constituent statements had valid, complete code.

3D3B Recursive Function Calls

Since the actions of the code created by a node are being done as that code is being constructed, some side effects of that execution may be expected to occur. In particular, recursive use of a specific node which has not completed creating code at some previous level of recursion can cause a problem. The program given previously in figure 3B2-1 will illustrate the difficulty.

The first time that the statement

F ← A + B * A;

is executed, B is called by the VALC(αB) node and B, during its execution invokes F (but only once). So the expression A + B * A portion of the parse tree will be in the state depicted by figure 3D3-1 :

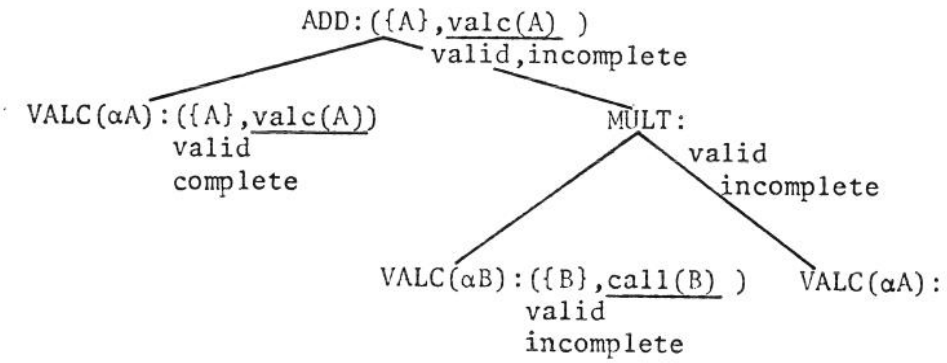


Figure 3D3-1: Partially Executed Parse Tree for A+B*A

Since the ADD node has incomplete code, that code is thrown away, and ADD begins blindly reinterpreting. VALC(αA), having valid, complete code will simply execute that code and return to ADD. MULT then meets the same fate as ADD and so,in turn does VALC(αB). VALC(αB) then causes B to be called recursively. This time B will return a value without calling F again, so the VALC(αB) node can finish at this level; it produces code "ctype(real)" to check that the value returned is of type real (for possible future executions). Execution then proceeds as in the previous examples.

But there is still the previous incarnation of B to deal with:

a problem: if the IF-expression has the value true, only the THEN-part should be executed, and if it has the value FALSE, only the ELSE-part should be executed. Since only execution causes compilation of code, if just the ELSE-part were compiled, then, were the THEN-part ever needed, it would not be available. And even if it could be acquired at that time, to acquire its code at the level of the IF-statement would require splitting the previously acquired IF- and ELSE-code to insert the THEN-code between them. This in turn would cause problems with control jumps in the code - such complicated solutions are not appealing! Also, because we are dealing with interactive systems, we would like to be able to execute incomplete programs, and such a program could be missing a THEN- or ELSE-part of an IF-statement. If that branch of the program were not needed until later in the development of that particular program, then attempting to "compile" it, even if it were not there, would rob the system of some flexibility since the user would necessarily have to insert it at that time.

Hence we must have an alternative which will allow a node not to acquire code from its subnodes if they have none (by reason of not having been executed), but which will allow the code to float up once it is all available.

Dividing the COMPLETE_CODE action into parts will accomplish this. The first action marks the code at the currently active node as complete. The second action is given explicitly by a non-terminal node and asks to acquire the code of a specific subnode, if it has any. Thus, the IF-statement could check whether the non-executing part (THEN or ELSE) had code. If it did, then the IF-statement node routine could ask to acquire all its sons' code sequences since they would then be available. A switch sent to CODE as a parameter could be used to mean "execute but do not store" so that the IF-statement could decide not to produce any code (which might be incorrectly acquired by its parent node) unless both the IF and THEN subtrees had themselves acquired code.

A construction such as "BODY" in the SLICE grammar would also have this property so as not to pass code up until each of its

MULT.2	PERFORM *2		VALC(αA):({A}, valc(A))
VALC(αA).0	since the code is valid, it is executed and control and code are returned to MULT		
		A ADD.3 B' A	MULT:({B',A}, valc(B)'valc(A))
MULT.3A	PERFORM CONVERT A ADD.3 B' A MULT.4		
CONVERT:	assume that CONVERT changes B' to be of same type as A using code convert and then returns;		
		$\underline{A}$ ADD.3 $\underline{B'}_{T_A}$ $\underline{A}$	MULT:({B',A}, valc(B)'valc(A) convert)
MULT.4	CODE(multiply top two stack items)	$\underline{A}$ ADD.3 $\underline{B'\ *A}_{T_A}$	MULT:({B',A}, valc(B)'valc(A) convert mult')
MULT.5	COMPLETE_CODE	$\underline{A}$ $\underline{B'\ *A}_{T_A}$	ADD:({A,B'}, valc(A)valc(B)' valc(A)convert mult')

For the example it was assumed that the value of B on the stack was converted to be the same type as A (i.e., the same type as B was previously), so ADD will produce the same code as it had before. Thus, the sections of the tree which might have required a change of action we reinterpreted, while those not affected by the semantic change simply executed their valid, compiled code. Such a feature has obviously high value is we picture one of the VALC(αA) nodes being replaced by an entire program block which escaped reinterpretation.

3D3 The TFI Method for Some Classes of Statements

This section simply contains a small collection of "implementation-type" problems encountered in implementing the TFI algorithm for some classes of statements.

3D3A IF ... THEN ... ELSE

Using the TFI algorithm for an Algol-type if-statement causes

stimulus: only code dependent on B will be reinterpreted. Any new code or dependency will be denoted in the following execution trace by a prime (') attached to the element. All other notation is the same as previously.

Executing Routine and Line	Operation	Stack	Tree Nodes
ADD.0	code is found invalid when ADD node is PERFORM-ed and is then discarded and reinterpretation is initiated by calling the ADD routine;		
ADD.1	PERFORM *1	ADD.2	VALC(αA):({A}, valc(A))
VALC(αA).0	code is found valid and is executed and control returns to ADD which acquires the code A		

Since the VALC(αA) node's code is valid, it is simply executed; reinterpretation is not necessary.

ADD.2	PERFORM *2	A ADD.3	MULT:({B,A}, valc(B)valc(A) mult)
MULT.0	code found invalid and discarded		MULT:
MULT.1	PERFORM *1	A ADD.3 MULT.2	VALC(αB):({B}, valc(B))
VALC(αB).0	code found invalid and discarded		VALC(αB):
VALC(αB).1	CODE(B to stack) - copy and parametrize code - execute code A ADD.3 MULT.2 B'		VALC(αB):({B'}, valc(B)')
VALC(αB).2	COMPLETE_CODE	A ADD.3 B'	MULT:({B'}, valc(B)')

For the first time in this process, the actions and result on the stack are really different than when the expression was originally interpreted.

the level of the ADD node, and will continue to rise and become imbedded in ever longer strings of code until some scheme causes it to halt (at the statement level for instance). This method of combining an FI with parse tree interpretation we will call a Tree Factored Interpreter or TFI.

The next time that the ADD node is to be executed, the compiled code is used if it is still valid. The manner in which validity is maintained and determined needs to be discussed. One of the primary data structures in an IPS is the symbol or name table, which associates the print name of a variable with information needed to obtain its value, and more importantly, semantic information about the variable. This semantic information can be composed of quantities such as type (e.g., real or integer), scope (e.g., local or global), structure (e.g., array or scalar), and so on. It is the constancy of this information over a period of time which allows the code produced by an FI or a TFI to be re-executed.

One way of determining code validity is to scan all nodes of all parse trees to find any code which depends on some variable, whenever the semantic information for a variable is changed, and to mark such code invalid. This will work in simple cases, even though terribly inefficient, and is presented here only as an example of a method for checking the validity of compiled code. An efficient and complete algorithm is developed in chapter 4.

3D2 Movement Between Interpretation and Compilation

One of the points previously made about the TFI was that it allowed a reasonably smooth flow between interpretation and compilation. The interpretation to compilation direction has been demonstrated, and we will show here what happens in the opposite direction when some semantic change forces reinterpretation of previously compiled code. The algorithm is efficient and reinterprets only what has been (potentially) affected by the semantic change.

We will assume that the semantics of B are changed and show how the previously interpreted tree for A + B * A responds to this

For the first time, the calling routine (MULT) has a non-empty dependency set and a non-empty code buffer. The subnode's dependencies {A}are united with its parent's dependency set {B} to get {B,A}; and VALC(αA)'s code valc(A) is concatenated to the MULT node's code buffer. The next action by MULT is to produce the code to multiply the values computed by its sibnodes.

MULT.4	CODE(multiply top two stack items) - copy code into buffer - execute code	A ADD.3 B*A	MULT:({B,A}, valc(B)valc(A) mult)

Not only has MULT completed its action at this point, but it also has compiled code to do the actions of its entire subtree without reinterpretation. If any node in the tree is executed and all its subnodes (or subtrees) acquire valid compiled code by executing, then that node can acquire that quality also. In this way, if all the program and its variables were to remain semantically constant (at some level of semantics such as the types of variables, as opposed to their values, for instance), then the executed portions of the program would have compiled code for future executions. It is this which represents the automatic movement from interpretation to compilation. Movement in the opposite direction will be described later. The evaluation continues with MULT returning code to its caller, ADD:

MULT.5	COMPLETE_CODE	A B*A	ADD:({A,B}, valc(A)valc(B) valc(A)mult)
ADD.4	CODE(add top two stack items) - copy code		ADD:({A,B}, valc(A)valc(B) valc(A)mult add)
	- execute code	A+B*A	
ADD.5	COMPLETE_CODE - at this point the ADD node's caller can acquire its code.		

Finally the code for the entire expression A + B * A has floated up to

as is A above. At this point ADD, by invoking its *1 node (see figure 3B4-2 for a diagram of the parse tree for A + B * A) using a PERFORM action, has specified that any code produced by that node is also to be placed in the ADD node's code buffer when control returns to ADD. The code is passed back to ADD when the VALC node invokes COMPLETE_CODE:

VALC(αA).2	COMPLETE_CODE	A	ADD:({A}, valc(A))

COMPLETE_CODE does the following:

(1) mark the code of the currently active node (each node has its own code buffer) as valid and complete (these are two separate flags associated with each node);

(2) if this routine was invoked by a PERFORM, then add its code to that of the calling routine, and unite its dependency set with the calling node's (in the above case, all that happens is that ADD gets VALC's dependency set {A} and its code valc(A));

(3) return control to the calling routine, passing back any value left on the stack (A in this case).

ADD.2	PERFORM *2	A ADD.3	MULT:
MULT.1	PERFORM *1	A ADD.3 MULT.2	VALC(αB):
VALC(αB).1	CODE(B to stack)		
	- copy and param-etrize code		VALC(αB):({B}, valc(B))
	- execute code	A ADD.3 MULT.2 B	
VALC(αB).2	COMPLETE_CODE	A ADD.3 B	MULT:({B}, valc(B))

At this point, MULT acquires the code and dependency set of its *1 subnode, VALC(αB)

MULT.2	PERFORM *2	A ADD.3 B MULT.3	VALC(αA):
VALC(αA).1	CODE(A to stack)		
	- copy and param-etrize code		VALC(αA):({A}, valc(A))
	- execute code	A ADD.3 B MULT.3 A	
VALC(αA).2	COMPLETE_CODE	A ADD.3 B A	MULT:({B,A}, valc(B)valc(A))

```
ADD:
      1  PERFORM *1;
      2  PERFORM *2;
      3  IF TYPE_OF (TOP_OF (STACK)) ≠ TYPE_OF (SECOND_OF (STACK))
         3A  THEN PERFORM CONVERT;
      4  CODE(add top two stack values and replace them by their sum);
      5  COMPLETE_CODE;

MULT:
      1  PERFORM  *1;
      2  PERFORM  *2;
      3  IF TYPE_OF (TOP_OF (STACK)) ≠ TYPE_OF (SECOND_OF (STACK))
         3A  THEN PERFORM CONVERT;
      4  CODE(replace top two stack items by their product);
      5  COMPLETE_CODE;
```

PERFORM is a special form of routine call used to activate the interpretive routine specified in a subnode or a specific interpretive routine (such as CONVERT above); it will become more important in the next chapter.

We will again trace the execution of the parse tree for A+B*A, also showing the run-time stack and the level of interpretive control. Only the changes being made to the node which is active are displayed under the heading Tree Nodes. Execution begins at the ADD node.

Executing Routine and Line	Operation	Stack	Tree Nodes
ADD.1	PERFORM *1	ADD.2	ADD:
VALC(αA).1	CODE(val A to the stack); - copy code and parametrize - execute code return control to VALC node	ADD.2 A	VALC(αA):({A}, valc(A))

Now the VALC(αA) node has attached to it the dependency set {A} and the code string valc(A); only compiled code will be so underlined, and anything on the stack which is a value will be doubly underlined (A)

It is impossible to prepare (and execute) code such as

c(jump ahead) c(some other code) (place to be jumped to)

which occurs in the Algol if-statement, without a mechanism to stack "labels" as is done in most compilers. Backward jumps are easier if the interpretive routine can be sure that the code being branched to will not disappear. The problem of goto's (as used in Algol), in an FI environment such as we are suggesting, is sufficiently difficult that we choose to ignore it until some fairly sophisticated mechanisms have been developed in the next chapter.

Another drawback of an FI is that statements whose scope extends over more than one line (we are assuming that individual lines are alterable entities) are difficult to execute. Our next extension of the FI concept to interpretation on parse trees will provide a more elegant solution to the problem.

3D Extending an FI to Parse Tree Interpretation

Combining the notions of interpreting a parse tree with the FI method yields some very interesting phenomena. Among these are a smoother flow between interpretation and compilation, minimal interpretive overhead when responding to semantic changes, and easy extension to classical compilation.

3D1 The TFI Algorithm

The explanation of the mating of the FI concept and parse tree interpretation is best explained by redoing the A + B * A example given previously, with the following (slightly altered) ADD, MULT, and VALC routines to include the code bracket concept of the FI. The Algol-like notation used is an extension of the SLICE language, which is given in Appendix A of this chapter.

```
VALC(αA):
        1  CODE(push the value of A onto the stack);
        2  COMPLETE_CODE;
```

act of interpreting the program; thus, it provides a simple connection between one type of interpretation and compilation. Secondly, statements are changeable, and this action may be interlaced with execution of the program, the only restriction being that a statement which is being executed may not be changed. The ability to change programs and the data on which they operate while those programs have been executing and using the data we will call dynamic changeability. Its counterpart, static changeability is limited to changes which can be done only when the programs and data concerned are not being used or are not in any stage of execution. The ability to suspend program execution by some form of user interrupt and during that suspension alter program or data structures - while still maintaining the ability to resume the program - is a good example of dynamic changeability.

If each statement for which code has been created during interpretation has a list of the variables in its dependency set, then, whenever the semantics of a variable are changed, one could scan the entire list of statements and mark as invalid all those having code which depends on the old semantics of that variable. If no statements depending on that variable are active, then semantic changes to variables can be interlaced with execution in an FI system. Also, except for a truly active statement, i.e., one which is being executed of which is in the hierarchy of statements awaiting completion of a called routine, any statement can be changed. Hence, there is a modicum of dynamic changeability of programs inherent in what has been described so far.

With this method, then, we could handle a system in which variables did not change type, for instance, at arbitrary times during execution, but which would nevertheless allow typeless or undeclared variables as are allowed in JOSS, LC^2, and APL. As well, the program could be changed even during program execution, except for active statements. The BASIC system described earlier fits these constraints, and could be implemented in such a manner.

There are some other problems with an FI which is operating from a linear pseudo-code such as postfix.

Basically, the idea is that the actions associated with a statement, once interpreted successfully, can be kept for future execution. There are some obvious restrictions on this scheme and they are largely concerned with the type of semantic changes allowed. Clearly, since no check is made of the validity of the code for a given statement, any changes in the semantics of the variables which that statement uses could mean that the code is not correct. Nevertheless, such a scheme could be useful in a conversational Fortran system, for instance, in which variable declarations (e.g., REAL A(20)) were not alowed to be altered, but in which statements could be changed. Undefined variables (variables which have not been assigned a value before they are used) could be checked, and if no error occurred during interpretation, code could be created to access the variable from that time on. In fact, a variant of this idea has been suggested for a Fortran system with debugging capabilities [Mit 69].

Since such "interpretively compiled code" is dependent on variables which it uses, some means of denoting (and implementing a check for validity with) this dependency will be necessary if the code is to respond to semantic changes in variables. If there were some sequence

$$p_1\ p_2\ \cdots\ p_n$$

of interpretive pseudo-code (postfix for example) and the code for it, created by an FI is

$$c(p_1)\ c(p_2)\ \ldots\ c(p_n)$$

which in turn depends on the semantics of a set $V=\{v_1,v_2,\ldots,v_k\}$ of variables (hereafter called a dependency set), then we need at least a mechanism which can decide that unless V is semantically changed,

$$c(p_1)\ c(p_2)\ \ldots\ c(p_n)$$

can be considered as valid code, and take some action if V is changed. We shall denote such dependencies by simply describing interpretively compiled code as a pair (V,C), where V is the dependency set of variables on which the code C depends.

3C2 Incremental Differential Compilation from an FI

The FI method has some interesting, though transparent characteristics. Firstly, the code which is compiled is created during the

for each statement to point to the compiled code, if any, and a bit in each entry to specify whether or not the code for that statement is valid.

we are dealing with s-actions which contain no x-actions within them and conversely. Thus, any action carried out and required as part of the execution of an interpreted program is reflected entirely by the sequence of x-actions performed during that execution.

This last remark is crucial. If each use of an interpretive routine for some specific operation of the source program causes a different sequence of x-actions to occur, then interpretation is certainly always necessary. For that could only happen if semantic changes occurred which changed the meaning of the particular operation of the program each time that it was executed. As has been pointed out previously, such behavior for all the parts of a program is exceedingly rare: entire programs do not constantly change even in the most interactive of environments.

For purposes of explication we will assume that all the parts of an interpretive routine not marked by "code brackets" are s-actions. And each time an x-action is to be performed, a routine called the CODE routine is invoked. This routine has as parameters the limits and position of the x-action code; and the place being interpreted in the source program is known as a global semantics variable. We will assume two levels of interpretation so that another routine, INITIALIZE, is invoked just before each statement is to be interpreted, and a routine called COMPLETE_CODE is invoked after each statement is completed. X-actions may need to have addresses and values substituted into the code produced, much like macro substitution, and we therefore need some means of specifying those parameters. For this initial outline we will assume that such a mechanism exists and will not describe it in any detail.

The algorithms for INITIALIZE, CODE and COMPLETE_CODE are given in "pseudo-Algol" in Appendix 3B. All that is required in terms of data structures for these routines is a pointer in the data structure

actually cause part of the interpreted program to be executed. We will distinguish these two classes as semantic actions (s-actions) and execution actions (x-actions).

Figure 3C-1 is a simple representation of the execution of a semantic routine in a system such as FSL. S-actions are denoted by directed line segments and horizontal bars delimit x-actions. It is important that this diagram is also representative of the execution of an interpretive routine in which semantics are checked dynamically possibly depending on execution parameters, and that in this case the code within the code brackets is executed instead of just saved for later execution. Normally such distinctions are buried in the code of interpreters (and compilers also). We will call an interpreter in which x-actions are factored out or made distinguishable from s-actions a Factored Interpreter (FI).

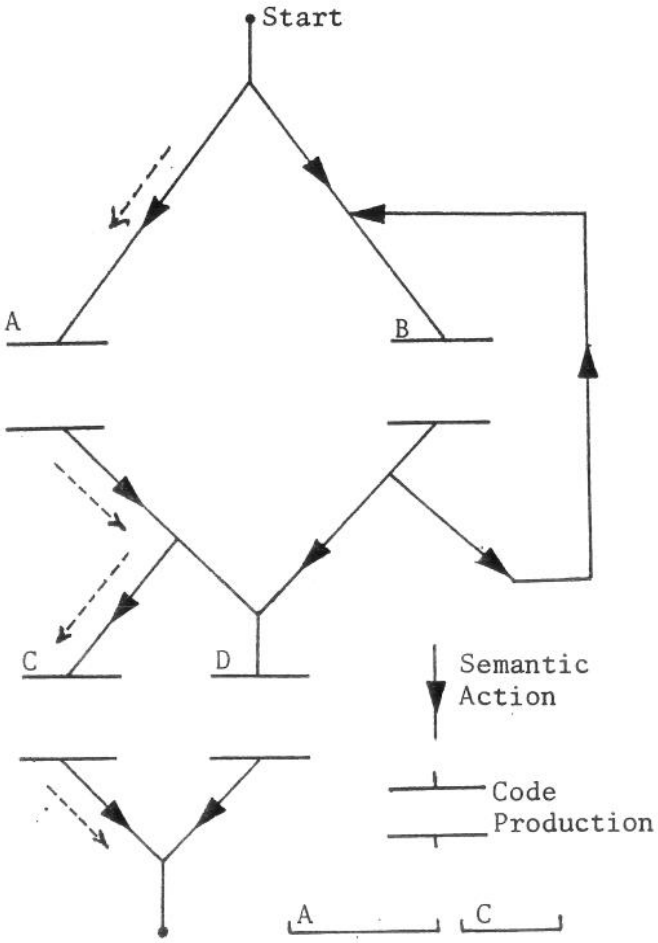


Figure 3C-1: Schematic of the Execution of a Semantic Routine

3C1 Using an FI for Both Interpretation and Compilation

The introduction of factored x-actions provides a link between interpretation and compilation. Since s-actions do not themselves affect any action on the part of the program being interpreted, they can have no direct effect on the program or its data. If a presumed s-action does cause a change in the control or data environment of the program which affects its execution, that s-action is factorable into a sequence of smaller s- and x-actions. Hereafter we will assume that

between compilers and interpreters.

As discussed earlier, interpreters are mainly used when the semantic content of a program can vary during execution due to changes in variables, meanings of operators, or changes in the program text itself. However, the purpose of debugging some representation of an algorithm is to fix the program and data which describe it in such a way that the program is a valid statement of that algorithm over some domain of data. It is reasonable therefore to expect that more and more of a program remains constant as it approaches correctness. This is not always the case since there are some algorithms which may modify themselves, but even in these cases there is probably some portion of the program which is constant, namely, the part that is modifying the rest of the program. If we could so arrange matters that constant parts of the program become compiled - in the sense that R is performed on them only when they vary, and remains fixed when they and their associated semantics remain fixed - then we would have a connection between the spectral extremities of interpretation and compilation.

In compiler systems, the semantics R is decomposable into two parts: semantic analysis and code production. That is, some set of semantic information is used by R to determine which of a variety of explicit actions is required to perform an operation (for instance, an add). The semantic portion of R may also access and change a class of variables not connected with the execution of the program but only with its translation; these are normally called compile-time variables. The object code for the selected explicit actions is then created by a (possibly) different set of constructs in the compiler. In FSL [Fe 64] "code brackets" surround object code which is to be created. This code is parametrized by using compile-time variables to contain the addresses of run-time quantities. Indeed, one of the main contributions of FSL was the realization that many of the same facilities are necessary at both compile and run-time: the code brackets simply qualify the time at which an action is to occur. The same distinction can be made in an interpreter between actions which are done solely for the sake of interpretive control and information and those which

each routine then handles control by deciding whether and in what order its subnodes should be "executed" by calling them. And not only is control distributed throughout the tree, but each X-routine such as ADD also has a fair bit of structural syntactic and semantic information locally available through its subnodes.

3C Factored Interpretation

A compiler system for programs in a language L on a machine M can be viewed as having the following components:

(1) a syntax analyzer, or parser, P, for the strings of L into some intermediate language L_p (i.e., $P: L \rightarrow L_p$);

(2) a set of semantic routines, R, which maps L_p into a run-time language L_R;

(3) a set of run-time support routines E which together with M can execute L_R programs on M (and therefore can execute L programs on M).

Now, let $S=S_1;S_2;\ldots;S_n$, $S \varepsilon L$ (the S_i are the statements in S). Then, for a compiler system, the execution of S on M can be described as

$$\begin{aligned} & E\ (\ R\ (\ P\ (\ S\ )\ )\ ) \\ &= E(R(P(S_1;S_2;\ldots;S_n))) \\ &= E(R \circ P(S_1;S_2;\ldots;S_n)) \\ &= E(\ R \circ P(S_1);R \circ P(S_2);\ldots;R \circ P(S_n)) \end{aligned}$$

That is, we can distribute the language translation process (or compile-time as it is commonly called) over the program, after which E is applied to the translated program. Here we are speaking of E as the hardware and software needed to execute an L_R program.

The same is not generally true in an interpretive system where P may be pre-applied to S, but R is applied at the same time as E; i.e., the process is grouped as

$$E \circ R(P(S_1);P(S_2);\ldots;P(S_n)).$$

Thus, the number of times R is applied to S_i in an interpreter is proportional to the number of times that S_i is executed, whereas R is applied only once to each S_i in a compiler system. It is this simple distinction which accounts for the difference in run-time efficiency

Now, let us trace the execution of the parse tree in the same manner as done previously for its construction, noting the order in which operations are performed. The stack used to hold intermediate values can also be used to record the control in the parse tree, i.e., the calls made on subnodes. The notation "name.k" in the stack stands for the information necessary to return control to the routine "name" at line k when the called routine returns.

Executing Routine and Line	Operation	Execution Stack
ADD.1	call *1	ADD.2
VALC(αA).1	place value of A on the stack	ADD.2 A
VALC(αA).2	return value	A
ADD.2	call *2	A ADD.3
MULT.1	call *1	A ADD.3 MULT.2
VALC(αB).1	place value of B on the stack	A ADD.3 MULT.2 B
VALC(αB).2	return value	A ADD.3 B
MULT.2	call *2	A ADD.3 B MULT.3
VALC(αA).1	place value of A on the stack	A ADD.3 B MULT.3 A
VALC(αA).2	return value	A ADD.3 B A
MULT.4	multiply	A ADD.3 B*A
MULT.5	return value	A B*A
ADD.4	add top two stack items	A+B*A
ADD.5	return value	A+B*A

The order of operations, listed linearly, is

VALC(αA) VALC(αB) VALC(αA) MULT ADD

which is just the Polish postfix for the expression: thus, this method of using parse trees as the control for an interpreter can mimic interpretation of postfix notation. It is important also to note that only a single stack is necessary to maintain interpretive control and the values being computed. In fact, the interpreter's F-level really does nothing except call the routine indicated by a certain node, and

3B5 Interpreting Parse Trees

We previously showed three methods of program interpretation: from source text, lexically scanned text, and parsed text (postfix). One can also interpret a parse tree of the type in figure 3B4-2 above.

Consider each node of the tree, whether terminal or non-terminal, as the name of an interpretive routine to be called to accomplish the action required by the node which contains its "name." Each semantics routine called has access to its subnodes (if any) as the names *1, *2, etc. from left to right. Considering just the parse tree in figure 3B4-2 for the expression A + B * A, we can describe the action of each of the routines named in that routine in the order of execution, starting at the root of the tree, the ADD node.

ADD:

1. call *1 to execute;
2. call *2 to execute;
3. check the top two values on the stack, performing any necessary conversion to make their types compatible;
4. add the top two stack values and replace them by their sum;
5. return the value on the top of the stack as the value of the ADD routine at this node.

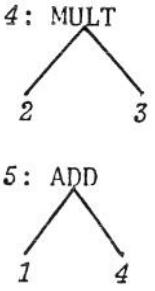
VALC(αA):

1. push the value of A onto the stack (the value is obtained by using the parameter αA in the nodal information);
2. return with the top of stack item as the value of the node.

MULT:

1. call *1 to execute; (i.e., the node VALC(αB))
2. call *2 to execute; (i.e., the node VALC(αA))
3. check the top two stack items and perform any necessary conversion;
4. replace the top two stack items by their product;
5. return the top of stack value as the value of the MULT routine

Rule and Depth	Action	Stack [head→]	Tree
EXP			
TERM			
FACTOR			
PRIMARY	.ID	αA	
	:VALC	αA VALC	
	[1]	*α1*	*1*: VALC(αA)
FACTOR			
TERM			
EXP			
EXP			
TERM			
FACTOR			
PRIMARY	.ID	*α1* αB	
	:VALC	*α1* αB VALC	
	[1]	*α1 α2*	*2*: VALC(αB)
FACTOR			
TERM			
FACTOR			
PRIMARY	.ID	*α1 α2* αA	
	:VALC	*α1 α2* αA VALC	
	[1]	*α1 α2 α3*	*3*: VALC(αA)
FACTOR			
TERM	:MULT	*α1 α2 α3* MULT	
	[2]	*α1 α4*	*4*: MULT
EXP	:ADD	*α1 α4* ADD	
	[2]	*α5*	*5*: ADD



Thus the final parse tree for A + B * A is

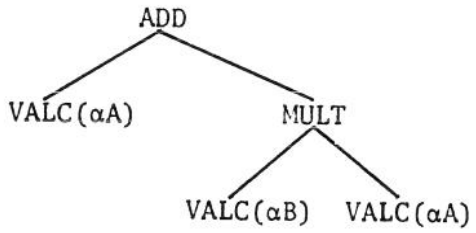
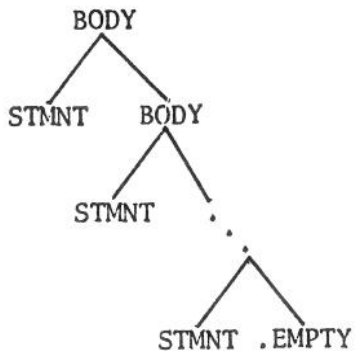


Figure 3B4-2: SLICE Parse Tree for A + B * A

and we will make extensive use of this in the remainder of this chapter as well as the next.

instead of:



Since the "$" is acting very much like a counting mechanism anyway, we allow ".$" to be used to request counting of the repetitions and allow "$" to be used in the output brackets. Thus, the BODY rule becomes

```
BODY = STMNT .$( ";" STMNT ) :BODY[1+$] / .EMPTY :BODY[0];
```

The output brackets are written as [1+$] to indicate that the first STMNT (which is not in the range of the ".$") is to be output as a subnode as well.

Thus the SLICE rules

```
FACTOR = "-" FACTOR :NEGATE[1] / PRIMARY;
PRIMARY = .ID :VALC[1] / .NUM :LITC[1] / "(" EXP ")";
```

applied to the string "-A" would produce the following stack transformations and tree creation:

stack	tree
αA	
αA VALC	
α1	*1* : VALC(αA)
α1 NEGATE	*2* : NEGATE
	1 : VALC(αA)
α2	

The rules for EXP, TERM, FACTOR, and PRIMARY (in figure 3B3-1) together applied to "A + B * A" would result in the following sequence of actions:

by α identifier) onto the stack.

(2) A simple ":name" construct places "name" on the stack.

(3) The construct [n] (where n is a numeric constant) creates a tree whose root is the item on the top of the stack, with n items as that node's subnodes, in order from right to left from the second item on the stack down; then (n+1) items are popped from the stack and a reference to the created tree replaces them.

(4) A [1] following a .ID in a rule will cause the address of the symbol table address for the identifier to be placed <u>in</u> the created node itself rather than making an extra node. Thus, in the PRIMARY rule (figure 3B3-1),

```
.ID :VALC[1]
```

will produce a node such as

```
VALC( α identifier)
```

and <u>not</u>

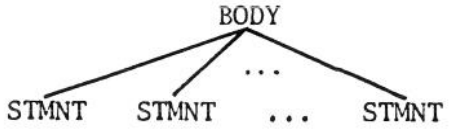
```
VALC
 |
α identifier
```

as would normally be the case with such a construct.

(5) Lastly, we generalize the [n] construct to allow the creation of trees whose number of subnodes is determined by the use of the grammar. This situation arises when one has a rule such as

```
BODY = STMNT $( ";" STMNT ) / .EMPTY;
```

and would like to make each of the STMNT's a direct subnode of the body node instead of a right-recursive binary tree: i.e.,



The allowable segments, or partial statements, in the SLICE grammar are defined by the non-terminals PROG, FDEF, STMNT, BODY, THENST, ELSEST, FNAME, HEAD, and TAIL. Hence, the SLICE program

```
BEGIN
     IF A = B THEN A ← B +
       C;
END
```

is invalid because an EXP such as B + C cannot be split across lines.

3B4 Sample Parse Tree for a SLICE Statement

In this section we will show how the parser works on the statement F ← A + B * A, which is statement 1C2 in the sample program in figure 3B2-1. The parse tree for this statement is the following, generated from the SLICE grammar above:

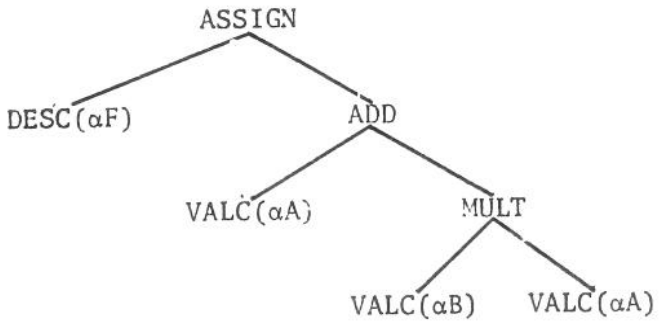


Figure 3B4-1: Parse Tree for F ← A + B * A

The MULT node was created in the TERM rule of the SLICE grammar using the Tree-Meta construct :MULT[2] which means the following:

> output a tree which has as its root node, MULT, whose right subnode is the item on top of the stack, and whose left subnode is the item immediately below the tope of the stack. Then pop the stack twice, and push a pointer to the created tree onto the stack.

Items are placed on the stack in five ways:

(1) A lexical routine such as .ID places a reference to a symbol table entry for identifiers which it collects (denoted

these operations, of course, require some care in writing the incremental grammar, but few complications arise if the set of potential dangling non-terminals is reasonable. Also, these methods require that certain nodes of the parse tree be marked as being associated with a line of text - they are exactly the mated nodes. It is also reasonable to keep the text for the lines attached to the mated node for that line, and we will assume this hereafter.

3B3 SLICE: A Small Language for Interpreter/Compiler Examples

In order to have some specific base for many of the algorithms which follow, we have constructed a grammar for a language called SLICE and have described it in Tree-Meta. In fact, we shall extend SLICE throughout the next chapter in order to deal with some more difficult problems of an IPS. This extension will be manual for the most part, although we will later touch upon some problems posed by extensibility in an interactive environment.

```
PROG = HEAD FDEFS BODY TAIL :PROG[4];
  HEAD = "BEGIN" :BEGIN[0];
  TAIL = "END" :END[0];
  FDEFS = .$( FDEF ";") :FDEFS[$];
    FDEF = "FUNCTION" .ID :FNAME[1] ";" BODY TAIL :FDEF[3];
  BODY = STMNT .$( ";" STMNT) :BODY [1+$] / .EMPTY :BODY[0];
    STMNT = (( .ID ":" .$( .ID ":") :LABELS[1+$] SIMPST :LBSTMNT[2]
           / SIMPST ) :STMNT[1];
      SIMPST = ASSIGN / EXP / IFST / COMPOUND / GOST;
        ASSIGN = .ID :DESC[1] "←" EXP :ASSIGN[2];
        IFST = ( "IF" EXP "THEN" ( STMNT :THENST[1]) "ELSE"
                 ( STMNT :ELSEST[1] ) ) :IFST[3];
        COMPOUND = HEAD BODY TAIL :COMPOUND[1];
        GOST = "GOTO" .ID :GOTO[1];
        EXP = TERM ( "+" EXP :ADD[2] / "-" EXP :SUB[2] / .EMPTY );
        TERM = FACTOR $( "*" FACTOR :MULT[2] / "/" FACTOR :DIVD[2]);
        FACTOR = "-" FACTOR :NEGATE[1] / PRIMARY;
        PRIMARY = .ID :VALC[1] / .NUM :LITC[1] ? "(" EXP ")" ;
```

Figure 3B3-1: The SLICE Grammar