A REVIEW OF DESIGN TECHNIQUES FOR PROGRAMS AND DATA

Ben Shneiderman

Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT No. 25

A REVIEW OF DESIGN TECHNIQUES
FOR PROGRAMS AND DATA

BEN SHNEIDERMAN

APRIL, 1975

A Review of Design Techniques For Programs and Data

Ben Shneiderman
Computer Science Department
Indiana University
Bloomington, Indiana 47401

I. INTRODUCTION

The strong advocacy position assumed by several authors of program development papers has been interpreted, by some, as dogmatic and narrowly doctrinaire. In particular, critics have suggested that the top-down programming [1] and stepwise refinement [2] techniques were limited in their applicability to certain types of problems. Gries [3] defends Wirth and acknowledges that there are a variety of program development strategies which can be applied. One of the goals of this paper is to distinguish the development process from the resulting program product and to present a catalog of program designs with examples of their applications.

Wirth is especially careful to point out that "the program and the data must be refined in parallel" as the development proceeds from an abstract problem oriented description to a more machine-oriented, concrete, implementation specification. This discrete stepwise transition along a continuum of data structure possibilities is an extremely difficult process to teach and to learn. Various authors have found anywhere between two and six steps along the path to implementation, but as yet, no clear-cut pattern has emerged [4,5,6,7,8]. A second goal of this paper is to provide a

catalog of possible data structure way stations on the path from the problem definition to the implementation.

Finally the relationship between the data and the program must be defined. The nested block structure of many popular languages provides only one possibility for handling local and global variables. FORTRAN COMMON areas, the COBOL data division and database schema descriptions are other useful constructs for certain problems. Kieburtz [9] proposes a new strategy for sharing data among subprograms and Parnas [10] supports the modular design notion of "information hiding". The third goal of this paper is to catalog alternatives for sharing data among the modules of a program.

None of these catalogs is intended to be complete. Programming is an art [11] and like painting, sculpture, prose or poetry, no complete specification of techniques is possible or even advisable. Still the basic methods need to be learned as a basis for discovery of new techniques. The creative programmer carefully selects the right combination of standard design patterns but is ready to devise new techniques if these fail.

Variety of Programming Situations

It is the wide variety of programming situations that compels us to think about alternate design strategies. First the structure of the problem may have a serious influence on the design of the program. Of course, the structure that we perceive is strongly a function of our experience, but the problem itself has a structure. This structure is partially determined by the presentation or definition of the problem. Some problems have clear, well-understood descriptions, for example, find all the integers in the range 1 to

10,000 which are the sum of their divisors or given an employee number find the employee name. Other problems have a more vague description, for example, write a program to understand spoken English or to produce a weekly payroll. In these cases a great deal of design and specification work must be done before program development can begin. Even if the problem has been stated clearly, the program which solves the problem will still have to be created. Although some work has been done on Problem Specification Languages [12], we are a long way from having a useful general facility. In summary, the problem, as perceived by the programmer, will have a direct influence on the design strategy.

A second factor which influences the program design is the size of the problem. Although it is difficult to provide a good metric of problem or program size or complexity we will use the number of program statements as a rough guide. A small program (less than 100 statements) often yields to a simple straightforward method. If the underlying problem is complex more involved procedures can be applied. For medium sized programs (100 to 1000 statements) careful planning and study of the problem are necessary to select a useful strategy. For large programs (1000 to 10,000 statements) greater care and effort are required. Incorrect designs can destroy the usefulness of the resultant program or make completion impossible. An evolutionary development strategy which allows for a phased or staged implementation may be required. Failure at any one point should not jeopardize the completion since sufficient time for backtracking and re-design should be included in the schedule. In the case of very large programs (more than 10,000 state-

ments) several levels of design work are necessary. The problem must be divided into distinct components that multiple design teams work on independently. Each component can then be designed and developed separately. Here it seems inconceivable that anything but an evolutionary process would allow for partial development, testing, redesign, and improvement. The first versions should solve a useful subset of the problem specifications with modest attention to efficiency. When this initial version works, the remaining specifications should be handled and the efficiency issues dealt with. Finally, improvements and modifications of the original specifications should be implemented.

The third key factor in the program development process is the experience of the programmer/designer. Familiarity with similar problems is most helpful since the new problem can be analyzed and related to previous successful experiences. A useful set of abstractions, such as graph theory, matrix manipulation, queueing models, set theory, or decision tables, can greatly assist in the analysis. Knowledge of programming concepts such as hash coding for symbol tables, bit maps, linked list strategies or string manipulation techniques can be applied where required.

Previous programming experience also has an influence. Novice programmers who have dealt with simple small programs often find the extra details of subroutine argument passing confusing and useless since they can perceive a linear organization for programs. More experienced programmers recognize the usefulness of modularity and design their programs with independent functionally defined modules. Past experience with particular programming languages

will influence the designer. The kind of modularity afforded by COBOL paragraphing is very different from that offered by ALGOL 60. The shallow subprogram specification permitted in BASIC is worlds apart from the complex and rich facility of LISP. Operating systems also influence program design since they permit a range of features within the job control language, program and data libraries, linkage editing, and independent compilation of subprograms.

The issue of past experience is reviewed in Weinberg's text [13] which provides a broad insightful discussion of "programming as a human activity". He reviews individual personality issues, discusses the group interaction difficulties and describes the effect of various programming tools and aids.

II. PROGRAM DESIGN METHODOLOGIES AND PROGRAM DESIGNS

The first reading of the papers by Mills [1] and Wirth [2] produces a sense of enthusiasm and the feeling that these authors have lucidly explained themselves. They are both proposing a process for developing programs from a problem statement which differs from the classic idea of bottom-up development. This latter process starts with the construction of low-level routines and through a series of integration steps arrives at a complete tree-structured program. The top-down and stepwise refinement approaches start with a high-level description of the program, possibly in English or some "notation which is natural to the problem in hand" [Wirth] and through a series of refinement steps produces a complete program. In both cases the lower levels contain an increasing amount of implementation detail which reflect efficiency issues. Mills suggests that the result is a tree-structured program where the root segment contains a broad general description of the program and the lower levels contain ever more implementation detail:

> Our end result is a program, of any original size whatsoever, which has been organized into a set of named member segments, each of which can be read from top to bottom without any side effects in control logic, other than what is on that particular page. A programmer can access any level of information about the program, from highly summarized data at the upper level segments to complete details at the lower levels.

During the development process the unwritten segments are replaced "with program stubs standing in for functional specifications".

Wirth is never so clear that the resultant program structure will be a tree. He writes:

> Program construction consists of a sequence of refinement steps. In each step a given task is broken up into a number of subtasks.

The reader is left with the impression that a tree-structured program will result but this point is never clarified. The small elegant example he presents does result in a tree-structured program, but we are not given any guidelines for other examples.

Although the top-down and bottom-up processes seemingly produce tree-structured programs, stepwise refinement as defined by Wirth apparently may result in other program structures. Ledgard [14] in a meta-stepwise refinement has taken Mills' top-down notions, Wirth's stepwise refinement, and Dijkstra's [15] level structuring and has proposed a combined program development methodology which produces a level-structured tree-structured program as its product.

Unfortunately, the pursuit of the universal program development methodology is just as empty as the pursuit of the universal programming language or natural language. Each individual or group of individuals will select the methodology which is natural to them and to the problem at hand. Machines deal with precise, uniform and unvarying situations, but humans are much more complex, variable, and resentful of narrow doctrinaire constraints.

Having attempted to separate the program development process from the program product, we now review a variety of program structures. Programmers may select, according to their expertise, from this brief catalog or combine these structures to create new design possibilities.

Single Module Programs

A single module may consist simply of the input of two values. followed by the computation and printing of their average (Figure 1). In other situations a module may be an entire sort routine or even a compiler. The point is that programmers can often understand the solution of their problem as a single step: find the average of two numbers, sort an array of numbers or compile a program. The situations which manifest themselves as one step solutions depend on the programmer's experience and the programming tools available.

Linear Program Structures

An obvious program structure is a linear sequence of modules (Figure 2). To the novice, a simple program might be composed of three modules which mimic the read-process-print sequence; read a pair of data values, compute the average and print the result. A more sophisticated application might be the classic design for a compiler: lexical analysis, syntactic analysis, code generation, code optimization. A still higher level situation is the usual compile, link-edit and execute sequence. In each of these situations the execution sequence is linearly through the modules; once a module is exited control never returns to it. Hopefully the modular design is on functional separation enabling independent modification. A bad choice of modules can result in a program which is difficult to comprehend, debug, and modify.

A generalization of linear programs would be the repetition of the program until some termination condition is reached. Thus, the compile-load-execute sequence may be repeated for several input programs of the read-process-print sequence may be repeated for several sets of data.

Tree-Structured Programs

This commonly used structure is the result of a classic top-down or bottom-up development or of a stepwise refinement process (Figure 3). The root module contains an outline or general image of the function of the program, while the lower levels contain increasing amounts of implementation details. The root module might contain instructions which merely invoke read, process, print, and termination modules. The read module may invoke modules to process initialization information, raw data and summary data and to print the input data. The print module may invoke heading modules, column print modules and footing modules. Wirth's [2] solution of the eight queens problem is an example of a simple tree-structured program.

We distinguish linear program structures from unary trees. In linear program structures the execution of each module proceeds in sequential manner to termination while in a unary tree program structure control passes from the root module to the other modules and can oscillate between modules by a series of invocations and returns.

Level-Structured Programs

A number of authors have recognized that by the proper organization of a program it is possible to functionally separate independent levels or layers (Figure 4). Typically the higher levels are close to the problem description while the lower levels have increasing amounts of implementation detail. In such a design strategy it should be possible to develop or modify each level independently [16]. The basic restriction is that modules at a specific level invoke modules only at the next lower level. Lower level modules can never invoke higher level modules. The program structure was elegantly

applied and described by Dijkstra [17] in the T.H.E. operating system. His six-level approach was: level 0, real time clock and processor allocation; level 1, segment controller; level 2, message interpreter; level 3, buffering of data; level 4, independent user programs; and level 5, the operator. Standish [18] describes a generalized five-level structure with specific application to orbital mechanics: level 0, knowledge of machine representations (machine words); level 1, programming language implementation techniques (list structures); level 2, knowledge of representation techniques in programming languages (polynomials); level 3, mathematical knowledge (Taylor series); and level 4, problem domain knowledge (orbital mechanics). The Data Independent Accessing Model (DIAM) [19] contains four levels: level 0, physical device model; level 1, encoding model; level 2, string model; and level 3, entity set model. Finally, Parnas [10] gives a short example and suggests that the level separations should be made on the basis of "information hiding" [10]. Details of the implementation within a module should have no effect on other modules. Each module performs a well defined function, but the method of performance is hidden from other modules.

There is some conflict over the development process for level-structured programs. In his description of the implementation of the T.H.E. operating system, Dijkstra [17] clearly states that the lowest numbered levels having the closest ties to the machine hardware were developed and tested first. In his elegant description of a line printing control program, Dijkstra [15] presents level-structured programs as a "necklace of pearls" where each pearl is a module at a specific level. In this description the higher levels,

which are closer to the user's perception of the problem, are developed first. Both strategies seem viable. In the former case the definition of the lowest was strongly influenced by the hardware, and the higher levels could easily be altered to meet changing demands and desires. In the latter case the highest level routine was most clear in the programmer's mind, while the lower levels could be more easily modified to fit a particular environment. A third alternative of developing all levels in parallel is possible (and advisable for large systems) if the levels are functionally well defined and the interfaces rigidly determined. Of course, level-structured programs can be tree structured as well.

Network Program Structures

Often the complex interrelationships among the components of a problem do not permit a simple tree or level-structured decomposition. In this case, a network of functionally defined modules may be the most obvious program structure (Figure 5). If done poorly this method may result in a complex and confusing program since any module may invoke any other module with potentially chaotic results.

The co-routine structures described by Dahl and Hoare [21] are excellent examples of SIMULA 67 network program structures. His program for reading 80-column data cards and printing 125 character lines required incard, outcard, dis-assembler, squasher, assembler, and main program modules. In a simple SNOBOL SCRABBLE playing program [22] there was a network interaction among the game manager, referee, and player who had access to the dictionary module while the scorer could be invoked by the game manager and the player. As a final example, network program structures have been efficiently employed in human speech understanding systems [23].

III. DATA STRUCTURE DESIGN

While Wirth [2] repeats the admonition to define the data structure in parallel with the program structure he does not provide guidelines for the refinement of data. In a later paper [24] he suggests that a language like PASCAL, which provides extensive facilities for structuring data, would be useful. Excellent extensive discussions of data structures are found in Knuth [25,26] and Hoare [27].

The multi-level structures proposed by Shneiderman and Scheurmann [28] can be useful in a stepwise refinement situation when combinations of linear and tree structures are required. The general problem of the refinement of data structures is more complex. A number of authors distinguish between abstract and concrete representations [8] or modelling and implementation domains [29]. There is, in fact, a continuum from the abstraction to the implementation and, to complicate matters, there is no agreement about where a particular data structure belongs on this continuum. This disagreement is a result of the confusion over the distinction between logical and physical structures; data structures, and storage structures. References to stacks, queues, binary trees, etc. often mix logical and physical aspects. Even the logical description of a simple structure such as a stack may be incomplete since the permissible operations are not fully described. While stacks are usually characterized by the operations push and pop, some descriptions permit the examination of the top element without the necessity of a pop operation. Still other descriptions permit examination of the internal nodes. The automata theorists have long ago made the distinction

between stack and pushdown-list automata, but data structure researchers rarely make this distinction explicit. At the implementation level a stack may be constructed by the use of contiguous storage locations and a top pointer or by a linked list strategy. In either case, the issue of actual or relative storage addressing is often treated thinly.

While we await the development of more precise logical and physical description facilities [5,30,31] and the mappings between them, the practical problems of data structure refinement persist. To a assist the programmer we provide a classification of data structures and examples at the logical and physical levels.

Single Node Data Structures

The number of bits in a simple single node data structure depends on the operation that is applied. In the simplest case a single bit, representing a Boolean value, may be considered as the entire single node data structure.(Figure 6). More commonly, a single byte or word which stores the value of a single variable is a node. If array operations are permitted, then the entire one, two, or n-dimensional array is treated as a single unit. For certain operations a COBOL structure, PL/I structure, or PASCAL record are a single node. At a still higher level, operating system utilities deal with entire files as a single node. Since the operations at this single node level are relatively simple, careful checking at compile and execution time can be made to guarantee the correctness of the operations.

Linear Data Structures

Examples in this category include strings of bits or characters,

subscripted arrays, COBOL or PL/I structures (although there may be a tree structure of names, the data is in a linear form), linear linked lists and sequential files (Figure 7). A number of special purpose structures fit into this category as well: stacks, queues, diques and rings. In each of these cases the permissible operations of insertions, deletions, copy, update, and searching can be carefully and clearly defined so that accurate checking can be done to ensure the correctness of the operations and the well-formedness of the results [27,29,32].

Tree-Structured Data Structures

The most popular tree structure is the binary search tree and its variations: height balanced, weight balanced, minimum path length (internal, external or total) [26,33] (Figure 1). Multiway trees, digital trees, digital tries and B-trees are important variations. Tree structures appear in more obviously problem oriented situations such as family trees, organization charts, multilevel indexes (or tables of contents) and product distribution trees.

Insertions or deletions of subtrees or leaf nodes are usually straightforward, but insertions and deletions in the interior of the tree often require complex algorithms. Few systems contain declarative facilities for tree structures which permit the user to specify the attributes of the tree and thereby enable compile and execution time validity checks. The most advanced work in this area can be seen in various database management systems which contain data definition languages permitting tree-structured data structures, for example, System 2000, IMS, or DBTG-like systems.

Acyclic Network Data Structures

This class of structures is characterized by the existence of at least one node for which there is more than one search path and no cyclic search paths (Figure 9). The standard PERT chart, flow graphs and legal precedent citations are classic examples of these structures. Indexed sequential access methods which permit direct access of keyed records through an index or sequential search through the records are available on many computer systems. Sparse array techniques and Iliffe vectors are implementation level network data structures designed to save unused space or eliminate redundancy.

Although the search paths in these structures are all finite, insertion and deletion can be problematic. In the special case of indexed sequential access methods the insertions and deletions are hopefully performed correctly by the manufacturer supplied software, but no generalized facility provides a data structure declaration which permits the user to specify his/her intentions.

These acyclic network data structures may exhibit a level structured nature as well; the levels of an indexed sequential file or certain tree-structured indexes (level 0, index to countries; level 1, index to provinces; level 2, index to cities) are but two examples.

These complex structures have only the limited restriction that each node of the data structure must be accessible along at least one search path (Figure 10). Generalized communication or transportation networks and scientific journal citation search paths can contain cyclic structures. Semantic networks for representing knowledge and natural language comprehension systems display cyclic structures as well [34,35,36].

Cyclic networks are difficult to deal with since not only are insertion and deletion difficult, but search paths can have infinite length.

This crude classification of data structures is based on the topological configuration and the operations that are applied. Unfortunately no algorithm exists to guide the programmer in the proper selection of a data structure. The impression that the programmer has concerning the problem statement will lead to the selection of a particular abstract structure. This in itself can be a complex decision, but it should be independent of implementation considerations. Then the abstract structure is refined and converted into an implementation structure based on various efficiency considerations. If efficiency is an overriding concern, then the hardware details will have to be examined carefully and an implementation facility which permits precise low-level control is necessary. More commonly, the abstract structure can be molded to fit the structures commonly available in high-level programming languages: arrays, PL/I or COBOL structures, sequential files and the various data types. If complex network structures are to be used, a graph oriented language such as GROPE [37] may be used to simplify implementation. For very large volumes of data the facilities of a database management system may similarly simplify the task. Once again the experience of the programmer and the availability of the right software aids will substantially affect the decision.

IV. SHARING DATA AMONG MODULES

Having selected a program design and a data structure design, the programmer must decide how the data is shared by the different modules. Assembly language provides the most powerful and flexible facilities, but the least protection from error. The high level languages and database management systems provide less flexibility but in some cases greater protection. In this section we examine some of the alternatives to sharing data among modules.

Restricted Access

Using this technique, the data is tightly controlled and access is limited to precisely those modules which require access. This technique can be accomplished in FORTRAN if there is no auxiliary storage, no COMMON areas, and if array overruns are not permitted. Stated positively, all data is transferred as arguments to subprograms. Of course a subprogram may maintain its own data in local arrays or variables. Parnas' notion of "information hiding" can be successfully implemented under such a discipline [10]. The higher level modules do not have access or knowledge of how the low level modules store their data. The low level modules are defined by their function; the implementation technique may be changed without affecting the high level modules. Similarly, the high level modules may also be modified independently. Furthermore, the low level modules cannot access the data in the high level modules. Kieburtz [9] discussed an extended version of this idea which further refines the passed arguments as having READ-WRITE, READ-ONLY, and NO-ACCESS attributes. A WRITE-ONLY attribute might be a worthwhile addition to permit low level modules to only set values.

Block-Structured Data Sharing

This technique is available in PL/I, ALGOL, and other block-structured languages which permit low level procedures to have access to all declared variables of the procedures in which they are nested. This results in low level modules having access to not only their own data, but the data of modules at higher levels. Kieburtz [9] characterizes this process as asking the janitor to empty your office garbage pail and them giving him/her the keys to your desk, office, car and home. Such broad powers can lead to unpleasant results in either case. On the other hand, the relative ease of implementation, the simplicity of use, and the power of such a mechanism is tempting.

Global Access

Early FORTRAN programmers accomplished this technique by declaring a large COMMON area and then provided each subprogram with a copy of the COMMON declaration. COBOL programmers were led to this method by the existence of the DATA DIVISION for declaring all variables. Contemporary database management systems carry this process one step further by having an independent compilation of the "data schema". As Bachman [38] has remarked, this doctrine fundamentally changes the perspective of the programmer. No longer is the program seen as a processor of data which absorbs input and produces output. The program and the programmer are but a small mechanism navigating through a vast sea of data. The data has a life of its own and exists independently of any particular program or application. This is the idea of data independence. Any number of programs may add, delete, update, or retrieve from the data structure, and none

need be aware of the implementation details. The data itself is a model of its real world counterpart: corporation, geographic area, molecular structure, etc.

V. CLASSIFICATION OF PROGRAM AND DATA STRUCTURE

The obvious parallel classification of program structures and data structures is not coincidental. Using graph theoretic abstractions enables us to perceive the close relationship between these fundamental constructs. The notion of an unlabeled, well formed list structure developed earlier [32] is matched by the notion of a well formed program structure (WFPS). In a WFPS there are no "unsatisfied externals" (referenced subprogram names which have not been provided), and all subprograms are "invokable" from the initially invoked unique main program. In a data structure these restrictions can be interpreted as the absence of invalid pointers and the reachability of every node from the unique entry node (square nodes in figures).

For simplicity of discussion let us establish the following classification of the well formed structures:

Class I: Linear structures

Class II: Tree structures

Class III: Acyclic network structures

Class IV: Cyclic networks

Notice that each class contains the lower numbered classes and that Class IV includes all of the well formed structures.

Each of the four classes of structures is recursively enumerable and recursive. This can be demonstrated since we can generate all of the possible directed graphs. For a given number of nodes, n, we generate all the 2^{n*n} Boolean adjacency matrices. For each matrix the transitive closure matrix can be generated and the reachability of every node from the entry node can be tested. Graphs satisfying

this test constitute Class IV. If the transitive closure matrix contains no ones along the diagonal the graph is free from cycles and belongs to Class III. If every node in the adjacency matrix has in-degree one, then the graph is a Class II structure. Finally, if every node in the adjacency matrix has in-degree one and out-degree one, it is a Class I structure. Performing this procedure for $n = 1,2,...$ we can generate every member of every class of structures. Given a graph with a finite number of nodes n^* we can similarly test it for membership in any of the four classes.

In some instances the parallels between classes of data and program structures are obvious. If the examination of each node of a linear data structure takes finite time, then the examination of the entire structure takes finite time. Similarly, if the execution of each module of a linear program structure takes finite time, then the execution of the entire program takes finite time. In fact, the examination (execution) time of a linear data (program) structure is simply the sum of the examination (execution) times of the nodes (modules).

Further parallels exist between classes of data and program structures. To examine each path of an acyclic network data structure a stack of node addresses is necessary. Similarly, to perform all execution sequences of an acyclic network program structure, a stack of module addresses is necessary.

VI. CONCLUSION

Programmers must have more than a program development methodology in mind when developing programs. They must consider the possible resulting program structures. A wide variety of data structures, abstractions, and implementations, each having their advantages and disadvantages, are available. Finally, the designer must select a method for sharing data structures among the program modules.

On a more theoretical level, the parallel classification of program and data structures gives new insight to the programming process.

Summary

The proliferation of papers on programming methodology focus on the program development process but only hint at the form of the final program. This paper distinguishes between the development process and the program product and presents a catalog of possible program organizations and data structures with examples drawn from the published literature. The methods for sharing data among modules and a classification scheme for programs and data structures is presented.

REFERENCES

1. Mills, H. Top down programming in large systems. Debugging Techniques in Large Systems, R. Rustin (Ed.), Prentice-Hall, Inc. (1971), 41-55.

2. Wirth, Niklaus. Program development by stepwise refinement. Comm. ACM 14, 4 (April, 1971).

3. Gries, D. On structured programming. Letter to the ACM Forum, Comm. ACM 17, 11 (Nov., 1974), 655-657.

4. Earley, J. Toward an understanding of data structures. Comm. ACM 14 (1971), 617-626.

5. Earley, J. Relational level data structures for programming languages. Acta Informatica 2 (1973), 293-309.

6. Childs, D.I. Feasibility of a set-theoretical data structure--a general structure based on a reconstituted definition of relation proceedings. IFIP Congress, North Holland Pub. Co.

7. ----------. Extended set theory: a formalism for the design implementation and operation of information systems. Unpublished manuscript.

8. Schwartz, J.T. Abstract and concrete problems in the theory of files. Data Base Systems, R. Rustin (Ed.), Prentice-Hall (1972), 1-22.

9. Kieburtz, Richard B. Steps toward verifiable programs. Technical Report No. 12, Department of Computer Science, State University of New York at Stony Brook (Nov., 1972).

10. Parnas, D.L. A technique for software module specification with examples. Comm. ACM 15, 5 (May, 1972).

11. Knuth, D. Computer programming as an art. Comm. ACM 17, 12 (Dec., 1974), 667-673.

12. Teichroew, D. A survey of languages for stating requirements for computer based information systems. Proc. AFIPS 41, AFIPS Press, Montvale, NJ (1972)

13. Weinberg, G. The Psychology of Computer Programming, Van Nostrand Rheinhold Co. (1971).

14. Ledgard, Henry F. The case for structured programming. BIT 13 (1973), 45-57.

15. Dijkstra, Edsger W. Notes on structured programming. Structured Programming, Academic Press (1972).

16. Woodger, M. On semantic levels in programming. Information Processing 71, North-Holland Publishing Co. (1972).

17. Dijkstra, Edsger W. The structure of the "THE"-Multiprogramming System. Comm. ACM 11, 5 (May, 1968).

18. Standish, Thomas A. Representation cascades, heterarchy and knowledge structures in automatic programming. Bolt Beranek and Newman (Oct., 1973).

19. Senko, M.E.; Altman, E.B.; Astrahan, M.M.; and Fehder, P.L. Data structures and accessing in data-base systems (three parts). IBM Systems Journal 12, 1 (1973), 30-93.

20. Parnas, D.L. On the criteria to be used in decomposing systems into modules. Comm. ACM 15, 12 (Dec., 1972).

21. Dahl, O.J., and Hoare, C.A.R. Hierarchical program structures. Structured Programming, Academic Press (1972).

22. Ambrose, Margaret, and Rasche, Barbara. A computerized scrabble player. Unpublished report, Computer Science Department, Indiana University (1974).

23. Erman, L.D.; Fennell, R.D.; Lesser, V.R.; and Reddy, D.R. System organization for speech understanding. Proc. of the Third International Joint Conference on Artificial Intelligence (1973).

24. Wirth, Niklaus. On the composition of well-structured programs. Computing Surveys 6, 4 (Dec., 1974).

25. Knuth, D. The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, MA (1968).

26. --------. The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, MA (1973).

27. Hoare, C.A.R. Notes on data structuring. Structured Programming, Academic Press (1972).

28. Shneiderman, Ben, and Scheuermann, Peter. Structured data structures. Comm. ACM 17, 10 (Oct., 1974).

29. Rowe, Lawrence A. Modelling structures formalism. Technical Report No. 52, Department of Computer Science, University of California, Irvine, CA 92664 (1974).

30. Stored Data Definition and Translation Task Group Report (to be published 1975).

31. Sibley, Edgar H., and Taylor, Robert W. A data definition and mapping lanauage. Comm. ACM 16, 12 (Dec., 1973), 750-759.

32. Shneiderman, Ben. Data structures: description, manipulation, and evaluation. Ph.D. Thesis, Department of Computer Science, SUNY at Stony Brook, NY (1973).

33. Nievergelt, J. Binary search trees and file organization. Computing Surveys 6, 3 (Sept., 1974).

34. Shapiro, S.C. A net structure for semantic information storage deduction, and retrieval. Second International Joint Conference on Artificial Intelligence (Sept. 1-3, 1971).

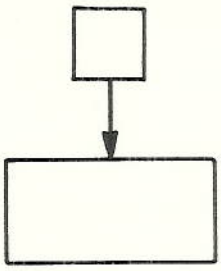
35. Quillian, M.R. Semantic memory. Semantic Information Processing, M. Minsky (Ed.), MIT Press, Cambridge, MA (1968), 227-270.

36. Simmons, R.F. Semantic networks: their computation and use for understanding English sentences. Computer Models of Thought and Language, Schank & Colby (Eds.), W.H. Freeman and Co., San Francisco (1973).

37. Friedman, D.P. GROPE: a graph processing language and its formal definition. Ph.D. Dissertation, University of Texas at Austin (1973).

38. Bachman, Charles. The programmer as navigator. Comm. ACM 16, 11 (Nov., 1973).

Typed by Christopher Charles



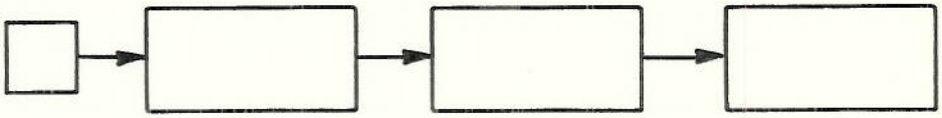


Figure 2: Linear Program

Figure 1: Single Module Program

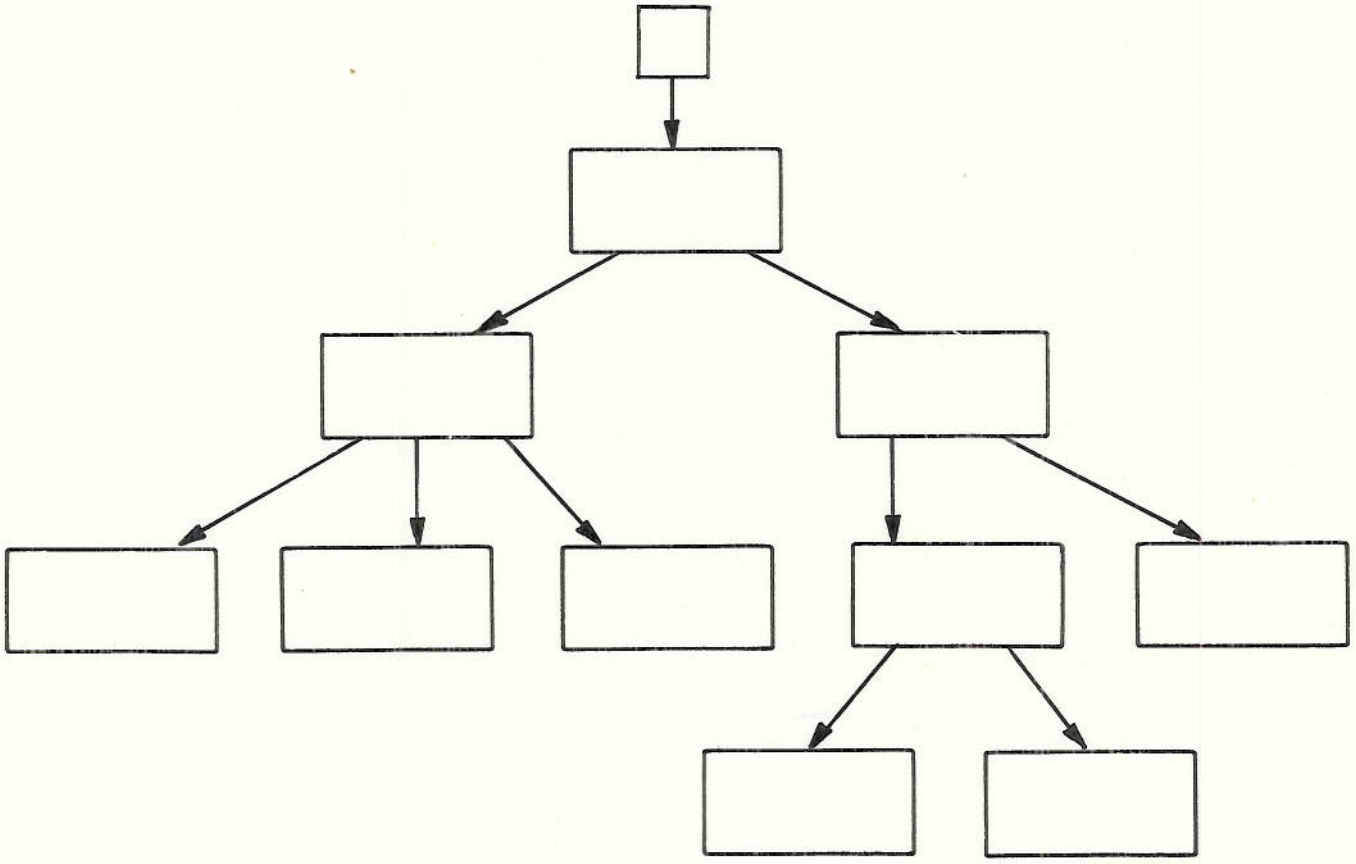


Figure 3: Tree Structure Program

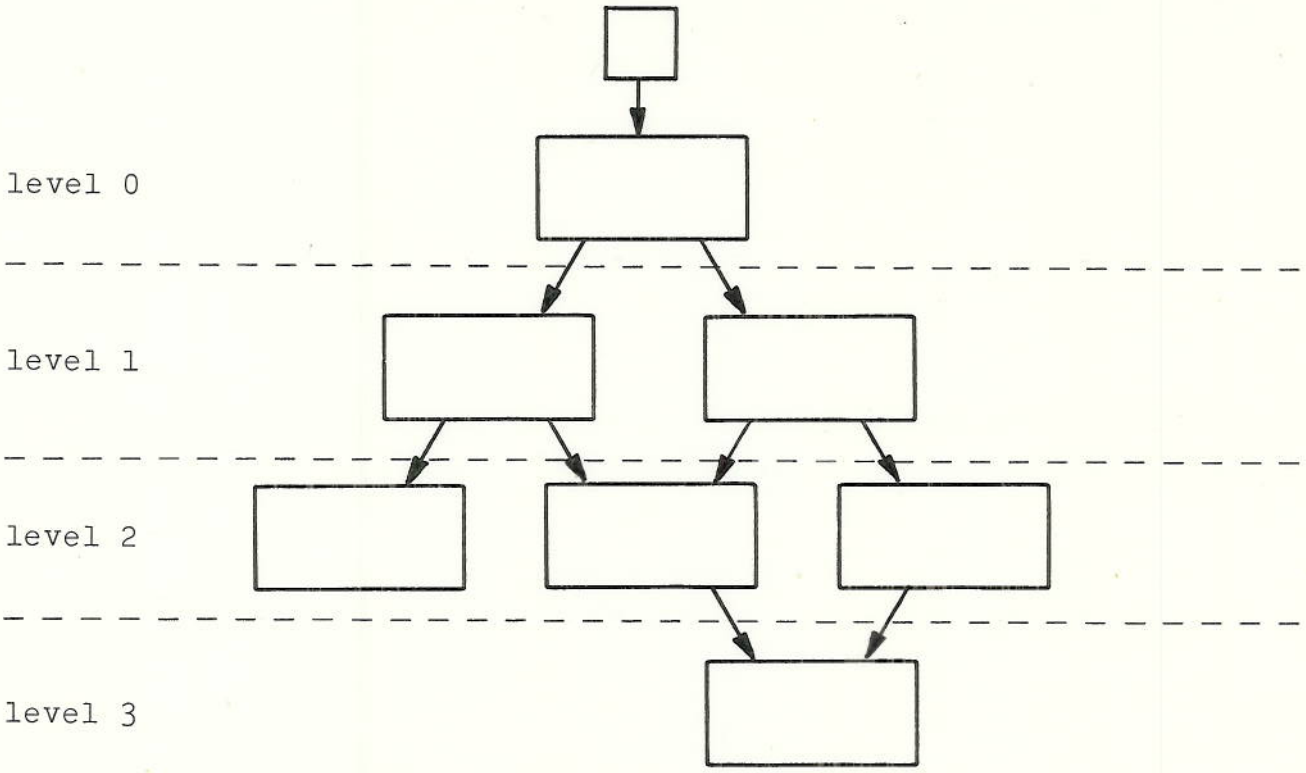


Figure 4: Level Structured Program

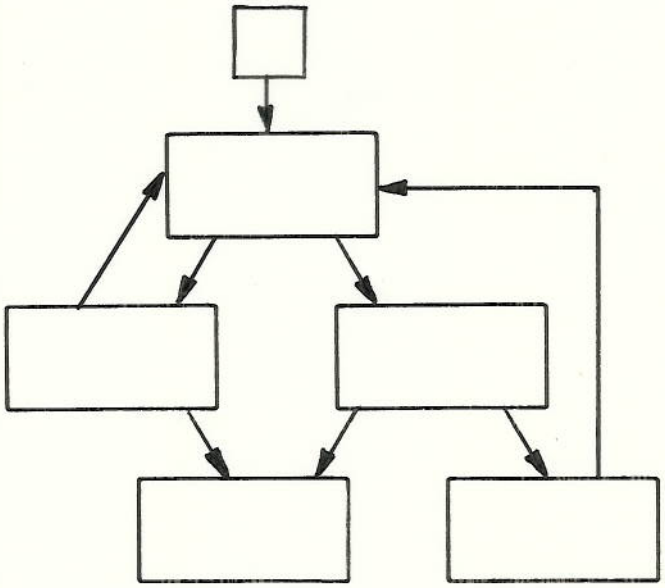


Figure 5: Network Program

Figure 6: Single Node Data Structure

Figure 7: Linear Data Structure

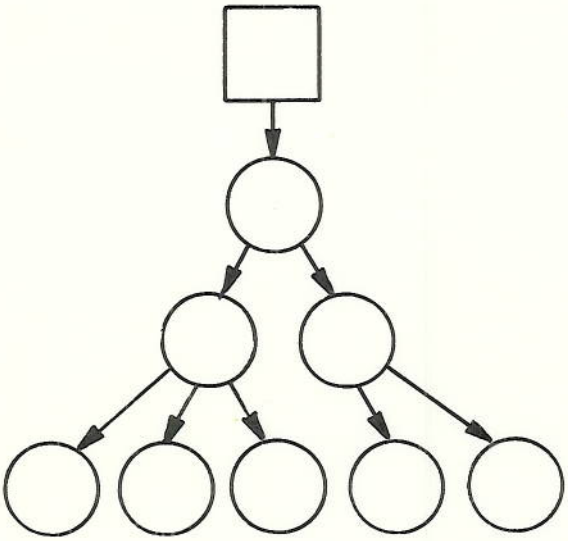


Figure 8: Tree Structured Data Structure

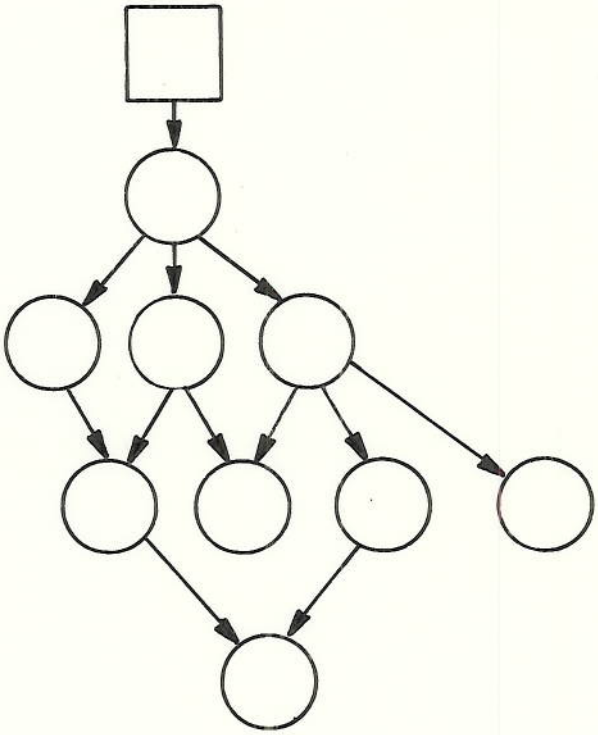


Figure 9: Acyclic Network Data Structure

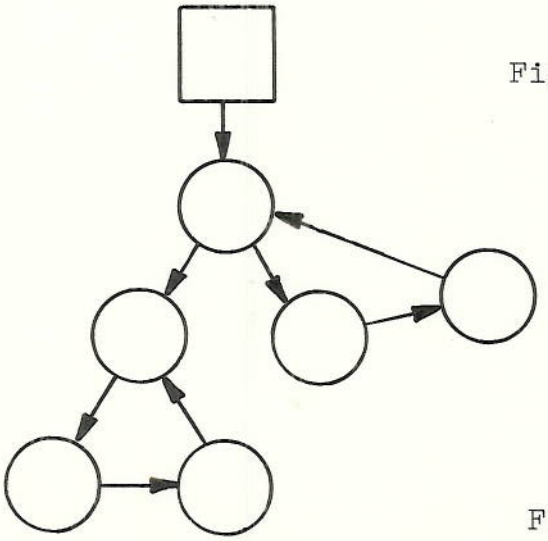


Figure 10: Network Data Structure