Using this approach, the A+B*A example could have the following tree and code after once being executed in this manner:

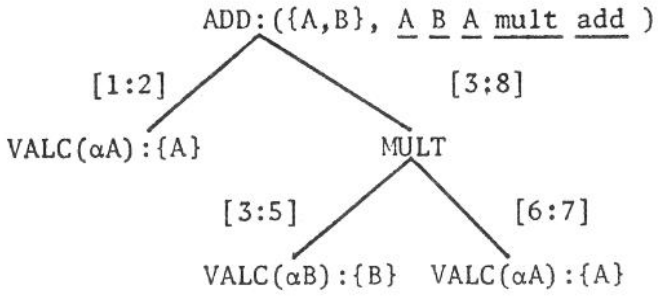
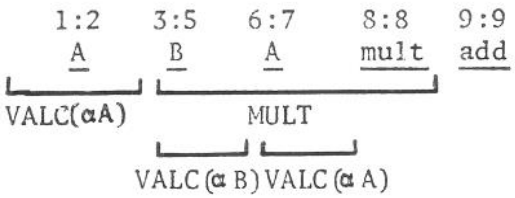


Figure 4C1-2 :A+B*A Tree with Single Buffer

The [FIRST:LAST] pairs are given as attached to the arcs leading to subnodes.

Which sections of code are generated by a non-terminal node can be deduced from the position pair on the arc leading to it and those leading to its subnodes, even if that node generates more than one piece of code (scattered between the code from its subnodes, for instance). Thus, we can easily write down the scopes of parts of the code at the ADD node from the position pairs:



The code generated by MULT is [8:8] and [9:9] is that generated by the ADD node. This relatively innocent-looking addition to the TFI scheme will also prove of considerable value in aiding jumps within and around generated code, in recovery from semantic changes affecting active statements, and in allowing a TFI to be operated as a "normal" compiler.

The last item which needs to be mentioned is that nodes

We have now generated two apparently opposing constraints:

(1) in order to avoid complete reinterpretation after a semantic change, each node needs a copy of its own code which it can use; and

(2) in order to conserve memory space, only certain nodes should have code attached to them (at the statement level, for instance).

A means of satisfying both constraints is to place minimal information at a non-terminal node concerning the code generated by each of its subnodes. This information is simply

(a) the starting position of the code from the beginning of the code buffer (counting from one) for that subnode, and

(b) the index of the last word of code generated by that subnode.

Both these numbers include all the code which appears to have been generated by a subnode, whether generated directly by that subnode or by that subnode's subnodes.

This code position pair we will designate as [FIRST:LAST] and is generated in the following way:

(1) whenever a node activates one of its subnodes, the next available code buffer position becomes the FIRST value associated with that subnode; and

(2) whenever a subnode returns control to its parent, the LAST value is set to the value of the code buffer index minus one.

and, in general, $$F_{n+k} = F_n + 2\times \sum_{i=1}^{k} (n+i) + k$$

$$= F_n + 2nk + k(k+1) + k$$

$$= F_n + k^2 + k(2n+2) \qquad \text{(L5)}$$

Now, $F_0=1$; and substituting n=0 in (L5) gives

$$F_k = k^2+2k+1 = (k+1)^2 \qquad \text{(L6)}$$

Hence, for the lower-bound case, $F_k/f_k \simeq \frac{k}{2}$; i.e., only (2/k) of the space would be occupied by the code which is used most often.

For the upper-bound case, the recurrence relation defining f_{n+1} is

$$f_{n+1} = 2f_n+1 \qquad \text{(U1)}$$

and, in general, $$f_{n+k} = 2^k f_n + \sum_{i=0}^{k-1} 2^i$$

$$= 2^k f_n + 2^k - 1 \qquad \text{(U2)}$$

Now, $f_0=1$, so substituting n=0 in (U2) yields

$$f_k = 2^k + 2^k - 1 = 2^{k+1} - 1 \qquad \text{(U3)}$$

The relation which defines the total amount of code in a full tree of depth n+1 is

$$F_{n+1} = 2F_n + f_{n+1} = 2F_n + 2^{n+2} - 1 \qquad \text{(U4)}$$

and, in general, $$F_{n+k} = 2^k F_n + k\times 2^{n+k+1} - \sum_{i=0}^{k-1} 2^i$$

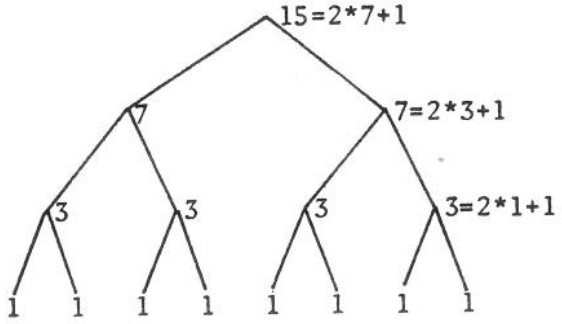
$$= 2^k F_n + k\times 2^{k+1+n} - 2^k + 1 \qquad \text{(U5)}$$

And, since $F_0=1$, we get $F_k = 2^k + k\times 2^{k+1} - 2^k + 1 = k\times 2^{k+1} + 1$ (U6)

Thus, in the upper-bound case, $F_k/f_k \simeq k$, meaning that only 1/k of the storage used for code is filled by the code at the topmost node.

In both these cases, then, the ratio of total code to most used code is of the order of the depth of the parse tree. It is clearly to our advantage to attempt to eliminate this wastage.

Upper-Bound Case: The parse tree is a full, balanced binary tree: i.e., every node but those at the leaves has two subnodes, and the tree has the same depth to the right and left:



What we would like to find are expressions for each case for

(a) the amount of code at the top node in a tree of depth k, called f_k; and

(b) the total amount of code in a tree of depth k, obtained by adding together the code lengths at every node; we will call this F_k.

Then, the ratio F_k/f_k is an indication of what proportion of the space is "wasted" due to the presence of the copies of the code.

For the lower-bound case, f_n can be defined as

$$f_{n+1} = f_n+1+1 = f_n+2 \qquad \text{(L1)}$$

and, in general, $$f_{n+k} = f_n+2k. \qquad \text{(L2)}$$

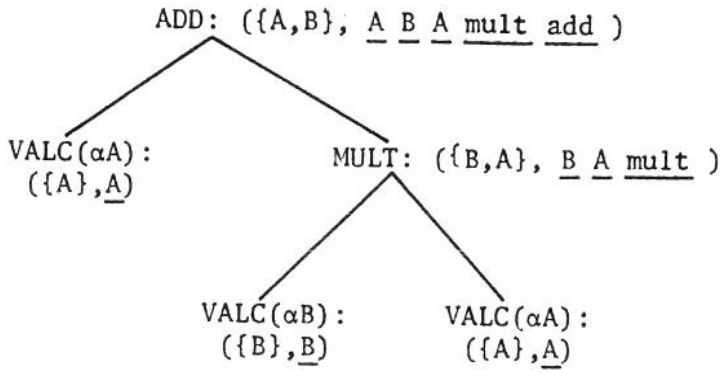
Now, $f_0=1$, and letting n=0 in (L2) gives

$$f_k = 1+2k \quad (k\geq 0) \qquad \text{(L3)}$$

for the lower-bound case.

Similarly, we can easily write the following recurrence relation for F_{n+1}:

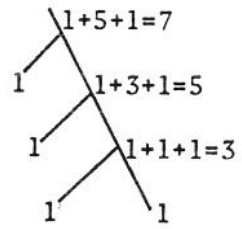
$$F_{n+1} = F_n+f_{n+1} = F_n+2(n+1)+1 \qquad \text{(L4)}$$



Now, if we count each underlined unit in the code portions of this tree, we find there are 11 "pieces of code", and 7 entries (such as {A}, {B}) representing threads in the dependency chains associated with those variables. Since it is hoped that the code at the ADD node is all that will be needed for some period of time, approximately twice as much code is kept the tree as will normally be needed.

We can obtain lower and upper bounds on the potential wastage by considering the following two extreme cases of a parse tree (assumed to be binary). In each case we are making the assumption that each node adds one word of its own code to that which it acquires from its two subnodes.

Lower-Bound Case: The parse tree has the form



or its left-recursive counterpart, which is really an approximation to a linear list.

4C The TFI Generalized

The TFI algorithm as described in chapter 3 has some limitations and a few associated difficulties. The primary restriction is its inability to allow semantic changes to a variable when part of a program which depends on that variable is active and has compiled code for accessing the variable. One of the main problems revolves around the issue of jumps or branches in the code generated by the TFI, especially for statements like IF...THEN...ELSE.

In this section we will develop solutions for these difficulties. Also to be presented is a formal discussion of the adequacy of the TFI method for handling semantic changes to variables. Finally, we will develop a means by which the TFI can also act as a "normal" compiler, and discuss some space trade-offs of using the dynamic TFI over compiling programs in the IPS.

4C1 Code Creation by the TFI

The TFI as described thus far has been almost flagrantly excessive in its use of storage (this was done for purposes of ease of explanation). The fact that an entire parse tree is kept in memory is part of this misuse. However, a much larger portion of storage is consumed by each node's keeping a copy of its generated code, as well as its parent nodes also acquiring copies of that code.

In the previous chapter we applied the TFI discipline to the execution of the expression A + B * A, finally arriving at the following tree:

Ultimately it is clear that good compiled code for using a reference variable can only be obtained if the user is able to declare either or both of: (1) the depth of the reference chain, and (2) the type of the ultimate referent (as is done in Algol 68 [VW 69] for instance).

In order to allow this we extend the syntax for declaring a reference variable in the following manner:

```
REFERENCE = "REFERENCE" 0$1( "+" INTEGER) 0$1("TO" ATTRLIST ":");
```

Some examples are

```
1 DECLARE REFERENCE A;
2 DECLARE REFERENCE TO REAL B;
3 DECLARE REFERENCE 2 C;
4 DECLARE REFERENCE 3 TO ARRAY[10] D;
5 DECLARE REFERENCE 1 TO REAL ARRAY[10]:ARRAY[10] MATRIX;
```

meaning:

(1) A is a reference variable without depth or the type of what it points to being specified;

(2) B is a reference of unknown depth which must refer to something of type REAL;

(3) C is a two-level reference to something of unspecified type;

(4) D is a three-level reference to a 10 element vector of unspecified type; and

(5) MATRIX is itself a vector of 10 elements, each of which is a direct reference to a REAL ARRAY[10] — this could be considered an implementation of two dimensional arrays.

that label (e.g., in a GOTO statement). Labels are identifiers attached to lines in programs; they therefore act as reference variables pointing to program statements. Since it is an identifier, there is an entry in the containing program's symbol table for each label in that program. Any code dependent on a label (such as a GOTO) is on the dependency chain for the label; hence, deleting a labelled statement can be viewed as a semantic change to its labels. Code dependent on the labels can be invalidated using the standard method and traversing the labels' dependency chains. Finally, each line of program text (or, equivalently, each node in the parse tree corresponding to a line of text) must have a list of pointers to the symbol table entries for the labels in that line in order to drive the above scheme for handling changes to labelled statements.

Reference variables cause some complications for the routines such as VALC which must access them. VALC could assume that the depth of the reference chain or the type of the ultimate referent are fixed. The more it assumes, of course, the more likely it is that those assumptions will prove incorrect, causing reinterpretation. This last statement recalls what was described for the case of variables with inherited scope, and, indeed, the situations are very similar. Inherited variables are implemented just as are reference variables, except that they are bound by the scope algorithm instead of by statements in the user's program, a distinction which can become arbitrarily fuzzy in a bootstrappable IPS. All the comments about what to consider as fixed in the case of inherited variables thus apply equally well for reference variables.

or one entries in the variable's personal stack; any new allocation of the variable causes the system to delete the previous one. If the previous value is to be retained, then one need only set one of the fields in the newly created item to reference the old before assigning the new value to the variable.

One thing that is apparent from all the above accessing methods is that any variable whose scope is known may be implemented using the LC^2 method with the naming structures previously described. It is therefore an implementation which ought always to be present. The presence or absence of other representations taking advantage of declarative knowledge about some variables' attribute-values is only necessary where efficiency and bootstrapping (to be mentioned later) are concerned.

4B4G Reference Variables

Sequences of machine code are not the only class of items which may depend on the semantics of a variable. Any variable which refers to, or names another must also be considered as dependent on the variable to which it points. Thus, any change in the critical attributes of the variable referenced must not only cause reinterpretation of any statements using that variable, but may cause reinterpretation of statements using variables which refer to it. Reference variables must therefore (like inherited scope variables) be on the dependency chain of the variable to which they refer.

A special case of the situation just described is the deletion of a labelled statement in a program which is referred to by

A simpler method is to implement the variable using the LC^2 method, outside of the stack, until the routine in which it is active is finished. We also set an attribute-value field specifying how it is implemented. This implementation attribute is essentially an internal system attribute (although there is no reason why a user could not specify the implementation to be used in a declaration) set at the time a declaration takes effect and decides on the representation for that variable. At the next block entry the declaration will take effect since we are assuming the variable has lifetime AUTOMATIC and scope LOCAL. Then we can decide to change the implementation to the more efficient stack method, using the standard TFI mechanism for invalidating any code which depends on the value of the previous attributes, including its implementation.

4B4E Fortran Method

Any variables whose scope and lifetime are declared, the lifetime being STATIC, may be implemented and accessed in the manner most common on computers, namely as a group of contiguous words whose absolute address is known. The only difference between such a variable and one implemented in the LC^2 manner is that the program may feel free to use the address of the value as given in a name table entry rather than always accessing it via that entry.

4B4F Other Considerations

Allocated variables may also be implemented using the LC^2 method of a stack per variable, but there will always be either zero

attributes (those being depended upon by some code) is not restricted to the case of inherited-scope variables, but rather applies to all code for all variables. Hence, if a change in attribute-values, however done, does not alter the values of any critical attributes, no code will be invalidated.

4B4D Algol Stack Method

The value entries for all variables in some program whose scope and type are declared and whose lifetime-value is AUTOMATIC can be implemented by using a contiguous block of storage (sometimes called a "frame") which is allocated on a stack used for this purpose; the same stack can be used for recording control information as well as the values of partially evaluated expressions. Indeed, the most common method incorporates the control information into the frame used for local storage, thus allowing one "push" and "pop" operation to push or pop both the control information and the values of any local variables in this class.

Since such a frame, being imbedded in the stack, is normally fixed in size, a declarative change to a variable so implemented can present problems. Of course, the old value in the stack can simply be discarded since it will be reclaimed when the program exits. However, we are left with the case of a variable's being declared while its containing program is active. It is unlikely that we will be able to implement it in the stack at that time since the frame size is fixed, although occasionally we may be able to use the stack cell which was discarded because of the changed declaration.

Any code generated for accessing an inherited variable,V, will assume some things (such as type and structure) as fixed. If separate calls to the routine using V as inherited do not present V with attribute-values different from those used in the code accessing V, which was created on a different call, then that code will remain valid. Now, in order to allow such code to be as efficient as possible, we could attempt to use all the available semantic information about V, including the scope of its different parent variables. But this could easily mean that every call on the function would invalidate the code depending on V. On the other hand, if only some strict subset of the semantics is assumed by the code, then so long as that subset was constant from call to call, the code would remain valid. To do this it is necessary that each symbol table entry contain an extra set of bits which correspond to the different semantics fields, say one for each. Whenever a piece of code is created which depends on the i'th attribute value, the interpreter routine is required to set the i'th bit in that extra set in the symbol table to mark that there is some code depending on the value of the i'th attribute. Then, on each call, when inherited variables are being attached to the correct parent, only those attributes marked by the code would need to be the same as the corresponding attribute values of the parent. If the generated code expects little in the way of semantic constancy then this method should give it a longer lifetime; the converse is also true.

What is more, this method of selectively marking "critical"

represents. Any declarative change to the prototype must affect all its son variables since they are semantically the same entity. The simplest way to ensure that the sons "know" of any semantic changes to the prototype is to include them on the prototype's dependency chain as if they were code which was dependent on it. Whenever the prototype changes, its dependency chain will be traversed to mark all code on it as invalid. If a son variable is encountered on the chain, its dependency chain is also traversed in the same manner in order to invalidate code which may "indirectly" depend on the prototype.

4B4B The LC^2 Method

In this implementation, each entry in a name table points to the top of a non-contiguous stack of values for the variable whose name is in that entry. These stacks are attached to the individual entries which are listed from each name entry — thus there is one stack per unique scope of the variable. Each time a declaration for such a variable is executed, for example on block entry, a new value entry for the variable is pushed onto its stack. Accessing then only requires knowledge of the scope of the variable in order to insure that a particular entry in the list from a name table entry may be used to access the variable's stack.

Hence, this method, while independent of the type and structure of the variable in question, is constrained by the scope of that variable being fixed. Thus, any change in the scope of a variable renders code generated for it under this method invalid.

4B4C Variables With Inherited Scope

An inherited variable is assigned a name table entry, which is marked as inherited, even though no declaration for it appears at that level. Accessing such a variable then becomes a matter of accessing the name table entry, using it as a reference to the value entry to which it points, as in the LC^2 method, and then using the value as a pointer to the actual value cell for the variable to which the one of inherited scope is attached. Let us call a variable which has inherited scope a son of the (prototype) variable which it

and the harder it is to recover from declarative changes to the variable.

The accessing of which we speak is that which takes place when a low-level semantic routine such as VALC(variable) appearing in the parse tree is to be executed. Such a node will be a terminal node with a parameter specifying the variable. At that time, the VALC routine would have to decide, on the basis of the way in which that variable is stored in memory, as well as other attributes, which accessing function is to be used in the code which it generates and executes. Its partner is the routine used at the time a declaration takes effect which decides how a variable is to be implemented.

4B4A Full Interpretation

When a VALC node is encountered during execution of the program, the exact variable to which it corresponds must be determined using the SCOPE algorithm described earlier; once it has been found in the naming structures, then other information available for the variable can be used to construct accessing code for it. Thus, this type of accessing is normally only done the first time that a VALC node is encountered which has no valid code associated with it. The method always works since the only semantic information for a variable which never changes for that variable is its symbol table address.