structures, Comm. of the ACM 14, 10 (Oct. 71) 617-627.

8. J. Earley, On the Semantics of Data Structures, Data Base Systems, R. Rustin (Ed.), Prentice Hall (1972) 23-32.

9. J. Earley, Relational Level Data Structures for Programming Languages, Acta Informatica 2, (1973) 293-309.

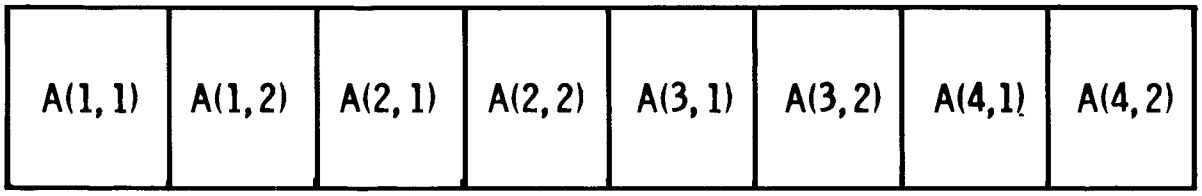
10. J. Earley and P. Caizergues, VERS Manual Version 4, Computer Science Department, U. of California, Berkeley (Oct. 71).

11. D. L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, Comm. of the ACM 15, 12 (Dec. 72) 1053-1058.

12. B. Shneiderman and P. Scheruerman, Structured Data Structures, Department of Computer Science, State U. of New York at Stony Brook (June 73) Technical Report No. 16.

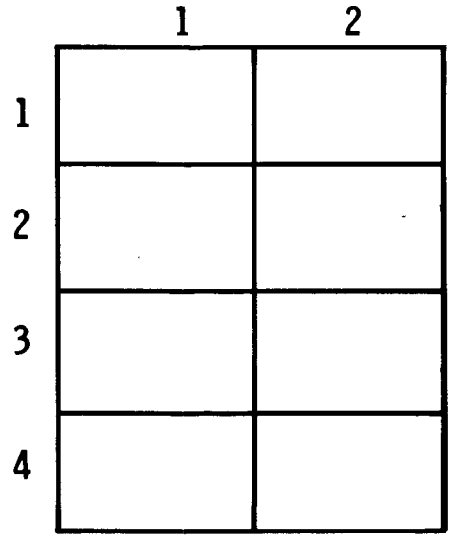
13. D. P. Smith, A Method for Data Translation Using the Stored Data Definition and Translation Task Group Languages, Proc. 1972 ACM SIGFIDET Workshop (Nov. 29 to Dec. 1, 1972) 107-124.

A

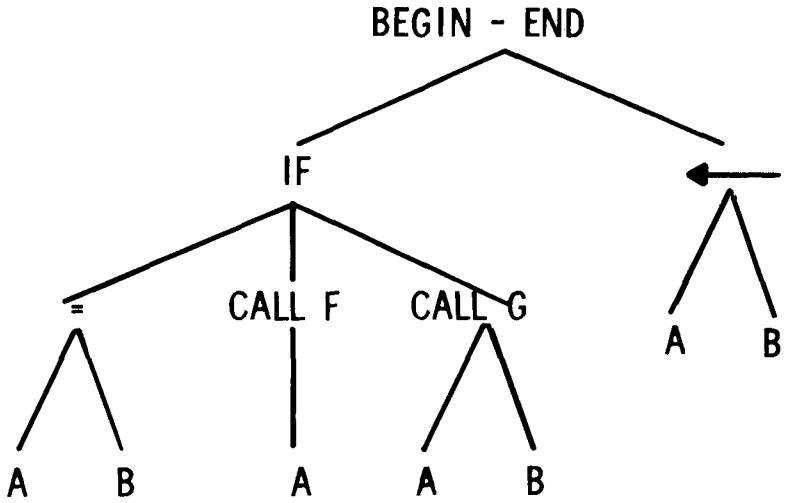



ASSEMBLY LANGUAGE ARRAY
FIGURE 1

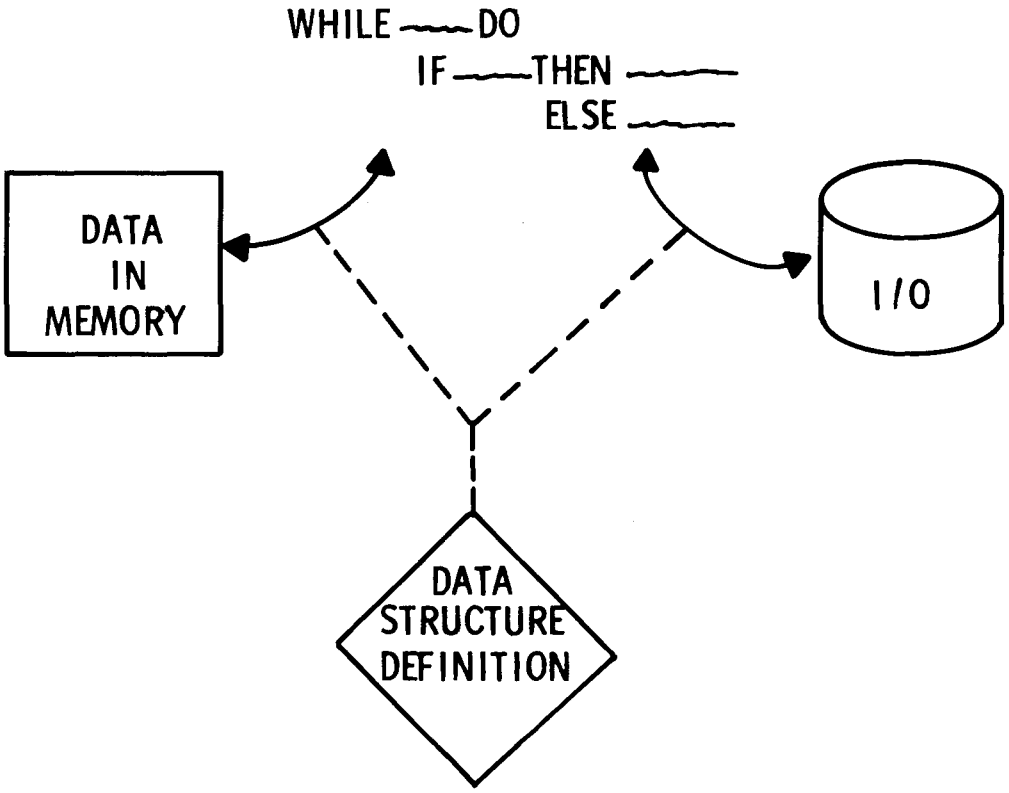
A




FORTRAN ARRAY
FIGURE 2




COMPILER SYNTAX TREE
FIGURE 3




HYPOTHETICAL SYSTEM
FIGURE 4