point is reached, since it is only there that it and the operand stack can be "out of phase". Hence, beyond the interpretation-point, PERFORM will use the operand stack for call-information also; this keeps the time during which calls must be treated specially to a minimum.

The second loose end has to do with the fact that the old value of A is on the operand stack even though the leftmost VALC(αA) created code to fetch the new value of A onto the stack, and the ADD node will have trouble deciding whether to assume the type of the value which the VALC node describes (INTEGER) or the value on the stack, which is REAL. The answer is that this situation normally would not have been allowed to exist because of the Visual Fidelity arguments (and algorithms) in sections 4B3A and 4B3B. This inconsistency was allowed here only for the purposes of being able to use a familiar example.

Nevertheless, SETUP does need to be extended to cover the case when a variable is changed after it has been used in a statement and will not be used again in that execution of the statement. In that case, the easiest action is simply to mark the code invalid but finish executing it. This is a viable method since the code will be seen to be invalid the next time that statement is used, and the original, simple TFI can handle that case.

The state of the MULT node is (INVALID,NON-TERMINAL,HAD THE POINT) since CONTROL is in the range [3:9]. Hence, ACTION 1 is taken and MULT begins. The call-stack is

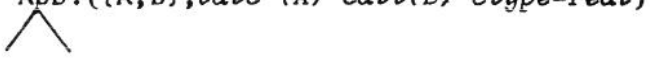
SETUP.6 ADD.2

MULT's first action is

1 PERFORM *1;

which activates REVERT. The state of the VALC(αB) node is (VALID, TERMINAL,HAD THE POINT); we are finally at the interpretation-point! The sequence of actions by REVERT for a node in this state is

(1) copy code [3:5] from OLD_CODE into the new buffer, making the ADD node



(2) set EXECUTE_CODE to TRUE (i.e., turn execution back on);

(3) branch to the "right" place in the code buffer; the right place is (CONTROL - XFIRST + β.FIRST) = (5 - 3 + 3) = 5 in the new code buffer.

The call-stack is

SETUP.6 ADD.3 MULT.2

and when the code is executed, REVERT will simply return to MULT.2, and interpretation will continue with REVERT still being activated on each PERFORM because REVERT_SWITCH remains TRUE. When control finally reaches SETUP.6, REVERT_SWITCH is set to FALSE, the OLD_CODE is discarded, and control returns to the original caller of the ADD node (and the separate call-stack can be dispensed with).

There are two loose ends which need to be tied. The first is that the separate call-stack is only needed until the interpretation-

activated by the PERFORM, since REVERT_SWITCH = TRUE. The state of the ADD node is (INVALID,NON-TERMINAL, HAD THE POINT) since the CONTROL certainly resided in its code. The action taken is ACTION 1: allow ADD to operate with execution remaining off.

The operand stack remains as it was when REVERT was started: (the heads of stacks are to the right)

operand-stack: A

The (separate) call-stack looks like

call-stack: SETUP.6

and ADD is now allowed to execute. ADD's first action is (see section 3D1: The TFI Algorithm):

1 PERFORM *1;

to call VALC(αA) to interpret. REVERT intervenes, and the state of the VALC(αA) node is (INVALID,TERMINAL,NOT THE POINT) since CONTROL does not lie in [1:2], which is this node's code-position pair. Therefore, the necessary action is ACTION 1. The call-stack is now

SETUP.6 ADD.2

Ultimately VALC will produce new (and probably different code), *valc'(A)*, to push the value of A onto the operand-stack, but since EXECUTE_CODE = FALSE, it will only be generated and not executed. VALC then returns to the ADD routine. The environment is now the following:

call-stack: SETUP.6

operand-stack: A

the ADD node: ADD:({A},*valc'(A)*)

Line 2 of ADD now executes "PERFORM *2" and REVERT intervenes.

4C3D An Example of the Use of REVERT

We will trace the application of REVERT for the case of the example of A + B * A above, noting which action is taken at each node in the tree. The trace will have a form similar to that used in chapter 3: line and call depth, the call-stack, the operand-stack, and the tree node being "executed" are displayed. A separate call and operand-stack are needed in order to avoid incorrect intermingling of operands and call information caused by the fact that the calls are taking place after the time they normally would had we been interpreting. Since call-markers are linked anyway, this simply means that another stack is used for calls, which stack links into the former at the point where REVERT returns control (line 7 in figure 4C3C-5). From that time on the call and operand-stacks are again one single stack.

We will assume the tree in figure 4C3C-4 and begin applying REVERT to the ADD node. First, we do SETUP for REVERT. This will leave the following assignments to variables:

EXECUTE_CODE = FALSE;

OLD_CODE = *valc(A) call(B) ctype-real valc(A) real-mult real-add*

The ADD node (β is its address) now has an empty buffer and dependency set:

And, finally,

REVERT_SWITCH = TRUE.

Also, the value of CONTROL is assumed to be 5, placing it within the code *call(B)*.

The ADD node is now PERFORMed from line 5 of SETUP, and REVERT is

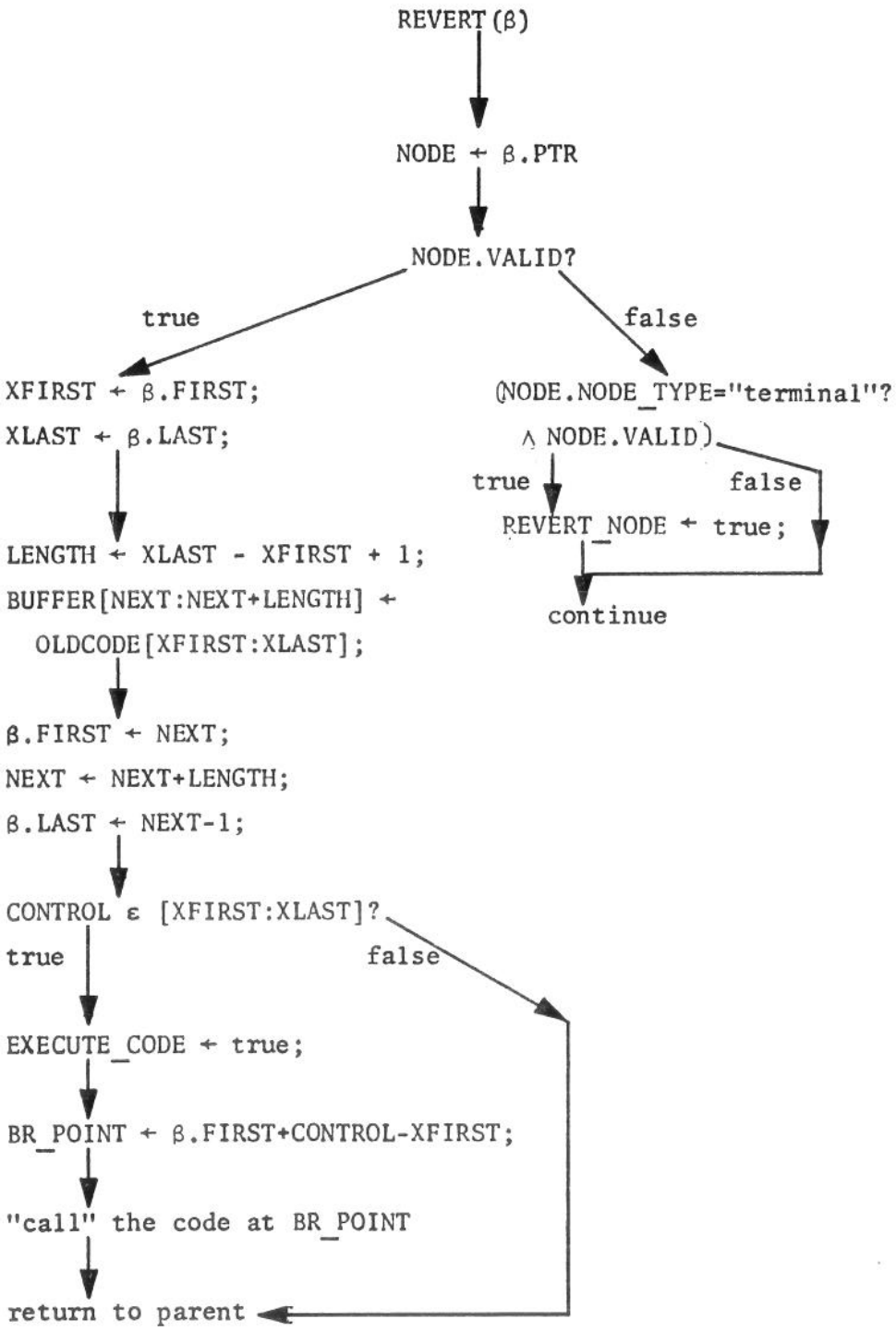


Figure 4C3C-6 : The REVERT Algorithm

(VALID,TERMINAL,HAD THE POINT): do ACTION 4.
[VALC(αB) in the example]

REVERT_HERE is a switch which tells the VALC or DESC node that it must finish its previous action which was stopped before completion (if, in fact, that has a valid meaning) and then create new code to take its place. Each of the other actions are fairly straightforward. A flowchart for the described algorithm follows. is a pointer to the subnode descriptor in the parent node containing a pointer (PTR) to the subnode being PERFORMed and the [FIRST:LAST] pair for that subnode.

attaches a new code buffer for β_1 and calls the routine specified by the node at β with EXECUTE_CODE set to FALSE in order to initially suppress the execution of generated or copied code. REVERT_SWITCH is tested by each TFI routine upon entry via a PERFORM and if it is TRUE, the following part of the REVERT algorithm is activated.

For each node which is PERFORMed while REVERT_SWITCH is TRUE there is a set of three mutually exclusive conditions which describe the state of that node:

(1) it is either terminal or non-terminal;

(2) it either has valid code or is invalid [such as ADD]; and

(3) the interpretation-point resides in it or it does not.

For each combination of these three state sets we can list the action to be taken at that node:

(INVALID,NON-TERMINAL, NOT THE POINT): [not in the example]	(ACTION 1):allow the routine to operate; execution stays off.
(INVALID,NON-TERMINAL,HAD THE POINT): [ADD,MULT in example]	do ACTION 1.
(INVALID,TERMINAL,NOT THE POINT): [the first VALC(αA)]	do ACTION 1.
(INVALID,TERMINAL,HAD THE POINT): [not in the example]	(ACTION 2): REVERT_HERE←TRUE; do ACTION 1;
(VALID,NON-TERMINAL,NOT THE POINT): [not in the example]	(ACTION 3): copy appropriate piece of OLD_CODE into new code buffer.
(VALID,NON-TERMINAL,HAD THE POINT): [not in the example]	(ACTION 4): do ACTION 3; EXECUTE_CODE←TRUE; GOTO correct place in new code.
(VALID,TERMINAL,NOT THE POINT): [not in the example]	do ACTION 3.

that node, some action must be taken and execution resumed. The action to be taken is a function of the routines VALC and DESC (Descriptor Call); They are the only interpretive routines which must be cognizant of the reversion to interpretive control.

Although we could predetermine exactly which node is the critical switching point from non-execution to execution, and set a bit in that node to cause some action, we must not do so. For it is possible that control may never reach that terminal node. After all, its code is not necessarily invalid, and the TFI will not reinterpret nodes with valid code. And, if one of its parents has valid code, reinterpretation will bypass that point. In that case, the bit in that VALC or DESC [VALC(αB) in figure 4C3C-4] would remain set — incorrectly — so we are constrained to do this checking "on the fly" as each PERFORM of an interpretive node takes place (see Appendix 3A for a description of PERFORM).

In any case, we will start the faked interpretation by the following simple sequence. β is the address of the node to which the code was attached [the ADD node in figure 4C3C-4].

```
1  EXECUTE_CODE ← FALSE;
2  OLD_CODE ← β.CODE_STRING;
3  β.CODE_STRING ← α(NEW CODE_BUFFER);
4  REVERT_SWITCH ← TRUE;
5  PERFORM  β.ROUTINE(β);  COMMENT: begin fake interpretation;
6  REVERT_SWITCH ← FALSE;  COMMENT: clean up - we are finished;
7  RETURN to original caller of the code;
```

Figure 4C3C-5: SETUP for the REVERT Algorithm

This program saves the old (and now invalid) code, creates and

valc(A) call(B) ↑ *ctype-real valc(A) real-mult real-add*

Figure 4C3C-3

for the expression A + B * A under the assumptions of the previous examples. The "↑" represents the execution-point as being within the code which accomplishes a call on B.

The parse tree, with dependencies and code position pairs noted, is:

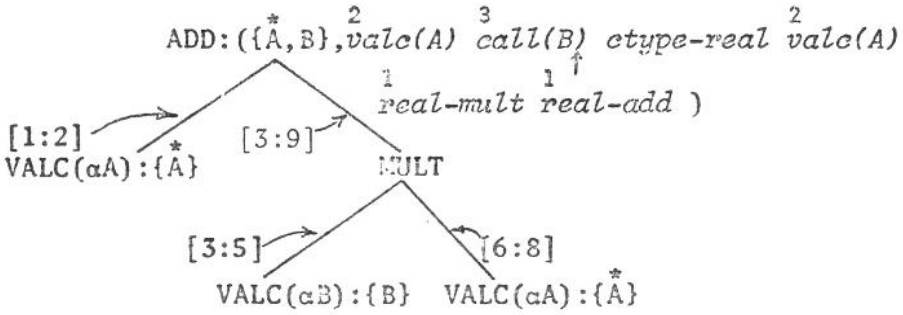


Figure 4C3C-4

The number above each code piece represents a postulated length for that piece. Since it is A whose semantics have changed, we have marked the dependency elements for A with an "*"; internally this would correspond to a validity bit having been set to zero (or FALSE).

Now, because of previous considerations we know that control must reside at a VALC node which produced code for a function call [VALC(αB) above]. It is also possible to detect exactly which terminal node this corresponds to using a simple tree search directed by the [FIRST:LAST] pairs on each arc. Thus, in the above, the control pointer is at position 5 of the code buffer, say. Therefore the right subnode of ADD is suspect, and MULT's left subnode, being terminal and responsible for the code from positions 3 to 5 inclusive, is identified as the culprit. Thus, when the "faked" interpretation reaches

discussion of REVERT. Scattered throughout this development are references to the above example, enclosed in brackets. These references supply concrete points of comparison for the more general algorithm being described.

The code string being executed at any one time belongs to some node and represents the code for the entire program whose parse tree is rooted at that node [the ADD node]. Now, if we have partially executed the code at that node, and wish to re-create interpretive control to the interpretation-point then the following problem arises. Under interpretation, the call hierarchy would probably indicate that we were somewhere in the middle of the routine for the topmost node [the ADD node] as well as any nodes which it had called and which had not returned [MULT in figure 4C3C-2]. We must get to that same situation without disturbing the environment of the user's program.

The statement which was invalidated had been partly executed; therefore we cannot simply reinterpret it from the beginning since that could easily affect the environment. We must be able to cause interpretation to start at the node to which the code is attached without actually executing anything until we have reached the interpretation-point. Two things are implied by this: (1) that we can detect when the interpretation-point is reached; and (2) that we can turn execution off and on at will. The second requirement has already been dealt with and poses no problem.

The first requirement is somewhat more difficult. It and the REVERT algorithm as a whole are best illustrated by the example of executing the code:

unlikely (indeed, undesirable) that one routine could be written which could allow for this variability and still re-create the interpretation-point corresponding to the execution-point in code previously created by those routines. For after all, the reason that this must be done is that the compiled code is invalid. If we had been interpreting the statement, at least one of the interpretive routines would have produced code which was different than that existing (the VALC(αA) nodes in the example will probably produce different code). In general, therefore, we make no assumptions about the way in which a routine produces code.

A situation under which REVERT may need to be used is the following: A function call occurs in a statement, and during the execution of that function, some semantic change occurs which invalidates the code to which control would normally return on completion of the called function.

Another possible circumstance might be a user-generated interrupt when the user wishes to suspend execution and regain control of the interaction. However, we restrict this interrupt to be processed only at (a) the end of a statement, (b) at the end of one iteration of a FOR-loop, or (c) at a function call. In this way, it is guaranteed that the user will regain control quickly but that the only way that an expression can be incompletely executed is if it called a function which has not yet returned a value. This is exactly what occurred in the example. The REVERT algorithm, then, is activated when a return to an invalid code string is attempted, and this is the only way in which it can be activated. We will now proceed to a detailed

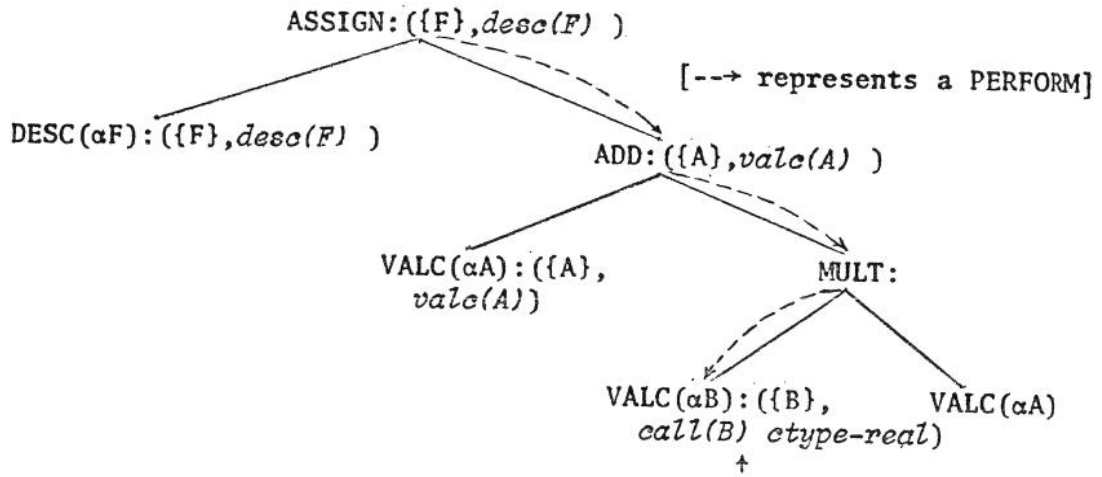


Figure 4C3C-2: The Interpretation-point for F ← A + B * A

VALC(αB) node in the code it produced to call the function B. The name of the REVERT game is to "redo" statement 1C2 interpretively, allowing the x-routines (ASSIGN, VALC, ADD, MULT) to pretend they are executing until the interpretation-point is reached. This will establish the call hierarchy and interpretive control without disturbing the program's environment. Then we will turn execution on and allow interpretation to continue from the interpretation-point as it would normally.

One of the prime constraints in re-creating interpretive execution from the execution of compiled code is that it be transparent to the interpretive routines involved. That is, the TFI-type routines need not be aware of this special mechanism and should only reflect the normal code creation/execution mode.

A second constraint which has a strong influence on this is that the routines to be used to reestablish interpretive control must be the normal TFI routines themselves. Each interpretive routine has its own way of making decisions and performing x-actions, and it is highly

IPS: N = 1
IPS: N = 0 (these lines typed by B)

At this point, assume that the user interrupts the system. The IPS responds with

IPS: INTERRUPTED AT LINE 1B

and gives the user control at his terminal.

Since we are at line 1B and N is 0, we must be about to return from B to the original incarnation of F which was called from line 1G. When control returns to F, the execution-point will be in the middle of line 1C2. Let the compiled code for line 1C2 be

desc(F) valc(A) call(B) ctype-real valc(A) real-mult real-add store
↑

Figure 4C3C-1: Compiled Code for F ← A + B * A

The execution-point is denoted by the "↑". We will change A's semantics while we have control at the terminal, and then allow the program to resume execution:

```
DECLARE INTEGER A; A ← 3;
GO
```
IPS: RUNNING FROM LINE 1B

A was previously assumed to be REAL; we have made it an INTEGER and told the program to continue. Now, when control returns from B to line 1C2, the code in figure 4C3C-1 will be invalid (because we changed A), the execution-point is as shown in figure 4C3C-1, and we have created a situation where REVERT is needed.

Assume we had been interpreting instead of executing compiled code: the parse tree for line 1C2 is shown in figure 4C3C-2 as if each node had its own code buffer. The interpretation-point is at the

since it includes the control involved in calling interpretive routines as well as executing any code produced. Nevertheless, these are meant as parallel notions, and when we speak of the execution-point and interpretation-point together, they are always assumed to be equivalent in terms of the user's program.

It should be pointed out that the situation with which we are dealing is not uncommon in an IPS. Since execution and text alteration can be interleaved, the user can interrupt his running program and, using direct statements, change the type or value, etc. of any variable. Consider the following short interaction (the program is essentially that given in figure 3B2-1). Lines typed by the user are indented, and those typed by the system or the user's program are preceded by "*IPS:*".

```
1  PROCEDURE PROG;
   1A  FUNCTION B;
      1A1 TYPE N;
      1A2  IF N ≠ 0
         1A2A  THEN B ← F
         1A2B  ELSE B ← 0
   1B  END
   1C  FUNCTION F;
      1C1  N ← N - 1;
      1C2  F ← A + B * A;
   1D  END
   1E  N ← 1;
   1F  A ← 1;
   1G    F;
2  END
PROG;  COMMENT : execute PROG; (this is a direct statement)
```

4C3C The REVERT Algorithm

In chapter 3 we showed how a TFI system could respond to semantic changes, as long as those changes occurred between the execution of statements. However, what about changes which affect a statement which is only partially executed and is executing compiled code? The compiled code is very myopic toward such changes and will simply perform as if these changes had not occurred. What is needed is a mechanism for rescuing the too inflexible compiled code from such predicaments.

If a statement is being interpreted (by a "normal" interpreter) when a change to some variable occurs, it can respond to that change by using the context exactly as it exists (change and all), rather than presuming some previous context — which is, of course, what the compiled code does. If we could cause the system to revert to interpretive mode when control attempts to return to the invalidated code, our problem would be solved; that is the approach to be used, and the controlling algorithm will be called REVERT.

We will define execution-point to mean the point which was next to be executed in the compiled code. Assume that we had been interpreting the same statement as was being executed. Then, the TFI would have been in the act of executing code created by an interpretive routine, and executing a piece of code identical to that at the execution-point in the compiled code when the change was discovered. This point we will call the interpretation-point ; it is equivalent to the execution-point insofar as the program being executed is concerned. The interpretation-point is a more complex concept than the execution-point

An example of the use of the EXECUTE_CODE option occurred in the discussion of the IF-statement. The code created for jumping around other code is not in reality executable while the statement is being initially interpreted since the code to be jumped to may not necessarily exist. Hence, the IF routine sets EXECUTE_CODE to false while generating the test and jump to ELSE code string. It does the same with the interp-call put into the code to cause interpretation of the THEN_CLAUSE at some later time. Other routines could require the creation and execution of code but wish not to save it: some simple immediate statements could be handled this way, hence the existence of SAVE_CODE.

The means by which a node acquires code from its subnodes requires clarification because not every node has a code buffer of its own anymore. A problem only arises when a node such as a statement node, which has its own code buffer, wishes to pass back only a call on its code rather than an actual copy of it. In that case it need only make the current code buffer what it was when the routine was entered and then use the CODE routine to place a comp-call into that buffer without executing it.

for correcting already compiled code when interleaved execution/alteration is allowed.

4C3B Generalizing the Code Routines

Before discussing the generalization of the TFI to handle dynamic execution/alteration interleaving, we need to describe a few necessary changes to the CODE routines which output and execute code as requested by the TFI routines. Indeed, we have already used some of these features in the discussion of the internal relative jumps above.

The CODE routine, as given previously, performs the following actions:

(1) move code to code buffer and substitute parameters into it;

(2) execute the generated code;

(3) change the buffer index so that further code will follow that just placed in it.

Now, the first action must always be done if either of the other two are to be performed. However, action (2) is not required in order to do (3) and (3) is not necessary unless the code is to be saved — indeed, if (3) were never done, the TFI would simply degrade into a tree interpreter.

We propose to make actions (2) and (3) optional and dependent on the values of two logical variables EXECUTE_CODE and SAVE_CODE. If EXECUTE_CODE is true then the code will be executed, otherwise it will not. Similarly, the code will only be saved if SAVE_CODE is true.

re-established later. In the case of variables on the dependency chain (meaning that they are references to this variable), apply case (a) to those entries and then use the SCOPE algorithm to attach the reference variable to the correct semantics entry and incarnation of the deleted variable.

Case (c):

Either there are code sequences in the scope of B using x_i or there are not. If not, then there is nothing to do. Assume, therefore, that code sequences using x_i exist but that they depend on another semantics entry for x_i. Now, we will check each code string,C, on the dependency chain from that semantics entry as follows:

(1) using n_C to start, search the parse tree in an upwards direction until a numbered node is encountered;

(2) if that number has an initial sequence which is the same as the level of the newly declared variable, then mark the code string as invalid and delete it from the dependency chain being searched.

(3) move to the next element on the dependency chain if there is one, and repeat this procedure from step (1).

Thus, we will have removed and invalidated all code strings affected by the added declaration. This completes the proof.

The theorem imposes a requirement on code strings, namely that they must contain,in a header, at least a reference to the parse tree node to which it is attached. This solves the problem of detection; the next two sections will give a method

those variables.

Let S_C be the list of dependency chain entries associated with C and let n_C be a reference (in the "head" of C) to the parse tree node which owns C.

Then S_C is sufficient to guarantee that if C is marked valid then the attribute-values of the variables in X have not changed.

Proof:

The semantics of a variable x_i can change in one, and only one of the following three ways:

(a) the values of some subset of the attribute-values of x_i in a name table entry referencing S_C are altered;

(b) a name table entry for x_i is deleted completely (this corresponds to a change in the scope of the variable);

(c) a new name table entry for x_i is created in some block B which previously had no x_i entry in the name table.

When a code sequence, C, is initially constructed, it is assumed that it is valid code.

Case (a):

This case is the most straight forward. Any change to the semantics of the variable will cause all items on the dependency list for that x_i to be marked invalid.

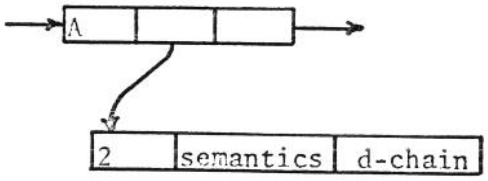
Case (b):

Proceed as in case (a). Since the addresses in any parse tree node which refers to that variable only points at the name table entry and not to the semantics entry, correct code will be

The program:

```
1  PROCEDURE F;
2  LOCAL A ← 0;
  2A  BEGIN
    2A2  A ← A + 1;
  2B  END
3  A ← A + 1;
4  END
```

has been entered into the system, and so the name table for the procedure F has an entry for the identifier A which looks like the following:



The dependency chain (d-chain) links together statements 2, 2A2, and 3.

Now, if the user types

```
2A1  LOCAL A;
```

this will cause another entry for A in the list hanging from its name table entry and will not affect the entry already in that list. But statement 2A2 is then incorrect and its code must be marked invalid if this change in scope is to take effect. We now give a constructive proof of a theorem showing how this can be handled.

Theorem

C is a sequence of code involving some set $X=\{x_1,x_2,\ldots,x_k\}$ of variables, and C is dependent on the attribute-values of

4C3 A TFI Which Responds to Dynamic Changes

We have described data structures for the naming of variables and ways of representing values which allows interleaving of program execution and semantic changes. It only remains to complete the specification of the TFI method in order to give it the ability to respond to semantic changes affecting active pieces of programs.

Before altering the TFI, however, we need to establish the sufficiency of the dependency chains and the naming structures described for detecting all the semantic changes which can affect a given statement.

4C3A Semantic Changes: Their Detection and Range of Effect

Each entry for a specific variable in a name table has a sequence of bit fields associated with it which are the attribute-values for that variable. Also a part of each name table entry for a variable is a reference to the head of a list of all the variables and code strings which depend on the present semantics of that variable. This chain is really a ring; i.e., the last element of the chain points back to the name table entry. Clearly, any change in the semantics of variable can then easily invalidate all those things which are dependent on it — except for a change in the scope of a variable, which may not directly affect that particular name table entry. Consider the following situation.

The indirect relative jumps are still valid since the β and γ pairs have been correctly altered.

It is not necessarily desirable that all the code from a program be generated into one single buffer since that would incur a large amount of overhead when inserting code into the buffer. A better compromise is to allow any node to possess its own code buffer instead of using its parent's. Then the code passed back to its parent is not a copy of that code buffer but a call on the code which that node owns: we shall denote such a call as a comp-call in what follows. Its counterpart which we have already used in the IF-statement example is a call on the interpretive routine for a node from a string of compiled code. The THEN-call in figure 4C2-2 is an example of an interp-call which is used to cause the THEN_CLAUSE node to interpretively execute. That particular interp-call was in fact created by the IF-node, but attributed to the THEN_CLAUSE simply by setting the value of the code position pair for the THEN_CLAUSE node to indicate that the code belonged to it.

One last item of note is the generation of backward relative jumps in order to implement loops, etc. They can be done in exactly the same way as forward jumps except that it is the FIRST-field of the code position pair addressed by the backward jump which must be used, and any increment or decrement from that position must also be specified as a field in the instruction. Both this and normal jumps only work if the beginning address of the executing code string is available since it is needed in order for the jump mechanism to work.

subnodes can be accomplished by using the LAST field for that subnode. In this way, even if the code from that subnode changes, the relative jumps will remain valid since the code position pair for the subnode will have changed. Indeed, this would allow us to replace the THEN-call code in the above example by the actual code generated whenever the THEN_CLAUSE was executed, without touching any of the other code in the buffer — simply by altering the code position pair for the THEN_CLAUSE node.

The tree before this transformation might be:

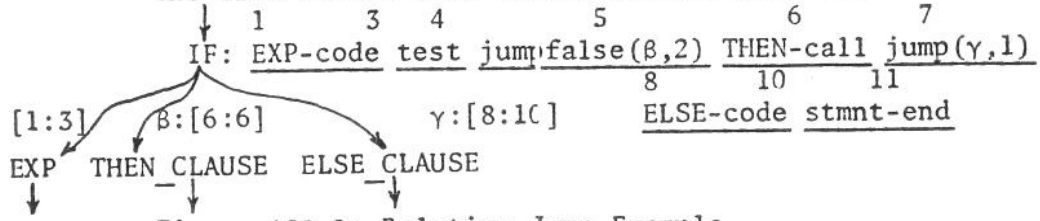


Figure 4C2-2: Relative Jump Example

β and γ represent the memory locations of the code position pairs for the THEN and ELSE clauses, respectively.

When the THEN-call is executed, the result is the establishment of a code buffer assumed to be indexed from 6 rather than 1. After the THEN_CLAUSE node has generated and executed the necessary code, the THEN-call will be excised from the original code buffer and replaced either by the actual THEN_CLAUSE code or a call on it. The code position pair for the THEN_CLAUSE might, therefore be [6:10] after this operation. In that case, the difference between the new LAST value (10 in this case) and the old LAST value for the THEN_CLAUSE must be computed (10-6=4) and added to the [FIRST:LAST] pairs for each sibling node to the right of the THEN-clause. The new tree would then be

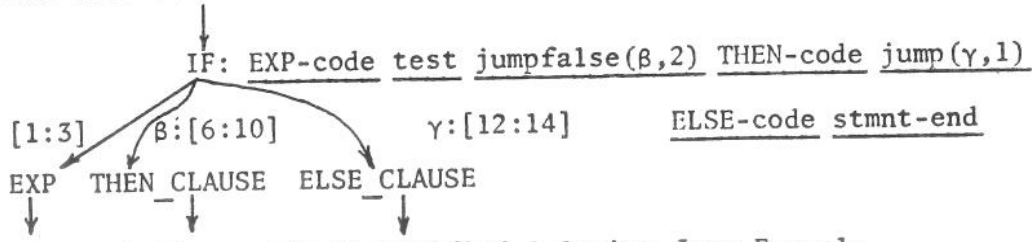
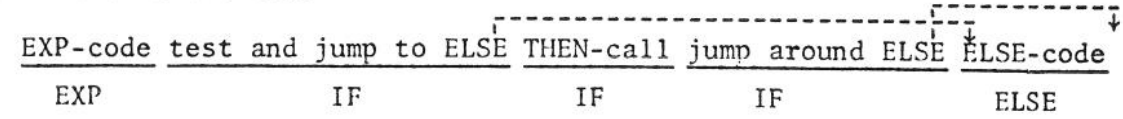


Figure 4C2-3: Modified Relative Jump Example

which is not necessary to the program's execution at that time. To make the code for the IF node logically complete, it places into the code buffer (without executing it) code to make an interpretive "call" on the THEN_CLAUSE should it be needed in the future.

So the IF node first executes and acquires the code from each of its EXP and ELSE_CLAUSE subnodes as well as its own code to jump to the ELSE code when EXP is false, and to call the THEN_CLAUSE node if EXP is true sometime in the future. The generated code then looks like



The names under the code descriptions indicate which node actually generated that code. The THEN-code, it should be noted, represents a case where the code was created but not executed.

The problem is that if we would like to replace the THEN-call by code generated by the THEN_CLAUSE when it is later executed, the relative jump to the ELSE-code created by the IF node will be incorrect. The same situation can also occur for the transfer of control which is used to bypass the ELSE-code since the ELSE-code itself may contain incompletely executed statements such as an IF-statement.

This problem is easily solved using the code position indices described in the last section. For each subnode, a parent node contains three pieces of information: a reference to the subnode, and the [FIRST:LAST] numbers for the subnode. Thus, any relative jump in the parent's code to bypass the code from one of its

of an IF-statement as shown below:

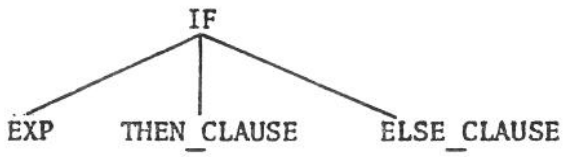


Figure 4C2-1: Tree for an IF-statement

Now, only one of the THEN_CLAUSE or the ELSE_CLAUSE will be executed the first time that the IF node is interpreted, depending on the value computed by EXP. Assume that EXP produces the value false the first time, and only the ELSE_CLAUSE is to be executed. We can do one of two things with the THEN_CLAUSE:

(a) let the TFI act as a compiler (this is described in section 4C4) and compile — but do not execute — the code for the THEN_CLAUSE; or

(b) place into the code buffer an operation which will call the THEN_CLAUSE to interpret should it ever be needed, but do not use the THEN_CLAUSE this time.

Version (a) if logically extended will cause complete compilation of an entire program whether or not it was all executed. But it is undesirable for the more important reason that the program may not even have a THEN_CLAUSE at this point in time. Recall that we wished to be able to execute incomplete programs as an aid in development and debugging of programs. Requiring the user to supply the THEN_CLAUSE when it is not even being used seems a strong concession to the implementation, at his expense.

We prefer method (b) since it allows code from the IF-statement to be logically complete, while not requiring the user to supply text

must still denote themselves as having created code which is dependent on the semantics of certain variables. However, a node such as MULT which has no code buffer associated with it and which does not directly access B or A need not denote itself as being on their dependency chains, since that information is easily deducible from its two subnodes. ADD is denoted as dependent on A and B because it has actual code associated with it; and the VALC nodes are dependent directly on the variables which they access, so they are placed on the dependency chains.

This manner of noting dependencies requires some extra work on the part of the routine which traverses a dependency chain when a variable's semantics are changed. Whenever a node is encountered which has no code buffer of its own but has produced code, all its parent nodes up to the closest node with a code buffer attached to it must also be marked invalid. This will keep the TFI's behavior as though each node had its own code buffer.

4C2 Simple Jumps and GOTO's

As mentioned previously, the jumps which are required in the code generated by a TFI, such as are needed in an IF-statement or looping statements, need to be relative jumps. By relative we mean relative to the beginning of the code string in which the jump code exists. This is necessary because of the manner in which code can change in the parse tree; i.e., it must be easily relocatable.

Nevertheless, even relative jumps are insufficient for the case