STRUCTURED DATA STRUCTURES*

Ben Shneiderman
Computer Science Department
Indiana University

Peter Scheuermann
Department of Computer Science
State University of N.Y. at Stony Brook

Introduction

Much attention has been focused lately on the notions of structured programming, a crucial factor when dealing with the design of large programming systems. Rather than viewing programming as an art and the programmer as the perpetual artist, structured programming provides a more systematic (and in a way more restricted) approach which facilitates debugging and proving assertions about programs.

One of the ideas developed by advocates of structured programming is the top-down elaboration of program control structures by a recursive process of successive refinements [1]. No such process has been developed for dealing with data structures. The main reason for this is the improper intermixing of the semantic and implementation concepts [2] of a data structure. The failure to distinguish between these concepts is a result of the conflicting factors involved in choosing a data structure: simplicity of element access, minimization of search time, dynamics of growth or elimination of data, simplicity of restructuring and extension, efficiency of storage utilization and others.

With these in mind we propose a "structured" Data Structure facility, which we call a Data Structure Description and Manipulation Language (DSDML) to maintain a similar terminology to other groups (see CODASYL Report [3]). The DSDML provides data structure definitions in addition to the data definitions available in the host language (e.g. PL/1 or COBOL). It will include explicit declarations of commonly used data structures and information about their access and manipulation characteristics. These characteristics include such features as reset pointers or end pointers and search rules.

The main advantages of such a facility can be summarized as follows:

(i) It provides (some) control mechanism over the behavior of data structures. Just as the "go to" is considered harmful to modular programs, in dealing with data structures, we want to eliminate unrestricted branches or edges. The permissible operations in the DSDML are more restricted than those in the CODASYL Report, but allow the creation of a wide variety of commonly used structures. The DSDML will be a useful tool in verifying that our structures are indeed "well-formed" (i.e. enabling us to prove assertions about data structures). Declarations of variables in a programming language provide the compiler with the information necessary to prevent incorrect mixed mode operations from occurring. In FORTRAN, for an example, one can declare variables to be REAL, INTEGER, LOGICAL, DOUBLE PRECISION, or COMPLEX. In a similar way, the DSDML will prevent the programmer from mistakenly converting a binary tree into a three-way tree or from inserting a queue where a ring is expected or from obtaining undesired cycles and so on. The semantics will include provisions to prevent invalid operations. For example, if a one-way list with a bottom pointer and with reset pointers (pointing back to the first node from each node) is defined, it is not possible to make insertions along the bottom pointer or reset pointers.

(ii) Top-down programming can be achieved in terms of data structures, too. This follows from the fact that the DSDML allows for definitions of multilevel data structures, in which the nodes of a given level structure serve as headers for the structures at the next level.

(iii) It might be possible to obtain a more optimal storage allocation. By providing a data structure definition, the compiler (or run-time package) may have the ability to allocate storage in a more efficient way than the usual "space available stack" technique employed for most dynamic storage allocation schemes. For example, by declaring a binary tree with reset pointers, some information is provided about the amount of storage necessary for the nodes of the structure.

Estimation of the DSDML and Comparison with Other Systems

Most commercial systems which provide some kind of data structure description and manipulation facility limit themselves

*This abstract was accepted on the basis of the full paper which will appear in Communications of the ACM.

to one or two structure types. For example, the Integrated Data Store [4] system mainly restricts the programmer to the use of rings and although networks can be obtained by interconnecting different rings, storage is not efficiently used and the underlying structure is hidden behind the superimposed structure supplied by the system. It appears that the problem is created by the fact that their "world-view" is essentially based on records with fixed format, fixed fields (a specified number of fields have to be in a record). This consideration limits the access mechanisms to operate on specialized structures.

The major design effort by the Codasyl Data Base Task Group [3] to develop a Data Description Language and a Data Manipulation Language for a generalized Data Base Management System was very much concerned with selecting a useful set of data manipulation primitives. They have solved some of the problems posed by earlier systems, but the complexity of the system makes it (sometimes) difficult to follow the effect of the operations. The situation is that the same operations might produce different effects depending on a detailed set of declarations made in a complicated Data Definition Language. For example, the STORE operation creates a node and possibly connects it to one or more structures, depending on how the structure has been defined. The complexity and danger in deleting nodes has led the designers to define no less than four DELETE operations each of whose functions vary depending on which node is referenced and upon the structure definition. As a result, the user who implements a complicated structure in this system would have to be extremely careful when invoking the operations.

A different approach to describe the various structures and operations may be found in attempts to define a generalized graph programming language. Each of these systems give the necessary primitives to manipulate a graph in an arbitrary manner. Complex structures such as acyclic graphs and networks can be created, but no guarantee is given to prevent the user from obtaining an illegal logical structure. Note that the structures are built in terms of primitives which resemble the level of assembly language operations.

The attitude taken for designing the DSDML was to choose and implement higher level primitives with just the right amount of power. A host programming language like PL/1 provides list processing facilities, allowing the addition or deletion of branches and the creation or destruction of nodes by the use of pointer variables and the ALLOCATE or FREE operations. However, these primitives are too powerful since they permit programmers to create structures which are not well-formed. Unrestricted use of the built-in ADDR primitive can be made by assigning the returned value of ADDR to a pointer variable. Since the argument of the ADDR function can represent any physical address, a pointer can be defined to point to any object in the program.

In choosing the primitives for the DSDML, we did not want restrictive operations which are meaningful in only a very limited environment, nor did we desire a set of primitives which are so powerful that the user can unwittingly destroy a structure. The development of the DSDML depended very much on the restriction to basically two types of structures: linear and tree-like, and combinations of these. The explicit recognition of these distinct classes of data structures enabled us to assign a unique meaning to the operations, which depends only on the class of the structure being operated on.

References

1. Mills, H. Top down programming in large systems. From "Debugging techniques in large systems." Gourant Computer Science Symposium, pp. 41-53.

2. Earley, J. Toward an understanding of Data Structures. Comm. ACM 14, 10 (October 1971), pp. 617-627.

3. CODASYL - Data Base Task Group Report. Available from ACM, 1133 Avenue of the Americas, New York, New York 11306. (April 1971).

4. Integrated Data Store. Honeywell Information Systems, Inc. Wellesly, Massachusetts (December 1971).