MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 70

Modularity*

by

Jack B. Dennis

Notes Prepared for an Advanced Course on Software Engineering,
Technical University of Munich, February 1972

June 1972

This research was done at Project MAC, MIT, and was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr N00014-70-A-0362, and in part by the National Science Foundation under grant GJ-432.

1. INTRODUCTORY CONCEPTS

The word "modular" means "constructed with standardized units or dimensions for flexibility and variety in use." Applied to software engineering, modularity refers to the building of software systems by putting together parts called program modules.

The dictionary meaning applies very well in, for example, the construction materials trade: In the United States floor tile comes in nine-inch squares (the modules) which may be conveniently adjoined to fill up any shape of floor area with just a bit of trimming at the boundary. A great variety of patterns may be produced by using modules of differing color and texture.

In modular software, clearly the "standardized units or dimensions" should be standards such that software modules meeting the standards may be conveniently fitted together (without "trimming") to realize large software systems. The reference to "variety of use" should mean that the range of module types available should be sufficient for the construction of a usefully large class of programs.

In July 1968 a two-day symposium was held in Boston on the subject of Modular Programming [1]. The preprints of papers for this meeting probably form the only collection of material representing a significant range of viewpoints on the nature and purpose of modular programming. In this collection of papers various concepts of program modularity are described ranging from vaguely defined principles to definitive formal concepts. Yet there is an important objective common to all. It stems from recognition of the high cost of producing correctly functioning software systems; it is to realize the benefits promised by the saying: "divide et impera."

To many people in software practice, modular programming means the division of the whole of a program into parts so "the interactions between parts are minimized" or so "the parts have functional independence." Frequently, the assumption is made that in modular programming the program and its parts are designed at the same time and under the same authority. There is little appreciation that the objective of simplifying program construction by dividing the task into parts has definite implications regarding the structure of programs and the characteristics of computer systems.

Nevertheless, several thoughtful and precise notions were also expressed at the symposium. The designers of the Integrated Civil Engineering System (ICES) [2] emphasized the importance of being able to use together independently written program modules. Boebert [3] also recognized that the success of modular programming depends on characteristics of the linguistic level at which the modules are expressed. He points out that modularity should be regarded as a property of a computer system or linguistic level rather than a property possessed or not possessed by some program. E. W. Dijkstra's concern [4] with principles of "structured programming" is closely related.

Our goal in these lectures is to develop further understanding of these notions of modular programming, and to derive their implications for the design of programming languages and computer systems.

1.1. DEFINITION OF MODULARITY

We take the following statements to be the objectives of modular programming:

1. One must be able to convince himself of the correctness of a program module, independently of the context of its use in building larger units of software.
2. One must be able to conveniently put together program modules written under different authorities without knowledge of their inner workings.

These statements embody the concept of "context-independence" discussed by Boebert [3], and the concept of non-interference stated by Dijkstra [4].

We consider modularity to be a property of computer systems:

> A computer system has modularity if the linguistic level defined by the computer system meets these conditions: Associated with the linguistic level is a class of objects that are the units of program representation. These objects are program modules. The linguistic level must provide a means of combining program modules into larger program modules without requiring changes to any of the component modules. Further, the meaning of a program module must be independent of the context in which it is used.

In previous publications [5,6] I have applied the term "programming generality" to computer systems that have this property of modularity.

Two relatively precise concepts regarding the form of a program module occur in the literature on modular programming. On one hand, a module is viewed as a procedure: At any point during the progress of a computation, one module (procedure) may initiate an activation of another procedure by specifying a set of input data. The new procedure activation is carried on, possibly making use of additional procedures, until it terminates, leaving a set of output data for use by the procedure from which it was activated. In this concept, a modular program is a collection of non-interferring procedures. Characteristic of programs constructed as combinations of procedures is the flow of control in a pattern described by a tree. The notion of procedure is a central feature of most modern programming languages, Algol 60 being the classical model [7,8]. But, as we shall see, the procedure in its usual form does not meet our requirements for [illegible] modular programming.

On the other hand, a module may be conceived as an entity that is joined to other modules by communication links. Each module receives [illegible] data over its input links, transforms it in some way, and sends it on to other modules over its output links. In this picture, each module is continuously active, processing data so long as inputs are available. [illegible] Concurrency of operation is an inherent part of this notion of modularity. The links connecting one module to another are thought of as channels through which data flow. First in-first out queues may be introduced in the links as a means of improving the efficiency of an implementation without altering the semantics of a modular program. This form of modular programming is advocated [3,9] for data processing applications where the links are implemented as "buffer files." The concept is closely related to Conway's coroutines [10] and Dijkstra's cooperating sequential processes [11]. The only programming languages having features suitable for this form of modular programming are certain simulation languages, in particular Simula 67 [12].

In these lectures, we study the limitations on modular programming found in the linguistic levels defined by certain computer systems. We consider the well-known programming languages, Fortran and Algol 60, to understand the issue of clashes of identifiers. We then consider the problems of handling dynamic data structures in modular programs and the problems of combining program modules expressed in different representations. Multics is studied as a system in which sharing of procedures and data is possible with considerable generality. Finally, we consider the definition of a hypothetical linguistic level within which a very general form of modular programming is possible.

1.2. MODULARITY IN FORTRAN

Let us start by considering the forms of modular programming possible at the linguistic level defined by the ANSI Fortran language standard. We will not consider here the features of Fortran for input, output and transfer of data between storage levels, and we assume that subprograms in other languages are not permitted.

A Fortran program consists of a sequence of statements that make up a main program and a collection of separate sets of statements that represent function subprograms and subroutine subprograms. Since there is no provision in the Fortran standard for combining separately written Fortran programs, a complete Fortran program consisting of main program and subprograms cannot serve as a program module at the linguistic level defined by the standard.

The obvious choice as a unit for modular programming is the Fortran subprogram. We encounter one difficulty immediately: The only method of combining several subprograms is to collect them together with a main program, yielding an executable Fortran program. Alas, this is not a program module, and therefore cannot be further combined with other units to form larger modules.

Thus Fortran fails by not permitting hierarchical structure in a modular program. Nevertheless, let us disregard this defect and look for other problems. It will be useful to have in mind a picture of the computation states occurring during execution of a Fortran program. The structure of a state is shown in Fig. 1 as an <u>object</u> of the variety used by the IBM Vienna Group in their work on formal definition of programming languages. This object represents an execution state, and therefore the operation of putting several modules together to form a program has been performed. The 'text'-component of the state is an object having as its components the compiled form of each source language subprogram, including one subprogram identified as 'main', and the remaining subprograms identified by names chosen by their programmers. The 'private'-component of the state has, as its leaf nodes, data entities and other values that are accessed only during execution of the corresponding subprogram text (except , of course, when these values are passed as arguments to other subprograms). These values are values of Fortran variables and arrays not mentioned in COMMON statements of the source language subprogram, and additional variables generated by the compiler.

The 'common'-component of the state contains several vectors of data items that are accessed during execution of statements in several subprograms. The computation state of a Fortran program has a fixed structure during execution of the program, only values at the leaf nodes are changed (two exceptions: adjustable arrays and extension of COMMON).

Limitations on the generality of modular programming in a linguistic level arise from points of interaction between program modules. For Fortran subprograms these points of interaction are: calling a function or subroutine; the naming of subprograms; and the use and naming of COMMON.

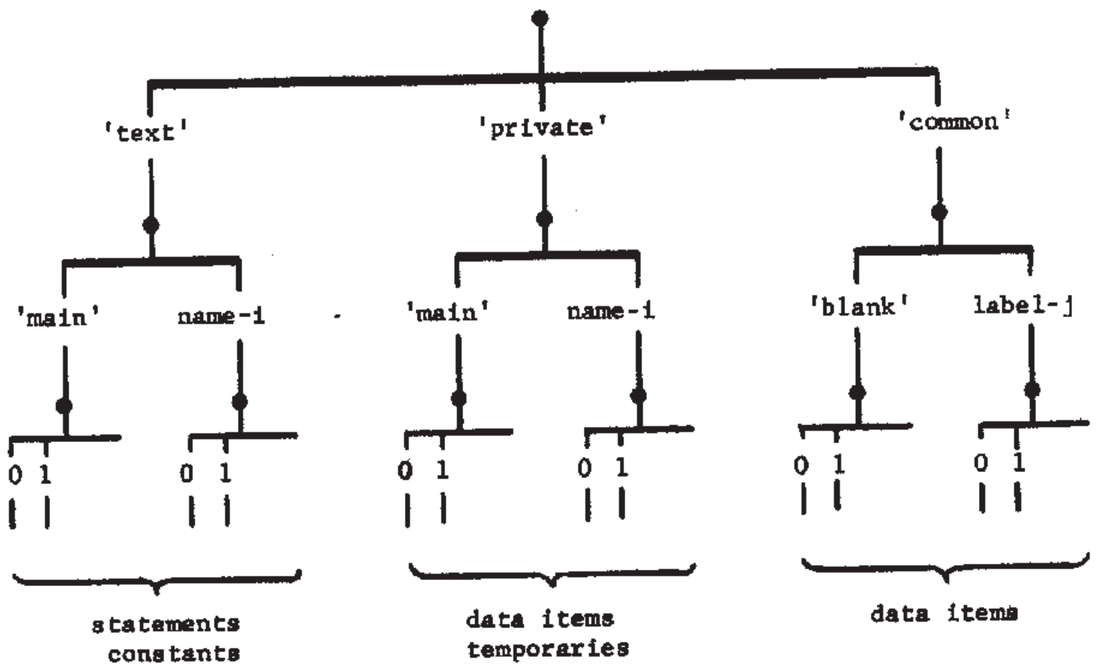


Figure 1. State of a Fortran program.

If two authors have chosen the same name for their independently written subprograms, a clash of names occurs when these subprograms are used together. Similarly, two authors may choose to use blank COMMON for different purposes, or may use the same names for labelled COMMON storage. These are violations of our definition of modularity since alteration of the representation of a module may be required before it can be correctly combined with other modules.

These names clashes may be removed by changing the names of subprograms and choosing new labels for COMMON storage areas. Matters would be more difficult if a program module were to consist of several subprograms, possibly independently written, working together. The problems introduced by attempting to remove clashes through substitution are discussed below.

1.3 MODULARITY IN ALGOL 60

In Algol 60 the procedure is clearly the candidate for consideration as the form for program modules. Since procedures may be combined without modification to form larger procedures, a modular program in Algol 60 may be a hierarchy of modules having an arbitrary depth of nesting. The modules are represented as Algol 60 source text. Compiled Algol programs are not program modules of the Algol-defined linguistic level and cannot be combined.

The instances of the identifier y in the Algol procedure

```
real procedure f(x); real x;
                begin  f := x + y;
                       y := y + 1; end
```

are nonlocal references and therefore y must be a local identifier in some enclosing procedure if the complete Algol program is to be meaningful. A person using procedure f as a module must know about all such external references occurring in f (including those arising within procedures enclosed by procedure f) since external references are a form of interaction of a procedure with external objects. One may wish to use two Algol procedures, f and g, in the construction of a modular program where each procedure makes use of the identifier y to reference some external object. If both procedures are placed in the program as declarations within the same enclosing procedure, there is a clash of names. Thus the use of nonlocal references in an Algol 60 program module is a violation of our concept of modularity.

Several means are available to remove or avoid clashes of names between procedures in Algol 60 programs:

1. Substitute an alternate identifier for each appearance of y as an external reference in one of the procedures. For reasons to be discussed shortly, the use of substitution has significant disadvantages.
2. Enclose one of the procedures within an "interface procedure" that renames the external object by assignment:

```
          real procedure f1(x)  real x;
               begin  real y;
               real procedure f(x); real x;
                    begin f := x + y;
                          y := y + 1; end
               y := y1;
               f1 := f(x)
               y1 := y
               end
```

   This would be awkward to do for arrays, and impossible in Algol 60 if the external object is a procedure. Moreover the choice of identifier y1 depends on the text of the procedure that encloses f1.
3. Enclose one of the procedures in a procedure declaration in which y is a local identifier and formal parameter:

```
          real procedure f1(x,y); real y
               begin
               real procedure f(x); real x;
                    begin f := x + y;
                          y := y + 1; end
               f1 := f(x)
               end
```

This has the effect of substitution for y, but takes effect at procedure entry.

4. Organize the modular program that uses procedure f and g so that the scopes of y do not overlap, by placing the declarations of f and g within distinct procedures or blocks of the program.

The need for any of these schemes would be avoided if y were included as one of the formal parameters of procedures f and g.

The mechanism of non-local reference in Algol 60 was inspired by the evaluation rules of the lambda calculus, and reduces the number of required formal parameters in procedure application. At the interface between independently written program modules, the need to discover and resolve name conflicts makes external references from program modules an unattractive form of interaction. For this reason, we shall adopt as a principle of modular programming, that the only means of communicating data to and from a procedure module is by its formal parameters (and resulting value, if any). Note that this principle rules out "side effects" of the kind observable in Algol 60: Operation of a module can only affect information explicitly passed to it.

1.4. SUBSTITUTION

The names (identifiers) that occur in a representation of a program module can be divided into two groups -- bound and free. By definition, if a name has a free occurrence in the module, it refers to some object bound to the name outside the module. Hence substitution of an alternate name for all instances of the name within the module without rebinding names outside will change the effect of the module. All names that identify primitive operations, constants, etc. of the linguistic level at which the module is expressed are free and have permanently fixed meaning.

Names that are bound in a program module may be uniformly replaced throughout the module without altering its meaning.

If name conflicts occur when two program modules are combined, it is because the same identifier occurs free in both modules, and with different intended meanings. We have seen how such conflicts can arise from function names, subprogram names, and labels for COMMON in Fortran, and from nonlocal identifiers in Algol 60. We have noted that name conflicts may be removed by substituting an alternate name for a free name at each appearance as an external reference within a program module. This substitution must be made before the modules to be combined have lost their separate identity, for example before an Algol program is compiled or before Fortran subprograms are linked.

There are several difficulties with name substitution as a means of resolving name conflicts. Firstly, performing the substitution may involve considerable information processing. A program module may itself be a combination of many simpler modules and the substituted name must be chosen so that no new conflicts are generated either inside or outside the program module.

The most important consequence of name substitution is that the possibility of sharing a representation of a program module among users of the module is foreclosed. A substitution required to remove a conflict cannot be made in a representation of a module already in use as part of another modular program. A copy of the module must be made first.

The importance of being able to share representations of program modules is gradually becoming recognized. In Multics [13, 14], the idea has been carried furthest: Every procedure written for operation in the system may be shared by all authorized users without the making of copies. We expect sharing to be increasingly important in future computer systems. Therefore, as a requirement of our concept of program modularity, we adopt the rule that names occurring free in a program module may refer only to fundamental entities of the linguistic level.

1.5 REFERENCES

1. T. O. Barnett, Modular Programming: Proceedings of a National Symposium, Symposium Preprint. Information and Systems Press, Cambridge, Massachusetts 1968 [Out of business].

2. J. M. Sussman and R. V. Goodman, Implementing ICES module management under OS/360. Published in [1], pp 69-84.

3. W. E. Boebert, Toward a modular programming system. Published in [1], pp 95-111.

4. E. W. Dijkstra, A constructive approach to the problem of program correctness. BIT (Nordisk Tidskrift for Informations-behandling), Vol. 8, No. 3, 1968, pp 174-186.

5. J. B. Dennis, Future trends in time-sharing systems. Time-Sharing Innovation for Operations Research and Decision-Making, Washington Operations Research Council, Rockville, Maryland 1969, pp 229-235.

6. J. B. Dennis, Programming generality, parallelism and computer architecture. Information Processing 68, North-Holland Publishing Co., Amsterdam 1969, pp 484-492.

7. E. W. Dijkstra, Recursive programming. Numerische Mathematik, Vol. 2, 1960, pp 312-318.

8. P. Naur, et al, Report on the algorithmic language ALGOL 60. Comm. of the ACM, Vol. 3, No. 5 (May 1960), pp 299-314.

9. E. Morenoff and J. B. McLean, Program String Structures and Modular programming. Published in [1], pp 133-143.

10. M. E. Conway, Design of a separable transition-diagram compiler. Comm. of the ACM, Vol. 6, No. 7 (July 1963), pp 396-408.

11. E. W. Dijkstra, Co-operating sequential processes. Programming Languages, F. Genuys, Ed., Academic Press, New York 1968. [First published as Report EWD 123, Department of Mathematics, Technological University, Eindhoven, The Netherlands, 1965.]

12. O. J. Dahl and K. Nygaard, SIMULA -- an Algol-based simulation language. Comm. of the ACM, Vol. 9, No. 9 (September 1966), pp 671-678.

13. F. J. Corbato, C. T. Clingen, and J. H. Saltzer, MULTICS -- The first seven years. AFIPS Conference Proceedings, Vol. 40, SJCC, 1972, pp 571-583.

14. R. C. Daley and J. B. Dennis, Vurtual memory, processes, and sharing in MULTICS. Comm. of the ACM, Vol. 11, No. 5 (May 1968), pp 306-312.

2. DATA STRUCTURES IN MODULAR PROGRAMMING

The achievement of program modularity becomes increasingly difficult as the linguistic requirements for representing program modules move further from the linguistic level defined by the computer system on which the modules are to be run. In this lecture, we explore issues arising in the construction of program modules that require the ability to create, extend, and modify structured data. We conclude that, to achieve modularity, a computer system must define a linguistic level that provides a suitable base representation for structured data, a requirement not satisfactorily met by conventional computer systems or by implementations of contemporary programming languages.

2.1. ADDRESS SPACE AND MODULARITY

First we note that conventional computer memories and addressing schemes impose a limitation on modular programming. When a program is run on a contemporary computer system, all procedures and data involved in the computation must be assigned positions within the address space provided for the computation by the computer system. If more than a single object -- whether procedure or data -- is assigned to some area of the address space, the meanings of addresses must change during the computation. This violates our principles of modular programming because some program modules will require knowledge of the internal construction of others in order to determine which objects should occupy the shared areas of address space. Thus the finiteness of address space limits the size of modular programs. To support modular programming a computer system must provide an address space of size sufficient to hold all procedures and data structures required for the execution of any modular program. A more complete presentation of this argument may be found in [1].

The addressing limitations of finite main memories have been reduced through the brute force expedient of using larger and larger main memories. Yet practical main memories are still small in comparison to the extent of data bases and program libraries we wish to use in constructing modular programs. A more sophisticated approach to overcoming the finiteness of main memory is to arrange a computer system to provide a large virtual address space for each user. In effect, a process is given a large address space without tying up a corresponding amount of main memory. As it is currently implemented, the virtual memory idea also has limitations, for chunks of address space are reassigned from one physical storage device to another in relatively large units (512-word pages, for example); and it is difficult for the programmers of a module to map his data structures into the address space in such a way that related items will be moved together between physical storage levels.

2.2. REPRESENTATION OF PROGRAM MODULES

Other implications of modularity concern linguistic features of the linguistic level at which modules are represented for combination into larger units. We noted earlier that all identifiers occurring in a program module must be bound within the module unless they refer to primitive constructs of the linguistic level. Otherwise identifier clashes can occur when independently prepared modules are used together. From this premise it follows that any information to which a program module requires access to perform its function must be part of the module itself, or must be passed to the module by means of formal parameters of the calling statement. Any information created or modified by the module and intended for use outside must be passed to the caller through formal parameters.

Since the objective of modularity is that any program may be used as a program module, it must be possible to treat any entity to which reference may be made by a program module as an actual input or output parameter of the module. A program module that implements a certain algorithm should be applicable to any input data to which the algorithm applies. It is possible to design algorithms that work effectively for a wide range of inputs as, for example, a procedure for matrix inversion or one for constructing the parse of a sentence according to a formal grammar. The representation of such program modules requires linguistic primitives for building and altering data structures of extent not known until the time of execution.

In summary, we have three requirements to be met by a linguistic level intended as a foundation for modular programming:

1. Any data structure may occur as a component of another data structure.
2. Any data structure may be passed (by reference) to or from a program module as an actual parameter.
3. A program module may build data structures of arbitrary complexity.

The linguistic levels defined by conventionally organized computer systems have a linear address space as their fundamental notion of data structure, and indexing as $\overset{\text{their}}{\wedge}$ fundamental means of data access. Such a level is not an acceptable foundation for modular programming because the primitive constructs do not provide for altering one data structure without interfering with the representations of others. To enlarge one structure may require rearrangement of other structures in address space and cannot be done without knowledge of their scheme of representation.

There are three ways in which a satisfactory linguistic level for modular programming can be realized starting from a host level H defined by some computer system:

1. Use a "standard" programming language L with an available translator to level H and having an adequate class of data structures and primitive operations.
2. Extend a programming language L' that does not offer an adequate class of data structures, to realize a new linguistic level L that is adequate.
3. Design and implement a new language $\overset{L}{\wedge}$ by constructing either
   a. A translator from L to H.
   b. An interpreter of L that runs at level H.

Suppose the host level H is defined by a conventional computer which provides the user with a linear address space. Whichever of the above means is used to realize the desired linguistic level L, the data structures of L must be mapped into the linear address space of H in such a way that the primitive operations of L can be implemented effectively in terms of the primitives of H. The difference between means (1)above and means (2) or (3) is that in (1) the language L is standard and the mapping of L into H is uniform over all program modules expressed in L; in cases (2) and (3) the mapping of structures in L into the linear address space of H is chosen independently by the designer of each program module and the same choice is unlikely to be made for any pair of modules.

To be more specific, suppose the designer of a program module is using the second approach. Let the language L' be a language (Fortran or Algol 60, for example) that does not provide adequate primitives for manipulating structured data. To implement the program module, the designer must extend L' by adding a <u>memory</u>. He does this by setting aside some portion M of the linear address space of H to hold representations of the data structures of L as they are created and operated upon during operation of the program module. The memory may be viewed as a pair (M, C) where M is a one-dimensional array, and C is a collection of procedures that implement the primitive data structure operations of L. If L' is Fortran, the memory array M may be allocated within a block of COMMON storage and the procedures of C may be realized as a group of subprograms. If L' is Algol 60, the memory array and the procedures of C would be declared within the outermost block of the program module.

There are serious problems with an approach in which the memory is separately implemented in independent program modules. Suppose A and B are two such modules. Then:

1. Either the base linguistic level H includes an allocation mechanism for units of address space, or arbitrarily chosen areas of address space must be set aside as the memory arrays for modules A and B.

2. A structure created by module A cannot be directly accessed from within module B, for the primitives of A are not used within B.

Partitioning the address space into separate areas for each module requires that each area be large enough to hold any structure that could be created. The idea of segmentation [1] is a way of meeting this requirement. If the host level H provides a facility for management of address space, then introducing a second layer of memory management mechanism aggravates the inefficiency of program execution.

The problem of communicating data structures between program modules expressed in different representations may be discussed in terms of Figure 2. Modules A and B are expressed in different extensions L_A and L_B of a host linguistic level H. Sets S_A and S_B represent the classes of data structure representations in L_A and L_B. The maps f_A and f_B (which may be relations) relate representations in L_A and L_B to corresponding representations at the host level H.

If the linguistic levels L_A and L_B are different, then a data structure produced by module A cannot be directly accessed by module B. Nevertheless, modules A and B may be used together if no data structures are exchanged between them, or if we can prepare routines t and t^{-1} at the host level H which convert structures from their representation in L_A to their representations in L_B and vice versa. Of course, the need to write these routines is a violation of modularity since knowledge of how the data structures of L_A and L_B are represented at H is required, and this knowledge concerns the internal construction of modules A and B.

We have discussed Figure 2 assuming modules A and B include the definitions of and L_B as internal components. The same picture holds if modules A and B are expressed in "standard" languages L_A and L_B that define primitive operations on data structures by two different extensions of a host level H. If L_A and L_B are "standard" languages, then knowledge of the mappings f_A and f_B does not involve internal knowledge of modules A and B. Thus the construction of the conversion routines t and t^{-1} depends on knowledge of the implementations of L_A and L_B rather than the workings of the modules. However, now these routines are subject to invalidation if the implementation of either L_A or L_B is changed.

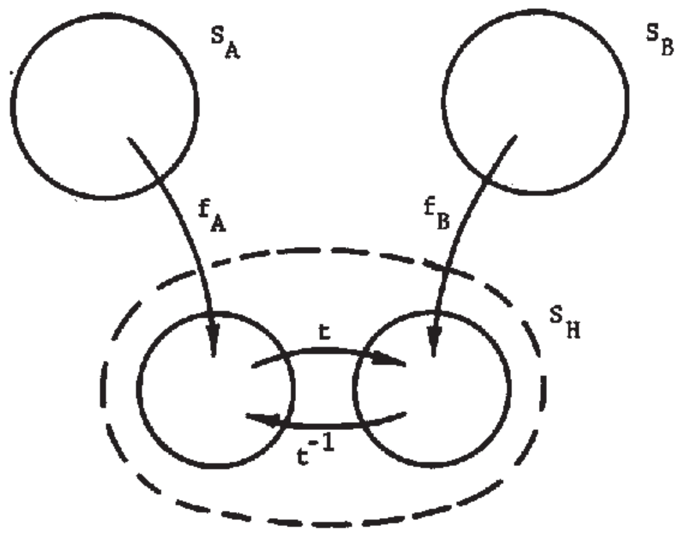


Figure 2. Exchange of data structures between program modules.

If the host level H defines a linear address space, construction of the conversion routines can prove difficult. This is because level H lacks notions that would save the programmer from the need for complete knowledge of the data structures being transformed. A data structure represented in a linear address space is referenced by an *address*, but there is no uniform rule for locating all items in the address space that are parts of the data structure. Also there is no uniform convention regarding how individual data structures may be combined into a single object.

That two program modules are represented at the same linguistic level L does not ensure that consistent representations are used for objects dealt with by the algorithms of the modules. For example, there are many ways in which a directed graph may be represented by a vector of integers. If a community of users interested in sharing program modules that manipulate directed graphs can agree on a standard representation in L for directed graphs, then programs contributed by the community may be used together without difficulty. Otherwise conversion routines are required. Nevertheless, if the representation of a directed graph is to be passed as an argument or result of computation by a module, the scheme of representation in L must be given as part of the functional specification of the module. The necessary conversion routines can be written in L from the module specification. Without adequate data structure primitives in the common language L, the conversion routine would be difficult, if not impossible, to write.

2.3. LINGUISTIC LEVELS FOR MODULAR PROGRAMMING

We have argued that a satisfactory linguistic level for modular programming must provide adequate primitive features for building and transforming data structures, and that the linguistic levels defined by computer systems of conventional organization are inadequate. Next we examine the linguistic levels defined by several well-known programming languages for their suitability to modular programming, particularly in regard to their provisions for building and transforming data structures. The two most familiar languages, Fortran and Algol 60, are inadequate by default, since arrays are the only sort of data structure provided, the bounds of arrays are inflexible once storage has been allocated, and the dimensionality of arrays is fixed by the program text. The languages PL/I, Algol 68, and Lisp are considered in the following paragraphs.

2.3.1. PL/I

In PL/I [2] the principal data types that may be used to represent and manipulate structured data are arrays, structures, based variables and pointers. Arrays in PL/I are subject to similar limitations as arrays in Fortran or Algol 60: an array identifier may only name arrays of the declared dimensionality; subscript bounds cannot be changed once storage for an array is allocated; all elements of an array must be of the same data type. These limitations are imposed so that a permanent assignment of array elements to a contiguous portion of address space is possible, and the efficient indexing access mechanism of present day computers may be used.

In PL/I structures, components are accessed by means of a sequence of symbolic names called selectors; the length of the selector sequence is the depth of the component in the structure. Components of a structure may be further structures, arrays, etc. So that each generation of a structure may be permanently assigned to a contiguous portion of address space, each component of a structure is restricted to a size stated in the structure declaration. Structures (all satisfying the same declaration) may occur as elements of arrays.

Structures as in PL/I do not meet the requirements of modular programming. It is not possible during a computation to make an arbitrary structure a component of another structure -- the entire form of a structure must be specified before any of its components may be given a value. Furthermore, since the depth of a structure is implicit in the program text, there is no way of representing data structures of arbitrary extent as PL/I structures.

Use of PL/I pointer variables, the addr primitive, and variables, arrays and structures declared based permits the construction of arbitrarily complex address-linked storage structures. The only correct interpretation of PL/I pointer values is as locations within a linear address space. Pointer values may occur as elements of arrays and as components of structures as well as values of simple variables declared as pointer variables. A pointer value is created either by applying the primitive function addr to a name, or by explicitly allocating storage for a variable declared to be based, the pointer value returned being the origin of the allocated region of address space.

Although PL/I pointers provide a very general facility for building representations of data structures, the needs of modular programming are not met. A pointer value cannot be regarded as a reference to a data structure because PL/I provides no convention for identifying the set of elements belonging to the structure referenced by a pointer value. There is no built-in concept of one linked structure being a component of another. There is no guarantee that an element pointed to has the data type intended by the programmer. Further, deletion of elements must be done by explicit free statements; an element disconnected from a linked structure through reassignment of pointer values remains in existence until its storage is explicitly release ~~[illegible]~~.

Each programmer is forced to adopt his own conventions regarding extent of data structures, a notion of component, and when storage may be reclaimed. Hence the use of PL/I linked structures for communication between independent program modules offers no advantage over a bare machine having a linear address space.

Unsuitability of the data structure facilities is not the only problem PL/I presents for modular programming. Since PL/I refers to "external" procedures and data sets in the same manner as Fortran, and since procedures may have nonlocal identifiers, name clashes are possible whether one choses the PL/I program or the PL/I procedure as the form of program module. In addition, the introduction of new language features such as tasking has not considered the requirements of modularity.

2.3.2 ALGOL 68

In an Algol 68 program[3,4], each occurrence of an identifier has an associated *mode* that determines the set of values permitted for the named variable. The modes that provide representations for data structures are *multiple values* and *structures*. Multiple values are similar to PL/I arrays and, in themselves, do not provide an adequate foundation for modular programming.

A structure mode declaration in Algol 68 specifies that any value of the mode being declared is an object having a fixed number of component objects identified by *field selectors*, each component being an object of specified mode. Through use of several mode declarations one may define a class of objects having graphs that are trees. Each node of such a tree has an associated mode and is the origin for a fixed number of arcs, each bearing a field selector as specified in the mode declaration.

Since recursive mode declarations are permitted, the objects of a given mode may be of unbounded depth, as for example, the class of binary trees. Yet no Algol 68 structure mode permits values that range over all Algol 68 data structures. Thus there is no means for substituting an arbitrary structure for some component of an existing structure. Specifically, it is not possible to write an Algol 68 procedure that obtains a data structure from one program module and gives it to another module without knowing enough about the data structure to specify its mode. Also, a program module expressed in Algol 68 cannot build an arbitrary Algol 68 data structure because a finite set of mode declarations is insufficient to describe the complete class of Algol 68 objects.

Since Algol 68 includes suitable conventions for delineating the extent of data structures, and has satisfactory provisions for building and accessing complex structures, the data structure primitives of Algol 68 are superior to those of PL/I as a foundation for modular programming. However, the requirement that the mode of every variable be explicit is an unfortunate limitation.

Other limitations of Algol 68 for modular programming stem from the design of the language primarily as a means for one programmer to write a complete program for a computation of interest to himself. A prime example is the concept of coersions by which conversion of values from one data type to another is implicit in many circumstances. A consequence of coersion is that a scan of an entire Algol 68 program may be necessary to fix the meaning of statements in a deeply nested procedure.

2.3.3 LISP

In Lisp[5,6], data structures are represented as *lists*. A region of a linear address space (the *memory*) is reserved for *cells* from which lists are built to represent data structures. Each cell has two fields which may contain addresses (called *pointers*) of other cells in the memory. A list is specified by the address of a cell and consists of all cells that can be reached by tracing pointers from the starting cell. Thus a list is essentially a rooted, directed graph in which each node is the origin of at most two arcs that define the left and right sublists for the corresponding cell. In most applications, lists containing directed cycles do not occur, and lists have the form of a binary tree with shared subtrees.

Lisp includes primitive operations for building lists, for obtaining the left or right component sublist of any list, for testing whether two lists are equal or are the same list, and for making one list the new left or right sublist of an existing list.

The leaf cells of lists are called *atoms* and have associated named values called *properties*. A property of an atom may be an elementary object such as a character string, an integer or a real number, or may be an arbitrary list. Lisp includes basic functions for performing operations on property values. It is easy to devise ways in which lists may be used to represent any of the commonly used data structures in programming practice.

Since any list may occur as an actual parameter of a Lisp function application, and Lisp has primitives for building, disecting and rearranging specified lists without disturbing the meaning of other lists sharing the memory, Lisp meets our fundamental requirements for modular programming with respect to data structures. The principal weakness of Lisp for modular programming arises from its inability to exploit indexing as an efficient access mechanism for arrays. For applications where an array is a natural representation for a data structure, many representations as lists have been designed to yield efficient operation for a variety of different expected patterns of access. Because these representations are generally in conflict, conversion of data structures is often required to combine independently written Lisp functions, where conversion would not be required if the modules were expressed in a language offering arrays as a basic data type.

Lisp also shares with the other languages we have discussed the failing of having a global level of nomenclature. Programmer defined functions and constants are given names that are global in a Lisp program. There is no provision for ensuring freedom from name conflicts when independently written Lisp programs are combined.

2.3.4 DISCUSSION

On one hand, Lisp is superior to PL/I and Algol 68 as a foundation for modular programming because PL/I and Algol 68 fail to provide an adequate foundation for representing and manipulating data structures. The limitations of PL/I and Algol 68 can be traced to the desire of the designers of these languages to make efficient implementations possible.for conventional computers that implement a linear address space. Thus it was considered essential that arrays be included as a fundamental data type and that arrays be implemented using the indexing hardware of contemporary machines. On the other hand, Lisp has achieved a more satisfactory concept of data structure by giving up the array as a fundamental notion and ignoring the use of indexing. By making these concessions, the address space may be divided uniformly into list cells so that the allocation and deallocation of cells become trivial operations. In this way a powerful language for expressing computations on symbolic data has been realized.

Is there a way to combine the best aspects of these three languages? In the final section of these notes we explore the definition of a base linguistic level for modular programming using a concept of data structure that yields natural representations for a wide variety of data structures commonly applied in programming practice, including lists, arrays, and structures. Although this concept may prove impractical to implement for general use on computers of conventional organization, it should prove valuable as a standard of achievement, and as a guide for the design of computer systems intended to advance the prospects for modular programming.

2.4. REFERENCES

1. J. B. Dennis, Segmentation and the design of multiprogrammed computer systems. J. of the ACM, Vol. 12, No. 4 (October 1965), pp 589-602.

2. S. V. Pollack and T. D. Sterling, A Guide to PL/I. Holt, Rinehart and Winston, Inc., 1969.

3. A. Van Wijngaarden, Ed., Report on the algorithmic language ALGOL 68. Numerische Mathematik, Vol. 14, No. 79 (1969), pp 79-218.

4. J. E. L. Peck, An ALGOL 68 Companion. Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada, October 1971 (preliminary edition).

5. M.I.T. Computation Center, LISP 1.5 Programmer's Manual. Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Mass., August 1962.

6. E. C. Berkeley and D. G. Bobrow, Eds., The Programming Language LISP: Its Operation and Applications. Information International, Inc., Cambridge, Mass. 1964.

3. MODULARITY IN MULTICS

We have seen that most contemporary computer systems and programming languages do not support a very general form of modular programming. Yet one advanced computer system comes significantly closer to defining a linguistic level suitable for modular programming. A major objective of the development of Multics at Project MAC [1] has been to create an environment within which programs developed independently and expressed in different source languages may be combined with minimum difficulty. In this lecture we shall study how well this objective has been achieved.

First, we present a model for those aspects of Multics that must be understood to discuss modularity from the viewpoint of the Multics user. Then we discuss the achievements and limitations of Multics for modular programming in terms of the model. The model consists of a representation for the states of Multics processes as an augmented class of objects, and an informal discussion of certain state transitions that occur during execution of procedures by Multics processes. We do not attempt to model the mechanisms of Multics for protection, access, control, and interprocess communication.

3.1. THE MODEL

3.1.1. THE FILE SYSTEM

The file system of Multics [2] retains the programs and data of all Multics users in the form of a hierarchical structure of directories and segments. We represent a directory by an object as in Figure 3. A directory has arbitrarily many components, each of which may be a directory entry, a segment entry, or a link — an example of each type is shown. The selectors for the entries and links of a directory are called entry names, and are character strings. Each entry name in a directory must be unique. A directory or segment entry has an 'attr'-component that gives attributes such as access rights, date of last change, etc. The second component is an object that represents either another directory, or a segment. A link is a pathname composed of a sequence of entry names. The Multics file system is an object that represents a particular directory called the root directory. Each item (directory or segment) in the file system is specified by the unique sequence of entry names by which the item may be reached from the root of the directory tree. The sequence of entry names is a pathname of the directory or segment.

A segment in Multics is a linear address space of 2^{18} addresses which may hold either data or one or more procedures. A segment is represented by an object having elementary components selected by the integers 0, 1, ...

In the root directory of the file system, the entry names are user names and the entries are user directories. A user is the owner of all directories and segments that are entries in his user directory, and is the owner of directories and segments that are entries in owned directories.

We will simplify the representation of the file system state by omitting attribute components and omitting the branches labelled 'directory' or 'segment'. This simplified form is illustrated in Figure 4. Entry names of links are distinguished by an asterisk. The link shown is to the item having pathname 'b.b.a'.

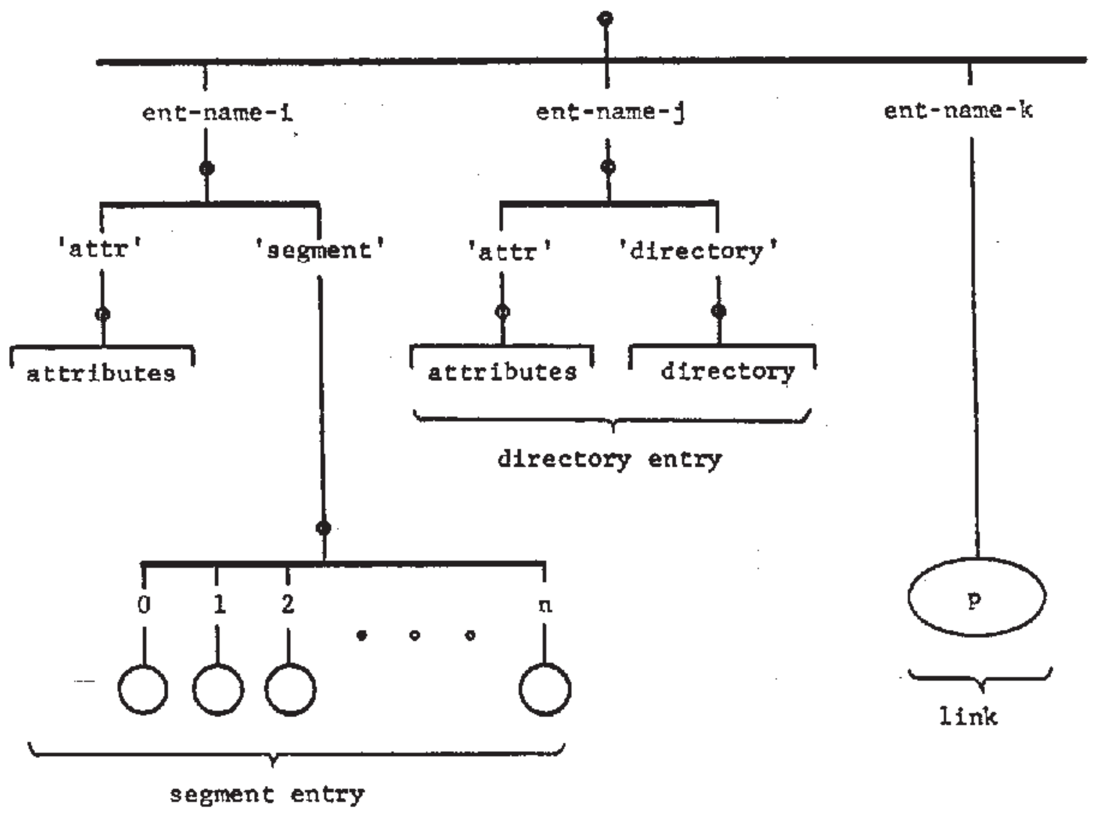


Figure 3. Model for the Multics file system.

3.1.2. PROCESSES AND ADDRESS SPACES

When a Multics user begins a console session, a process is created for him. By typing commands at the console, the user causes the process to execute procedures. The execution of commands results in changes in the file system state. Normally a user process ceases to exist when his console session is terminated, and the changes to the file system are the only record retained in Multics of the user's activity.

For our purposes a state of Multics may be represented as an object having a component for the file system, and one component for each process in existence. In Figure 5 we have identified each process by a distinct user name.

The state of a process is an object having components as follows (Figure 6):

1.	'memory'	process address space
2.	'stack'	stack segment and pointer
3.	'kst'	known segment table
4.	'sng'	segment number generator
5.	'link'	linkage segment and pointer
6.	'w.dir'	working directory

In fact, components of the process state are implemented as segments in the Multics file system which are accessible to system procedures. We choose to model them as separate objects for ease in discussing their function from the user's viewpoint.

The 'memory'-component of a process state is the address space implemented by the hardware and software of Multics for each Multics process. The object that models the address space of a process is shown in Figure 7. It is a two-level tree. The selectors at the first level are integers called segment numbers. Each segment number identifies a segment which may contain up to 2^{18} words. Since the segments of an address space are not distinct from segments of the file system, the nodes selected by segment numbers are, in fact, identical with segment nodes of the file system state. The address spaces of Multics processes are implemented by a complex arrangement of hardware-accessed tables in core memory, a small associative memory, and auxiliary storage devices (drum and disc) to hold pages of segments not allocated space in the core memory [3, 4]. A two-component address consisting of a segment number and a word number that specifies a word in the address space of a process is called a generalized address.

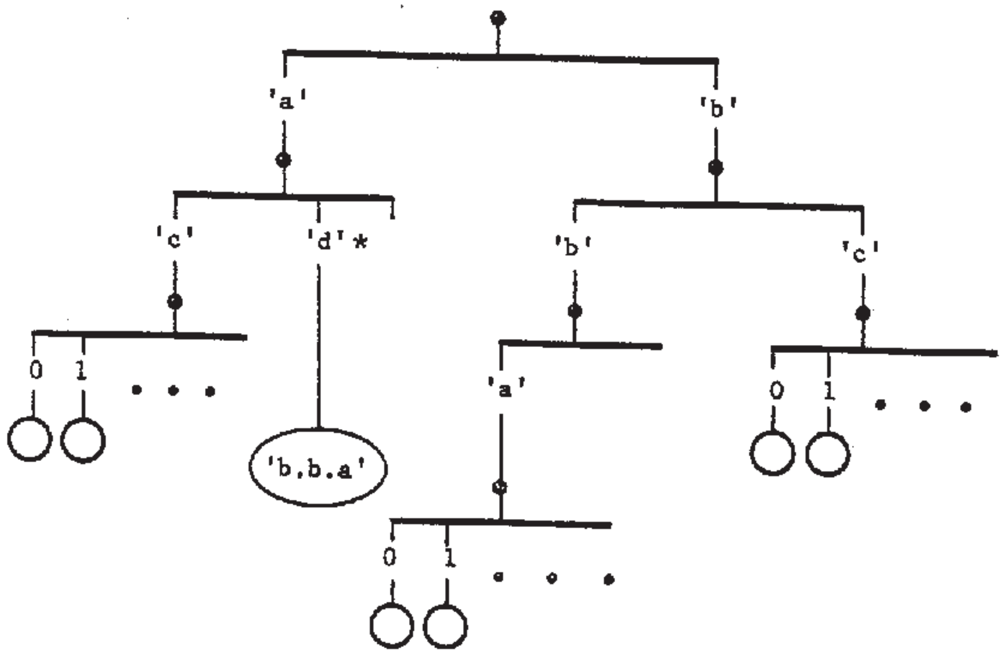


Figure 4. Simplified model for the file system.

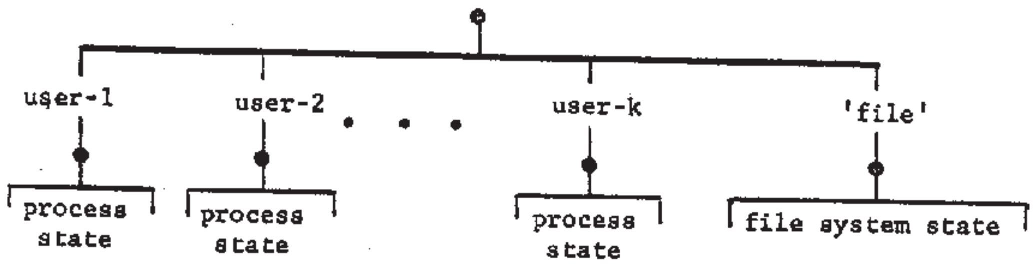


Figure 5. Model for a state of Multics.

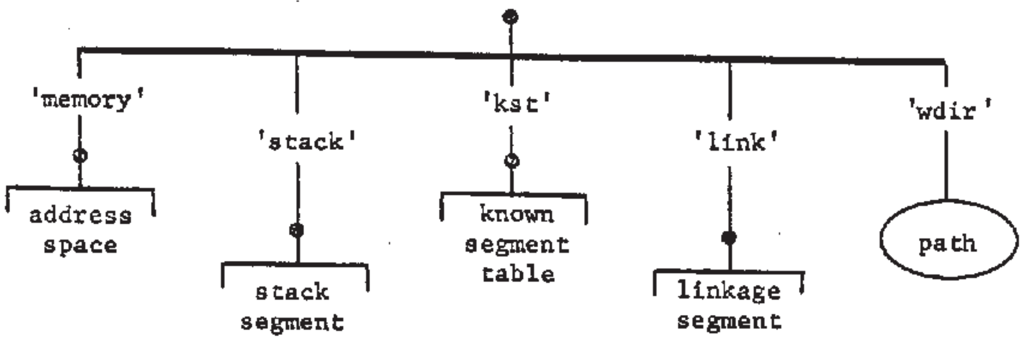


Figure 6. Model of a Multics process.

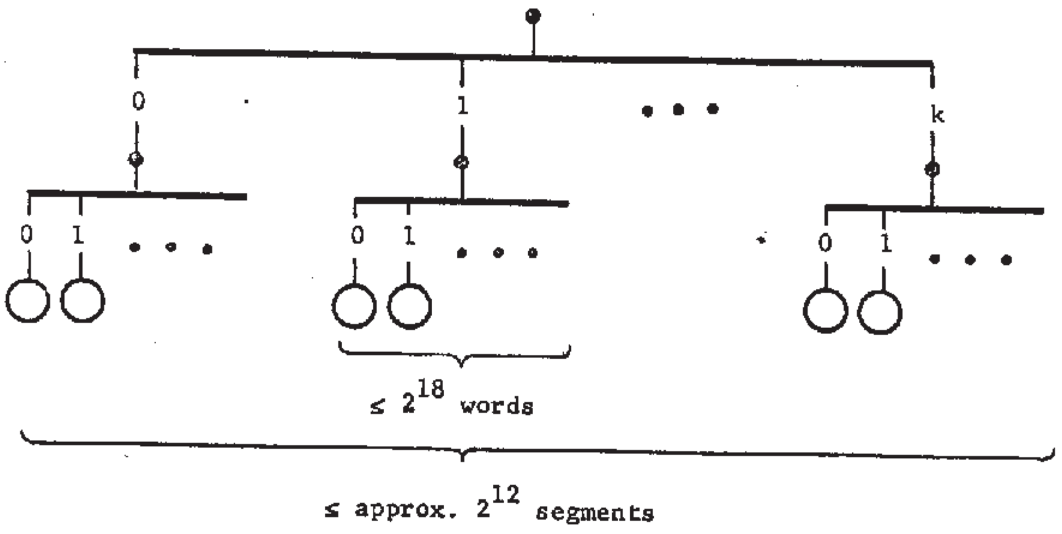


Figure 7. Model for the address space of a process.

The 'stack'-component of a process state consists of a segment (for our purposes not part of the file system) and a pointer variable. Variables assigned by the programmer to "automatic" storage are accessed by addresses relative to the stack pointer. On procedure entry the pointer is advanced to the end of the stack area used by the calling procedure; on procedure exit the stack pointer is returned to its value before entry. In this way, all Multics procedures that use the standard call and return conventions may be used recursively.

3.1.3. MAKING A SEGMENT KNOWN TO A PROCESS

The assignment of a segment from the Multics file system to the address space of a process is called making the segment known to the process. This action occurs when the process, in executing a procedure, encounters a symbolic reference to a segment. The symbolic name used in the code of the procedure segment is called a reference name. The path name of the segment in the file system to which a reference name refers is found by a system procedure directed by a set of search rules in a manner to be discussed later. A segment known to a process has an associated segment number; segment numbers are assigned to segments sequentially as they become known to the process.

The associations between segment numbers, reference names and path names for all segments known to a process are held in a data structure called the known segment table which is the 'kst'-component of the process state. The known segment table is modelled as an object in Figure 8. For example, the figure shows that segment number i of this process has the path name 'x.y.a' and the reference names 'a' and 'b' have been used to refer to the segment during operation of the process. The 'n'-component of the known segment table is the highest integer in use as the segment number of a segment known to the process. It is given the initial value 0 when the process is created, and is incremented by 1 for each segment made known to the process.

An illustration of the state transition that occurs when a segment is made known to a process is shown in Figure 9. The value i of the 'n'-component of the known segment table is incremented and used as the selector for a new entry in the known segment table. The new entry contains the reference name 'a' used by the procedure in execution and the path name 'x.y.a' obtained by system routines directed by the search rules. Segment i + 1 of the address space of the process is identified with the segment having pathname 'x.y.a' in the file system.

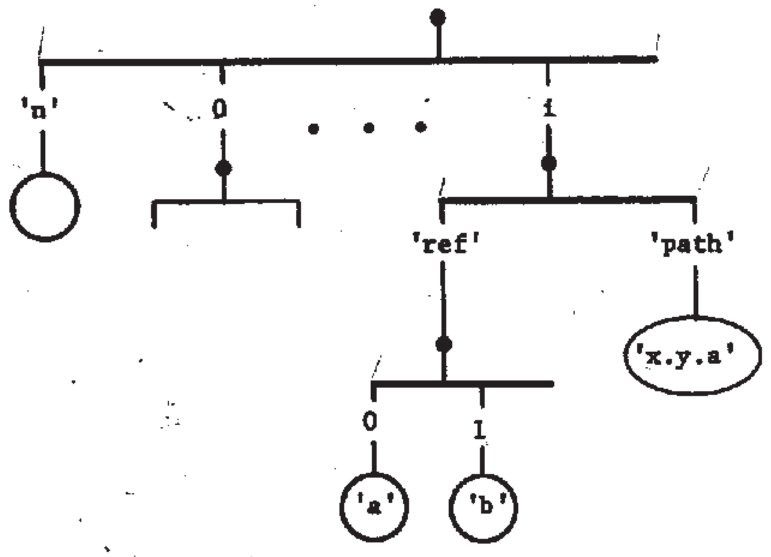
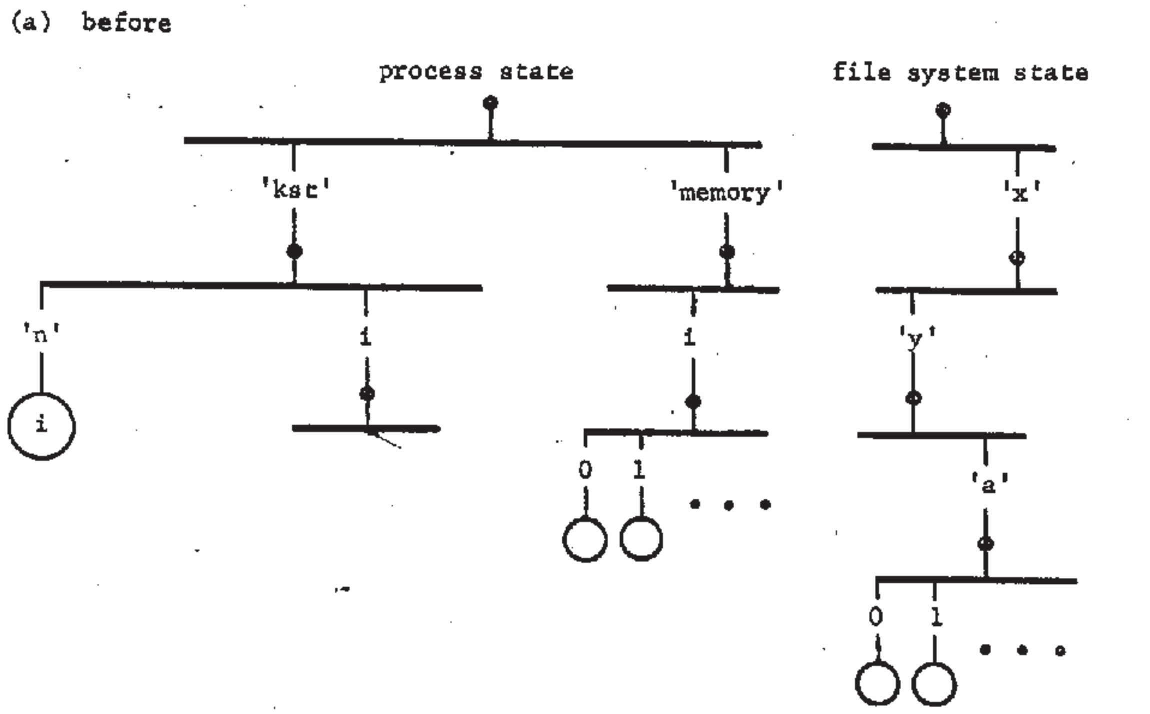


Figure 8. Model for the known segment table.



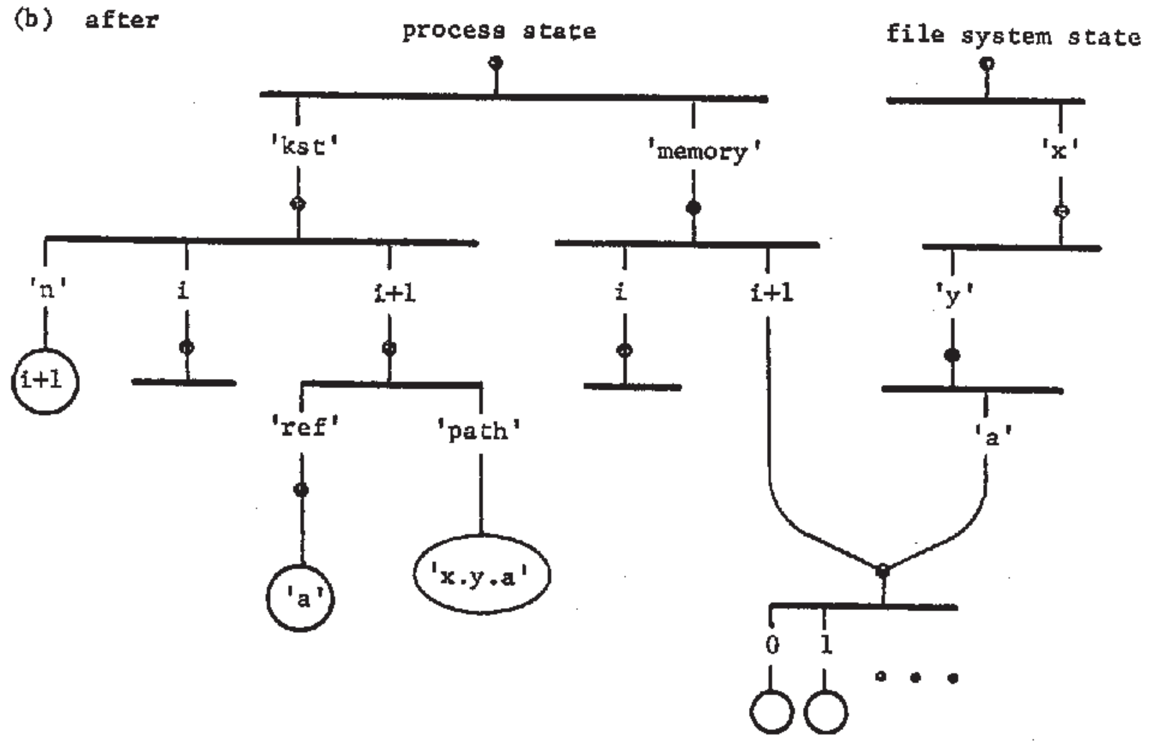


Figure 9. Making segment 'a' with pathname 'x.y.a' known to a process.

3.1.4. DYNAMIC LINKING

For a segment S to be made known to a process, reference to S by means of a reference name must occur from within some procedure segment P. Once segment S is known to the process, references to it should use the hardware-implemented addressing mechanism provided for generalized addresses. The Multics state transition that realizes this objective is called linking. Linking a site of reference in segment P to segment S cannot involve any change in the content of segment P, because procedure segments in Multics are shared among processes. The scheme used is to implement references to other segments from segment P by indirect addressing through items called links that make up a linkage section for segment P. The linkage sections for all procedure segments known to a process form the 'link' component of the process state. When a procedure segment is made known, its linkage section is added to the 'link'-component, with each of its links set to cause transfer of control to a system routine. The system routine reads the reference name from the procedure segment and determines whether the referenced segment is known. If not, this segment is made known as described above. Then the link is replaced by the generalized address of the referenced segment. The details of this mechanism have been published [4].

3.1.5. SEARCH RULES AND THE WORKING DIRECTORY

A Multics user must specify an owned directory of the file system as the working directory for his process when he begins a computation. The working directory of a process may be changed by a system command procedure which may also be called by the user's program. The pathname of the working directory is the 'wdir'-component of the process state.

The search rules of Multics specify how reference names encountered during procedure execution are to be converted into pathnames. The search rules are stated as a list of data structures in the sequence they are to be searched for an entry named by the given reference name. The usual search rules specify the following order of search:

1. known segments
2. referencing directory
3. working directory
4. system libraries

The search begins by testing whether the segment is represented by an entry in the known segment table. This is done so that links to segments already known to the process may be completed without any directory searching, which consumes significant processing time. If the reference is not to a segment already known, a search is made of the "referencing directory" -- the directory from which access to the procedure currently in execution was obtained. This search rule supposes that procedures that form a subsystem are grouped together in directories, and gives preference to such a related procedure over a procedure of the same name in the user's working directory.

A program expressed in Fortran or PL/I for execution by Multics normally references its user owned procedure and data segments in the working directory, and accesses library procedures in the system libraries directory.

3.2. ACCOMPLISHMENTS

Multics has realized a number of significant advances in computer system design, and has made them available to a large community of users for the first time. These unique characteristics of Multics include some features of major importance for modular programming.

1. A large virtual address space (approximately 2^{30} elements) is provided for each user.
2. All user information is accessed through his virtual address space. No separate access mechanism is provided for particular sorts of data such as files.
3. Any procedure activation can acquire an amount of working space limited only by the number of free segments in the user's address space.
4. Any procedure may be shared by many processes without the need of making copies.
5. Every procedure written in standard Multics user languages (Fortran, PL/I and others) may be activated multiply through recursion or concurrency.
6. A common target representation is used by the compilers of two major source languages -- PL/I and Fortran.

These achievements are major contributions toward simplifying the design and implementation of large software systems. They were made possible by building the Multics software on a machine expressly organized for the realization of a large virtual memory and shared access to data and procedure segments [5].

3.3. UNRESOLVED ISSUES

The ease of modular programming in Multics is limited by certain design problems that remain unresolved issues. One problem Multics shares with all computer systems in which data structures must be mapped into a linear address space. As observed earlier in these notes, each author of a program module must adopt his own private conventions for introducing the concepts of "the extent of a data structure" and "component of a data structure", for no conventions are established by the Multics virtual machine nor by the standard user languages of Multics. This problem can be solved only through the adoption of a more suitable model for structured data as the basis for computer system design. A model having the essential attributes is discussed in the final section of these notes.

3.3.1. TREATMENT OF REFERENCE NAMES

Another problem for modular programming in Multics concerns the treatment of reference names. Basically, reference names are identifiers that occur free in the text of Multics procedures. Since reference names occur not only as identifiers of fixed elements of the Multics linguistic level, absence of name conflicts cannot be ensured when a user attempts to combine independently written procedures. The following discussion of this issue is based in part on a study by Clingen [5].

The set of search rules given earlier for determining the pathname of a segment specified by a reference name is an attempt to avoid the undesired consequences of name conflicts. To see how the set of search rules evolved to this form, we first consider the problems of modular programming with the search rules

1. working directory
2. system libraries

This combination of search rules is appropriate where a user has defined a collection of procedure and data segments and entered them in an owned directory. By making this directory the working directory of his process, all reference names designating members of the user's collection of segments will be associated with the correct segment, and so will references to library procedures so long as their reference names are not duplicated in the working directory.

The possibility of clashes between reference names chosen by the user and reference names of library procedures is not the only difficulty with this choice of search rules. If two programming languages are implemented independently for use in Multics, the sets of reference names used to access the run-time procedure libraries for the two implementations may include duplicate names with conflicting meanings. These names should identify entries in separate directories, but this is not provided for by the search rules. One could let the user specify one of several library directories in the second search rule, but this would not provide for programs that combined procedures expressed in the two source languages. Alternatively one could use a set of search rules such as

1. Working directory
2. Run time library A
3. Run time library B

but duplicated names would be misinterpreted if they were intended to reference segments in run time library B.

Another difficulty is that a mistake in use of a reference name may lead to successful search and linking to a strange procedure in a library directory, whereas one would prefer to have such mistakes produce an error response by the system.

In Multics, the natural form for a program module is a collection of procedure and data segments entered in a common private directory of the file system. If a user wishes to use two such modules together, some arrangement must be made so that reference names occurring in either module will be interpreted correctly. One scheme is to arrange that the working directory is always the directory containing the procedure in execution. This requires that the working directory be changed whenever control passes from procedures in one module to a procedure in the other module. Since changing the working directory of a process is an expensive task, this solution is not attractive, especially if control transfers between modules occur frequently. Also, this arrangement requires different call and return conventions (the inclusion of a command to change the working directory) for calls on procedures of other modules. This requirement conflicts with the concept that one should be able to apply a program module simply by using its name in a call statement.

The difficulty of making the working directory concept work satisfactorily led to addition of the "referencing directory" search rule:

1. referencing directory
2. working directory
3. system libraries

The referencing directory rule directs search for a reference name to the directory in which the procedure segment in execution was found. This is accomplished by using the segment number of the procedure in execution to locate its entry in the known segment table. The 'path'-component of the entry provides unambiguous identification of its directory. With this rule in effect, calling any procedure of a program module automatically makes the directory of that module the first directory to be searched for all reference names encountered during execution of procedures that are part of the module.

The "known segments" search rule was added to the set of search rules to reduce the time spent performing searches in directories of the file system, thereby improving system efficiency. This search is performed in such a way that it has the same effect as the referencing directory search rule. An entry in the known segment table is located that has the given reference name in its 'ref'-component. Then the 'path'-component of the entry is tested to verify that the entry is for a segment found in the same directory as the segment in execution. If the test fails, the entry is rejected and search for other entries having the given reference name is continued.

Thus the search rules of Multics implement the correct context for reference names occurring in procedures of program modules. Yet several difficulties remain:

1. Mistakes in use of reference names may lead to unsuspected linkage to library or system procedures.
2. Implementers of programming language subsystems must avoid name conflicts among their libraries.

3. No suitable means is provided for representing references among the data segments of a large data base. This is a problem because no mechanism has been implemented for creating links from uses of reference names in data segments.

In the final section of these notes, we present a conceptual basis for a computer system in which these issues of modular programming are resolved by providing the appropriate context for each use of a name.

REFERENCES

1. F. J. Corbato, C. T. Clingen, and J. H. Saltzer, MULTICS -- the first seven years. AFIPS Conference Proceedings, Vol. 40, SJCC, 1972, pp 571-583.

2. R. C. Daley and P. G. Neuman, A general-purpose file system for secondary storage. AFIPS Conference Proceedings, Vol. 27, Part 1, FJCC, 1965, pp 213-229.

3. A. Bensoussan, C. T. Clingen, and R. C. Daley, The Multics virtual memory. Proceedings of the Second Symposium on Operating Systems Principles, ACM, October 1969, pp 30-42.

4. R. C. Daley and J. B. Dennis, Virtual memory, processes, and sharing in MULTICS. Comm. of the ACM, Vol. 11, No. 5 (May 1968), pp 306-312.

5. E. L. Glaser, J. F. Couleur and G. A. Oliver, System design of a computer for time sharing applications. AFIPS Conference Proceedings, Vol. 27, FJCC, 1965, pp 197-202.

6. C. T. Clingen, unpublished memorandum prepared for the NATO Conference on Software Engineering Techniques, Rome, 1969.

4. A BASE LINGUISTIC LEVEL FOR MODULAR PROGRAMMING

In this lecture, we present informally the semantic concepts of a linguistic level (a common base language) that could serve as a common representation for program modules expressed in a variety of source programming languages. The objective is to describe a linguistic level such that the issues of modular programming raised in the preceding presentations have a satisfactory resolution. It is hoped that this material will serve as a guide or standard of capability for computer system designers so future computer systems will better serve as foundations for modular programming.

Our work toward the specification of a common base language [1] uses methods closely related to the formal methods developed at the IBM Vienna Laboratory [2, 3] and which derive from the ideas of McCarthy [4, 5] and Landin [6, 7].

4.1. OBJECTS

For the formal semantics of programming languages a general model is required for the data on which programs act. We regard data as consisting of elementary objects, and compound objects formed by combining elementary objects into data structures.

Elementary objects are data items whose structure in terms of simpler objects is not relevant to the description of algorithms. For the present discussion, the class E of elementary objects is

$$E = Z \cup R \cup W$$

where

Z = the class of integers

R = a set of representations for real numbers

W = the set of all strings on some alphabet

Data structures are often represented by directed graphs in which elementary objects are associated with nodes, and each arc is labelled by a member of a set S of selectors. In the class of objects used by the Vienna group, the graphs are restricted to be trees, and elementary objects are associated only with leaf nodes. We prefer a less restricted class so an object may have distinct component objects that share some third object as a common component. The reader will see that this possibility of sharing is essential to the formulation of the base language and interpreter presented here. Our class of objects is defined as follows:

> Let E be a class of elementary objects, and let S be a class of selectors. An object is a directed acyclic graph having a single root node from which all other nodes may be reached over directed paths. Each arc is labelled with one selector in S, and an elementary object in E may be associated with each leaf node.

We use integers and strings as selectors:

$$S = Z \cup W$$

Figure 10 gives an example of an object. Leaf nodes having associated elementary objects are represented by circles with the element of E written inside; integers are represented by numerals, strings are enclosed in single quotes, and reals have decimal points. Other nodes are represented by solid dots, with a horizontal bar if there is more than one emanating arc.

The node of an object reached by traversing an arc emanating from its root node is itself the root node of an object called a component of the original object. The component object consists of all nodes and arcs that can be reached by directed paths from its root node.

4.2. STRUCTURE OF A BASE LANGUAGE INTERPRETER

Figure 11 shows how source languages would be defined in terms of a common base language. A single class of abstract programs constitutes the base language. Concrete programs in source languages (L1 and L2 in the figure) are defined by translators into the base language. The structure of abstract programs cannot reflect the peculiarities of any particular source language, but must provide a

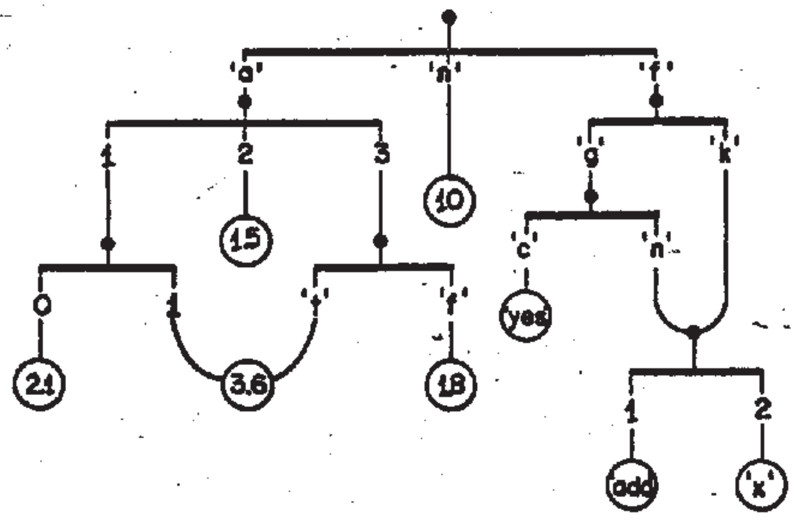


Figure 10. An example of an object.

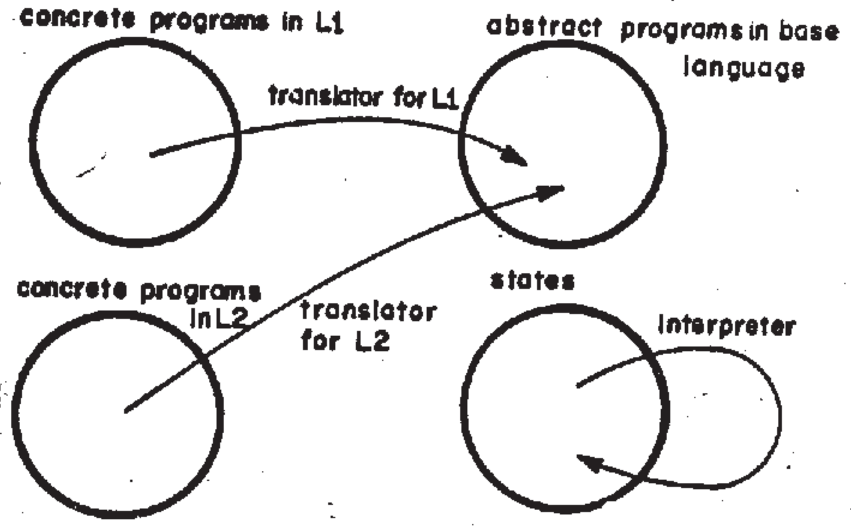


Figure 11. Language definition in terms of a common base language.

set of fundamental linguistic constructs in terms of which the features of these source languages may be realized. The translators themselves should be specified in terms of the base language, probably by means of a specialized source language. The semantics of abstract programs of the base language are specified by an interpreter which is a nondeterministic state-transition system, as in the work of the Vienna group. Formally, abstract programs in the base language, and states of the interpreter are elements of the class of objects defined above.

The structure of states of the interpreter for the base language is shown in Figure 12. Since we regard the interpreter for the base language as a complete specification for the functional operation of a computer system, a state of the interpreter represents the totality of programs, data, and control information present in the computer system. In Figure 12 the universe is an object that represents all information present in the computer system when the system is idle -- that is, when no computation is in progress. The universe has data structures and procedure structures as constituent objects. Any object is a legitimate data structure; for example, a data structure may have components that are procedure structures. A procedure structure is an object that represents a procedure expressed in the base language. It has components which are instructions of the base language, data structures, or other procedure structures. So that multiple activations of procedures may be accommodated, a procedure structure remains unaltered during its interpretation.

The local structure of an interpreter state contains a local structure for each current activation of each base language procedure. Each local structure has as components the local structures of all procedure activations initiated within it. Thus the hierarchy of local structures represents the dynamic relationship of procedure activations. One may think of the root local structure as the nucleus of an operating system that initiates independent, concurrent computations on behalf of system users as they request activation of procedures from the system files (the universe).

The local structure of a procedure activation has a component object for each variable of the base language procedure. The selector of each component is its identifier in the instructions of the procedure. These objects may be elementary or compound objects and may be common with objects within the universe or within local structures of other procedure activations.

The control component of an interpreter state is an unordered set of sites of activity. A typical site of activity is represented in Figure 4 by an asterisk at an instruction of procedure P and an arrow to the local structure L for some activation of P. This is analogous to the "instruction pointer/environment pointer" combination that represents a site of activity in Johnston's contour model [8]. Since several activations of a procedure may exist concurrently, there may be two or more sites of activity involving the same instruction of some procedure, but designating different local structures. Also, within one activation of a procedure, several instructions may be active concurrently; thus asterisks on different instructions of a procedure may have arrows to the same local structure.

Each state transition of the interpreter executes one instruction for some procedure activation, at a site of activity selected arbitrarily from the control of the current state. Thus the interpreter is a nondeterministic transition system. In the state resulting from a transition, the chosen site of activity is replaced according to the sequencing rules of the base language.

4.3. STATE TRANSITIONS OF THE INTERPRETER

Next we show how typical instructions of a rudimentary base language would be implemented by state transitions of an interpreter. This will put the concepts expressed above into more concrete form. For illustration, we will use a representation for procedures that employs conventional instruction sequencing. The instructions of a procedure are objects selected by successive integers, with 0 being the selector of the initial instruction.

The effect of representative instructions on the interpreter state is shown in Figures 13 through 19 in the form of before/after pictures of relevant state components. In these figures, P marks the root of the procedure structure containing an instruction under consideration as its i-component, and L(P) is the root of the local structure for the relevant activation of P.

The add instruction is typical of instructions that apply binary operations to elementary objects. The instruction

add 'u', 'v', 'w'

is an object having as components the four elementary objects 'add', 'u', 'v', and 'w'. These are interpreted as an operation code and three "address fields" used as selectors for operands and result in the local structure L(P). The

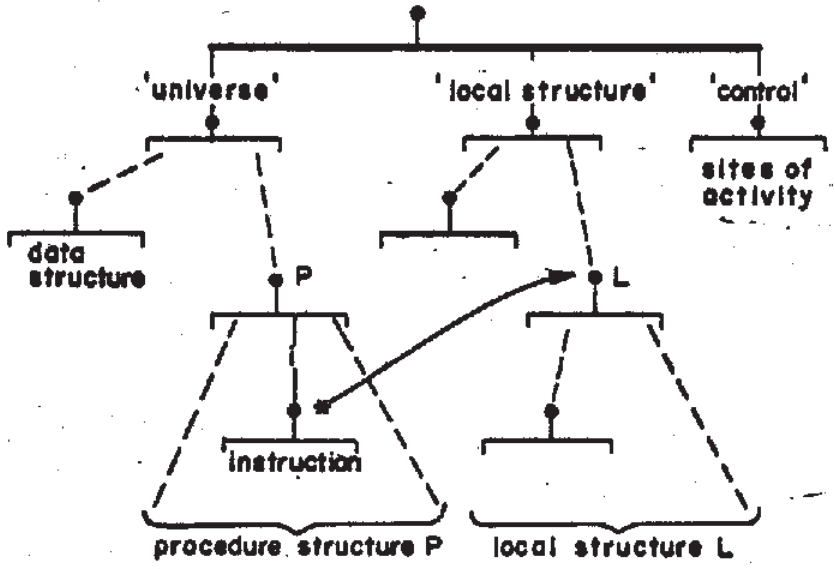


Figure 12. Structure of objects representing states of the base language interpreter.

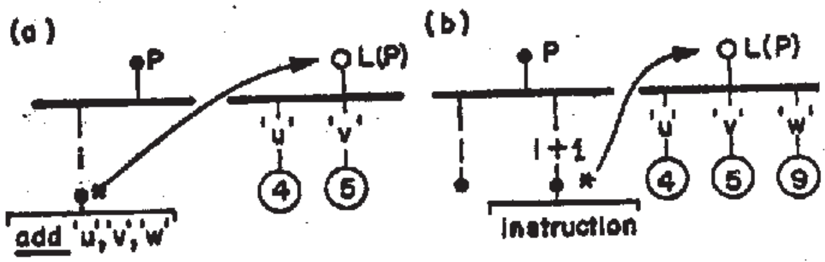


Figure 13. Interpretation of an instruction specifying a binary operation.

state transition is shown in Figure 13. Note that the site of activity advances sequentially to the i+1-component of P.

Let us say that a procedure activation has <u>direct</u> <u>access</u> to a data structure if the data structure is the s-component of the local structure for some selector s. The instruction

<u>select</u> 'p', 'n', 'q'

is used to gain direct access to the 'n'-component of a data structure to which direct access exists. This instruction makes the object that is the 'p'.'n'-component of L(P) also the 'q'-component of L(P), as shown by Figure 14.

Literal values are retrieved from the procedure structure by <u>const</u> instructions such as

<u>const</u> 1.5, 'x'

which makes the elementary object 1.5 the 'x'-component of L(P). <u>Select</u> and <u>const</u> instructions may be used to build arbitrary data structures as illustrated in Figure 15. Note that execution of <u>select</u> 'p', 'n', 'x' implies creation of an 'n'-component of the object selected by 'p' if none already exists.

Figure 16 shows how the instruction

<u>link</u> 'p', 'n', 'q'

establishes an arc between two objects (the 'p'- and 'q'-components of L(P)) to which direct access exists. Execution of this instruction makes the 'q'-component of L(P) also the 'p'.'n'-component of L(P). The <u>link</u> instruction is the means for establishing sharing -- making one object a common component of two distinct objects.

The instruction

<u>delete</u> 'p', 'n'

erases the arc labelled 'n' emanating from the root of the 'p'-component of L(P). Any nodes and arcs that are unrooted after the erasure cease to be part of the interpreter state, as shown in Figure 17.

Although we have not mentioned them in this brief summary, the base language will include appropriate instructions for implementing conditional and iteration statements, and for testing the presence and type of a component of an object.

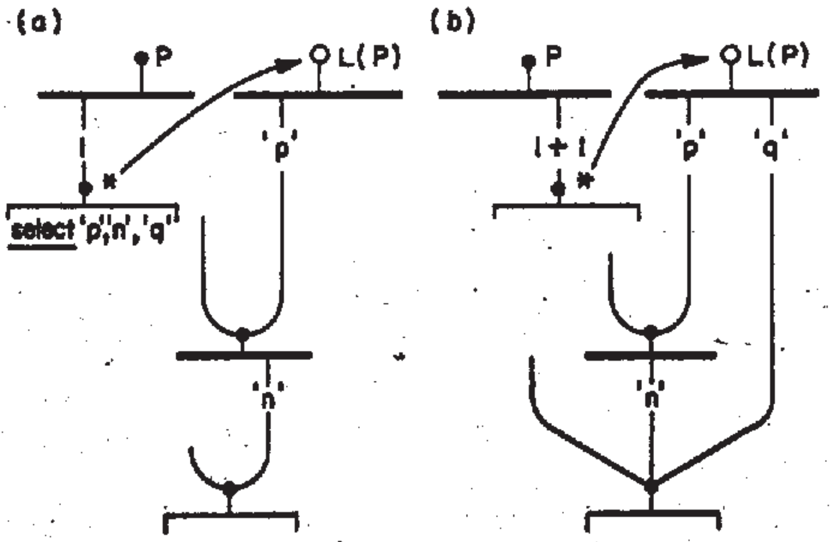


Figure 14. Interpretation of a select instruction.

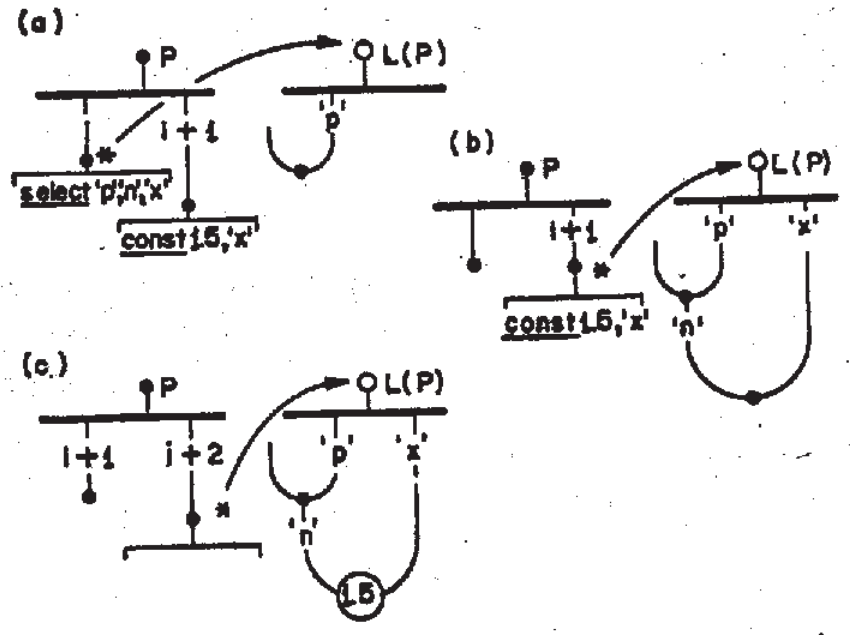


Figure 15. Structure building using select and const instructions.

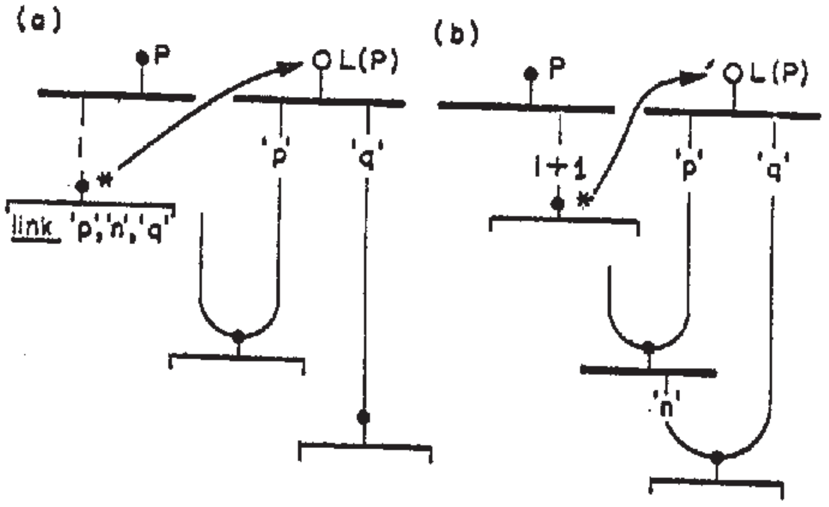


Figure 16. Insertion of an arc by a link instruction.

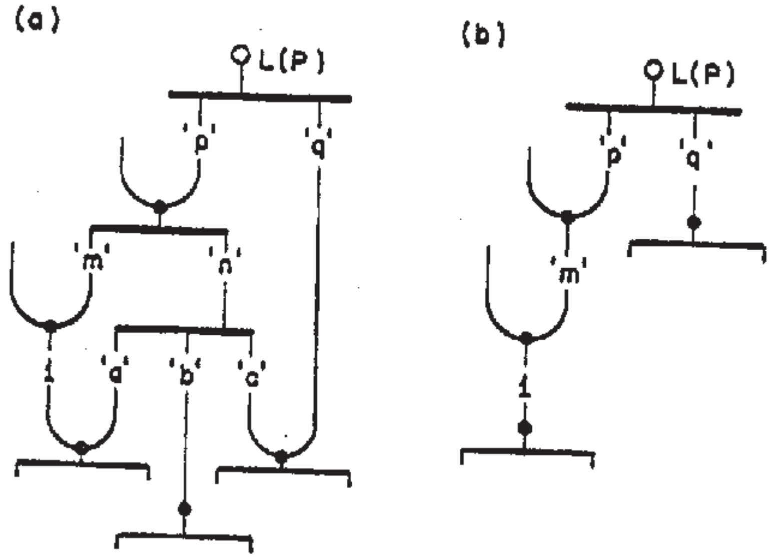


Figure 17. The effect of executing a delete instruction.

Activation of a new procedure is accomplished by the instruction

<u>apply</u> 'f', 'a'

where the 'f'-component of L(P) is the procedure structure F of the procedure to be activated, and the 'a'-component of L(P) is an object (an <u>argument structure</u>) that contains as components all data required by the procedure (e.g., actual parameter values) to perform its function. Execution of the <u>apply</u> instruction causes the state transition illustrated in Figure 18: A root node L(F) is created for the local structure of the new activation; the argument structure is made the A-component of L(F); a new site of activity is denoted by an asterisk on the 0-component of F and an arrow to L(F); and the original site of activity is advanced to the i+1-instruction of P and made doemant as indicated by the parentheses.

A procedure activation is terminated by the instruction

<u>return</u>

which causes the state transition displayed in Figure 19. The root node L(F) is erased, deleting all parts of the local structure of F that are not linked to the argument structure; the site of activity at the <u>return</u> instruction disppears; and the dormant site of activity in the activating procedure is activated. Note that the entire effect of executing procedure F is conveyed to the activation of P by way of the argument structure.

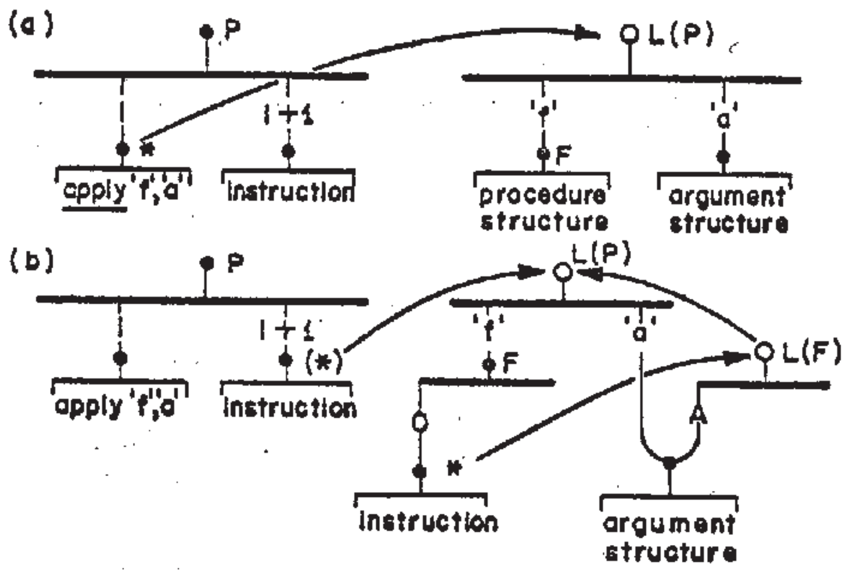


Figure 18. Initiation of a procedure activation by an apply instruction.

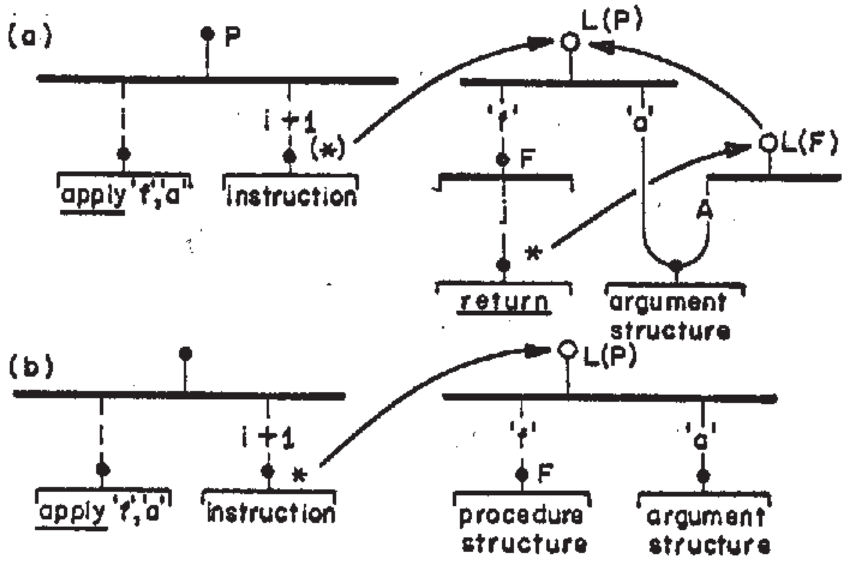


Figure 19. Termination of a procedure activation by a return instruction.

4.4. REPRESENTATION OF MODULAR PROGRAMS

With the foregoing introduction to base language concepts we may study how well the base language could serve the needs of modular programming. First we consider the adequacy of the base language for representing and transforming data structures.

The data types of many practical programming languages have natural representations as objects that are strictly trees (have no shared substructures). These include vectors, arrays, directories, symbol tables, and hierarchical data bases (files). Some data management systems employ representations that provide for sharing of substructures. Also, most data structures occurring in Lisp programs have the form of binary trees with shared subtrees. These structures are directly modelled as objects having shared component objects.

Some important languages, including PL/1, Algol 68, and Lisp, permit the programmer to build data structures containing directed cycles. Such structures do not have direct representations as objects of the base language. It is not yet clear to what extent use of cycles is an essential part of modelling real world semantic constructs in contrast to use of cycles as an implementation technique through which, for example, objects may be represented and efficiently manipulated as lists.

The primitive constructs of the base language provide a general facility for building and manipulating objects. Any object may be constructed by a base language procedure through repeated use of select and const instructions. Through use of link instructions, objects may be made shared components of several objects, and argument structures may be assembled from any finite set of arbitrary objects. In contrast to linguistic levels (such as defined by PL/I) closely tired to the concept of linear address space, passing an object to a base language procedure gives the procedure the ability to transform the object in any way without the possibility of affecting objects not passed to the procedure as part of the argument structure.

In the paragraphs below we show how the use of objects as the fundamental notion of data structure yields natural solutions to a number of issues of language implementation and modular programming.

Recursion: Recursion occurs when a procedure makes application of itself in order to perform its function. In the base language interpreter outlined above, there is no way, without introducing cycles, to make a procedure structure a component of itself so it may be applied recursively. However, as shown in Figure 20, the procedure P that makes the initial application of a recursive procedure F may include the procedure structure of F as a component of the argument structure for its call of F. In this way F may make F a component of its local structure and create recursive activations.

Block structure: Implementation of free variables in procedures requires the ability to access variables by means of nonlocal references, and is essential for many programming languages derived from Algol 60. Although nonlocal references are not permitted in the base language, we may include as part of the argument structure for a procedure application an object having as a component each object to which execution of the procedure may require access because of nonlocal references in the source language program (see Figure 21). In this way, block-structured programs can be translated into base language procedure structures and interpreted correctly. Further details are given in [1].

Procedure variables: Some advanced languages permit assignment of procedure values to variables. In a block-structured language, correct implementation of procedure-valued variables requires use of the notion of the closure of a procedure. In the base language a closure may be represented by an object having two components as shown in Figure 22. The T-component is the text of the procedure and the E-component is an object that contains as components values of the variables that have free occurrences in the procedure text. A closure serves as the value of a procedure variable.

Context: In the base language the correct context for interpretation of names is provided by objects. Each identifier encountered during execution of a procedure is interpreted as the selector of a component of some specific object. The object is the local structure for the procedure activation or some part of the procedure structure itself, if the identifier was chosen by the author of the procedure. Otherwise the object is part of the argument structure. In this way all usual sourees of name conflicts are avoided, and mistakes in use of names lead to error reports rather than unsuspected bindings.

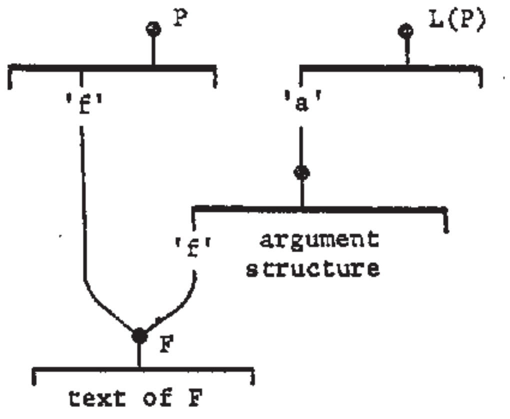


Figure 20. Implementation of a recursive procedure in the base language.

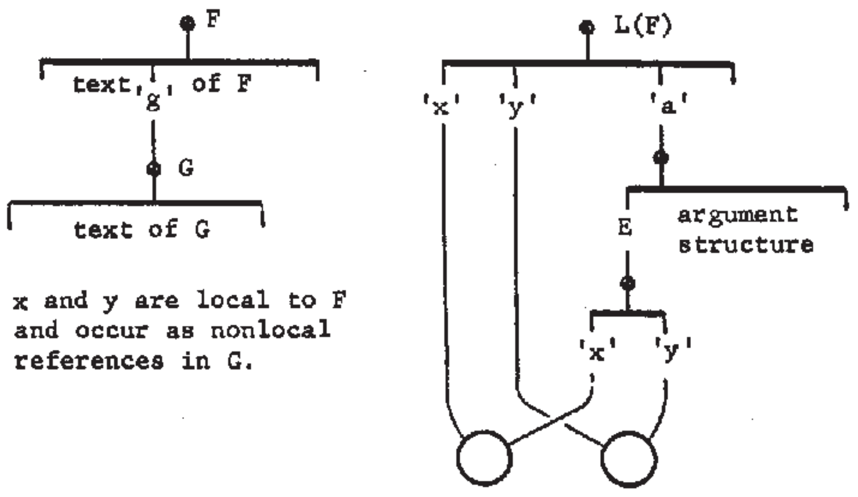


Figure 21. Principle used to translate block-structued programs.

<u>Run-time libraries</u>: Access to library procedures of the implementation of a particular programming language is readily handled in the base language. Each procedure structure resulting from translation of a program in source language A has as its 'lib'-component an object that represents the directory of run-time procedures for language A, as illustrated in Figure 23. This directory is a shared component of all procedure structures produced by translation of programs in language A. Procedures expressed in a different source language B become procedure structures sharing a separate directory of run-time procedures.

4.5. <u>USE OF THE MODEL</u>

The base language is founded on objects as the underlying notion of memory instead of the linear address space. Hence, it may turn out that radically new concepts of computer architecture [9] are required to bring the promised advantages into general practice. Nevertheless, the base language concepts presented here should be valuable to computer system designers interested in producing systems and languages that better serve the needs of modular programming. These ideas may be applied in several ways: They may serve as a guide for those proposing and evaluating advanced concepts of computer organization, and they may help the evolution of programming languages in directions favorable to modular programming. Moreover, the linguistic level of the base language can serve as a standard of achievement -- to be equaled or exceeded by the designer of practical computer systems. It should help designers better understand the true limitations of their systems for modular programming, and where design changes can correct defects that might otherwise plague users for many years after.

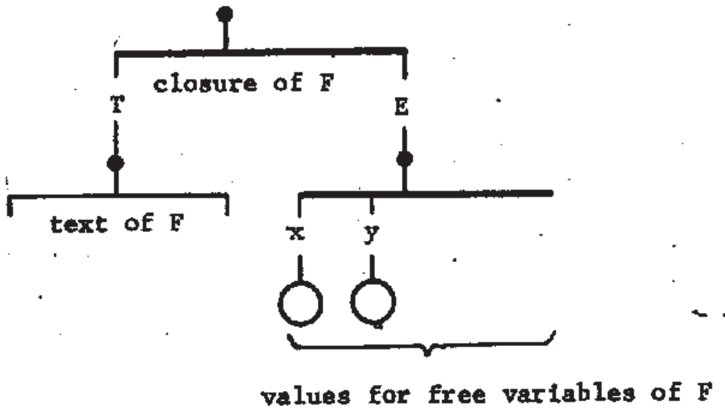


Figure 22. Base language representation for the closure of a procedure.

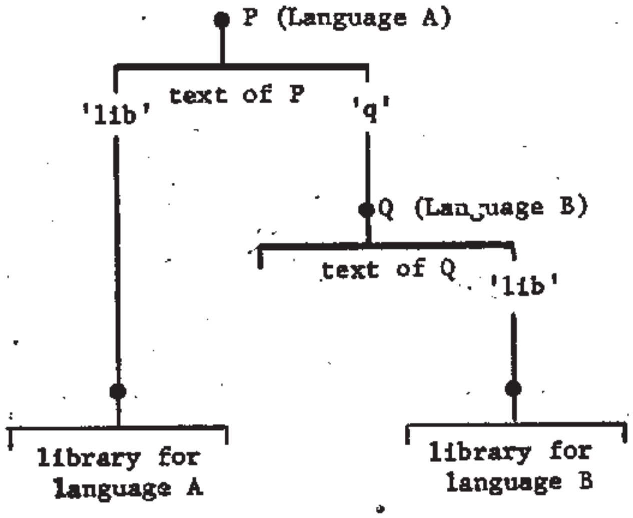


Figure 23. Providing separate libraries for two languages.

REFERENCES

1. J. B. Dennis, On the design and implementation of a common base language. Proceedings of the Symposium on Computers and Automata, Vol. XXI, MRI Symposia Series. Polytechnic Press of the Polytechnic Institute of Brooklyn, Brooklyn, N. Y., 1971.

2. P. Lauer, Formal Definition of Algol 60. Technical Report TR 25.088, IBM Laboratory, Vienna, December 1968.

3. P. Lucas and K. Walk, On the formal description of PL/I. Annual Review in Automatic Programming, Vol. 6, Part 3, Pergamon Press 1969, pp 105-182.

4. J. McCarthy, Towards a mathematical science of computation. Information Processing 62, North-Holland, Amsterdam 1963, pp 21-28.

5. J. McCarthy, A formal description of a subset of Algol. Formal Language Description Languages for Computer Programming, North-Holland, Amsterdam 1966, pp 1-12.

6. P. J. Landin, The mechanical evaluation of expressions. The Computer Journal, Vol. 6, No. 4 (January 1964), pp 308-320.

7. P. J. Landin, Correspondence between Algol 60 and Church's lambda-notation (Parts I and II). Part I: Comm. of the ACM, Vol. 8, No. 2 (February 1965), pp 89-101. Part II: Comm. of the ACM, Vol. 8, No. 3 (March 1965), pp 158-165.

8. J. B. Johnston, The contour model of block structured processes. Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices Vol. 6, No. 2, ACM, February 1971, pp 55-82.

9. J. B. Dennis, Programming generality, parallelism and computer architecture. Information Processing 68, North-Holland, Amsterdam 1969, pp 484-492.