The diagram represents a sequence of nested calls. Normally USER would have access to F's variables (via inherited scope), but that may "hide" some variables of interest in the incarnation of Q. Separating its context from F and reattaching it at Q (as indicated by the dashed line) overcomes this difficulty. Of course, USER must also be able to revert back to a previous context or nest them further; our remarks in chapter 2 on control symmetry apply mutatis mutandis to context control.

Since USER is really just a procedure like any other, this implies that any program should be able to mask off part of the control/context stack for scope purposes, as a separate consideration from the control information. Interestingly enough, this is exactly the mechanism used in many Algol systems for evaluating "by name" expressions in the context in which they were originally passed as actual parameters. The only difference is that the control is not available as an accessible structure in Algol whereas it must be in an IPS in order to provide these features, and allow the user access to his programming environment in as complete a manner as possible. Indeed, the ability to change contexts in this way is little more than an implementation of the by-name concept without a priori knowledge (in general) that such accessing will be required.

the former to locate a point in the context hierarchy, and then the notation F.A to access the A in the incarnation of F which is the closest to the specified control in a backward direction.

We will use the notation ↓↓.n to denote the control which is n levels back in the stack counting the current level as level 0. Thus, ↓↓.1.F.A would mean the following: beginning one level of control and context back, search the stack for an incarnation of the object F, either as part of the control or as a function named within the scope of some active context; then use the A for that particular function in that context (of course, this implies that A may have to be searched for by the SCOPE algorithm in the same way as F was found).

To thus completely specify a value each time it is to be used or inspected involves large overhead for the user in terms of the number of characters to be typed. It would be advantageous to define a default scope for some period of time during debugging so that one could simply type F.A instead of ↓↓.1.F.A each time that particular A was to be inspected or changed. Such a context level default could be associated with a particular incarnation of the USER procedure (USER was introduced in chapter 2), disappearing as the default when that level of USER did. Pictorially, what we are suggesting is that control be separable from context so that USER (and any program, in fact) may attach itself to context levels other than the one normally obtained in parallel with a subroutine call, thus bypassing part of the context which may be acting as a wall to hide the one desired: e.g.,

non-predictable: thus, in the above example, the user may not know that the A in the outer block should be inspected during the life of the inner block until some error occurs during testing of the program. Two main possibilities can be considered as candidates for a solution to this problem:

(1) control could be placed into the state of the outer block by "exiting" from the inner;

(2) a naming convention for the outer A could be established, allowing access to it without having to leave the inner block.

"Killing" the execution of the inner block as suggested by (1) seems too drastic just to inspect variables in the outer block. And when the value to be looked at is more than one level removed, even more execution must be deleted. However, (1) implies a rather neat method for accession of items in the call and block hierarchy as represented in the stack: namely, to establish a pointer into the stack at some control and context level, then using the SCOPE algorithm from there to find the A accessible at that level. This really amounts to being able to control the context as separate from the program control — and we will shortly do so.

The second suggestion is applicable in a static as well as dynamic sense. One would like to be able to access the variables in a given procedure, for instance, and a simple notation such as F.A meaning the A in F is ample for the static situation. However, when F is recursive, one needs to be able to talk about a specific incarnation of F, and then F.A becomes ambiguous.

An alternative is simply to combine the notions (1) and (2), using

which allows a process,Q,say to associate certain of its variables with their named counterparts in another process P for instance. P is probably not a simple name since it must at least contain some qualification as to the owner of the process, whether the user himself, the system, or some other user. Examples of such naming conventions have been implemented in a number of time-sharing systems such as MULTICS [Sa 66] and TSS/360 [IBM 69].

4A3 Circumventing Scope

One of the prime features of an IPS is the ability to inspect variables during program execution as an aid to program development and debugging. In accord with the philosophy of allowing the statements of an IPS language to be used in both direct and delayed (or stored) mode, inspecting the value of a variable is usually accomplished by the normal output statements of the language.

Since, in general, scope rules such as Algol's are intended to insulate variables and program sections from one another, it is common that not all variables are accessible at any given point in an Algol program. This may be so because of scope limitations or because of naming conflicts. An example of this is seen in the following block structure:

When control is within the inner block, the A of the outer block is inaccessible. However, in an IPS it is crucial that the user be able to circumvent such inaccessibility. Errors are generally considered

bypassed by the scope algorithm when encountered during the search of a name table.

4A2B Global Variables

Variables can be declared to have global scope, in which case, any use of a variable in a block in which it is declared global will access only the GNT to find it. Thus, all places in which A, for instance, is declared global will be using the same A. There will be an entry in the LNT of each procedure in which A is declared global for the block in which it was declared; but that entry is only used to relate that declaration to the GNT by means of a pointer to the GNT entry for A. Since there is then an entry in the LNT for A, the SCOPE algorithm can be expected to use it if encountered during a scope search. In order to prevent possible damage to global variables due to inherited scope, however, (e.g., by a called subroutine's unwittingly changing it) one can declare A as secret global, meaning that it should be bypasse in any scope search. Indeed, the attribute secret attached to a variable is more a protection mechanism than a scope and should be allowed as an extra attribute in declarations.

It is intended that global variables apply at the process level. Since a user may have more than one process or may be communicating with processes belonging to other users, a way is also needed to share "names" between such independent processes.

4A2C External Scope

Interprocess naming is provided by the scope declaration external(P)

are the same from use to use), then the TFI can produce stable code which will not be constantly invalidated. Moreover, after the initial use of a variable, separate functions using it will have a name table entry for it, created by the above method, thus decreasing the amount of overhead required to match a use in a function to a particular incarnation of a variable of that name. The actions necessary when a declaration is deleted are dealt with in section 4C3A: Semantic Changes: Their Detection and Range of Effect.

4A2 Other Scope Possibilities

4A2A Secret Variables

The mechanism of inherited scope allows a program to freely access its caller's variables, so it is necessary to provide a means whereby a calling program can insulate itself from the procedures which it calls. Otherwise, a "correct" program can be made to act incorrectly by procedures which are not lexically local to it but which it uses, and in particular, system routines written in the language could be susceptible to this kind of inadvertent damage.

A scope declaration such as secret can accomplish this: a secret variable obeys normal scope conventions except that it is not known by any lexically contained blocks or procedures, nor by any called procedures. To all other blocks, it is as if there were no declaration of secret variables in the block in which they are declared -- only their owning block knows of their existence. They are simply

nowhere in the call hierarchy to be bound to the global value. In any case, any existing lexical scope is honored before inherited scope whenever an identifier is undeclared in an executing program.

A large part of the motivation behind the TFI is that an IPS should attempt to bind semantics as early as possible even though the user has not declared such bindings. That is, (1) it is anticipated that statements and variables are changed much less frequently than they are used in normal execution in an IPS, and (2) there are de facto bindings of variables and programs which ought to be exploited, even though not explicitly given (See also section 2H Interpretation and Compilation).

In the case of inherited variables, it is not clear that an occurrence of such an identifier will be bound to the same variable from use to use of its containing block. But it is probable that other attributes of that variable will remain fixed over time (such as its type or number of dimensions, for instance), and it is that semantic constancy which should be exploited.

A simple modification to the above SCOPE algorithm which allows us to do just that for the case of inherited variables is the following: when an identifier is bound to a value and it represents a use of inherited scope, an entry for that variable is added to the function in which the identifier occurs. Its semantics then can include its type, etc., on that use of the function or block as well as the fact that it is an inherited variable. Then, on subsequent executions of that program, use of the variable will activate the SCOPE algorithm because it is marked as "inherited." However, if the other attributes of the variable, (from whom and when it is inherited

Since the outermost incarnation of USER has the GNT as its local name table, the logic of the above algorithm will allow an identifier which is declared

Q.1B means that statement 1B was the active line in procedure Q when P was called. Also, LNT(Q) is a pointer to the local name table for Q and is part of the state information which must be saved on a call. The call information saved for P calling F also marks that call as one to a LOCAL procedure.

We can now describe the algorithm for determining to which value the identifier D in F is to be bound in the above circumstances.

The SCOPE Algorithm

(1) Scan the current local name table (LNT(F) when the search starts) for an occurrence of the identifier D; if one is found, go to step (5). If the table scanned was the GNT and on occurrence was found, go to step (4).

(2) If no occurrence of the identifier is found at this level, then use the name table which is the parent of the one just searched and go to step (1) if the new table is not the GNT; if it is the GNT, proceed to step (3).

(3) Search back one level in the call stack; if that entry indicates a local call (as in the case of P calling F in the example), then repeat step (3). If the next level represents a non-local call (as in the case of Q calling P, or USER calling Q), then make the current local name (current in terms of this algorithm) be the one referenced by that entry. Go to step (1).

(4) Create an entry for D in the GNT and go to step (5), pretending that D was found.

(5) Bind the identifier to the value given by the "found" table entry.

There is an extra depth now to lexical scope since each procedure or function, being a namable entity must have its own name table; hence, we are faced with the problem also of maintaining lexical scope across name tables as well as within them. The algorithm for maintaining correct lexical and inherited scope is given in the following section.

4A1D Maintaining Lexical and Inherited Scope

Given a reference to some non-declared variable in a procedure, we would like to ascertain which incarnation of that variable is to be accessed. Which one it is will be determined using both the static name table hierarchy (for lexical scope - which must take precedence over inherited scope) and the call hierarchy, which can be viewed as implemented via a stack mechanism.

An example will best illustrate these points, and we will use the example from figure 4A1C-1 given previously. Assume that the user executes the direct statement

```
Q;
```

meaning "call the procedure Q". Then Q calls P, and P calls F in the example. When F executes statement

```
1B2  C ← D;
```

we need to determine which variable the identifier D represents.

When line 1B2 is executed, the call hierarchy is as given in the following stack description:

```
P.1D  LNT(P) local
Q.1B  LNT(Q)
USER  GNT
```

a declaration for the variable, one may be assumed to exist at the ever-present outermost (global) context. A more appropriate nomenclature is that the scope of such variables is "inherited" from their callers, and hereafter we will call this inherited scope.

Since variables and program names must be known in some outermost context, there must be at least one name table, called the Global Name Table (GNT), in a given process. Programs or procedures can thus be named, and each procedure has a name table as part of its description, which name table is structured as outlined previously and called a Local Name Table (LNT). An example of the naming structures for the following skeletal SLICE program is given in figure 4A1C-1 with it:

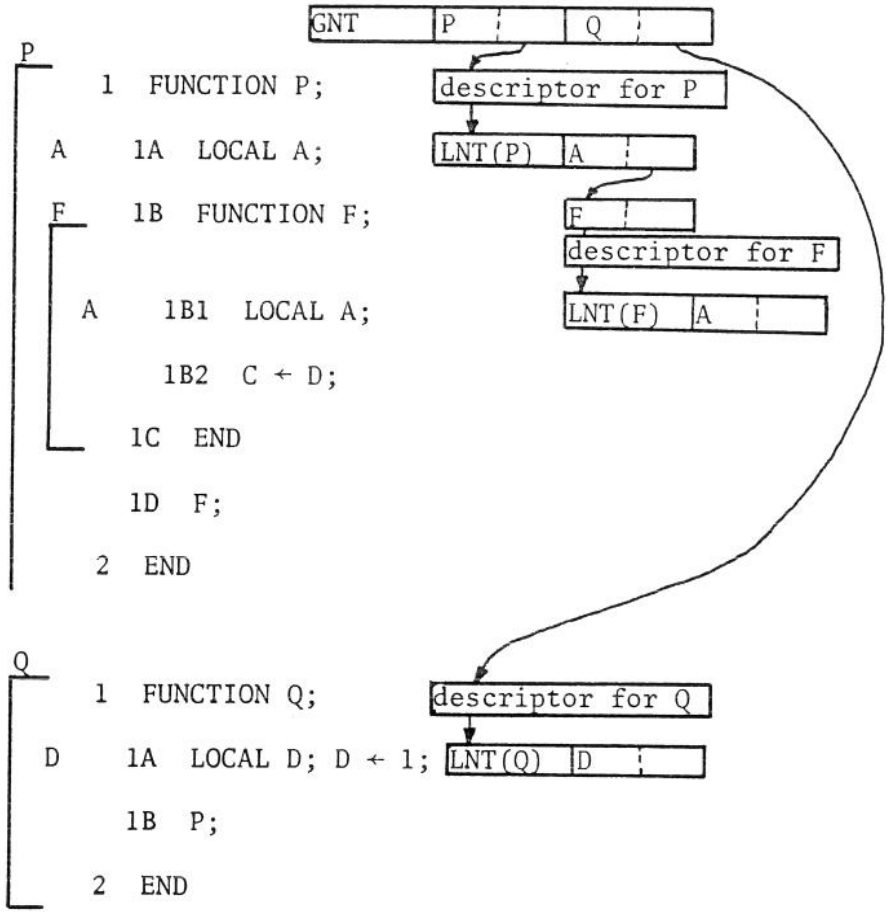

Figure 4A1C-1: Sample of Name Table Hierarchy

compiler system.

4A1B Maintenance of Lexical Scope

Lexical scope is well represented in the proposed name-oriented symbol tables, which have such scope described by a list appended to the name table entry for each name. The only relevant consideration is how and when that naming structure is altered. An easy and obvious solution is suggested by the nature of the declarations: namely, to effect those changes which are lexical in nature when a declaration is entered and parsed by the system. Exactly what those actions are, aside from altering the list attached to each name is detailed in section 4C3. Such semantic changes as are caused by altering, inserting or deleting declarations in a program can be termed global semantic changes since they may potentially affect many parts of a program. By contrast, local changes are those which can affect at most a single isolated line in a program: changing one line of text is the prototypical situation of this type of alteration.

4A1C Maintaining Dynamic Scope in a TFI

Earlier, a form of scope called **"dynamic"** was referred to. In this scheme, the scope of a variable is determined dynamically on each use of the program in which it resides (assuming it is not declared in that program, but only used). The incarnation of the variable used is that which is local to the program calling the one using (but not declaring) the variable. This is an inductive definition, since the scope of a variable in the calling program may be similarly determined, and if no program in the call hierarchy at that time contains

actual execution of the declaration, rather than entry into the block in which it appeared. Hence, there are potentially three times at which the declaration of a variable may have effect and which are suggested by the above. In order of earliest binding time they are:

(1) Activation-time: prior to the first entry into the block containing the declaration: this corresponds to compile or pre-run-time in a batch system. This time may, however, be anywhere between the time of entering the declaration statement (e.g., by typing it at the terminal) to just prior to the first execution of the block in which the declaration resides. In accord with a view to flexibility, the late end of this time-span seems the better of the two.

(2) Block-entry-time: upon entry to the block containing the declaration. This corresponds to a scheme used by Algol except that all the semantics is assumed to become effective at that time.

(3) Execution-time: each time a declaration is executed. This is the scheme used in LC^2; its benefit to the user is dubious except when a declaration is forgotten and one wishes to add it to an already active block before continuing the execution of that block.

Since (3) has been found to be of value in a limited way, it should be combined with (1) and (2) by generalizing these latter to "execute" any declaration of a variable entered into a block which is already active (including the possibility of multiple uses of a block due to recursion). Then we will call the declaration of type (1) lexical and those of type (2) dynamic, corresponding to compile and run-time in a

has been found to be a useful discipline in the use of the LC^2 system. The mechanism which makes it a viable and desirable method of program production is called "dynamic scope of variables" in [MPV 68]. Simply stated it says that any use of a variable in a module containing no declaration for it uses that incarnation of the variable local to the program which called it. If no module has declared a variable, then a global incarnation of it is used. In this way a module which has a number of different callers may, in fact, deal with different non-local variables from call to call, just as if a copy of it existed at each of the calling points and the normal Algol scope rules obtained. The ability to make copies of such modules and to place them at the calling points allows more efficiency of execution and a more lexically organized structure as the form and content of the program are finalized. Indeed, in many cases there is only one calling point and the dynamic scope facility is used simply as a device for segmenting the program into functionally separate units during design and development -- a most useful tool.

4A1A Binding Times of Declarations

In a compiler system such as Algol, the effect of a declaration such as real A may have an effect at both compile and run-time. At compile-time it causes information about the identifier A (including its scope) to be encoded in a symbol table. At run-time it causes the allocation of a place for A whenever the block in which it is declared is entered. In the interactive system LC^2 these two times were abandoned in favor of the

more suited to our needs. This statement will be justified throughout the remainder of this chapter.

In the next sections, we will investigate one of the prime usages of these naming structures, that of maintaining correct scope of variable names in the dynamic environment of an IPS. For the purpose of what follows we will assume that a global variable, CONTEXT, exists whose value is always the identification number of the program statement being executed, such as 1, 1A, 2B1, etc.

Now, there are a number of levels at which naming can occur. That just described can be thought of as being at the procedure, program, or module level, by which we mean relatively independent programs with no common lexical structure, communicating primarily by a calling and parameter passing mechanism such as subroutine or coroutine control.

Such modules may "share" variables in the following two ways: the first is the standard Algol concept of lexical scope which is handled within modules and by mechanisms such as declared global variables known to more than one module. The second method is more representative of the mode of program creation and debugging found in interactive systems such as LC^2 and APL. In an IPS it is common to develop programs as a number of fairly small and simple procedures which, although lexically separate, are expected to communicate large numbers of variables as if there were a lexical relationship between them. Often these modules will remain separated throughout the life of small or exploratory programs; in larger systems of programs, however, they may ultimately be merged and become part of an overall lexical and logical structure of the program set. This development sequence

The tree, complete with symbol tables, has the following form:

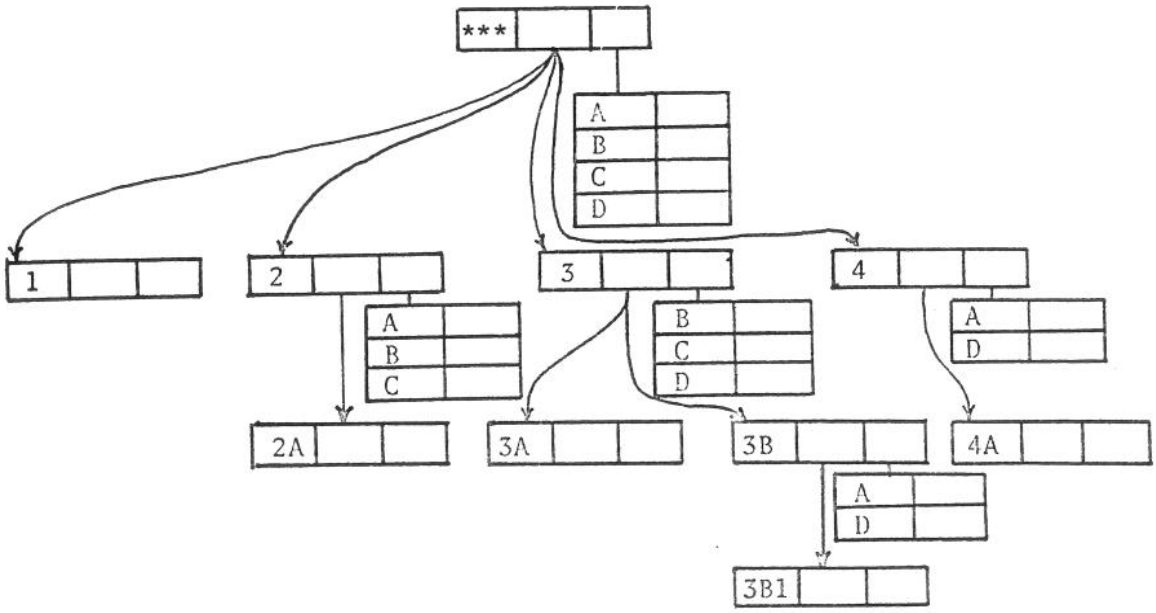

Figure 4A1-3: Scope-oriented Symbol Tables

The representation of this lexical scope which is oriented toward unique names rather than scope is given below:

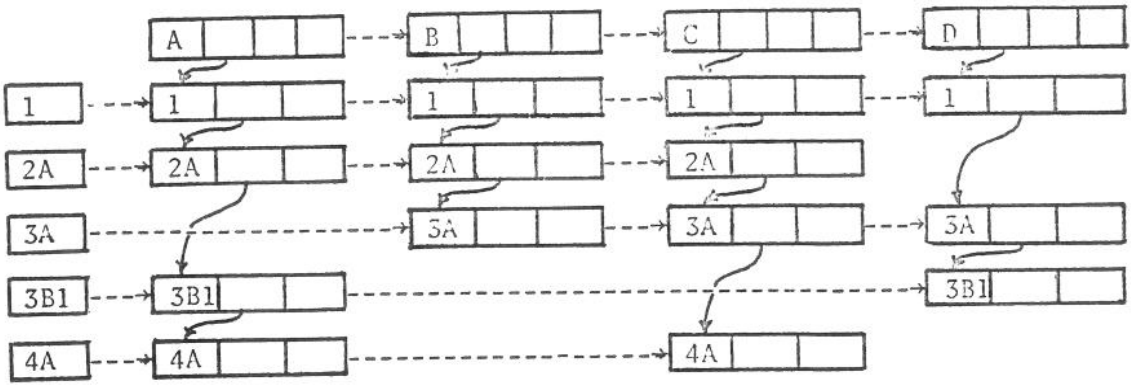

Figure 4A1-4: Name-oriented Symbol Table (Distributed Scope)

The similarity of the two structures at the elemental level is very striking, especially if we impose a threading of all occurrences of the same names in the scope-oriented representation. However, of the two, the latter (namely scope distributed over uniques names) seems

We will illustrate these two different structures, namely scope-oriented and name-oriented symbol tables by an example of a skeletal program containing only declarations. The program is the following:

```
1  real A,B,C,D;                          ]A
                                          ]B
2                                         ]C
                                          ]D
  2A  real A,B,C;                  ]A
                                   ]B
3                                  ]C
                                   ]B
  3A  real B,C,D;                  ]C
                                   ]D
  3B
                                   ]A
    3B1  real A,D;                 ]D
4
                                   ]A
  4A  real A,C;                    ]C
```

Figure 4A1-1: keletal program with lexical scope of names

The scope of identifiers is indicated by the bracketing at the right of the diagram. Now if we use the first method, there will be individual symbol tables associated with nodes delimiting the various blocks above. The form of such a tree is

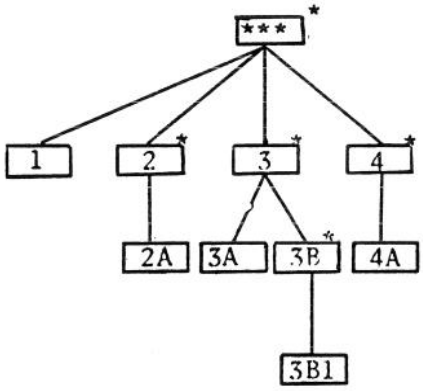

Figure 4 A1-2:

The nodes with *'s attached are those which would have symbol tables attached to them.

made which are of a global nature. A good example of this is the altering of the scope of a variable by insertion or deletion of a declaration in the program. Moreover, the parse tree is arranged for execution, and not necessarily to represent lexical structure: thus, the distribution of symbol table according to lexical scope must be "artificially" mapped onto that structure. Such an organization, therefore, is economical for execution, but clumsy with respect to global program alterations.

Another organization, suggested by the above, is name rather than scope oriented and uses but one table per program. Then, for each name there exists a list corresponding to the lexical structure of those program sections having a declaration of that name. This organization makes non-local information about a given name readily available, at the expense of a somewhat higher price for information about all the names in a given block, for instance.

However, when interpretation takes place, control information must be used to select a specific entry in the list attached to the symbol table entry for an identifier. There is an easy and obvious variant of this scheme which facilitates this kind of accessing: simply thread together the nodes in the symbol lists which correspond to the variables for a given block and place pointers to those threads in the parse tree nodes for the blocks, thus gaining some of the advantages of the scope oriented tables first described.

4A1 Naming Structures for an IPS

There are two major components of an IPS which can be expected to use the naming structures for variables: the translator, and the interpreter or execution system. These components access the table in different ways and with different demand rates, and its organization should reflect those differences. Associating an identifier in a line of program text with a specific symbol table entry (and therefore also a specific scope) is of prime concern to the translator. This depends on the logical position of the line of text in the program and the lexical structure of the program surrounding it. The interpreter, on the other hand, requires only local knowledge of the names in a block, and the semantic attributes (especially the location of the value for that variable) for each such variable. And since the program execution in the parse tree environment of the previous chapter views control as residing at the nodes of the tree, it is reasonable to associate all the names local to a block with the node defining that block.

This latter organization suggests a set of symbol tables, one per block, in the Algol sense. Thus, a given name might appear in more than one symbol (or name) table, and the variables local to a block are exactly those represented in that block's name table. This localization of names requires that the parser be cognizant of the block structure within which a statement appears if it is to bind occurrences of an identifier to the correct entry in the correct table. Most Algol compilers do indeed organize their symbol table this way. The distribution of names over the lexical program structure should be expected to cause problems when changes are

emphasis on names in programming. Such a set of tables is called a symbol table and is used in compiler systems to associate information about a named entity with that name. Normally this is semantic information, possibly including a run-time address for that variable. The information kept for the same purpose by an IPS is similar except that the information is used and changes in a much more dynamic way, partly because of the interlacing of translation and execution in these systems. Nevertheless, the primary means of accessing symbol tables is by name association, normally involving table searching. And here too, early binding can save overhead: for, if the symbol table entry for a variable occupied a fixed memory location, determined at the first "appearance" that name, then any subsequent references to that entry could be made directly without further table lookup for as long as that name existed anywhere in the system. This is what is done in the LC^2 and APL systems to speed the interpretation of variables.

In the following sections, alternative representations for IPS symbol tables are discussed. Then the enforcement of certain regimes of scope of names is outlined as a series of algorithms using those symbol table structures. Lastly, we will give a means of circumventing scope rules using the symbol table structures in the next sections.

that environment both flexibly and efficiently. Flexibility is desirable to minimize the user's overhead for a given task, and efficiency is necessary to minimize the computer's overhead in the same task. Indeed, the entire previous chapter on interpretation and compilation was an initial thrust in this area: the maintaining of flexibility without loss of efficiency.

In this chapter, we will give much more detailed methods for treating data and programs in this way. Also, the constraints placed on the environments in which those methods are practicable will be made more realistic and more in line with what is known about currently available interactive systems and their immediate extensions.

4A Names in an IPS

Naming data has always been of central concern in programming languages, from the simple naming of Fortran variables to the more elaborate block structure of Algol and PL/1 and the more recent dynamic scope of names as used in LC^2. And no matter what naming scheme one chooses for an IPS, one must also provide some means by which each named entity is accessible, because the user may intercede at any level of control and desire to inspect or change variables which are normally not in the program environment at that point (because of scope restrictions, for instance). Thus, whatever rules of lexically or dynamically determined scope of names are chosen as appropriate to an IPS, we are required to find a means of circumventing them within the IPS.

The importance of the data structure(s) used to represent the correspondence between names and values is a reflection of the

both of their systems, VERS and AMBIT/G respectively, have some interesting characteristics as a result of that logical separation.

Another implication of the above tenets is that programs in an interactive environment should be able to respond to changes in

4 Differential Data Structures and an Extended TFI

In the previous chapter, a method was developed which allowed programs to traverse the spectrum defined by interpretation and compilation as extreme points. In such a system, the programs act in a differential manner to changes in their environment. One major part of that environment is the data structures available to programs, and the representations of such data in the computer. Values whose representations can vary along a flexibility/efficiency spectrum will be called differential data structures. In general, they should have some of the same properties as differential programs (such as those in a TFI environment). Two of those properties, heretofore unstated, are the following:

(1) the ability to take advantage of any "effective" bindings, even if unstated by the user or his programs;

(2) the ability to allow changes in binding (such as the type of a variable changing) without affecting data or programs not connected with or dependent on the semantically object -- complete recompilation of a program because of changes to one variable would be in violation of this principle.

These two tenets are related and have some interesting implications. For instance, (2) implies that the way a data structure is implemented is an issue which is separable from its semantic content (i.e., its "meaning", how it is accessed, or how it may be changed). Earley [Ea 69] and Christensen [Chr 68] have both had some success in treating the accessing of data as separable from its representation, and

```
Boolean procedure INITIALIZE(STATEMENT);
begin
  if ~COMPLETE ∨~VALID then DELETE_CODE (STATEMENT);
  THIS_STATEMENT ← STATEMENT;
  if THERE_IS_CODE_FOR (STATEMENT)
    then begin
           COMPLETE ← true;
           INITIALIZE ← true;  comment : INITIALIZE = true means that
                                 the statement has been executed and
                                 the interpretive routine need do
                                 nothing;
           EXECUTE_CODE_FOR (STATEMENT);
         end
    else begin
           COMPLETE ← false;VALID ← true;
           GET_A_BUFFER (STATEMENT);
           SET_INCOMPLETE (STATEMENT);
           INITIALIZE ← false;
         end;
end;

procedure CODE (start,limit,params);
begin
  MOVE_CODE_TO_BUFFER (start,limit,buffer_address);
  SUBSTITUTE (params,buffer_address); comment : substitute parameters
                                                into the copied code;
  MOVE_CODE_TO_BUFFER(return_code,return_code_size,buffer_address+limit);
  TRANSFER_CONTROL(buffer_address); comment: execute the prepared code;
  buffer_address ← buffer_address + limit;
end;

procedure COMPLETE_CODE (STATEMENT);
begin
  SET_COMPLETE (STATEMENT);
  COMPLETE ← true;
end; comment:
```

Appendix 3A

The following syntax and semantics have been used in describing the ADD, MULT, and VALC routines on pages 3D-1 and 3D-2:

```
PERFORM = "PERFORM" ( .ID / SUBNODE ) :PERFORM[1];
  SUBNODE = "*" INTEGER .$( "*" INTEGER ) :SUBNODE[1+$];
```

PERFORM calls an interpretive routine; if the PERFORM .ID form is used, it simply calls the routine .ID, treating it as a nodal routine which has no code. If a SUBNODE is called, then PERFORM also does the INITIALIZE action described in Figure 3C1-1; i.e., if the subnode has valid code, PERFORM simply executes that code, and if not, it deletes any code which the node may have and gets a new code buffer for it before actually invoking the x-routine specified by that node.

In order to include these definitions in the SLICE grammar, the SIMPST rule must be amended to read

```
SIMPST = ASSIGN / EXP / IFST / COMPOUND / GOST / PERFORM;
```

We would also like to add syntax for line numbering:

```
LINENUM = $( INTEGER 0$1( APLHACHAR ) );
  ALPHACHAR = "A" / "B" / ... / "Z";
```

In order to place LINENUMs into the grammar, however, it would be necessary to optionally allow it before each non-terminal which can be a statement segment. If we altered the grammar so that a "*" precedes all rules which may be a segment, we could then simply let the parser mechanism worry about LINENUMs preceding segments of statements.

when it attempts to return, control will return to the point left by the first use of the VALC(αB) node. VALC will then attempt to create and execute the "ctype(real)" code as above. Finally, it can be revealed why the complete-flag is needed: using the fact that VALC(αB) already has valid and complete code, CODE can decide to execute the "ctype(real)" but will then throw it away. The same happens at the MULT and ADD nodes, thereby achieving the correct execution without erroneously attempting to concatenate extra code to already complete code sequences.