4B4 Some Representations for Variables in an IPS

Now that naming structures, attributes for variables, and the effect of changing declarations have been described, we can discuss how best to use information about variables in order to store and use them as efficiently as possible. In the following it is assumed that terminal nodes in the parse tree which access a variable have the symbol table address for the identifier as part of their nodal information. The information about the semantics of a particular variable can be found using the scope algorithm developed earlier to find the particular entry corresponding to a given occurrence of a variable.

A set of possible representations and accessing methods for variables will be developed; which implementation is used will depend on what information is available about a particular variable and when that information becomes known. In general, our aim is to use, to the best possible advantage, attribute information for a variable at the time it is declared(whether that is activation-time, block-entry time, or execution-time as described earlier) in order to decrease the cost of interpretation. The TFI may produce compiled code, but without further information about the variables and a set of possible accessing methods which can take advantage of the known attribute values, it will meet with a time barrier of the same sort as full interpretation.

As each method is discussed, we will also mention what must be declared (such as scope or type) about the variable in order for that method to work. Usually, the more attributes of a variable which are declared, the more efficiently can that variable be treated

(rather than just its presence or absence), any statements not using that attribute will not require changing by the TFI. In the case of X above, therefore, any statements which accessed but did not store into X would not require changing if the store monitoring were cancelled

to reasonably efficient but inflexible. Declaration changes in an active context can affect these representations, and where this situation obtains, we will present methods for handling it.

The ability to specify that an object is to be monitored in some way is a form of declaration. By "monitor" we mean that the object (value, variable, or program) is to be watched so that whenever it is used in a certain way, that will cause the activation of some procedure. The use which will trigger such an "interrupt" could be further constrained to be a store into the object, a change in the object, or accessing (or executing) it, and combinations of these. Of course, turning monitoring of an object on and off is a "declaration" which must be executed and not limited to taking effect only on block entry.

If a variable's being monitored (or not) is considered an attribute, then changing that attribute will invalidate any code which depends on that variable (or object, in general). Thereafter, when code is created which uses it, the code can include the action of invoking a procedure associated with the monitoring of that object (which procedure was specified when the monitoring declaration was made). And, if the monitoring is selective, only code which uses that object in such a way as to satisfy the selected monitoring need include the extra code to invoke its monitoring procedure. Hence, if only stores are being monitored for some variable X, accessing X would include no code to monitor it since accessing causes no stores into X.

When monitoring is discontinued or cancelled, the code invalidation scheme of the TFI is used to cause monitoring code to be replaced. Moreover, if the monitoring attribute specifies the type of monitoring

which could occur without declarations for the variable being altered. Typeless variables have this property. But such changes really say that the type of the value of that particular incarnation of X is different and may have no connection with the types of values of previous or future instances of that same variable.

If declarations are deleted, inserted, or altered in an inactive context, the TFI will still work because any code for the variables affected by those declarations will be marked as invalid, forcing reinterpretation of those statements when they are next executed. If, however, a declaration is changed which affects one or more incarnations of an active program (and therefore potentially one or more incarnations of some variable, D, in the declaration), the problem is not so simple. Consider the case in which only one incarnation of D exists: then that declaration must take effect immediately, in that context, in order to maintain Visual Fidelity of the program and its execution. This, it will be remembered, was the primary reason for the introduction of the third "time" (execution time) at which declarations could take effect. That is, we are stating that a declaration which is altered in an active context must take effect immediately. Moreover, if there is more than one active context, the VF principle implies that all the incarnations of D must be affected — declarations changed in an active context must be retroactive to suspended contexts in the call hierarchy. Since this may imply the same set of possible cases as in section 4B3B above, the same comments apply to each of those active contexts.

In the next section we intend to present several representations for variables, ranging from very flexible and inefficient

Case (2)

This case is probably the easiest to solve, since the new X should be used, there being no conflict between the past and present, at least at the single statement level.

Case(3)

This case presents a conflict between past and present. The Visual Fidelity Principle indicates that either the old X should be used in both places, or the new in both places, but not both mixed, since there is nothing in the text of the program to indicate that the two occurrences of X in the statement represent different variables. So, in effect, since the past is involved and the old value of X has already been used, this case degenerates to case (1). This case must be brought to the user's attention, however, because of the VF principle. In both cases (1) and (3), another option is available: tell the user of the problem and hold the change in abeyance until the end of that statement using the old X if that is desired, or make the change and disallow completion of the statement.

We have assumed here that the above cases are detectable by the system, as will be seen later in this chapter when the TFI is generalized to handle such dynamic changes. It is introduced here because of its relevancy to the following section on the representation of variables in an IPS under the execution/alteration interleaving constraint.

4B3C Declaration Alterations

The above paragraphs dealt with changes to variables

hard to provide, especially if variables can be shared between programs, for then the following situation can arise:

```
1  PROCEDURE MAIN;
2  DECLARE Y ← 1;
3  FUNCTION F;
  3A  Y ← Y + 1;
  3B  RETURN Y;
4  X ← Z + Y * F;
```

Assume that the user intervenes during execution of line 4 with the "*" about to be executed after F has returned its value. Then F has permanently changed Y, and redoing statement 4 because Z is changed by the user, for instance, cannot reverse the side effect of F in changing Y under any reasonable algorithm for execution. One would have to be prepared to execute the program backwards (literally) by having saved a complete history, none of which can be discarded until an entire program is finished, which decision could only be made by the user. It must be mentioned here that Bałzer [Bal 69] has developed a system for PL/I programs called EXDAMS which does allow one to "pretend" that a program is being run backwards. But it is not interleaved with the execution of the program and involves essentially playing a "history tape" of the program in reverse.

The main conclusion then is that using the old X is the only reasonable action in case (1) although one ought to be able to expect the IPS to inform him of such an event in order that the user may decide what to do.

(everything to the left of the pointer) and future (everything from the control pointer to the right); there is no present because we are speaking of the control pointer at a point in time between discrete actions, and it is the actions which concern us, not the time between them.

	Past	Future
		↑control pointer
Case 1	X	
Case 2		X
Case 3	X	X

Figure 4B3B-1: Schematic of Statement Execution

The three cases are

(1) X has been used in the past and will not be used in the future of this statement;

(2) X has not been used in the past and will be used in the future of this statement; and

(3) X has been used in the past and will also be used in the future in this statement.

Case (1)

In case (1), without some overhead it is difficult to use the new X instead of the old automatically, although the JOSS system [Ba 66] actually does just that by not permanently changing the value or type of a variable until a statement which changes it is successfully completed. Thus, changes to variables are not effective until a statement is complete, making it valid to simply redo the statement if the new X is desired, with assurance that the environment has not been changed in spite of the partial execution of the statement. This assurance, in more complicated languages is very

the change can be disallowed, or made so that if control ever attempts to return to the deleted statement, an interrupt or error action will occur. For statements having a scope of activity, the problem is more difficult but is solvable again either by disallowing the change or disallowing control to attempt blindly to return to that point.

Even though checking for a deleted statement on each return could cause a great deal of overhead, altering an active statement must be allowed in at least the following case. When an error occurs during execution of a stored statement,and the USER function is invoked, he must be able to correct the statement in error even though control clearly cannot be allowed to return there since the statement to which control points will have disappeared. But since the USER function is no different from any other procedure, the facility of checking this case on each return really is necessary.

Finally, since there is only one copy of the program in existence at one time, even during recursive use of a function, adhering to the VF Principle is almost trivial.

4B3B Type and Structure Changes to Variables

If no statement is active which accesses some variable, then simply changing the type or structure attributes of that variable can be handled by the TFI algorithm as it has been defined thus far. If, however, a statement is active which depends on X, say, and X is changed, there are three possible cases to consider. The diagram below represents a control pointer in a statement dividing the statement, in terms of execution, into past

Visual Fidelity Principle (VFP): the user must be able to expect that the appearance (text) of a program is a reliable indication of the way that program acts (its semantics). Let us observe first of all that this is always true in a compiler system - if not, any deviation is considered a compiler error and not a programming error. For the same reason, it should be the case that the appearance (i.e., the text) of an inactive program describes its semantics. However, some questions arise when speaking of active programs, and these will be delved into more thoroughly along with the above cases in the following paragraphs.

4B3A Statement Alteration

Altering a statement in an inactive program is simply the interactive parsing problem which was discussed in the previous chapter. And, because of the TFI algorithm, any code which a deleted statement possesses must be discarded as well as that from its parent nodes in the parse tree. The dependency chains of any variables on which the code of a deleted statement depended must also be changed to delete those entries, but that presents no difficulty since the dependency chain from a symbol table entry is in fact a ring. Thus, by traversing the chain, the deleted statement can be reached and removed from the chain.

If an active statement is to be deleted or altered (which implies a deletion) then the situation is more complex. For simple statements, such as assignment statements, which have no scope of action as do a BEGIN or an IF...THEN...ELSE statement, either

4B3 The Binding of Attributes

Throughout our discussions of an IPS, one major piece of philosophy has been the source of many of the system complications with which we have dealt: the ability to interleave program execution with program and data composition and alteration. As has already been remarked, facile interleaving is one of the major distinguishable features between batch and batch-like languages such as FORTRAN and BASIC and those designed to exploit the capabilities of interaction for the human user, such as JOSS, APL and LC^2.

Since interleaved execution and alteration can potentially affect one another, complications in the system's bookkeeping can arise. This increase in internal complexity is tolerable only insofar as it results in a meaningful decrease in unnecessary detail and overhead and a meaningful increase in ease of use and flexibility for the user. Since this impact on the user is the ultimate criterion against which an IPS must be measured, any design "principles" must view his requirements as a central constraint. This meta-criterion strongly influences any consideration of the dynamic execution/alteration environment. In particular, problems arise associated with the addition or deletion of declarations in a program which is active. Consider the case of a recursively used procedure having more than one incarnation of itself and its local variables in the control/context hierarchy. Should a declaration added to that procedure affect all its incarnations, just the most recent incarnation, or not be allowed? Since user convenience is considered central to the issue, one simple principle which must be observed is the following:

STRUCTCONST = STRUCTVAR "(" PLEXINIT ")";

PLEXINIT = PLEXCONST $("," PLEXCONST);

PLEXCONST = ARRAYCONST / STRINGCONST / VALUE / STRUCTCONST;

Example: COMPLEX(3.2,1.7)

This is a means of creating and initializing instances of structures to be used as structured constants, or as new incarnations of variables. When a new incarnation of a variable of lifetime ALLOCATED is required, the following notation is used:

ALLOCSTAT = "NEW" ACCESSOR "←" STRUCTCONST;

If the entity represented by ACCESSOR is not NIL, then its previous value will be lost, just as if the counterpart to NEW, vis. DELETE had been used.

The DELETE statement has the simple syntax

DELSTAT = "DELETE" ACCESSOR;

Examples: DELETE PROG;

DELETE PROG.SONS[1];

The latter example would delete the structure referenced by PROG.SONS[1] as well as setting PROG.SONS[1] = NIL; if it is desired to delete the value of PROG.SONS[1] but not what it references, such can be done by

PROG.SONS[1] ← NIL;

4B2E Some Pre-Declared Structure Classes

The system itself, that is its data structures and control must be describable in this same way. Accordingly, a program, the symbol table structure, etc., are declarable within the language itself for bootstrapping purposes. The exact structure of these and other key structures in an IPS will be detailed later.

TREE[n] example above, all of the following would be valid accessors:

PROG

PROG.INFO

PROG.SONS[2]

An operation similar to subfield accessing is treating an accessor as a reference to some structure to be accessed; ":" is used as the operator for this. Thus, if the SONS array in PROG referred to a TREE[3] structure, then the following would be allowable accessors:

PROG.SONS[2]:SONS[1]

PROG.SONS[1]:INFO

However, PROG.LIST.CAR is also correct, since the LIST component of TREE[n] is in fact a LISPLIST and not a reference to one. The complete syntax for ACCESSOR then is

ACCESSOR = SIMPVAR $(("."/":") SIMPVAR);

The last accessing primitive is that which yields the address of an ACCESSOR, namely the unary operation α; thus,

B ← α PROG;

would mean B is a reference to the variable PROG after the statement is complete, and

B ← α(PROG.SONS[2]);

means that B would be made a reference to the second element of the SONS array in PROG, not the contents of that element.

The other necessary primitives involve creation of new incarnations of structures and the destruction of unwanted ones. The syntax for a STRUCTURE declaration provides a convenient template for describing an instance of the structure in the following syntax:

```
STRUCTURE TREE[n] ( SONS: REFERENCE ARRAY [n],
                    ROUTINE: REFERENCE,
                    INFO: LOGICAL,
                    LIST: LISPLIST )
```

In the last example, the structure class is more than one class since it is parametrized. Examples of the use of structure classes in declarations are

```
DECLARE TREE[3] PROG;
DECLARE COMPLEX C1,C2;
```

which would declare PROG as a TREE[3] structure and C1 and C2 as structures of type COMPLEX.

4B2D Operations on Structured Data

Ultimately we would like to provide an operator extendibility feature to match the structure extendibility outlined above, but for the present we will only describe an accessing mechanism and a creation/destruction mechanism for structured data.

Earley has isolated the following primitive operations on structures (although not with the same syntax); however, the explanations of each are this author's.

The most important action on structured items is the ability to access their components, whether structures themselves or atomic data. A simple notation for this operation is the dot notation of PL/I, namely

```
SIMPVAR = .ID / .ID "[" SUBSCRIPTS "]";
ACCESSOR = SIMPVAR $( "." SIMPVAR);
```

where each SIMPVAR specifies a subfield of the structure whose name precedes it, except for the first name of an ACCESSOR. In the

4B2C Structure Class Declarations

In order to extend the number of data types of the system, a notation and a mechanism are needed for describing a new data type and the operations on it. As Earley has pointed out in his VERS paper [Ea 69], there are some operations, mainly accessing, and creation and destruction which are common to all structures and which can be viewed as independent of the exact representation of the data. Examples of this are the sequencing from item to item in a list structure, the extraction of a field within a table entry, or the accessing of a particular element in a tree. Standish's thesis [St 67] is another excellent work on extendible data structures, a subject of much controversy, and somewhat removed from the thrust of this work except as a necessary component of a bootstrappable IPS. Thus we feel no compunction about "stealing" a notation for the simpler forms of extendibility, and the following syntax can be found in both Standish's and Earley's works, as well as a report by Wulf and Mitchell [WM 69] for the declaration facilities for a particular IPS.

The syntax for declaring a new structure type follows.

```
STRUCTURE = "STRUCTURE" STRUCTVAR "(" PLEXLIST ")";
STRUCTVAR = 0$1( .ID 0$1( "[" .ID $( "," .ID) "]" ) );
PLEXLIST = PLEX $( "," PLEX);
PLEX = 0$1( .ID ":" ) SIMSTRUCT;
SIMSTRUCT = ARRAYATTR/STRING/TYPE/STRUCTURE;
```

Some examples are

```
STRUCTURE COMPLEX (RP: REAL, IP:REAL)
STRUCTURE LISPLIST (CAR:REFERENCE, CDR:REFERENCE)
```

variable is created.

Examples: DECLARE REAL ARRAY[10] B;

DECLARE ARRAY[10,-3:3,0:4] C, D;

Accessing an array element is the conventional notation

ARRAYNAME "[" EXP $("," EXP) "]"

with an additional notation for extracting subarrays in which any subscript position may be omitted or expressed as a bounds pair. When omitted, that subscript position is assumed to range over its declared bounds; if a bounds pair such as i:j is given, then that subscript position ranges from i to j inclusive. In this way, we can name "rectangular" subsections of arrays. The notation of APL [IF 68] which allows any subscript to be a sequence of single values is more general than this and is also allowed; the above notations are intended for common cases.

4B2B Strings

Strings, as opposed to vectors of characters, are elastic sequences of atomic types. Elasticity means that they have no predetermined length and may be altered by inserting or deleting other strings of length zero or greater, expanding or contracting in the process in order to remain logically contiguous. Since strings can be composed of any atomic type, in many ways they can behave like lists. Some of the attributes of a string which must be available are its length, and, if accessing the i'th element, an indication of whether or not it is past the end of the string. Strings are assumed to be homogeneous, i.e., only one type is allowed as the atomic units of a string.

4B2 Structured Data

Bootstrapping an IPS places demands not only on the efficiency of the system, but also places a strong requirement for flexibility of data structures on it. Data such as symbol tables, parse trees, machine code, etc., must be describable and manipulable; that is, not only must we be able to describe complex data structures and access their various components, we must also be able to define new operations on them and extend previously available operations to them. This has benefits for the user as well as the system builders as has been demonstrated by the "contagious" quality of Iverson's APL.

Three main items will be dealt with in this section. The first is simply the definition for the standard structured data available in almost all programming languages. Among these are homogeneous arrays, strings, etc. The second concern is the "theft" of some notation for speaking of extensible data structures and operations on them. Lastly, we will give examples of some system defined structures of note.

4B2A Arrays

An array is a set of values all of the same type, with individual elements being accessed by a sequence $i_1,\ldots,i_k$ of length k =(the number of dimensions of the array). The notation for declaring an array within a declaration is simply

```
ARRAYATTR = "ARRAY" "[" BOUNDS $( "," BOUNDS) "]";
BOUNDS = 0$1(EXP ":") EXP;
```

The time at which the bounds expressions are evaluated can be thought of as the time at which an incarnation of the array

Some of these combinations are very interesting. For example, SECONDARY AUTOMATIC would describe a temporary "file" whose lifetime was that of the block/routine in which it was declared.

4B1G Declaration Notation

For most cases, it is desirable that only what is necessary for the declaration of a variable need be stated. Naturally, it may be the case that many variables will never be explicitly declared, but for reasons of efficiency and completeness, declarations are very necessary addenda to many programs. When a declaration is used, it has the following syntax:

```
DECL = "DECLARE" ATTRLIST IDLIST;
IDLIST = .ID $("," .ID);
```

where ATTRLIST is a list of attribute-values, separated by blanks, with zero or one attribute-values for each possible attribute (except for the protection attribute which may have up to three). The list of sets below gives some notion of the possible combinations. Those values which are underlined are the defaults which the author would like chosen when no value for that attribute is specified. In general, we believe that defaults are a matter of personal choice and ought to be specifiable by the user. The command language CL-II of TSS/360 [IBM 69] is a good example of an IPS which can be easily tailored by the user to appear as he feels best.

The alternatives, grouped by attribute are the following:

attribute:	storage class	protection	scope	lifetime	type
values:	MAIN	NOREAD	LOCAL	AUTOMATIC	GENERAL
	SECONDARY	NOWRITE	SECRET	STATIC	INTEGER
		NOEXECUTE	GLOBAL	ALLOCATED	REAL
			EXTERNAL		LOGICAL
					CHARACTER
					REFERENCE

In an IPS there is one other type which, while not really simple, is basic to the notion of type-less variables, namely, type GENERAL. This is the default type assumed by all non-declared variables, although one can envision its use in a declaration even for compiled programs.

4B1F Value

The actual value of a variable is normally not considered as an attribute in the sense of scope, type, etc., primarily because a variable is used precisely because its value is not bound to be a specific number. Of course, many iterative procedures can be thought of as existing solely for the purpose of constraining the values of certain variables to lie within tighter and tighter bounds; however, that is outside our range of interest here. An application of the binding of a specific value to a variable which has some use is the naming of constants by binding a name, by declaration, to a specific value. Thus, one might say

```
DECLARE REAL A ← 3.2;
```

as a declaration which also causes A to be initialized to the value whenever an incarnation of it is created, or

```
DECLARE REAL A = 3.2;
```

to designate that A is simply another name for the constant 3.2, and any machine instructions which are available to make use of such constancy may be used by the system whenever A is used in the program.

4B1E Type

The most common use of declarations in programming languages is the assigning of types to variables. Simple types usually are those which have been representable and able to be operated on by the computer hardware. These include entities such as integers, floating-point numbers, characters, and logical values (bit strings) as well as addresses or pointers. We will denote these as INTEGER, REAL, CHARACTER, LOGICAL and REFERENCE respectively.

known only to the block in which the declaration appears, and are not known by its containing or contained blocks, whether in a lexical or dynamic sense; to other super- and sub-blocks it is as if there were no declaration of that variable in that block.

LOCAL corresponds to the normal scope rules of Algol 60 where only lexical scope is concerned; and EXTRENAL exists in a number of time-sharing systems (TSS/360, MULTICS) and is used for sharing programs and data between processes whether belonging to the same user or to different users.

4B1D Lifetime

The lifetime of a variable corresponds to how long a given incarnation remains in existence and is accessible. That lifetime can be one of:

STATIC: only one incarnation of the variable remains, existing as long as the program comprising its maximal scope exists; this lifetime corresponds to that of variables in Fortran and to Algol own — although own also implies a local scope;

AUTOMATIC: each time the block/procedure which is an AUTOMATIC variable's maximal scope is entered, a new incarnation of that variable is created; upon exit from that block (i.e., when that incarnation of the procedure or block ends itself) the corresponding incarnation of the variable disappears;

ALLOCATED: an incarnation for the variable will be made when it is first used, and may be explicitly deleted when the program no longer requires the information associated with that variable; of course, it can then be subsequently reassigned a value and be reincarnated.

Since data and programs are treated in only three primitive ways in computers (namely, being read, written, executed — and combinations thereof), we choose simply to protect against these primitive actions by allowing the following attribute-values, in any combination of zero or more in declarations: NOREAD, NOWRITE, and NOEXECUTE. The meanings of combinations of these are specified in the following table:

PROTECTION			INTERPRETATION
NOREAD	NOWRITE	NOEXECUTE	dead, inaccessible space
NOREAD	NOWRITE		pure procedure - execute only
NOREAD		NOEXECUTE	write-only data
NOREAD			modifiable but uninspectable code
	NOWRITE	NOEXECUTE	read-only data
	NOWRITE		inspectable pure procedure
		NOEXECUTE	unexecutable data
			unprotected data or program

Figure 4B1B-1: Protection Combinations

4B1C Scope

The scope of a variable may be one of the following:

LOCAL: known only to the block or routine in which the declaration lexically appears along with all its lexically contained and potentially its dynamically contained blocks/procedures;

GLOBAL: each block/procedure which declares some variable (A, say) as GLOBAL then uses the same variable A;

EXTERNAL: variables declared EXTERNAL are shared among processes (whether inter- or intra-user) in the same manner as GLOBAL variables between separate procedures in a single process;

SECRET: as mentioned earlier, variables whose scope is SECRET are

4B1 Attribute Sets

In this section some of the basic attributes and their corresponding value sets will be outlined. A following section will deal with a number of implementations and accessing methods for various attribute-value combinations.

4B1A Storage Class

The storage class of a variable declares where it will reside, and can be either MAIN or SECONDARY; data structures normally residing on secondary storage (drums, disks, tapes, etc.) are the equivalent of files or data-bases. Normally program and data structures reside in main memory, which is the only place where they are directly accessible and/or executable. Thus, the use of a function F whose storage class attribute-value is SECONDARY would imply the invocation of a transfer function to create a copy of it in main memory for execution.

Clearly, on a particular computer system, these declarations ought to be parametrizable over the various types of main or secondary storage. Nevertheless, the user should be able to specify the simplest options with the IPS concerning itself with the details.

4B1B Protection

Although protection is certainly necessary in single-user as well as multi-user IPS environments, the scope of the problem is too large for the present discussion. However, within a specific process it is still reasonable to protect data structures and programs from one another. A simple means of accomplishing this is to allow declarations to contain protection options.

which the variable is to reside;

values: MAIN, SECONDARY

(2) protection: the manner in which a variable may be accessed, either by its containing process or other processes (possibly belonging to different users);

values: NOREAD, NOWRITE, NOEXECUTE

(3) scope: the limits (lexical or dynamic) within which a specific variable is known;

values: LOCAL, SECRET, GLOBAL, EXTERNAL

(4) lifetime: the length of time from creation of a specific incarnation of a variable until its destruction;

values: AUTOMATIC, STATIC, ALLOCATED

(5) type: the form and semantic meaning of the variable with respect to specific classes of operations; e.g., arithmetic, string, etc.;

values: GENERAL, REAL, INTEGER, CHARACTER, LOGICAL, REFERENCE

(6) value: the actual value of the variable; in general, this is the last thing which is bound (if ever);

(7) structure: the number and form of the individual values which a variable represents.

values: ARRAY[n], SCALAR, LIST, etc.

These attributes will be discussed in detail in the remainder of this section. We will investigate the possible values of each of these attributes (called attribute-values), the binding of attribute-values to variables, and the representation and accessing of variables for various attribute-value combinations.

4B Attributes of Variables

As in a compiler system, the semantic attributes of variables in an IPS are directly connected with the potential representations possible for them. Since variables in an IPS can exist in different stages of declaration (at different times, of course), we must be concerned primarily with two things:

(1) representing a variable in the most efficient manner possible for the set of attributes ascribed to it either by context or by declaration, consistent with the flexibility to vary unstated attribute values;

(2) allowing the representation to vary as declarative information (however obtained) appears or is withdrawn.

This last requirement is simply the interpreter/compiler philosophy restated for the case of data structures. Requirement (1) is essentially a challenge to the implementers and designers of interactive systems to find representations for variables which can take advantage of many different combinations of attribute values for variables. Ideally, we should like to develop a method for moving representations in that multi-dimensional space as the TFI does with programs. For the present, however, we will simply define a subset of the possible representations and accessing methods for variables whose attribute values are partly known.

The attribute values which describe a variable are chosen from a set of attributes and possible values for them. The main attributes to be discussed are the following:

(1) storage class: the type of memory (active or passive) within