MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 60

# On the Design and Specification of a Common Base Language

Jack B. Dennis

A paper prepared for the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, April 13-15, 1971.

July 1971

(Corrected November 1971)

The work discussed in this article was done at Project MAC, MIT, and was supported in part by the National Science Foundation under research grant GJ-432, and in part by the Advanced Research Projects Agency, Department of Defense, under Naval Research Contract N00014-70-A-0362-0001.

# On the Design and Specification of a Common Base Language*

Jack B. Dennis†
Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract: This is a report on the work of the Computation Structures Group of Project MAC toward the design and specification of a common base language for programs and information structures. We envision that the meanings of programs expressed in practical source languages will be defined by rules of translation into the base language. The meanings of programs in the base language is fixed by rules of interpretation which constitute a transition system called the interpreter for the base language. We view the base language interpreter as the functional specification of a computer system in which emphasis is placed on programming generality — the ability of users to build complex programs by combining independently written program modules.

Our concept of a common base language is similar to the abstract programs of the Vienna definition method — but a single class of abstract programs applies to all source languages to be encompassed. The semantic constructs of the base language must be just those fundamental constructs necessary for the effective realization of the desired range of source languages. Thus we seek simplicity in the design of the interpreter at the expense of increased complexity of the translator from a source language to the base language. As an illustration of this philosophy, we present a rudimentary form of the base language in which nonlocal references are not permitted, and show how programs expressed in a simple block structured language may be translated into this base language.

The importance of representing concurrency within and among computations executed by the interpreter is discussed, and our approach toward incorporating concurrency of action in the base langauge is outlined.

---

† Computation Structures Group, Project MAC, MIT.

* The work discussed in this article was done at Project MAC, MIT, and was supported in part by the National Science Foundation under research grant GJ-432, and in part by the Advanced Research Projects Agency, Department of Defense, under Naval Research Contract N00014-70-A-0362-0001.

## INTRODUCTION

The Computation Structures Group of Project MAC is working toward the design and specification of a base language for programs and information structures. The base language is intended to serve as a common intermediate representation for programs expressed in a variety of source programming languages.

The motivation for this work is the design of computer systems in which the creation of correct programs is as convenient and easy as possible. A major ingredient in the convenient synthesis of programs is the ability to build large programs by combining simpler procedures or program modules, written independently, and perhaps by different individuals using different source languages. This ability of a computer system to support modular programming we have called programming generality [3, 4]. Programming generality requires the communication of data among independently specified procedures, and thus that the semantics of the languages in which these procedures are expressed must be defined in terms of a common collection of data types and a common concept of data structure.

We have observed that the achievement of programming generality is very difficult in conventional computer systems, primarily because of the variety of data reference and access methods that must be used for the implementation of large programs with acceptable efficiency. For example, data structures that vary in size and form during a computation are given different representations from those that are static; data that reside in different storage media are accessed by different means of reference; clashes of identifiers appearing in different blocks or procedures are prevented by design in some source languages but similar consideration has not been given to the naming and referencing of cataloged files and procedures in the operating environment of programs. These limitations on the degree of generality possible in computer systems of conventional architecture have led us to study new concepts of computer system organization through which these limitations on programming generality might be overcome.

In this effort we are working at the same time on developing the base language and on concepts of computer architecture suited to the execution of computations specified by base language programs. That is, we regard the base language we seek to define as a specification of the functional operation of a computer system. Thus our work on the base language is strongly influenced by hardware concepts derived from the requirements of programming generality [3].

In particular, the choice of trees with shared substructures as our universal representation for information structures is based in part on a conviction that there are attractive hardware realizations of memory systems for tree structured data. For example, Gertz [8] considers how such a memory system might be designed as a hierarchy of associative memories. Also, the base language is intended to represent the concurrency of parts of computations in a way that permits their execution in parallel. One reason for emphasizing concurrency is that it is essential to the description of certain computations — in particular, when a response is required to whichever one of several independent events is first to occur. An example is a program that must react to the first message received from either of two remote terminals. Furthermore, we believe that exploiting the potential concurrency in programs will be important in realizing efficient computer systems that offer programming generality. This is because concurrent execution of program parts increases the utilization of processing hardware by providing many activities that can be carried forward while other activities are blocked pending retrieval of information from slower parts of the computer system memory.

Our proposal for the definition of a common base language may seem like a rebirth of the proposal to develop a Universal Computer Oriented Language [24]. Thus it is reasonable to inquire whether there is any better chance that the development suggested here will succeed whereas this earlier work did not result in a useful contribution to the art. Our confidence in eventual success rests on important trends in the computer field during the past ten years and fundamental differences in philosophy. The most important change is the increased importance of achieving greater programming generality in future computer systems. The cost of acquiring and operating the hardware portion of computer systems has become dominated by the expense

of creating and maintaining the system and application software. At present, there is great interest in the exchange of programs and data among computer installations, and in building complex procedures from components through the facilities of time-shared computers. Computer users are often prepared to forsake efficiency of programs to gain the ability to operate them in different environments, and the ability to use the program in conjunction with other programs to accomplish a desired objective.

Furthermore, the pace of programming language evolution has slowed. It is rare that a fundamentally new concept for representing algorithms is introduced. Workers on programming language design have turned to refining the conceptual basis of program representation, providing more natural modes of expressing algorithms in different fields, and consolidating diverse ways of representing similar actions. Today, there is good reason to expect that a basic set of notions about data and control structures will be sufficient to encompass a usefully large class of practical programming languages and applications. In particular, the set of elementary data types used in computation has not changed significantly since the first years of the stored program computer — they are the integers, representations for real numbers, the truth values true and false, strings of bits, and strings of symbols from an alphabet. Also, considerable attention is currently devoted to the development of useful abstract models for information structures, and the prospects are good that these efforts will converge on a satisfactory general model.

We are also encouraged by others who are striving toward similar goals. Andrei Ershov is directing a group at the Novosibirsk Computing Center of the Soviet Union in the development of a common "internal language" for use in an optimizing compiler for three different languages — PL/I, Algol 68, and Simula 67 [7]. The internal language would be a representation common to the three source languages and is to serve as the representation in which transformations are performed for machine independent optimization.

The "contour model" for program execution, as explained by Johnston [10] and Berry [1] provides a readily understood vehicle for explaining the

semantics of programming languages such as Algol 60, PL/I, and Algol 68 in which programs have a nested block structure. It is easy to imagine how the contour model could be formalized and thus serve as a basis for specifying the formal semantics of programming languages. The contour model may be considered as a proposal for a common base language and as a guide for the design of computer systems that implement block structured languages.

John Iliffe has for some time recognized some of the fundamental implications of programming generality with respect to computer organization. His book Basic Machine Principles [9] is a good exposition of his ideas which are argued from the limitations of conventional computer hardware in executing general algorithms. Again, Iliffe's machine defines a scheme of program representation that could be thought of as a common base language. However, Iliffe has not discussed his ideas from this viewpoint.

## FORMAL SEMANTICS

When the meaning of algorithms expressed in some programming language has been specified in precise terms, we say that a formal semantics for the language has been given. A formal semantics for a programming language generally takes the form of two sets of rules — one set being a translator, and the second set being an interpreter. The translator specifies a transformation of any well formed program expressed in the source language (the concrete language) into an equivalent program expressed in a second language — the abstract language of the definition. The interpreter expresses the meaning of programs in the abstract language by giving explicit directions for carrying out the computation of any well formed abstract program as a countable set of primitive steps.

It would be possible to specify the formal semantics of a programming language by giving an interpreter for the concrete programs of the source language. The translator is then the identity transformation. Yet the inclusion of a translator in the definition scheme has important advantages. For one, the phrase structure of a programming language viewed as a set of strings on some alphabet usually does not correspond well with the semantic

structure of programs. Thus it is desirable to give the semantic rules of interpretation for a representation of the program that more naturally represents its semantic structure. Furthermore, many constructs present in source languages are provided for convenience rather than as fundamental linguistic features. By arranging the translator to replace occurrences of these constructs with more basic constructs, a simpler abstract language is possible, and its interpreter can be made more readily understandable and therefore more useful as a tool for the design and specification of computer languages and systems.

The abstract language that has received the most attention as a base for the formal semantics of programming languages is the lambda-calculus of Church. For several reasons we have found the lambda calculus unsuited to our work. The most serious problem is that the lambda calculus does not deal directly with structured data. Thus it is inconvenient to use the lambda calculus as a common target language for programs that make use of selection to reference components of information structures. It also rules out modeling of sharing in the form of two or more structures having the same substructure as a component.

A second defect in terms of our goals is that the lambda calculus incorporates the concept of free and bound variables characteristic of block structured programming languages. We prefer to exclude these concepts so the base language and its interpreter are simpler and more readily applied to the study of computer organization. Later in the paper we show how block structured programs may be translated into base language programs using the rudimentary version of the base language introduced below. This translation of block structured programs into programs that are not block structured is an important example of how simplicity in the interpreter may be obtained by translating source language constructs into more primitive constructs.

Our thoughts on the definition of programming languages in terms of a base language are closely related to the formal methods developed at the IBM Vienna Laboratory [17, 18], and which derive from the ideas of McCarthy [19, 20] and Landin [13, 14]. 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 <u>elementary objects</u> and <u>compound objects</u> 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 purposes of this paper, the class $\underline{\underline{E}}$ of elementary objects is

$$\underline{\underline{E}} = \underline{\underline{Z}} \cup \underline{\underline{R}} \cup \underline{\underline{W}}$$

where

$\underline{\underline{Z}}$ = the class of integers

$\underline{\underline{R}}$ = a set of representations for real numbers

$\underline{\underline{W}}$ = 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 $\underline{\underline{S}}$ 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 $\underline{\underline{E}}$ be a class of elementary objects, and let $\underline{\underline{S}}$ 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 $\underline{\underline{S}}$, and an elementary object in $\underline{\underline{E}}$ may be associated with each leaf node.

We use integers and strings as selectors:

$$\underline{\underline{S}} = \underline{\underline{Z}} \cup \underline{\underline{W}}$$

Figure 1 gives an example of an object. Leaf nodes having associated elementary objects are represented by circles with the element of $\underline{\underline{E}}$ written inside; integers are represented by numerals, strings are enclosed in single

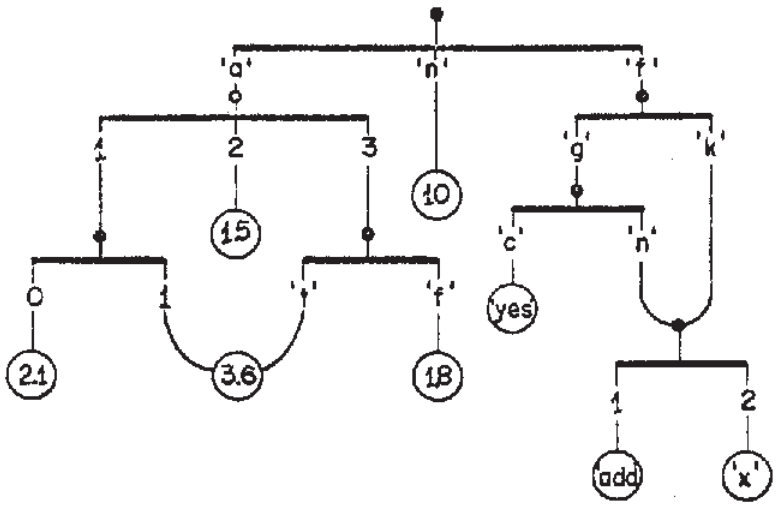



Figure 1. An example of an object.

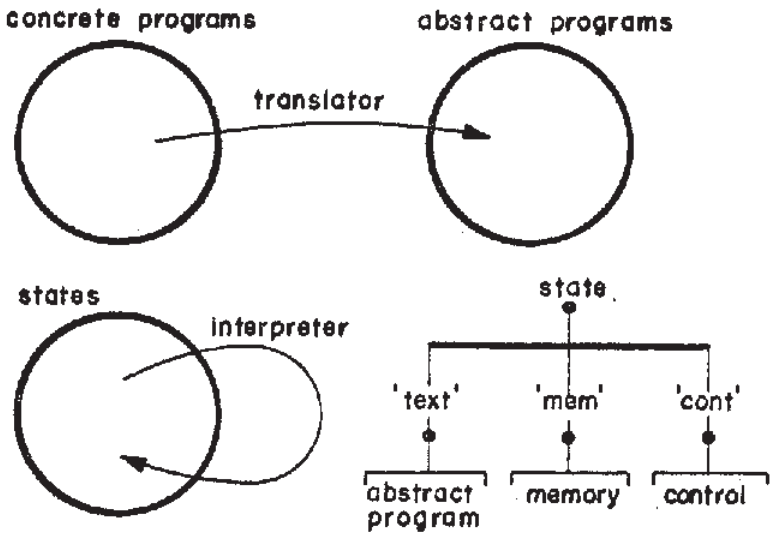



Figure 2. Language definition by the Vienna method.

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.

At present, we rule out directed cycles in the graphs of objects for several reasons: In the first place, the data structures of the most important source languages are readily modelled as objects according to our definition. Also, it seems that realizing the maximal concurrency of computations on data structures will be difficult to do with a guarantee of determinism if objects are permitted to contain cycles. Finally, the possibility of cycles invalidates the reference count technique of freeing storage for data items no longer accessible to computations, and some more general garbage collection scheme must be used. The general techniques do not seem attractive with regard to the concepts of computer organization we have been studying — especially when data items are distributed among several physical levels of memory.

It is convenient to introduce our concept of a base language and its interpreter by comparison with the Vienna definition method as represented by the formal definitions of Algol 60 [15] and PL/I [18]. The Vienna method is outlined in Figure 2. The concrete programs of the programming language being defined are mapped into abstract programs by the translator. A concrete program is a string of symbols that satisfies a concrete syntax usually expressed as a form of context free grammar. The interpreter is a nondeterministic state transition system defined by a relation that specifies all possible next states for any state of the interpreter. Abstract programs and the states of the interpreter are represented by objects (trees). Figure 2 shows the three major components of interpreter states. The 'text'-component is the abstract program being interpreted. The 'mem'-component is an object that contains the values of variables in the abstract program, thus serving as a model of memory. The 'cont'-component of the

state contains information about statements of the abstract program whose execution is in progress. The interpreter is specified as a non-deterministic system so activities may be carried out concurrently where permitted by the language being defined.

For comparison, note that a separate class of abstract programs and interpreter are sepcified for each formal definition of a source language; that states of the interpreter model only the information structures related to execution of one abstract program; and that statements in the concrete program retain their identity as distinct parts of the corresponding abstract program.

Figure 3 is the corresponding outline showing 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 class of abstract programs serves as the common target representation for several source languages. For this to be effectively possible, the base language should be the "least common denominator" of the set of source languages to be accommodated. The structure of abstract programs cannot reflect the peculiarities of any particular source language, but must provide a 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. Formally, abstract programs in the base language, and states of the interpreter are elements of our class of objects defined above.

The structure of states of the interpreter for the base language is shown in Figure 4. 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 a computer system. In Figure 4 the

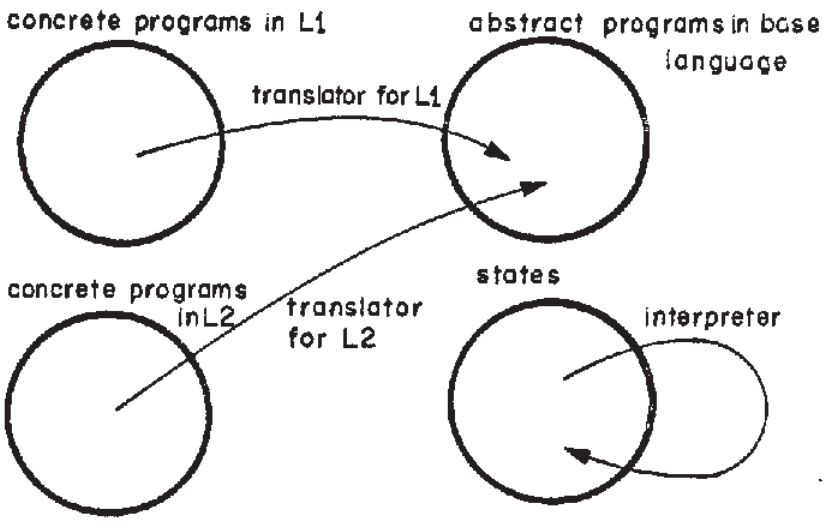



Figure 3. Language definition in terms of a common base language.

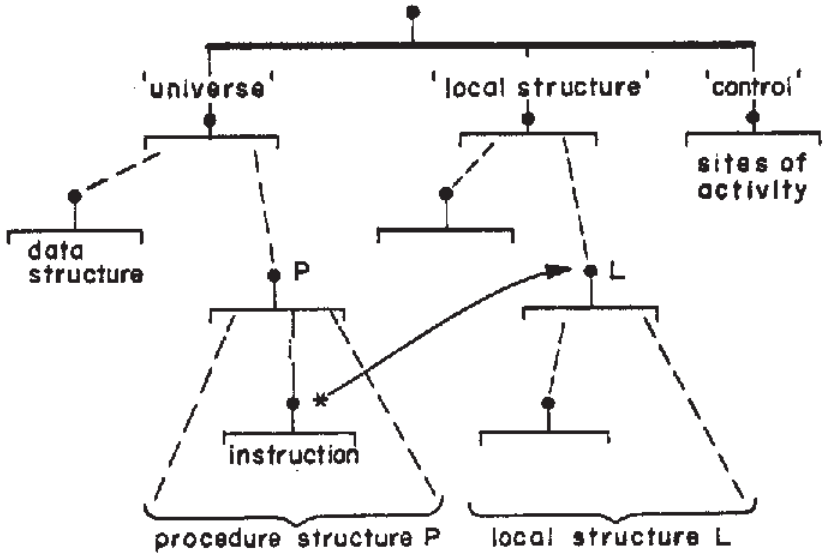



Figure 4. Structure of objects representing states of the base language interpreter.

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 [10]. 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. Replacement with two sites of activity designating two successor instructions would occur in interpretation of a fork instruction; deletion of the site of activity without replacement would occur in execution of a quit or join instruction.

## INTERPRETATION OF A RUDIMENTARY BASE LANGUAGE

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, and provide a basis for understanding the translation of block structured languages into the base language. Because consideration of concurrency in programs has led to concepts of program representation unfamiliar to most readers, and because these concepts are not sufficiently advanced, we will use for illustration a base language employing 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 5 through 11 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 state transition is shown in Figure 5. Note that the site of activity advances sequentially to the i + 1-component of P.

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

select '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 6.

Literal values are retrieved from the procedure structure by const instructions such as

const 1.5, 'x'

which makes the elementary object 1.5 the 'x'-component of L(P). Select and const instructions may be used to build arbitrary data structures as illustrated in Figure 7. Note that execution of select 'p', 'n', 'x' implies creation of an 'n'-component of the object selected by 'p' if none already exists.

Figure 8 shows how the instruction

link '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).

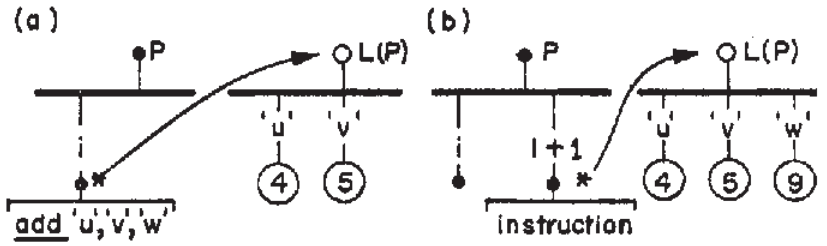



Figure 5. Interpretation of an instruction specifying a binary operation.

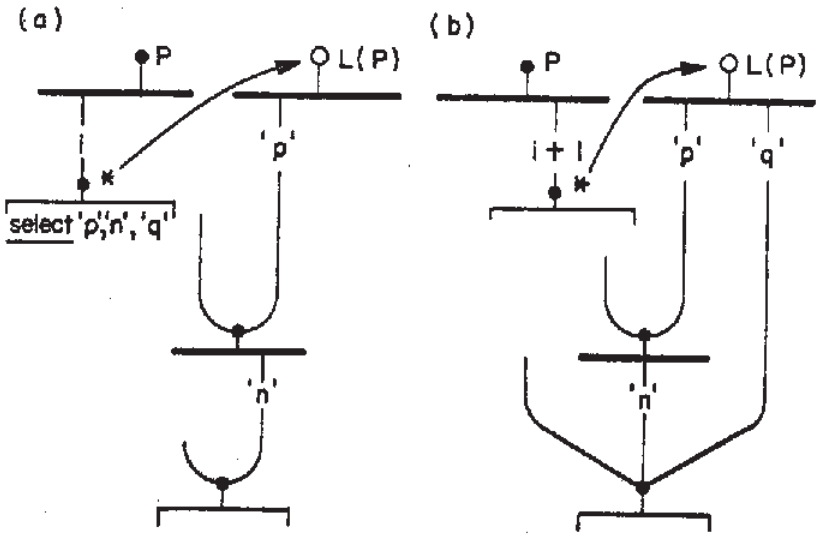



Figure 6. Interpretation of a select instruction.

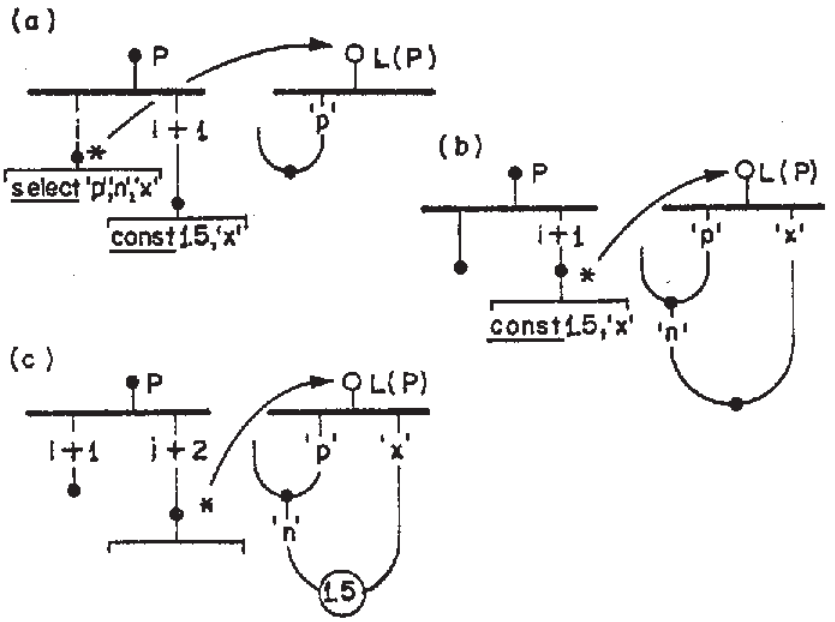



Figure 7. Structure building using select and const instructions.

The link instruction is the means for establishing sharing — making one object a common component of two distinct objects. Unless some restriction is built into the base language or its interpreter, use of link instructions can introduce cycles into the interpreter state. At present we do not know how use of link instructions should be limited so introduction of cycles cannot occur. One way in which cycles can arise occurs in the interpretation of block structured programs by the scheme given in the next section of the paper.

The instruction

delete '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 9.

Activation of a new procedure is accomplished by the instruction

apply '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 argument structure) that contains as components all data required by the procedure (e.g., actual parameter values) to perform its function. Execution of the apply instruction causes the state transition illustrated in Figure 10: 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 dormant as indicated by the parentheses.

A procedure activation is terminated by the instruction

return

which causes the state transition displayed in Figure 11. 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 return instruction disappears; 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.

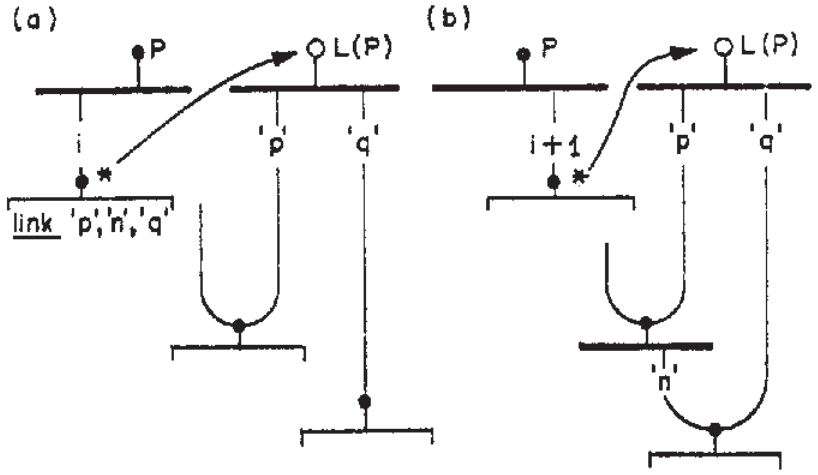



Figure 8. Insertion of an arc by a link instruction.

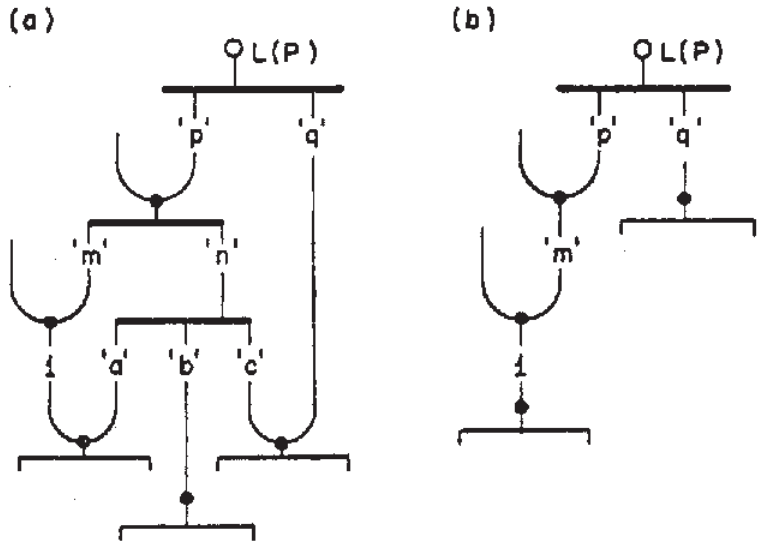



Figure 9. The effect of executing a delete instruction.

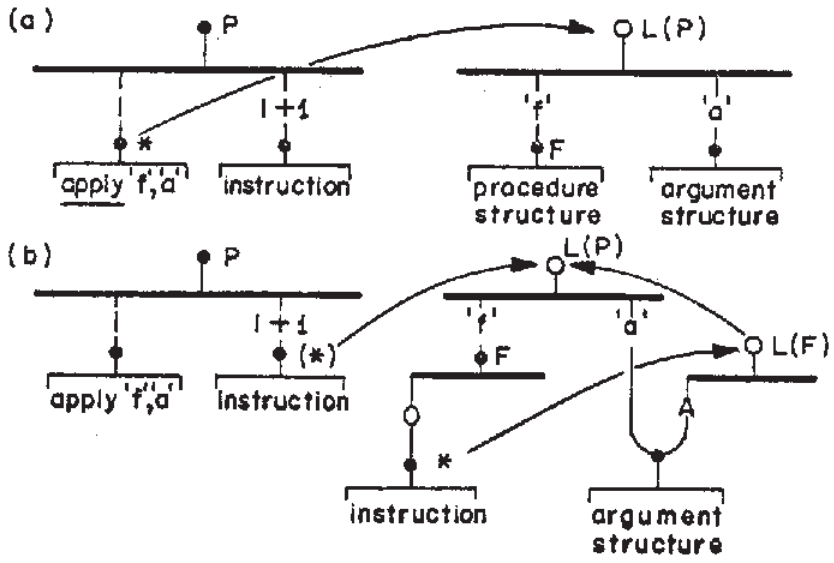



Figure 10. Initiation of a procedure activation by an apply instruction.

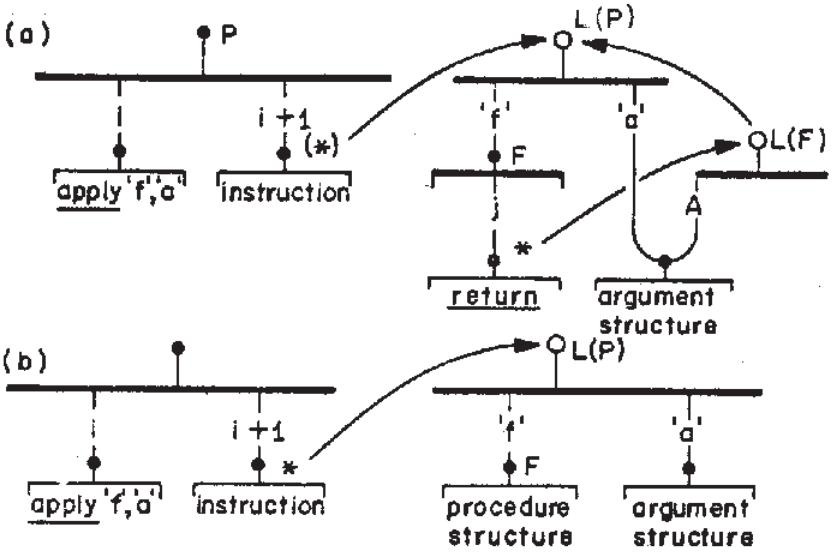



Figure 11. Termination of a procedure activation by a return instruction.

To apply a procedure, its procedure structure must be a component of the local structure of the current procedure activation. If the procedure to be activated is the 'g'-component of the procedure structure P in execution, execution of the instruction

move 'g', 'f'

will make it directly accessible by identifying the 'f'-component of L(P) with the 'g'-component of P.

## TRANSLATION OF BLOCK STRUCTURED LANGUAGES

Many important programming languages for practical computation are block structured; the texts of blocks and procedures are nested, and identifiers in one text may refer to variables defined in other texts. Since we do not plan to include in the base language provision for directly representing references by a procedure to external objects, we must show how the execution of block structured programs may be simulated through translation into the base language and execution by the base language interpreter. The following discussion gives one way in which this may be accomplished — a way that seems attractive in relation to the concepts of computer organization we are investigating. This discussion also serves as a good example of how complexity in a source language may be represented in the rules of translation rather than in the rules of interpretation of a formal definition.

For this discussion we will use an elementary block structured language. Identifiers are declared by the lines

integer x or proced x

to denote simple variables or procedures. Basic statement types include: Assignment statements such as

x := g(u, v)

where x, u, and v are simple variable identifiers, and g denotes an unspecified function; procedure applications of the form

apply f(x, y)  or  z := apply f(x, y)

where f is a procedure identifier, the second form being used for a value returning procedure; and conditional statements like

if p(x) then S1 else S2

and iteration statements like

while p(x) do S1

where p denotes an unspecified predicate and S1 and S2 are basic statements or a sequence of statements delimited by begin, end.

A procedure variable f may be assigned a value by a declaration statement having the form

```
f := procedure (x, ..., y)
     begin
       .
       .
       .
     end
```

where x,...,y are the formal parameters. A statement

return z

specifies the result of a value returning procedure. The lines between begin and end, together with the list of formal parameters, make up the text of the procedure.

A program in this language has the form of a nested set of procedure declarations. Except for the text of the outermost declaration, each text is enclosed by the text within which its declaration appears. As in Algol 60, each identifier is local to the text in which it is declared, and the meaning of a nonlocal appearance of an identifier is defined to be the same as its meaning in the enclosing text. The formal parameters of a procedure are local identifiers of the text being declared.

The meaning of block structured programs can be expressed in terms of a tree of symbol tables as has been explained by Weizenbaum [26], or in terms of the contour model. The interested reader should study the work of Berry [2] and Lucas [16] for other discussions of formal implementations of block structured programs and their equivalence.

To simulate the execution of a block structured program by a base language program, we need a scheme for implementing the nonlocal references of the source program. Our method is to augment the argument structure associated with a procedure activation in the base language interpreter so that all external objects to which reference is required by the block structured procedure are accessed as components of the argument structure.

To make matters precise, it is convenient to adopt some notation. Suppose T is the text of a procedure declaration. We write B(T) to denote the set of identifiers declared within T (local to T). The set X(T) of external identifiers associated with text T is defined as follows: We write $T' < T$ if text T' is nested within text T, that is, if there is a sequence of texts $T_0, T_1, \ldots, T_k$ such that $T = T_0$, $T' = T_k$, and $T_i$ encloses $T_{i+1}$ for $i = 0, \ldots, k-1$. Then X(T) contains each identifier x that has a nonlocal appearance in some text T', $T' \leq T$, and is not local to any text T'', $T' < T'' \leq T$.

In these terms we can describe the formats of the local structures and argument structures to be used in simulation of block structure in the base language. Corresponding to the activation record for an activation of procedure text T, a local structure (L-structure) is formed by the base language program. The L-structure has the format shown in Figure 12a. It has an E-component in which a value is associated with each identifier in $B(T) \cup X(T)$, that is, each local and each external identifier of T. The L-structure also includes components for temporary values required by the base language instructions that interpret the text T.

The argument structure (A-structure) for an activation of procedure text T will have one component for each formal parameter of the text T, and in addition, an E-component that conveys access to objects referenced by the external identifiers of T, as shown in Figure 12b.

A procedure identifier is given a value by a procedure declaration

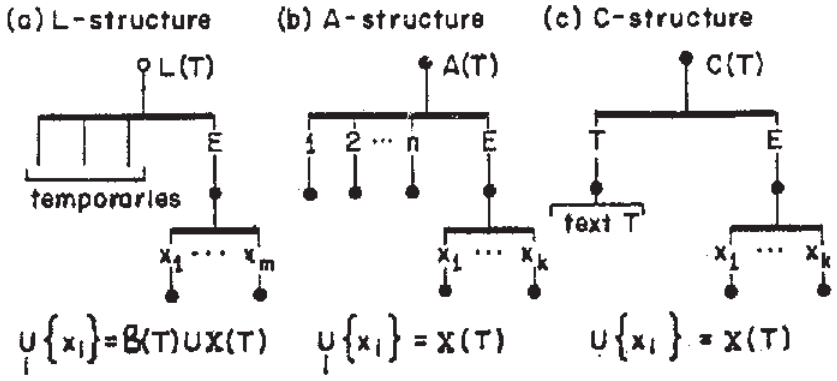



Figure 12. Formats of local, argument, and closure structures for the interpretation of block structured programs.

statement including a text T. Because procedure values may be assigned to nonlocal identifiers, and may be passed to the calling activation by a value returning procedure, activations of the text T may occur in situations where there is no clear meaning for the external identifiers of T. The usual solution to this problem is to let a procedure value be an object called a closure of the text T (a C-structure) having two components as in Figure 12c. The T-component of a closure is the text itself. The E-component (environment) includes an x-component for each x in X(T), and gives an activation of the text access to objects referenced by its external identifiers.

Usually, the meaning of the external identifiers of a closure of T is fixed at the time the closure is created by execution of the declaration of T. Each $x \in X(T)$ is given the same meaning as the current meaning of x in the text T' that encloses the declaration statement.

The way in which block structured programs may be simulated by the base language interpreter is best introduced by an example. The following program is adapted from Weizenbaum's paper [26]:

program 1:

```
p := procedure
begin proced f, q, r; integer u, v, z
        f := procedure (x); integer x
        begin proced g
                g := procedure (y); integer y
                begin integer t
                        t := (x + y) ↑ 2
                        return t
                end
                return g
        end
        q := apply f(1)
        r := apply f(2)
        u := apply q(3)
        v := apply r(5)
        z := u + v
end
```

The program consists of three procedure texts P, F and G having local and external identifiers as follows:

| | | |
|---|---|---|
| $B(P) = \{f, q, r, u, v, z\}$ | $B(F) = \{x, g\}$ | $B(G) = \{t, y\}$ |
| $X(P) = \emptyset$ | $X(F) = \emptyset$ | $X(G) = \{x\}$ |

Following Weizenbaum and Johnston, we display the progress of a computation by giving a series of snapshots of the interpreter state, chosen to illustrate points about the execution mechanism. For procedure P, the initial state of the interpreter (Snapshot 1, Figure 13) includes the text of P in the form of a procedure structure. This procedure structure is in fact a tree of procedure structures; for each text $T \leq P$, the procedure structure for T has as a component a procedure structure for each text enclosed by T. We will not describe further the coding of procedure texts as sets of instructions, as the required instruction sequences will be clear from the discussion of the state transitions seen in the series of snapshots. The initial state also includes a local structure L(P) that will serve as the activation record for procedure P; it is empty except for the argument structure A(P), which consists of an empty E-component.

For clarity, the arcs that make each argument structure a component of the local structures of the calling and called procedures are omitted from the snapshots. Also, we will not include the procedure structure for P in subsequent snapshots, its presence being understood throughout the computation.

The first step performed by instructions of the base language representation of P is to create an E-component of its L-structure, and an E·'x'-component for each identifier x in $B(P) \cup X(P) = \{f, q, r, u, v, z\}$. Execution of the declaration of text F yields snapshot 2. The E·'f'·C-component of L(P) is now a closure of F represented by a C-structure. Its T-component is the text of F and is shared with the text of P, its E-component is empty because $X(F) = \emptyset$.

The first step in the execution of

$$q := \underline{\text{apply}}\ f(1)$$

is to form an appropriate argument structure A(F1). Its 1-component is the actual parameter value, and its E-component is empty, again because $X(F) = \emptyset$.

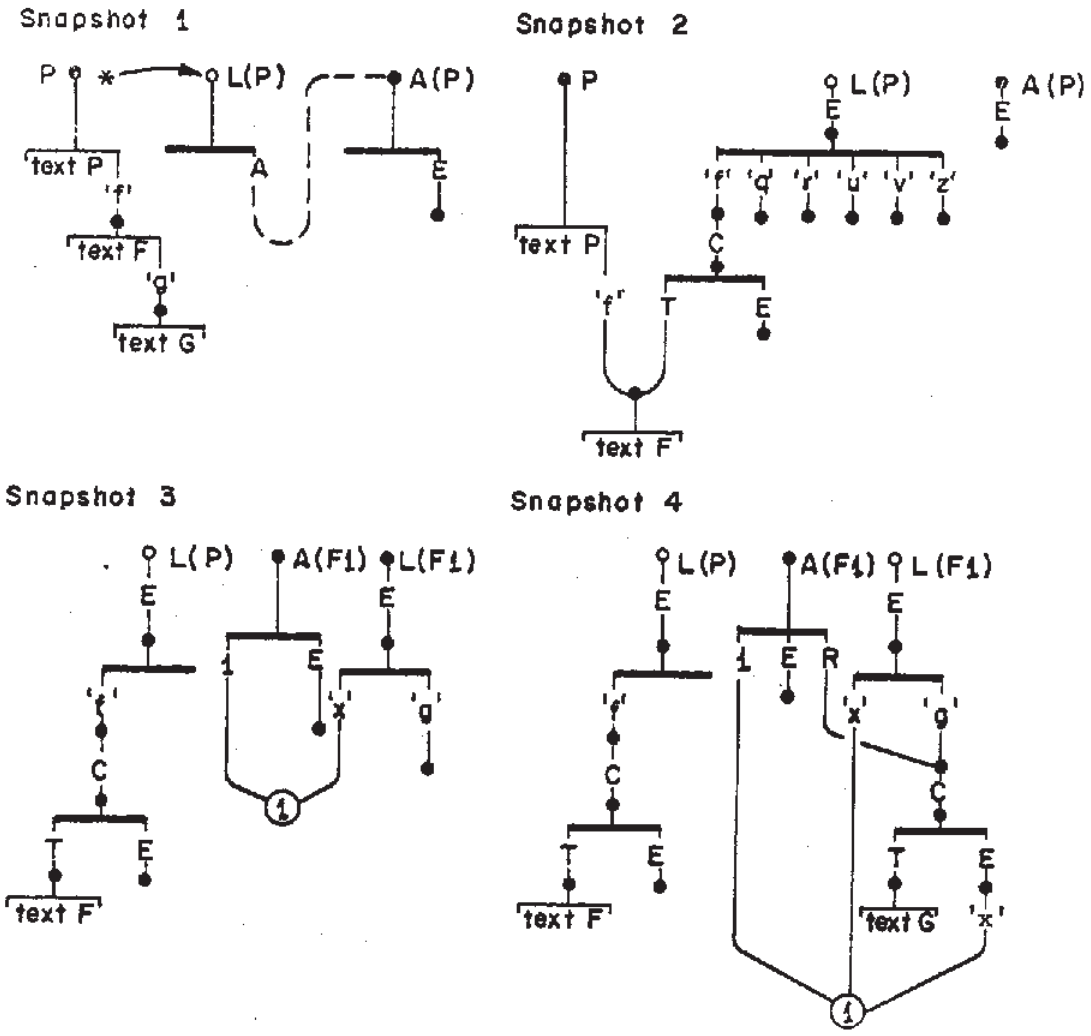



Figure 13. Interpretation of a block structured program — formation of a closure of text G.

Activation of the procedure structure for F creates an L-structure L(F1). The first action by instructions of F is to associate actual parameters with identifiers in B(F). Thus the E•'x'-component of L(F1) is linked to the 1-component of A(F1) as in snapshot 3.

Snapshot 4 shows the effect of interpreting the declaration of G. This adds a closure of G as the E•'g'•C-component of L(F1). The meaning of identifier x, which is an external identifier of G, is fixed in the closure by making the E•'x'-component of the closure identical with the E•'x'-component of the current L-structure. Snapshot 4 also shows the effect of the statement return g which links the E•'g'-component of L(F1) as the R-component (result value) of the argument structure A(F1). This action completes execution of the instructions of F, hence L(F1) is deleted and execution of instructions of P is resumed. To complete interpretation of the statement q := apply f(1), the R-component of A(F1) is made the E•'q'-component of L(P), and A(F1) is deleted. The result is shown in snapshot 5 (Figure 14), which also shows the effect of interpreting r := apply f(2) by a similar sequence of events

The progress of this computation through snapshot 5 illustrates how values required to interpret external references may be conveyed to a procedure activation via the argument structure, and how closures of a text may be formed to fix the meaning of the external (free) identifiers in a procedure declaration — all without going outside the base language features we have introduced. The remaining snapshots show what is involved in applying a closure with a nonempty E-component.

Interpretation of the statement u := apply q(3) begins with formation of an argument structure A(G1) as in snapshot 6, Figure 14. Here, since X(G) = {x}, an E•'x'-component of A(G1) is created and made identical with the E•'x'-component of the closure value of q in L(P). Then the initial instructions of G identify the E•'y'-component of L(G1) with the 1-component of A(G1), and, since $x \in X(G)$, identify the E•'x'-component of L(G1) with the E•'x'-component of A(G1). Instructions corresponding to the body of G compute the value t = (1 + 3) ↑ 2 = 16 which is returned as the R-component of A(G1). The result is snapshot 7 which includes the effect of interpreting the statements v := apply r(5) and z := u + v.

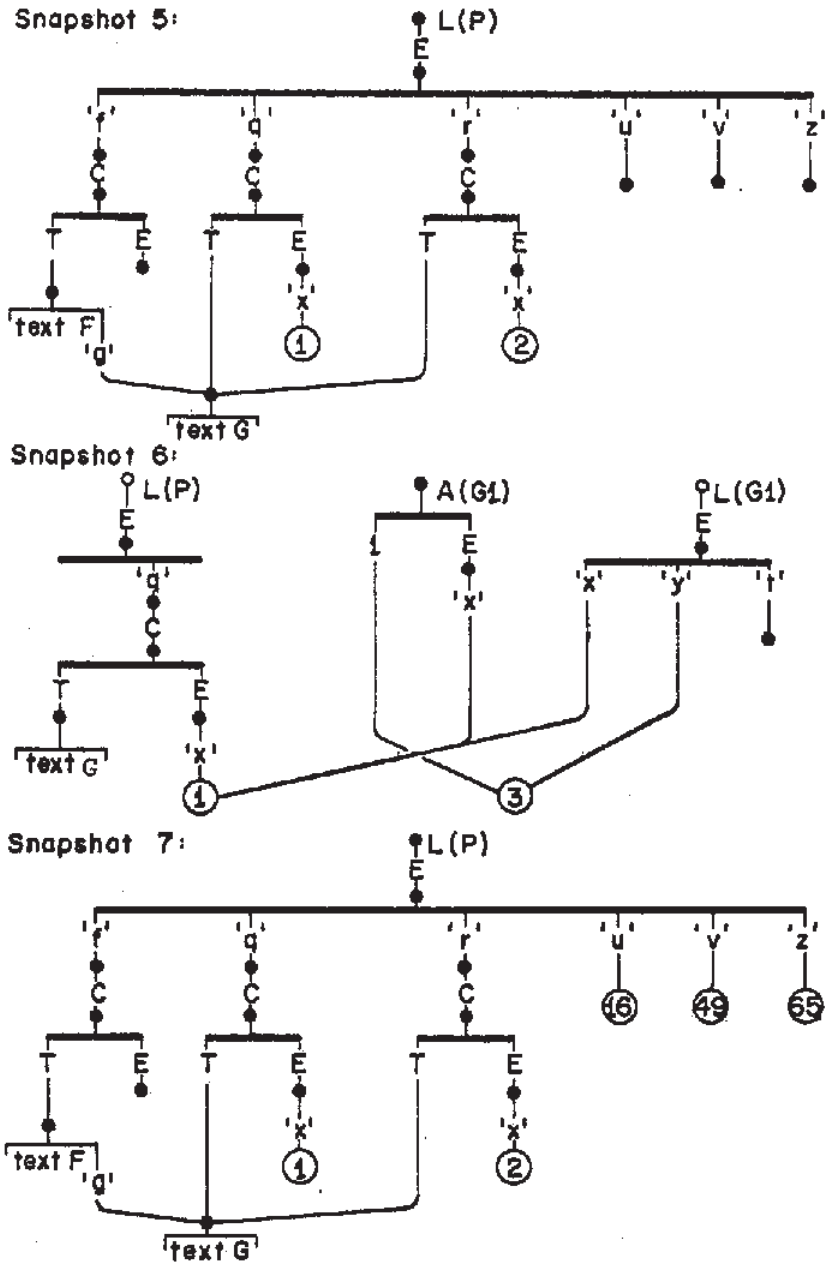



Figure 14. Interpretation of a block structured program — application of closures.

With the above example as a guide, we can formulate a general set of rules governing the simulation of block structured programs by the base language interpreter.

1. Formation of an argument structure for application of a closure of text T:
   a. The $i^{th}$ actual parameter for application of text T is made the i-component of A(T).
   b. For each identifier x in X(T), the E•'x'-component of the closure of T to be applied is made the E•'x'-component of A(T).
2. Initialization of the local structure L(T):
   a. For each x ∈ X(T), the E•'x'-component of A(T) is made the E•'x'-component of L(T).
   b. For each x ∈ B(T), an empty E•'x'-component is appended to L(T).
   c. Each actual parameter value (i-component of A(T)) is made the E•'x'-component of L(T), where x is the identifier of the $i^{th}$ formal parameter.
3. Return of value:
   a. Interpretation of the statement return x makes the R-component of A(T) identical with the E•'x'-component of L(T).
   b. Interpretation of z := apply f( , ..., ) in text T is completed by making the E•'z'-component of L(T) identical with the R-component of the argument structure for application of the closure f.
   c. The argument structure is deleted from L(T).
4. Formation of a closure of text T as the value of identifier f in an activation of text T':
   a. The new C-structure is the E•'f'•C-component of L(T').
   b. The text T is made the T-component of the C-structure.
   c. For each x ∈ X(T), the E•'x'-component of L(T') is made the E•'x'-component of the C-structure.

5. Interpretation of a procedure assignment statement f := g in text T:
   a. The E•'g'•C-component of L(T) is made the E•'f'•C-component of L(T), a previously existing C-arc from procedure node E•'f' being deleted.

## CYCLES AND THEIR PREVENTION

The method of simulating block structured programs presented above has a major defect in terms of our objectives for the base language: Interpretation of programs can lead to interpreter states for which the graph of the state has directed cycles and is not an object according to our definition. The simplest case is the following program:

program 2:

```
p := procedure (u)
begin
    f := procedure (x); integer x
    begin
        if x = 0 then return 1
        x := g(x)
        z := apply f(x)
        return z
    end
    apply f(u)
end
```

(P: p := procedure (u) ... end; F: f := procedure (x); integer x ... end)

The snapshot in Figure 15a shows the situation just after interpretation of the declaration of text F. The cycle arises because of the free occurrence of f within text F, where the value of f is a closure of F.

To understand in general the conditions under which cycles are introduced, it is instructive to use diagrams showing all nonlocal references to the procedure variables of programs being studied. A procedure variable is uniquely specified by a pair (x, T) where x is a procedure identifier and T is a text in which x is declared, that is, $x \in B(T)$. We write R(x, T) to represent

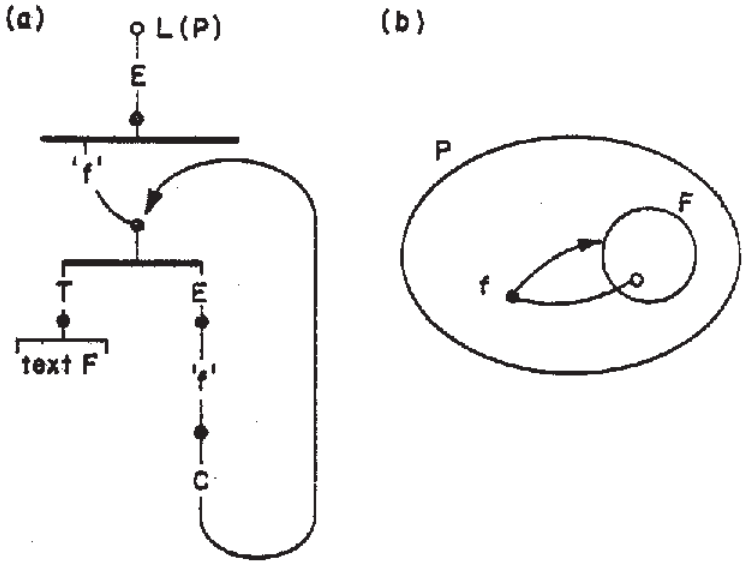



Figure 15. Interpreter state for program 2 showing the cycle introduced, and the corresponding procvar diagram.

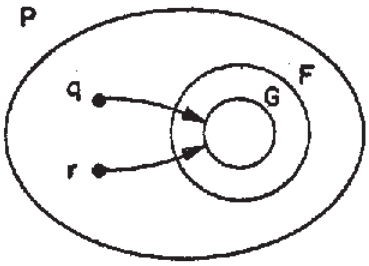



Figure 16. Procvar diagram for program 1 showing absence of necessary conditions for the occurrence of cycles.

the range of a procedure variable; R(x, T) is a set containing each text T' such that the variable (x, T) could be assigned a closure of T' as its value during program execution. The range of a procedure variable may be determined by tracing references to, and assignments of, the closures defined by procedure declarations. We suspect that, unless a program has redundant or unproductive statements, there will be some interpretation for its function and predicate symbols such that each element of the range of a procedure variable occurs as its value in some computation by the program.

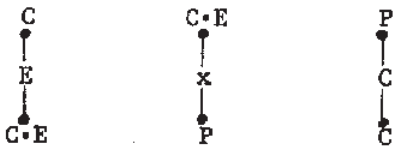
To construct the procedure variable diagram (procvar diagram for short) of a block structured program, represent the texts of the program by closed contours nested in the same way as the texts. The area inside the contour for T but outside contours for texts enclosed by T is the locality of T. Let (x, T) be a procedure variable of the program, and represent it by a solid dot labelled x and placed in the locality of T. Place a small open circle in the locality of T' for each text T' with T' < T in which identifier x refers to the procedure variable (x, T). Join each of these circles to the solid dot denoting (x, T) by arcs without arrows. For each text T' in the range of variable (x, T), draw an arrow from the solid dot representing (x, T) to the contour for T'. Repeat these steps for each procedure variable of the program.

The procvar diagram for program 2 is shown in Figure 15b, and the diagram for program 1 appears in Figure 16.

Next we formulate a necessary condition for a block structured program to generate cycles when interpreted according to our rules of simulation. First consider the forms a cycle must have in an interpreter state. There are nine kinds of nodes involved in the interpretation of block structured programs:

L: root nodes of L-structures
L•E: environment nodes of L-structures
A: root nodes of A-structures
A•E: environment nodes of A-structures
S: simple variable nodes
P: procedure variable nodes
C: root nodes of C-structures
C•T: text nodes of C-structures
C•E: environment nodes of C-structures

Of these, types L and A cannot occur in cycles because no action by the interpreter creates any arcs terminating on L-nodes or A-nodes (aside from the implicit links we have omitted from the diagrams). Further, arcs terminating on L•E- or A•E-nodes can only emanate from L- and A-nodes, respectively. Hence these node types cannot occur in cycles. No arcs emanate from S-nodes, and no arcs from procedure structures terminate on nodes of L-structures; therefore S-nodes and T-nodes cannot occur in cycles. These considerations leave just three kinds of arcs that can be members of any cycle (x is some procedure identifier):



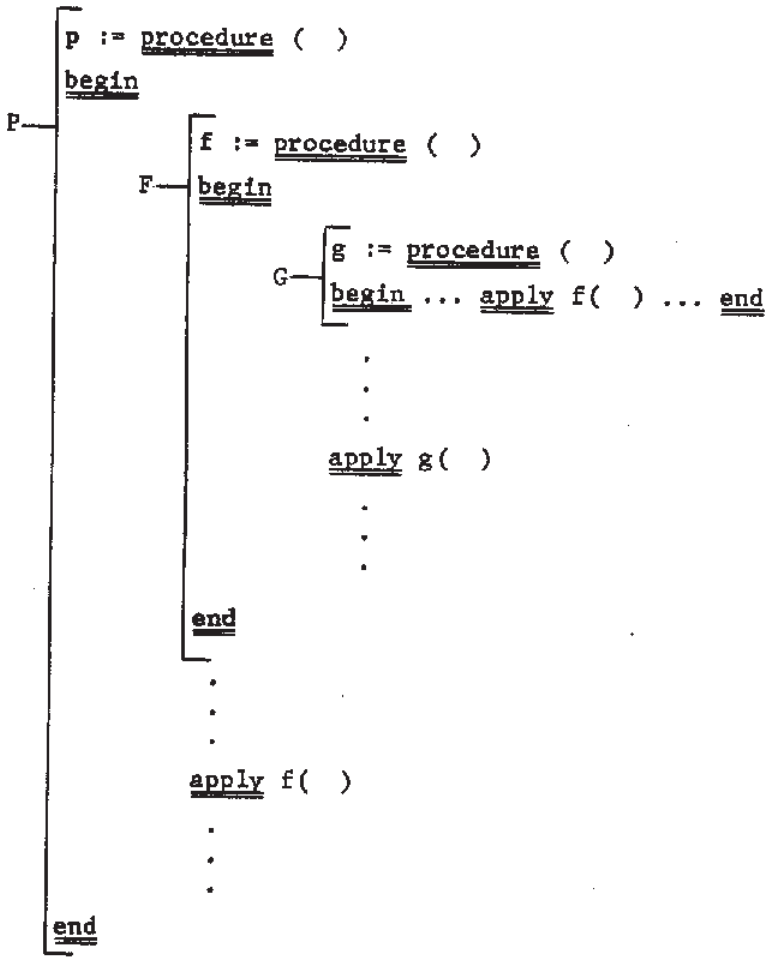

Thus a cycle in an interpreter state consists of a series of triplets, each triplet having one of each kind of arc, in the order shown above. From this reasoning, we deduce that a cycle arises from interpretation of a block structured program only if there is a finite sequence of texts $T_1, T_2, \ldots, T_k$, and a corresponding sequence of identifiers $x_1, x_2, \ldots, x_k$ that meet these conditions:

1. Each $x_i$ is an external procedure identifier of $T_i$: $x_i \in X(T_i)$. Let $(x_i, T_i')$ be the procedure variable denoted by $x_i$ in text $T_i$. Note that $T_i < T_i'$.
2. For each i and with $j = (i \bmod k) + 1$, $T_j$ is in the range of $(x_i, T_i')$.

These conditions imply that the procvar diagram of a program has a cycle of arrows such that each arrow terminates on the contour of a text that contains an external reference to the procedure variable from which the next arrow emanates. For program 2, Figure 15b shows a cycle that involves just one procedure variable (f, F).

Program 3 below is a nest of procedures activated recursively.

program 3:




The procvar diagram for thisprogram is shown in Figure 17a, and Figure 17b illustrates the interpreter state resulting from simulation through the first activation of text G. Still the cycle only involves procedure variable (f, P) because the only external reference is the appearance of f in text G.

The sort of program that leads to more complex cycles is illustrated by

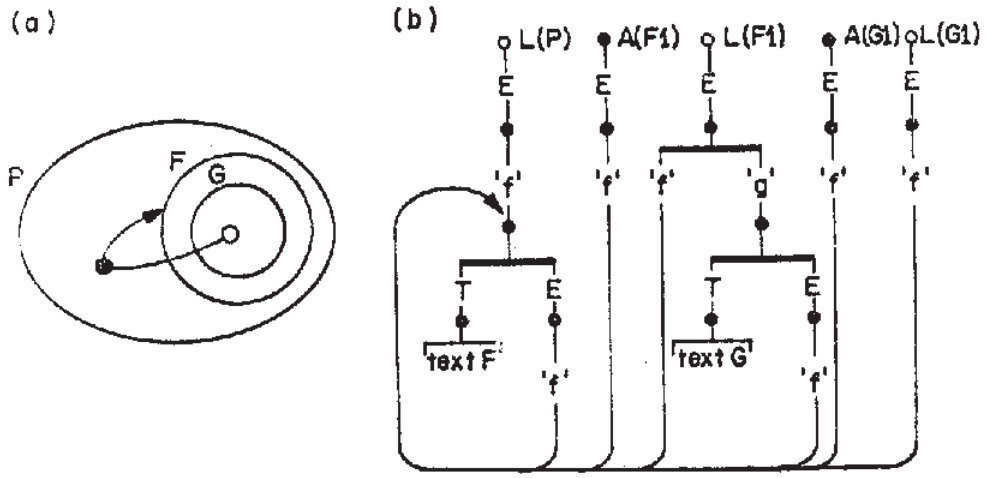



Figure 17. Procvar diagram and interpreter state for program 3.

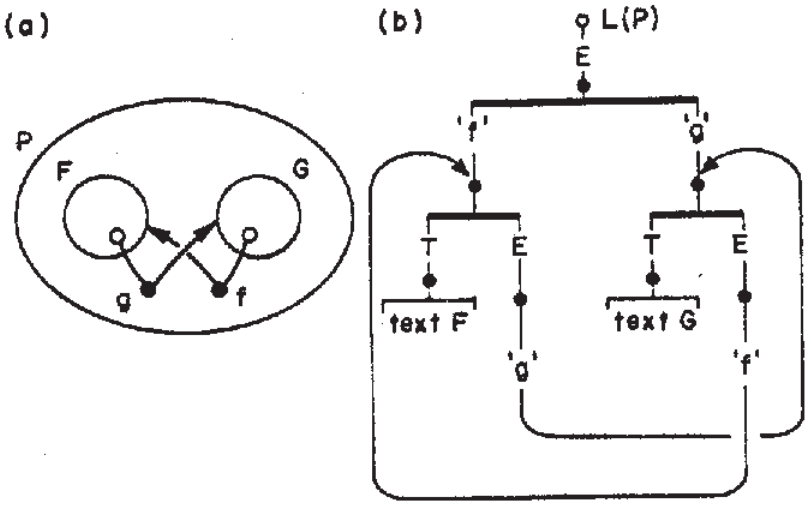



Figure 18. Procvar diagram and interpreter state for program 4.

program 4:

```
P─┌ p := procedure ( )
  │ begin
  │
  │      F─┌ f := procedure ( )
  │        └ begin ... apply g( ) ... end
  │
  │      G─┌ g := procedure ( )
  │        └ begin ... apply f( ) ... end
  │
  │          .
  │          .
  │          .
  │        apply f( )
  │          .
  │          .
  │          .
  │ end
  └
```

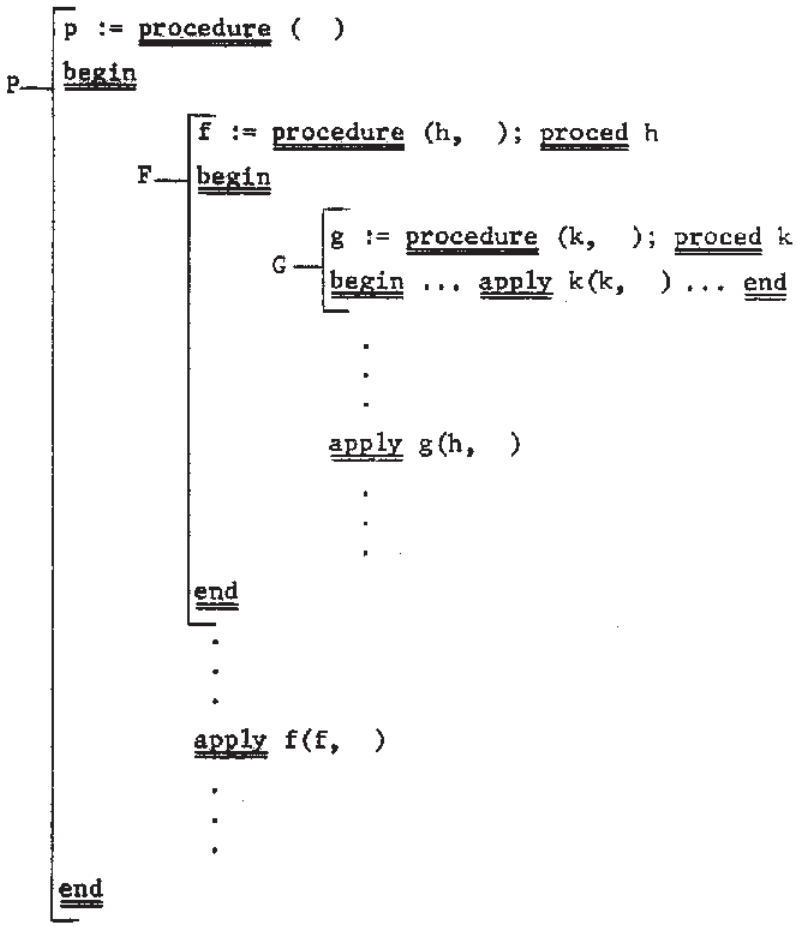
Figure 18 gives the procvar diagram for program 4 and shows the state of the interpreter after the declarations of F and G have been executed. The cycle involves procedure variables (f, P) and (g, P).

We have found that many block structured programs can be rewritten so they accomplish the original computation but no longer satisfy the necessary condition for the creation of cycles. The principle is to convey closures to and from a procedure activation by passing them as parameters or results rather than by external references. In this way, the three example programs may be rewritten as the three transformed programs given below. In each case the texts of the transformed programs do not contain any external references to procedure variables and therefore cannot lead to cycles when performed by the interpreter we have described.

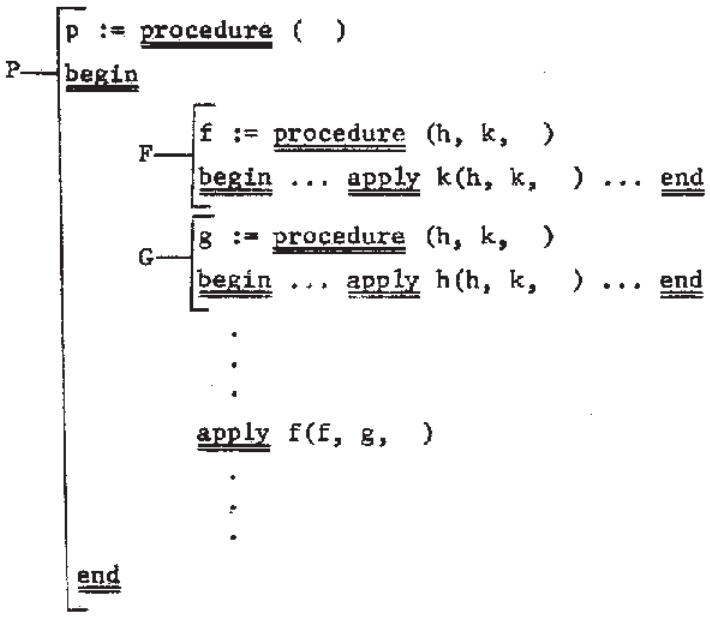
program 2':

```
  ┌ p := procedure (u)
  │ begin
  │
P─│      ┌ f := procedure (h, x) proced h, integer x
  │      │ begin
  │   F──│       if x = 0 then return 1
  │      │       x := g(x)
  │      │       z := apply h(h, x)
  │      │       return z
  │      │ end
  │
  │      apply f(f, u)
  │ end
  └
```

program 3':




program 4':



Several interesting questions are unresolved at this writing. We do not know in what sense, if any, the necessary condition formulated above is a sufficient condition for the formation of cycles during interpretation according to the scheme outlined. Also, we do not know a general method for rewriting block structured programs so that cycles will not arise during execution.

## REPRESENTATION OF CONCRRENCY IN THE BASE LANGUAGE

A subject of major importance in the design of the base language is the representation of concurrent activities. In the introduction we noted that some computations inherently involve concurrent processes and cannot be simulated by sequential programs — also, that a high degree of concurrency within computations may prove essential to the practical realization of computer systems with programming generality. To these motivations we may add that some contemporary source languages, notably PL/I, have explicit provision for programming concurrent processes.

We regard the state transitions of the interpreter as representing the progress of all activities in a computer system that is executing many programs simultaneously. The basic requirements for representing concurrent actions in the interpreter are met by providing for many sites of activity in the control component of the state (Figure 3), and by organizing the local structures of procedure activations as a tree so a procedure may spawn independent, concurrent activations of component procedures. Multiple sites of activity may represent many actions required to accomplish different parts of one computation as well as parallel execution of many independent computations.

Consideration of concurrent computation brings in the issue of nondeterminacy — the possibility that computed results will depend on the relative timing with which the concurrent activities are carried forward. The work of Van Horn [27], Rodriguez [22] and others has shown that computer systems can be designed so that parallelism in computations may be realized while determinacy is guaranteed for any program written for the system. The

ability of a computer user to direct the system to carry out computations with a guarantee of determinacy is very important. Most programs are intended to implement a functional dependence of results on inputs, and determinism is essential to the verification of their correctness.

There are two ways of providing a guarantee of determinacy to the user of a computer system. The distinction is whether the class of abstract or base language programs is constrained by the design of the interpreter to describe only determinate computations. If this is the case, then any abstract program resulting from compilation will be determinate in execution. Furthermore, if the compiler is itself a determinate procedure, then each translatable source program represents a determinate procedure. On the other hand, if the design of the interpreter does not guarantee determinacy of abstract programs, determinacy of source programs, when desired, must be ensured by the translator.

In the base language, it is necessary to provide for computations that are inherently nondeterminate, such as the example of a process awaiting the first response from either of two terminals. We want to include in the base language primitive features for representing essential forms of nondeterminacy. In principle, we wish to guarantee that any (base language) procedure that does not use these features will be determinate in its operation. Furthermore, use of base language primitives for the construction of nondeterminate procedures is intended to be such that the choice among alternative outcomes always originates from the source intended by the program author, and never from timing relationships unrelated to his computation.

Our current thoughts regarding representation of base language procedures so as to guarantee determinacy are based on <u>data flow</u> representations for programs in which each operation is activated by the arrival of its operands, and each result is transmitted, as soon as it is ready, to those operations requiring its use. Rodriguez [22] has formulated a data flow model that applies to programs involving assignment, conditional, and iteration statements, and data represented by simple variables. Procedures represented by Rodriguez program graphs are naturally parallel and the rules for their execution guarantee determinacy. In [3], Dennis has given a similar program graph model for procedures that transform data structures, but do not involve

conditional or iteration steps. Determinacy is guaranteed for these program graphs if they satisfy a readily testable condition.

We hope to be successful in combining and extending these two models to obtain a satisfactory data flow model for all determinate procedures. If this objective can be achieved, we expect to use program graphs as the nucleus of the base language. On the basis of improved understanding of parallel programs obtained by recent research on program schemes by Karp and Miller [11], Paterson [21], Slutz [23], and Keller [12], we are optimistic about finding an inherently determinate scheme for representing the concurrency present in most algorithms.

## CONCLUSION

This article has been an introduction to the goals, philosophy and methods of our current work on the design of a base language. The material presented is an "instantaneous description" of an activity that still has far to go — many issues need to be satisfactorily resolved before we will be pleased with our effort. In addition to the representation of concurrency, the base language must encompass certain concepts and capabilities beyond those normally provided in contemporary source languages. Four aspects of this kind are: 1. Generation and transformation of information structures that share component structures; 2. Concurrent processes that, in pairs, have producer-consumer relationships; 3. Programming systems that are able to generate base language programs and monitor their execution; and 4. Provision for controlling and sharing access to procedures and data structures among users of a computer system. We are continuing investigation of how these capabilities should be incorporated in the base language. Some ideas on intercommunicating processes have been reported briefly [5]. Some thoughts on program monitoring and controlled sharing of information are given by Dennis and Van Horn [6], and by Vanderbilt [25].

## REFERENCES


1. D. M. Berry, Introduction to Oregano. Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices Vol. 6, No. 2, ACM, February 1971, pp 171-190.

2. D. M. Berry, Block structure: retention or deletion? Proceedings of Third Annual ACM Symposium on Theory of Computing, May, 1971, pp 86-100.

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

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

5. J. B. Dennis, Coroutines and parallel computation. Princeton Conference on Information Sciences and Systems, Princeton, New Jersey, March 1971.

6. J. B. Dennis and E. C. Van Horn, Programming semantics for multiprogrammed computations. Comm. of the ACM, Vol. 9, No. 3 (March 1966), pp 143-155.

7. A. P. Ershov, Private communication.

8. J. L. Gertz, Hierarchical Associative Memories for Parallel Computation. Report MAC-TR-69, Project MAC, M.I.T., Cambridge, Mass., June 1970.

9. J. K. Iliffe, Basic Machine Principles. American Elsevier, New York 1968.

10. 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.

11. R. M. Karp and R. E. Miller, Parallel program schemata. J. of Computer and System Sciences, Vol. 3, No. 2 (May 1969), pp 147-195

12. R. M. Keller, On maximally parallel schemata. IEEE Conference Record. Eleventh Annual Symposium on Switching and Automata Theory, October 1970, pp 32-50.

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

14. 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.

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

16. P. Lucas, Two Constructive Realizations of the Block Concept and Their Equivalence. Technical Report TR 25.085, IBM Laboratory, Vienna, June 1968.

17. P. Lucas, P. Lauer and H. Stigleitner, Method and Notation for the Formal Definition of Programming Languages. Technical Report TR 25.087, IBM Laboratory, Vienna, June 1968.

18. 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.

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

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

21. M. S. Paterson, Program schemata. Machine Intelligence, Vol. 3, American Elsevier, New York 1968, pp 19-31.

22. J. E. Rodriguez, A Graph Model for Parallel Computations. Report MAC-TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969.

23. D. R. Slutz, The Flow Graph Schemata Model of Parallel Computation. Report MAC-TR-53, Project MAC, M.I.T., Cambridge, Mass., September 1968.

24. T. B. Steel, Jr., UNCOL: The myth and the fact. Annual Review in Automatic Programming, Vol. 2, Pergamon Press 1961, pp 325-344.

25. D. H. Vanderbilt, Controlled Information Sharing in a Computer Utility. Report MAC-TR-67, Project MAC, M.I.T., Cambridge, Mass., October 1969.

26. J. Weizenbaum, The Funarg Problem Explained. Unpublished memorandum, March 1968.

27. E. C. Van Horn, Computer Design for Asynchronously Reproducible Multiprocessing, Report MAC-TR-34, Project MAC, M.I.T., Cambridge, Mass., November 1966.