AN APPROACH TO INCREMENTAL SEMANTICS

by

Gregory F. Johnson

Computer Sciences Technical Report #547

July 1984

AN APPROACH TO INCREMENTAL SEMANTICS

by

Gregory F. Johnson

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN -- MADISON

1983

AN APPROACH TO INCREMENTAL SEMANTICS

Gregory F. Johnson

Under the supervision of Associate Professor Charles N. Fischer

The purpose of this thesis is to examine the problem of making updates to the static semantics of a program in response to small, incremental changes. The primary application of this work is in language-based text editors; after every editing operation a user performs on his or her program, an internal representation of the semantics of the program is modified in some minimal way so that any semantic errors can be reported immediately. The method studied here provides a new approach to the problem of incremental semantic evaluation in that attributes may flow directly to where they are needed, rather than being restricted in their flow to paths in the parse tree of a program. The communication paths take up space in memory and require time to construct, so their use involves trade-offs. Attribute evaluation processes somewhat different from those discussed in the literature are presented which allow for incremental updates and non-local attribute communication.

Acknowledgements

That this thesis has been completed is a tribute to the support and love I have received from many people. Most especially I would like to acknowledge and thank my wife Margaret. The motivation and intellectual leadership provided by my advisor, Dr. Charles N. Fischer, have been a constant help. My committee, Drs. Marvin Solomon, Sam Bent, Raphael Finkel, and Michael Meyer, were generous in their investment of time and effort to improve the quality of this thesis. Financial support for this research was provided by the National Science Foundation. Finally, I would like to thank the many friends who have come into my life at St. Paul's University Catholic Center for guidance and support in the many areas of life that fall outside of the realm of the academic world.

Table of Contents

Chapter 1 - Introduction

The recent computer science literature has contained numerous papers about a new concept in software development, the Language Based Editor, or LBE [1, 2, 3, 4, 5, 6]. An LBE is similar to standard text editors in that it has capabilities to create files, perform insertions, deletions, and other standard editing operations, but it differs from normal editors in that it possesses knowledge of the syntax and semantics of a particular programming language. With this knowledge, the text editor can notify a user immediately if an attempt is made to create an illegal program. Much attention has been focussed on the syntactic aspects of program editors, or more specifically on the context-free aspect of syntax analysis. Comparatively little attention has been directed at the semantic and context sensitive aspects of LBE's. Demers, Reps, and Tietelbaum [7] have suggested the use of attribute grammars [8] as an appropriate basis for the semantics of program editors. Reps [9] continued this work and presented an attribute evaluation technique tailored to incremental changes in an attributed parse tree. Skedzeleski [10] studied time-varying attributes, in which attributes may be evaluated more than once in the attribute evaluation process, and non-local attributes, which are attributes that are functions of attributes not localized within a single production. Although Skedzeleski's work was not developed for LBE's, it is similar to the approach presented here.

It has been our experience that pure attribute grammars have some shortcomings in the incremental update environment found in an LBE. Objects that are closely related in a semantic sense and which require frequent exchange of semantic information can be arbitrarily far from each other in the syntax tree describing the program. In order to alleviate this problem, we make use of 'non-local productions.' In the classic attribute evaluation framework, direct attribute

transmission may only occur across parent-child or child-child syntactic relations. We will extend this model to allow dynamic construction (at evaluation time) and dynamic use of relations over which attribute flow may take place.

The problem of how best to evaluate the attributes of a semantic tree has received much attention [11, 12, 13, 14, 15]. Previous work in attribute grammars has assumed the following sequence of operations:

```
parse;
eval_attributes;
```

or perhaps [16]

```
repeat
  parse partially;
  evaluate_as_much_as_possible;
until done;
```

We add a third phase to the latter scenario:

```
repeat
  parse partially;
  repeat
    evaluate_as_much_as_possible;
    update_flow_links_if_necessary;
  until done;
until really_done;
```

The update_flow_links phase bears some resemblance to parsing in that we are modifying the compound dependency graph [8, 17] of the program. It is also intimately related to the attribute evaluation process, since these links may themselves depend on attributes in a partially evaluated parse tree.

This work was motivated by the the University of Wisconsin Pascal Oriented Editor (POE) [18]. The semantic analysis that POE performs makes extensive use of context sensitive attribute flow. By far the most important application of the above ideas is to link the defining occurrence of an identifier with its uses. The example given at the end of Section 2 illustrates the specification of these

linkages. Also of importance is the use of attribute flow links to represent precedence relations in expressions.

1.1. Attribute grammars

An "attribute grammar" [8] is an extension of the context free grammar formalism which captures context sensitive requirements (we use the term "context sensitive" here in the loose sense to refer to compile-time checks of a program's correctness which are beyond the scope of a parser) by associating with each production a set of "attribute evaluation functions." Each node in a parse tree may have associated with it a set of values, or "attributes," which are the results returned by execution of such functions. A parse tree that has attributes associated with its nodes is called a "semantic tree". Evaluation functions take as their inputs attributes that already exist in the production instance with which they are associated. Some evaluation functions require no inputs, and some depend solely on informations that is pre-initialized in the tree before the evaluation process begins. For convenience we will lump these two categories together and say that the attributes output by these evaluation functions have in-degree zero. This term is suggested by the fact that if we drew a graph representing the dependencies among attributes in a semantic tree, the nodes representing these attributes would have no incoming arcs. Each attribute evaluation function has associated with it an indication of which symbol in its corresponding production is to receive its result. It is convenient to distinguish between attributes that are to be associated with the parent of a production, which are called "synthesized attributes" in the literature, and attributes that are given to children in the production, which are called "inherited attributes." We will also classify attributes as "input" and "output" attributes. The attributes that appear in the input list of some evaluation function in a given production will be

called input attributes to the production. Similarly, attributes that are created by some evaluation function in a production will be called output attributes of the production.

If the input and output sets of every production in a non-circular [8] attribute grammar are disjoint, we will say that the grammar is in "normal form". In the case of an attribute grammar that has no context sensitive relation sets, this definition is equivalent to the definition of normality given in the literature in terms of synthesized and inherited attributes [13]. In the remainder of the paper we will restrict our attention to attribute grammars in normal form.

1.2. Non-local attribute grammars

In an attributed parse tree derived from a standard attribute grammar, a given node may participate in at most two production instances; it may be a child in one production instance and a parent in another production instance. The root and leaves participate in only one production. In our new attribute grammar construction, we will allow a given node to participate in an arbitrary number of production instances. In addition to being a parent and a child, it may participate in several instances of "non-local productions". A non-local production is an association of nodes in a parse tree which may directly communicate attribute values among one another. As noted in the previous section, attribute evaluation rules are associated with productions. We allow parse tree nodes which are not adjacent in the parse tree to be grouped together into instances of non-local productions, and we associate attribute evaluation rules with such groupings. For instance, if the grammar has a non-local production that associates <use id>'s with their <defn id>'s, the <defn id> will participate in as many non-local productions as there are uses of the identifier for which it is the definition.

The construction of a non-local production may depend on attribute values of symbols that are far apart in the parse tree of a program. For instance, a non-local production may be established between two "identifier" terminals provided that they have the same value of the "name" attribute. Or, the construction may require that the symbols in an instance of a non-local production bear some context-free syntactic relationship to one another. For instance, Skedzeleski's non-local attributes allow communication between a node and any node between itself and the root of the parse tree. A non-local production might be the ordered set "<defn_id> - block", an evaluation function might be "put the <defn_id>'s name in the symbol table associated with block", and the grammar might require that <block> be the occurrence of the <block> non-terminal that is first encountered moving from the <defn_id> toward the root of the tree.

1.2.1. An example of a grammar that contains non-local productions

As an example of the above formalism, we present in Figure 1.1 an attribute grammar describing a simple language which has two data types and assignment statements.

S ::= Defs ; Uses	Uses.symtab := Defs.symtab;
Defs ::= <type> id_{def} , Defs	def.type := <type>.type; $Defs_1$.symtab := AddId($Defs_2$.symtab, id.STRING_REP);
Defs ::= epsilon	Defs.symtab := empty;
<type> ::= CHAR	<type>.type := character;
<type> ::= INT	<type>.type := integer;
Uses ::= Use ; Uses	$Uses_2$.symtab := $Uses_1$.symtab; Use.symtab := $Uses_1$.symtab;
Uses ::= epsilon	
Use ::= id_{use} := id_{use}	":=".error := use_1.type ≠ use_2.type; id_1.use := MakeLink(Use.symtab, id_1.STRING_REP); id_2.use := MakeLink(Use.symtab, id_2.STRING_REP);
def - use	use.type := def.type;

Figure 1.1.

If we constructed an LBE for this simple language, a user could change the defined type of a symbol, and the change would flow through the def_id straight to each use_id. It would not be necessary to re-evaluate every use_id in the entire program to see which ones were actually affected by the change. It is not uncommon, for example, for a programmer to write the statement part of a Pascal program first and then go back and create variables, types, and constants. This style of programming implies frequent re-definition of defined types of symbols.

1.3. Construction and operation of an evaluator

We will present a modification of the Kennedy and Warren planning approach [14] which is especially useful in an LBE environment. We will first present the new approach in the context of attribute grammars without non-local

productions; the problems of attributing a semantic tree from scratch and of incrementally updating a fully attributed tree that gets changed will be considered separately. Finally, the changes necessary when a grammar includes non-local productions will be discussed.

It turns out that there are several nested "evaluability classes" of attribute grammars which allow progressively simpler and more efficient incremental evaluators. We will present classification methods which can be used to determine the best evaluation algorithm for a given attribute grammar. The situation may be compared to that of context-free grammars: as we progress from the most general, unrestricted grammars through unambiguous grammars and the various sorts of LR(k) grammars, we have a sequence of progressively simpler and more efficient parsing and parser generation techniques.

1.3.1. Planning and evaluation for unattributed trees

We first consider the problem of evaluating all the attributes of an unattributed parse tree. Following the general approach of [14], we first construct a set of evaluation plans. This step is analogous to the use of a parser generator - an attribute grammar is given as input to the planner, and the planner creates a set of tables that will be used to perform evaluations of semantic trees. For each production in the grammar, the planner constructs a set containing all attribute instances in that production. For every collection of attributes which can become available together in a production in the course of evaluation, we construct a list of evaluation functions which can be invoked given that that collection of attributes is available. Thus, if in a semantic tree we happen upon a production instance with a set of available attributes, we know which new attributes to evaluate.

As a parse tree is being built, we do not do any attribute evaluation. We do, however, include in an "evaluate this production" set any production instance that the parser builds which has attributes with in-degree zero. When the parse tree has been completely built, we start the evaluation process. During evaluation, we maintain with every production instance a flag indicating which attributes are available in it. The evaluation process consists of an iteration of the following operations:

(1) select a production instance and evaluate attributes in it.

(2) Update flags of production instances to reflect newly evaluated attributes.

The above approach is useful only in a situation in which a representation of the entire program is available before the evaluation process begins. An LBE is, of course, such an environment. As will be seen below, the above approach lends itself to the problem of incremental parsing and to the case of attribute grammars that contain non-local productions. It also has the attractive property that no absolute non-circularity restrictions exist. Absolute non-circularity is basically a restriction on how divergent the attribute dependencies of a given node may be for the various subtrees that may be created below it. In the approaches taken by [14] and Cohen and Harry [11], the problem of absolute non-circularity must be addressed either at evaluator generation time or at evaluation time.

1.3.2. Incremental update of an attributed tree

In an LBE, as contrasted with a compiler, a user can make small, incremental changes to the semantics of a program. From an attribute grammar viewpoint, we have a fully attributed tree that is changed, perhaps slightly, and we want to avoid re-evaluating the entire tree. We would like to re-compute only the attributes that become invalid because of the change. This is the problem addressed by [7] and [9]. We will consider only operations involving the

substitution of one subtree for another. As pointed out in [7], insertions, deletions, and replacements can all be thought of as subtree replacements. We have the following situation: two fully attributed trees are to be merged. A node of the first tree which has the same grammar label as the root of the replacement tree is stripped of its subtree, and the replacement tree is attached in its place. The resulting tree must then be examined and perhaps partially re-evaluated.

The approach to be presented requires a minimum of information to be kept at each node in the attributed tree. A flag, which could be encoded as an integer, is all that is required. We follow the general principle that work is best done once at generation time, rather than being done repeatedly at evaluation time. Rather than manipulating graphs or other complex data structures at evaluation time, we perform look-ups in tables.

1.3.3. Evaluation and update in the presence of non-local productions

We now consider the changes that must be made to the above procedures when we are interested in attributing a parse tree or incrementally updating an attributed tree when the language describing the language contains non-local productions. The scheme can be summarized in the following way: when a subtree is replaced by another subtree, we first sever all connections between it and the parse tree. We then "wire in" the new subtree to the parse tree, connecting it through its root and also through a series of non-local productions. Finally, we incrementally update the attribute values in the new semantic tree. In POE a new subtree is connected through its root and also through every occurrence of an identifier. The first time a new subtree is put in place, the "wiring in" operation will involve an amount of work that is roughly equivalent to a normal evaluation of the subtree. After the subtree is in place, however, changes in other parts of the program which affect the new subtree can take place much faster, since the

leaves of the subtree are now "near" to other parts of the tree which originate semantic information that they need.

Chapter 2 - Efficient Linear Incremental Evaluation

2.1. Introduction

We present in this chapter a plan-oriented approach to incremental update in the setting of conventional attribute grammars. We will follow the definitions and statement of the problem as discussed in Demers, Teitelbaum, and Reps [7], and Reps [9]. The approach presented in this chapter is an alternative to that presented to the same problem in the two above-mentioned papers. Our approach will follow the Kennedy-Warren [14] philosophy of performing fairly extensive analysis of attribute dependencies at generation time in order to reduce the complexity of the evaluation task. In effect, we "compile" the attribute grammar into a set of plans, and the plans then drive a small, efficient evaluator.

The remainder of this chapter will be organized as follows: first an extension Knuth's IO graph construction will be mentioned that was first suggested by Katayama [19]. An algorithm will be presented for computing "upper" IO graphs, or as Katayama calls them, OI graphs. Several sets of assumptions and definitions that depend on the "OI graph" concept will follow.

Next, a simple initial attribute evaluation algorithm will be presented and proved to be correct. Initial evaluation is a crucial first step in anticipation of incremental evaluation; before we can perform a series of editing operations which require incremental evaluation, we must begin with a fully attributed parse tree. Moreover, a large parse tree fragment may have to be constructed in response to the input of a single token by the user, and such a fragment must be initially evaluated before it can be patched into the rest of the parse tree using incremental evaluation methods. Use of plans will be sketched in conjunction with the initial evaluation algorithm.

After the discussion of initial evaluation, a general algorithm for incremental re-evaluation will be presented for the case of a change in a single "global input" attribute whose evaluation function takes no inputs. An example of such an attribute might be the actual numeric value associated with an "integer" terminal node. The algorithm we present will be quite general: it will be valid for any non-circular attribute grammar. Next, a modification of the algorithm will be presented which allows incremental re-evaluation after a subtree has been deleted and replaced by another subtree. This result will be found to be as powerful as that of Reps, but will not require evaluation-time manipulation of graphs. The algorithm presented here is plan-oriented, depending on fast table look-ups.

2.2. Calculation of possible dependencies among attributes

The particular parse tree in which a given non-terminal is imbedded creates a partial order among the attributes of that non-terminal. The partial order represents the set of functional dependencies among the attributes of the non-terminal. The graph representing the part of the order generated by the subtree below the non-terminal is called the IO graph by Kennedy-Warren [14] and Katayama [19], and the characteristic graph or inferior characteristic graph by Cohen and Harry [11] and Reps [9]. It seems that the former term is becoming slightly more standard, and so we will adopt it. Similarly, the part of the partial order generated by the rest of the parse tree (that which would remain if the non-terminal's subtree were removed) is called the OI graph or superior characteristic graph. Again, we adopt the former terminology. Nevertheless, when referring to the set of all IO and OI graphs considered together, we will call them characteristic graphs.

An example of an attributed parse tree is shown in Figure 2.1, and Figures 2.2 and 2.3 show the IO and OI graphs generated at node 8 by the attribute

dependencies of the tree.

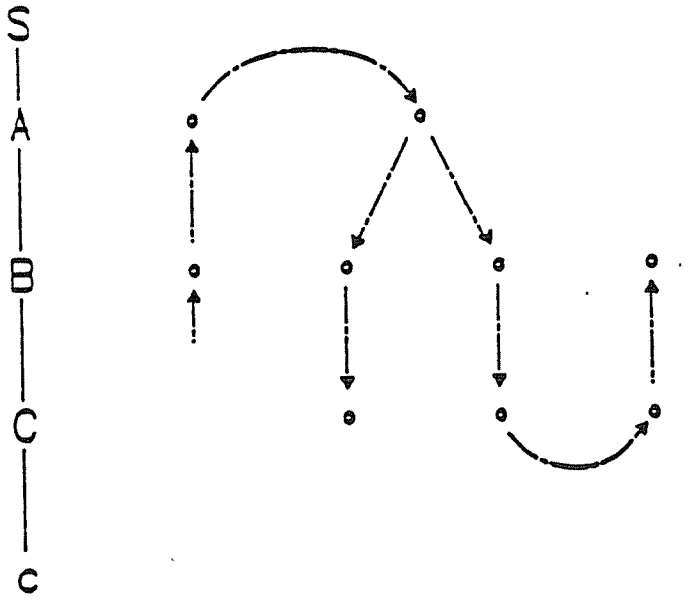

Figure 2.1.
A parse tree and associated attribute dependencies.

Figure 2.2.
The IO graph of the attributes of B induced by the parse tree in Figure 2.1.

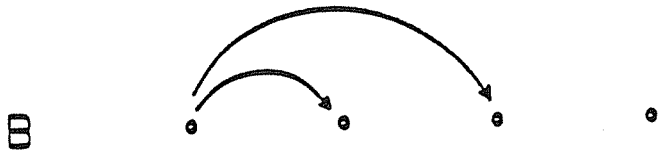

Figure 2.3.
The OI graph of the attributes of B induced by the parse tree in Figure 2.1.

In Knuth's original papers [8, 17] on attribute grammars he presented an algorithm to compute the finite set of possible IO graphs for every symbol in a given attribute grammar. We present below an extension of his algorithm which calculates both IO graphs and OI graphs for all symbols of a given grammar.

The proof that the algorithm produces the desired output is analogous to the proof found in Knuth; before presenting the algorithm and its proof we start with a few definitions.

Since we will be discussing IO graphs and the symmetric concept of OI graphs together, it will be helpful to extend the concepts of terminal symbols and exterior nodes to incorporate a corresponding symmetry. We refer to a grammar symbol as an "extremum" if it appears only in right-hand sides or only in left-hand sides of productions in a given grammar. For a reduced grammar which follows the convention that the start symbol appears in no right-hand side, the extrema are the terminals and the start symbol. In the sequel we will only consider grammars which have been given an extra production S' ::= S, if necessary, to insure that the start symbol is an extremum. We will analogously extend the concept of "exterior node" of a tree to "boundary node," by which we mean the root or a leaf node of the tree.

The "dependency graph" of a production has as its nodes the attributes of that production instance. There is an arc from attribute alpha to attribute beta if the value of alpha is an input to the evaluation function that computes beta. Evaluation functions and the dependency graph for the production "B ::= C" in Figure 2.1 are presented below.

```
B ::= C                          B.atr1 <- 47;
                                 C.atr2 <- B.atr2;
                                 C.atr3 <- B.atr3;
                                 B.atr4 <- C.atr4;
```

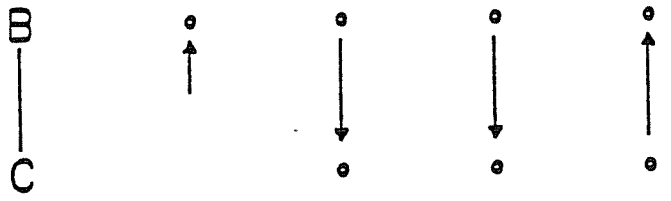

Figure 2.4.
A production and associated dependency graph.

The "compound dependency graph" of a parse tree fragment (some of whose boundary nodes may be non-extrema) has as its nodes the attributes of the various nodes in the parse tree fragment. The arcs of the compound dependency graph are determined by functional dependencies among the attributes of the parse tree. Intuitively speaking, there is an arc from attribute alpha to attribute beta in the compound dependency graph if alpha is an argument to the procedure which computes beta. More formally, if beta is a synthesized attribute of the parse tree node X, we look at the set of evaluation functions associated with the production which corresponds to node X and its children; if beta is an inherited attribute of X we look at the evaluation functions of the production corresponding to the parent of X and its children. In either case, alpha and beta are associated with identifiable nodes of the production, and we can examine the evaluation functions of the production to see if alpha is an input to the evaluation function

that computes beta.

Following Reps [9], we will define characteristic graphs in terms of two ancillary concepts: projections and what we choose to call near-complete parse trees.

The "projection" of a compound dependency graph on a particular parse tree node is the graph whose nodes are the attributes of the tree node; there is an arc from attribute alpha to attribute beta if there is a path from alpha to beta in the compound dependency graph of the parse tree which does not include any attributes of the node being projected upon. In the case of normal attribute grammars, a projection is just the subgraph of the transitive closure of the compound dependency graph which has as its nodes the attributes of the parse tree node being projected upon.

We define a "near-complete" parse tree to be a parse tree fragment in which exactly one of its boundary nodes is not an extremum. A near-complete parse tree may be either a fragment all of whose leaves are terminals but whose root is not a left-hand extremum, or a tree whose root is a left-hand extremum which has a non-terminal as one of its leaves. Near-complete parse trees are an extension of the concept of "derivation trees of type p" which is defined in Knuth [8].

The "characteristic graph" generated by a given near-complete parse tree at its non-extreme boundary node is defined to be the projection of the near-complete parse tree's compound dependency graph onto the non-extreme boundary node. Characteristic graphs are extremely useful for a variety of attribute grammar analyses. As mentioned above, they are the key concept in Knuth's original paper for non-circularity testing. Beyond that, they have been used for various sorts of evaluator plan creation and attribute grammar classification schemes.

Algorithm 2.1 computes all possible characteristic graphs for symbols in an attribute grammar.

```
Algorithm 2.1.
Input: an attribute grammar .
Output: the set of IO and OI graphs for each symbol in the grammar

{initialization; see Note 1.}
for those productions P which have only one non-extremum, do begin

  (say the non-extremum is at position "i" of the production)

  G := the projection of P's dependency graph on symbol "i";

  if i = 0
  then add G to the IO graph set of symbol "i"
  else add G to the OI graph set of symbol "i";

end for;

{main loop; see Note 2.}
while there remains an unexamined 3-tuple
    ( production P,
      position "i" in P,
      {characteristic graphs C_j for symbols of P
      other than the symbol at position "i"}
    )
do begin

  let D be the dependency graph for P augmented by the selected
  characteristic graphs;

  G := the projection of D on position "i";

  if i = 0
  then add G to the IO graph set of symbol "i"
  else add G to the OI graph set of symbol "i";

end while;
```

Note 1: Unless we are considering a trivial grammar which describes the empty language, there must be at least one production in the grammar whose right-hand side either consists entirely of terminal symbols or is epsilon. All such productions will be examined during the initialization step. Further, productions that have extrema as their left-hand symbols will be examined in the initialization step if at

most one of their right-hand symbols are non-terminals.

Note 2: As the algorithm progresses, it builds up sets of IO graphs and OI graphs for each grammar symbol. At the start of the "while" loop, we look for a production P and a particular collection of characteristic graphs for symbols of P that have not yet been examined together. For instance, if after the initialization step of the algorithm there is a production with non-terminals in its right-hand side all of which received IO graphs during initialization, then we can select that production, position zero within the production, and an element from the IO-graph set of each of the right-hand non-terminals of the production. Indeed, that production and the sets of IO graphs generated for its right-hand non-terminals by the initialization step could be used for the first $\prod_{i=1}^{n} C_i$ iterations of the "while" loop, where C_i is the number of characteristic graphs created by the initialization step for the i^{th} right-hand non-terminal of the production.

We now present and prove the desired theorem about Algorithm 2.1.

Theorem 2.1. A graph is generated by Algorithm 2.1 if and only if it is the characteristic graph of some near-complete parse tree derivable from the grammar.

Proof. We first prove that every graph produced by the algorithm is the characteristic graph of some near-complete parse tree. Certainly the assertion is true after initialization; the initialization step considers only those productions which are by themselves near-complete parse trees. We now show that the claim is an invariant for the main loop of the algorithm. Assume that the claim is true before an iteration of the loop. At the start of that iteration, a production instance "P", a position "i" in that production, and a set of characteristic graphs for symbols other than the one at position "i" are selected. By assumption, each of

the characteristic graphs selected has a corresponding near-complete parse tree. During the iteration, we create a new characteristic graph C for the symbol at position "i". A near-complete parse tree which corresponds to C can be created by taking an instance of production "P" and substituting for each characteristic graph used in C's creation a near-complete parse tree that corresponds to it. At the end of the loop, therefore, the claim still holds.

Conversely, if P is a characteristic graph generated by some near-complete parse tree at a symbol N, then the algorithm will have generated P. To see this, assume the contrary, that there is a near-complete parse tree whose characteristic graph is not among those produced by the algorithm. We know that the near-complete parse tree is not a single production, since its characteristic graph would have been generated by the algorithm during initialization. Thus, there must be at least two productions in our near-complete parse tree. In fact, one of the productions bordering the production containing N must also have a symbol with a characteristic graph not generated by the algorithm, since that is the only way the algorithm could have missed one of the characteristic graphs of N. We can, then, trim off the production containing N, consider the second production instead, and repeat the whole analysis, proving that it also must be in a near-complete parse tree with at least two production instances. This process clearly requires that there be an infinite number of production instances, contradicting the fact that every parse tree (including the one assumed to exist at the outset of this discussion) is finite.

□

We see, then, that the algorithm presented at the beginning of this section can be used to analyze any attribute grammar and determine all possible sets of dependencies among attributes of each symbol in the grammar.

An example of the execution of Algorithm 2.1 will now be presented and discussed in some detail.

(1) S ::= E	E.symtab := (("pi", 3.142), ("e", 2.718)); S.value := E.value;
(2) E ::= T	T.Symtab := E.Symtab; E.value := T.value;
(3) E ::= E + T	E_2.Symtab := E_1.Symtab; T.Symtab := E_1.Symtab; E_1.value := E_2.value + T.value;
(4) T ::= id	T.value := LookUp(T.Symtab);
(5) T ::= number	T.value := StringToFloat;

This grammar describes sums of floating point numbers; for the sake of illustration we allow the symbols "pi" and "e" to appear also. The "SymTab" attribute created in production (1) is a list of (name, value) ordered pairs. In this example, if a sentence has an identifier other than "pi" or "e", that is an error. The extrema of the grammar are the symbols "S", "+", "id", and "number."

In discussing the example, various graphs will be presented. As an illustration of the notation to be used,

a -> b UI -> d e

signifies a graph with five nodes labeled "a", "b", "UI", "d", and "e", with arcs from "a" to "b" and from "UI" to "d", and with no arcs from or to node "e." It will be convenient to include with characteristic graphs an indication of which attributes are inputs and which are outputs, so a dummy node "UI" (for "universal input") will be used as a predecessor of output attributes which have no inputs. (An input attribute is a synthesized attribute in an IO graph or an inherited attribute in an OI graph. An output attribute is an inherited attribute of an IO graph or a synthesized attribute of an OI graph.) Nodes which have no incoming or outgoing arcs can be

understood to represent input attributes which are not used in the computation of any other attributes.

During the initialization step of Algorithm 2.1, productions (1), (4), and (5) will be used to create characteristic graphs, since they are the only productions in the grammar which have no more than one non-extreme symbol. Production (1) is used to create the OI graph

E.value UI -> E.symtab

for grammar symbol "E," and productions (4) and (5) are used to create the IO graphs

T.symtab -> T.value

and

T.symtab UI -> T.value

respectively for the symbol "T."

After initialization, the "while" loop of Algorithm 2.1 can choose from among five alternatives. One alternative is to select production (2), position one, and the OI graph created for symbol "E." Production (2), position zero, and either of the IO graphs for symbol "T" account for two more alternatives. Finally, production (3), position one, the OI graph for symbol "E" and either of the IO graphs for symbol "T" account for the final two alternatives. Say production (2), position zero, and IO graph

T.symtab UI -> T.value

are selected. We take the dependency graph of production (2), the graph

E.symtab -> T.symtab T.value -> E.value,

and append to it at position one the selected IO graph. The resulting graph is

UI -> T.value E.symtab -> T.symtab T.value -> E.value.

Taking the transitive closure of this graph and then trimming out nodes and arcs other than those involving position zero (the symbol "E"), Projecting this graph on node "E" gives us the IO graph

E.symtab UI -> E.value.

Now that an IO graph is available for symbol "E," production (3) can be used either at position zero to try to find other IO graphs for "E," or at position 3 to compute an OI graph for symbol "T." After the algorithm has made its selection at this juncture, has gone through an iteration of the loop, and has continued the process until no more choices remain, IO and OI graphs will have been found for all symbols in the grammar. These IO and OI graphs are enumerated in the following table.

symbol	IO graphs	OI graphs
S	UI -> S.value	
E	E.symtab -> E.value E.symtab; UI -> E.value	UI -> E.symtab; E.value
T	T.symtab -> T.value T.symtab; UI -> T.value	UI -> T.symtab; T.value

2.3. Determination of applicable characteristic graphs at evaluation time

Several papers ([11,9,19], for instance) address the problem of determining during evaluation which of perhaps several possible characteristic graphs actually describes the attribute dependencies at each node of a given parse tree. We will adopt here the procedure suggested by Katayama [19], which will neatly reduce the general evaluation problem to consideration of "uniformly simple" attribute grammars, i.e. attribute grammars for which every symbol has exactly one IO graph and one OI graph. We form for every symbol X in the grammar a corresponding set

of augmented symbols X_{ij}), where i and j are respectively indices of the possible IO and OI graphs of X. Then, for any particular parse tree we simply re-label each node X with the appropriate X_{ij}).

In fact, we can extend the concept of Katayama and consider the indices i and j to be attributes. The algorithm given at the beginning of this section to compute characteristic graphs can be modified to provide the evaluation rules for i and j: each characteristic graph created by the initialization step is viewed as an attribute with in-degree zero whose value depends only on which production of the grammar was used to create the characteristic graph. Each characteristic graph created by the "while" loop has added to the evaluation function of its corresponding attribute an indication that when characteristic graph attributes at positions other than "n" have values corresponding to those selected in the "while" loop, the node at "n" is to be given a characteristic graph attribute whose value is the index of the characteristic graph created for "n."

It can be shown (the machinery to do so will be developed in a later chapter) that an attribute grammar created by the above method has a particularly simple structure, and can be initially or incrementally evaluated using any of the methods to be described in this chapter. So, when an editing operation is performed which changes the parse tree, we evaluate the characteristic graph index attributes first, and then for further attribute analysis we view the parse tree as having been generated from a uniformly simple attribute grammar.

2.4. Initial evaluation

In this section we present and demonstrate correctness of an initial evaluation algorithm. It is in essence the one used to perform initial evaluation in Poe. First, however, some necessary terms:

In the following discussion, we will use the terms "successor" and "predecessor" to indicate attributes that immediately depend on and are immediate ancestors of a given attribute. The terms "descendant" and "ancestor" will be used to refer to the "transitive closures" of the successor and predecessor relations; in a given attributed tree an attribute is an ancestor of another attribute iff there is a directed path from the first attribute to the second in the compound dependency graph. Also, when we say that attribute alpha depends on attribute beta, we mean that alpha is an descendant of beta.

The "universal input" attribute is an imaginary attribute that is the sole immediate predecessor of all attributes whose evaluation functions take no arguments. Use of this artificial construct turns out to simplify and consolidate notation. Formally, we have the following definition: the "universal input attribute" has the following properties:

(1) it is always available (i.e., any attribute that depends only on the universal input is immediately evaluable);

(2) any attribute that has the universal input as an immediate predecessor has no other predecessors;

definition: a "back path" is a function "f" from the positive integers to the nodes of a directed graph such that for all $i > 1$, there is a directed arc from node f(i) to node f(i-1); if for some i f(i) has no incoming arcs, then f(j), $j > i$ is undefined.

We can now make the following definition about evaluable attributed trees in terms of the universal input attribute:

definition: an attributed tree is "evaluable" if every back path contains the universal input attribute.

definition : an attribute grammar is "well-formed" if every attributed tree that may be constructed from it is evaluable.

We note that well-formedness implies non-circularity: if an attribute grammar is circular, then there is an attributed tree with attribute dependencies that form a directed cycle, and a back path can be formed whose range consists entirely of the elements of the directed cycle.

Algorithm (presented below) is the promised initial evaluation algorithm.

```
Algorithm 2.2.
Input: a well-formed attributed tree
Output: the same tree, with all of its attributes evaluated

for each production instance P in the parse tree that
has output attributes with in-degree zero, put P in Evaluable; (see Note 1.)

while Evaluable is not empty do begin
  select and remove an element E from Evaluable;

  evaluate attributes in plan( E ) (see Note 2.);

  for each production P that has a node N in common with E, do begin
    P.available := P.available + { newly available attributes of E at N};
    if plan( P ) is non-empty, put P in Evaluable;
  end for;
end while;
```

Note 1: We associate with each production instance a set called "available" which indicates which attributes of that production have been computed so far. At the outset, each production instance has in its available set only the universal input attribute. Attributes with in-degree zero are those which depend only on the universal input attribute, and hence they are immediately evaluable. This initialization step can actually performed as the parser is constructing the parse tree.

Note 2: The "plan" for a production instance with a given available set is simply the set of unavailable output attributes of that production all of whose inputs are available.

Algorithm 2.2 terminates, since there are a finite number of attributes in T, we evaluate each one at most once, and we evaluate at least one attribute during each iteration of the "while" loop.

To show that algorithm 2.2 correctly evaluates all the attributes of T, we first prove two lemmas.

Lemma 2.1. A production instance is in "Evaluable" if and only if it has evaluable output attributes.

Proof. The lemma is certainly true after initialization: all direct descendants of the UI are by definition evaluable, and production instances for which they are outputs are in "Evaluable." Conversely, in an evaluable attributed tree the graphis dominated by the UI, and hence by the set of its direct descendants. Since no direct descendant of the UI has been evaluated, no non-direct descendant is evaluable.

Now, we establish that the property mentioned in the lemma is an invariant of the "while" loop: a production instance can change in only two ways inside the loop: it can be removed from the "Evaluable" set and have its evaluable attributes evaluated, or it can have a neighbor evaluate an attribute of a node shared by the production instance and its neighbor. In the first case (by normality), after evaluable attributes are evaluated, that production instance has no more evaluable attributes, and hence does not belong in Evaluable (by normality, the evaluation step cannot change any inputs to the production being evaluated, and hence cannot make formerly unevaluable outputs evaluable.) The inner "for" loop examines every production instance that shares a node with the production being evaluated, and so no production instance that may experience a change in evaluability will go unexamined. After a neighbor has been examined, it will be in Evaluable if

(1) it was in Evaluable before being examined or

(2) it has newly evaluable attributes.

In the latter case we are done; in the former case we are done also, since no output attribute of a production instance can be rendered unevaluable by the process of making previously unavailable input attributes available.

□

Lemma 2.2. If there is an unevaluated attribute then there is an evaluable attribute.

Proof. If the set of unevaluated attributes is non-empty, then there is a "least" unevaluated attribute (since by assumption we can create a topological sort of the attributes), and by the well-formedness assumption this least unevaluated attribute is evaluable.

□

We can now prove the desired theorem about the evaluation algorithm.

Theorem 2.2. All attributes will have been correctly evaluated upon termination of algorithm 2.2.

Proof. By Lemma 2.1, algorithm termination (emptiness of the Evaluable set) implies that there are no evaluable attributes. But this implies (by the contrapositive of Lemma 2.1) that the set of unevaluated attributes is empty.

□

2.5. Change of a single attribute of in-degree zero

We now consider a mild form of incremental re-evaluation. A parse tree that has been initially evaluated is given, and a single "global input" attribute (one

whose evaluation function takes no attributes as inputs) is then given a new value.

Algorithm 2.3.
Input: a fully attributed tree from a well-formed attribute grammar and a new value for an attribute alpha of in-degree zero

Output: a consistent, fully attributed tree containing the new value for the attribute alpha

```
for all productions P containing alpha, do begin
  put ( production := P,
       active := {alpha},
       passive := {all attributes of P that don't depend on alpha} )
  in Interior (see Note 1.);
  if alpha is an input to P, then put P in Evaluable;
end;

while Evaluable is not empty do begin

  select and remove an element E from Evaluable;

  for each element beta of plan(E) (see Note 2.), do begin

    if beta depends on E.active,
    then new_beta := evaluate beta
    else new_beta := old_beta.

    if new_beta = old_beta
    then E.passive := E.passive + { new_beta }
    else E.active  := E.active + { new_beta };

  end;

  make passively available in E any attributes coming from Exterior
  neighbors that do not depend on E.active or E.unavailable (see Note 3.)
  Put E in Evaluable if it has a non-empty plan.

  for every adjoining production instance A, do begin
    if A is in Interior, then update A.active and A.passive, and
                         put A in Evaluable if it has a non-empty plan.
    else begin
      if A is receiving an active attribute then begin
        put ( production := A,
             active := {active attributes from E},
             passive := {attributes that don't depend on
                      E.active or E.unavailable} )
        in Interior;

        put A in Evaluable if it has a non-empty plan;
      end;
    end;
  end;
end.
```

Note 1: The set "Interior" contains those production instances that have active input attributes. Production instances that have no active inputs need not be examined by the algorithm, since it is known that none of their outputs will change value. We associate with each Interior production instance a set indicating which of its attributes are active and which are passive. Thus when a production instance gets placed in the Interior set its passive and active attribute sets must be initialized.

Note 2: The "plan" for a production instance that has a set of available attributes is the set of output attributes for that production which are not available and depend only on available attributes. The plan for a production may be empty (if, for instance, it has no unavailable attributes).

Note 3: As will be seen in the proof of the algorithm, attributes which are outputs of Exterior neighbors and do not depend on active or unavailable attributes of the production being examined do not need to be evaluated; they already have their correct values.

To prove that the algorithm is correct, we start with the following lemma.

Lemma 2.3. At the top of the main "while" loop, every attribute in Interior is either available, evaluable, or dependent on an unavailable attribute in Interior.

Proof. The claim is true after initialization, since if an attribute beta is unavailable then it must depend on alpha, and all successors of alpha are in Interior. Either beta depends on successors of alpha or it does not. In the former case, the assertion is seen to be true, since all successors of alpha are unavailable and in Interior; in the latter case the assertion is also true, since beta is then evaluable.

Now, we will show that the claim of the lemma is an invariant of the main loop. If the main loop selects a production instance with no Exterior neighbors, the claim will still be true after the loop: some evaluable attributes will become available,

and some unevaluable attributes will have the status of predecessors change. If all of the attributes in Interior that are an attribute's predecessors get evaluated, then it becomes evaluable, since (as we will show) it cannot depend on non-Interior attributes. To depend on a non-Interior attribute, beta would have to be an output of an Exterior production instance, and so beta could not have a predecessor in E.

Now, say E has an Exterior neighbor. We evaluate the attributes of plan(E). By the above discussion, attributes of Interior neighbors of E will satisfy the conditions of the lemma. So, we consider the attributes of the exterior neighbor. If an attribute beta of N depends on an unavailable interior attribute, then the lemma is not violated. If all of the Interior attributes that are ancestors of beta get evaluated, either it is a successor of an Exterior attribute or it is not. If it's not, then it becomes evaluable. If it is, then it's an output of an Exterior production. If the attribute's dependence on Interior is only via passive attributes, then it becomes available. If not, it must depend on an active Interior attribute. In that case, its production will be made Interior by the algorithm. This latter case will be covered below in the discussion of new Interior productions.

We now consider attributes of a new Interior production NI: all attributes of NI that do not depend on alpha become available since non-alpha Interior attributes are passive. Of those that remain, some depend only on available attributes of NI and are thus evaluable. Some depend on alpha and also on non-NI attributes. These attributes must depend on unavailable Interior attributes: they are outputs of Exterior productions, and hence cannot have predecessors in E. They must depend on (as yet unavailable) attributes in NI which in turn depend on attributes in E.

□

Now, at the top of the loop, say we have some unavailable attribute in Interior. By the well-formedness of the attribute grammar we can topologically sort all the attributes in the whole tree, and the unavailable attributes in Interior thus have an ordering. Specifically, at least one of them does not depend on any of the others. By the lemma, it must be evaluable.

2.6. Subtree replacement

We now consider the culmination of the results obtained so far, the process of incrementally re-evaluating the necessary attributes after a subtree replacement operation takes place. The key difference between the situation we consider here and the one discussed in the last section is that some attributes will have different evaluation functions after the replacement. We refer to the node whose subtree is changed as the intersection node and the two productions that contain the intersection node as the intersection productions. Attributes of the intersection node may have new evaluation functions. Thus, even if all inputs to a given attribute retain their old values, care must be taken to insure that that attribute ends up with the correct new value, and that attributes which depend on it are re-evaluated if necessary. Also, the initialization procedure must be somewhat different. Explanatory notes will appear after the algorithm.

```
Algorithm 2.4.
Input:  an attributed parse tree with a designated node
        and an attributed subtree

Output: an attributed parse tree with the input subtree spliced
        to the designated node of the parse tree

put productions containing the intersection node in Interior
  (see Note 1 of algorithm 2.8);

Make independent attributes not at the intersection passive (see Note 1.)

for each independent intersection attribute beta, do begin
  if donor.beta <> recipient.beta (see Note 2.)
  then put beta in active sets of the intersection production instances
  else put beta in passive sets of the intersection production instances;
end;

put each intersection production with a non-empty plan
in Evaluable;

while Evaluable is not empty do begin

  select and remove an element E from Evaluable;

  for each element beta of the plan for production instance E, do begin

    if beta is not an intersection attribute then begin
      if beta depends on E.active,
      then new_beta := evaluate beta
      else new_beta := old_beta.
      beta.active := new_beta <> old_beta;
    end

    else begin
      if beta depends on E.active
      then new_beta := evaluate beta
      else new_beta := donor_beta;

      beta.active := new_beta <> recipient_beta;
    end;

    if beta.active
    then E.active  := E.active + { new_beta }
    else E.passive := E.passive + { new_beta };

  end;

  make passively available in E any attributes coming from Exterior
  neighbors that do not depend on E.active or E.unavailable.
  Put E in Evaluable if it has a non-empty plan.
```

```
for every adjoining production instance A, do begin
  if A is in Interior, then update A.active and A.passive, and
                       put A in Evaluable if it has a non-empty plan.
  else begin
    if A is receiving an active attribute then begin
      put ( A,
          {active attributes from E},
          {attributes that don't depend on E.active or E.unavailable}
        )
      in Interior;

      put A in Evaluable if it has a non-empty plan;
    end;
  end;
 end;
end.
```

Note 1: At first, the only production instances in Interior are those that contain the intersection node. The "independent" attributes of a given production and position within that production are those attributes which do not depend on any input attributes of that position. Independent attributes are computed by Poegen by appending to the dependency graph of the production the characteristic graphs of the non-terminals of the production and then determining the attributes that are unreachable from the input attributes of the position in question. Independent attributes of the intersection productions are known to have their correct values, and need not be computed, since by definition they do not depend on the part of the tree that gets changed.

Note 2: At the intersection node, we have two different versions of each attribute, one from the production above the intersection, and one from the production below the intersection. For an intersection attribute beta, we call that version of beta the "donor" which comes from the production for which beta is an output attribute, and we call the version of beta the "recipient" which comes from the production for which beta is an input. In terms of synthesized and inherited attributes, the donor attributes are

(1) the inherited attributes of the production for which the intersection attribute is a right-hand symbol, and

(2) the synthesized attributes of the production for which the intersection node is the left-hand symbol.

Similarly, the recipient attributes are

(1) the synthesized attributes of the production for which the intersection node is a right-hand symbol, and

(2) the inherited attributes of the production for which the intersection node is the left-hand symbol.

We now show that after initialization Lemma 2.3 of the previous section will still hold. For reference, we repeat the statement of the lemma here:

Lemma .2.3. At the top of the main "while" loop, every attribute in Interior is either available, evaluable, or dependent on an unavailable attribute in Interior.

Proof. First, note that all independent attributes are known to be available. Now, a given dependent attribute may either depend on other dependent attributes or not depend on any dependent attributes. In the former case, the dependent attribute satisfies the third clause of the lemma. In the latter case, the dependent attribute depends only on independent attributes, since inputs to an attribute may only be other attributes of that production, and by definition all attributes of the production are either dependent or independent. Independent attributes are all available, and so in the latter case the dependent attribute is seen to be evaluable.

□

The loop invariant of Algorithm 2.3 is seen to hold after initialization, and the situations of algorithms 2.3 and 2.4 are identical with the sole exception that

recipient attributes of the intersection node may have different evaluation functions after the subtree replacement. Except for this one difference, the main result of the previous section will immediately follow. We now detail the reasoning behind the method used by Algorithm 2.4 to handle this one special case that is different from Algorithm 2.3.

At the intersection node, we must take new_beta to be donor_beta in the case that all of beta's ancestors are passive, since the evaluation function used to compute beta is associated with the donor production. Since none of beta's inputs have changed, we know that if we did evaluate beta we would get the value of beta possessed by the donor production.

On the other hand, even if all of beta's inputs are passive, beta itself may be active. This is because the evaluation function used to compute the beta possessed by the recipient production may be different from the evaluation function used to compute the version of beta possessed by the donor production. Attributes which depend on beta were computed using the value of beta possessed by the recipient production, and this is why the active/passive classification must be made using the recipient version of beta.

We see, then, that algorithm 2.4 will define the interior region based on which productions have active input attributes (with the sole exception of the intersection productions), and will never evaluate an attribute unless all its inputs are known to have their final values. Moreover, a production instance is visited by the evaluator only if it has evaluable attributes, so this evaluation scheme is optimal in the Cohen-Harry [11] sense in addition to being optimal in the Reps [9] sense.

2.7. Example of the behavior of Algorithm 2.4.

In the following figure, we have a fragment of an attribute grammar that describes "if" statements and expressions. In the process of initial evaluation, first the "symtab" attribute flows down from the part of the parse tree above the "Stmt" node to the leaves of the expression. Then the leaves determine their types by looking in the symbol table. Type information is passed back up the tree until it reaches the "if" statement production, at which time it is verified that the type of the expression is boolean. A boolean "error" attribute is sent back down to the leaves of the tree which indicates whether the expression has the correct type. A screen manager might take leaves with an "error" attribute that has the value "true" and display them in a stand-out mode to draw the user's attention to the error.

(0) ... ::= Stmt	Stmt.symtab := (symbol table built in declaration section)
(1) Stmt ::= if E then Stmt <else clause>	E.symtab := Stmt.symtab E.error := E.type <> Bool_type
(2) E ::= E <op> T	E_2.symtab := E_1.symtab T.symtab := E_1.symtab E_1.type := TypeCalc(E_2.type, <op>.optype, T.type) E_2.error := E_1.error <op>.error := E_1.error T.error := E_1.error
(3) E ::= T	T.symtab := E.symtab E.type := T.type T.error := E.error
(4) T ::= id	T.type := LookUp(T.symtab) id.error := T.error
(5) <op> ::= <	<op>.optype := less_op "<".error := <op>.error
(6) <op> ::= <=	<op>.optype := less_or_equal_op "<=".error := <op>.error
(7) <op> ::= +	<op>.optype := plus_op "+".error := <add op>.error

Let us say that the user starts by typing in the statement

if i < j then ...

The evaluator follows the rules of the attribute grammar and attributes the tree. Now say the user changes the "<" node to a "<=" node. The <op> node is the intersection node, and the optype attribute is the only independent attribute of that node. Since the optype attribute is different for subtrees "<op> ::= <" and "<op> ::= <=", the optype attribute is made active, and the production instance "E ::= E <op> T" is made a member of the Interior set. The "symtab" attributes of the new production instance do not depend on attributes of <op>, so they are

inferred to be available. Similarly, the "type" attributes of the nodes E_2 and T are made available. So, the attribute which is unavailable and depends only on available attributes is E_1.type. That attribute is evaluated, and is seen to retain its old value (the type of the expression remains boolean after a change of the <op> from "<" to "<="). So, E_1.type is a passive attribute. We therefore do not include the production

Stmt ::= if E then Stmt <else clause>

in the Interior set. On the other hand, we must make passively available attributes coming from that (Exterior) production which do not depend on active or unavailable attributes, and E_1.error is such an attribute. The other "error" attributes are then made passively available, again not causing any new productions to be included in Interior, and finally the "error" attribute of the node "<=" is set to be false.

If on the other hand the user changes "<" to "+", then the "type" attribute of the node E_1 changes, and the "if" production must be included in Interior. The "if" production sets "error" to be true, since the type of the expression is no longer boolean, and as the new (active) "error" attribute works its way back down the tree it causes inclusion of all of the "E ::= T" and "T ::= id" productions in Interior. Eventually all the leaves receive the new error attribute, and the whole expression becomes highlighted on the screen.

Chapter 3 - Priority-based incremental evaluation

3.1. Introduction

We now direct attention to the problem of eliminating unnecessary evaluations of attributes. We include in the work of evaluating an attribute the task of manipulating the state of production instances in which the attribute appears; thus, elimination of unnecessary evaluation implies eliminating manipulation of the states of productions containing the attribute.

The analyses presented in this section will result in a degree of increased efficiency in the case of normal (context-free) incremental update, and will be seen to be crucial for obtaining reasonable time bounds on incremental evaluation algorithms in the presence of non-local productions.

Although the Reps algorithm and the equivalent plan-oriented algorithm presented in the previous chapter are in one sense optimal (they both have O(n) space and time bounds, where "n" is the number of attributes that receive new values), attribute grammars can be constructed for which an arbitrarily large number of unnecessary visits to production instances will be performed in the incremental evaluation process. We deem a visit to a production instance to be unnecessary if it can be shown that any evaluation which may take place during that visit will result in an attribute value which is the same as that possessed by the attribute before the evaluation took place. This chapter will be devoted to derivation of various classes of algorithms which do not exhibit this undesirable behavior. For the sake of motivation, a class of attribute grammars will be presented such that for arbitrarily large N an attribute grammar can be constructed for which the new class of algorithms will perform N times faster than algorithms based on techniques alluded to in the previous paragraphs.

In order to lay the groundwork for determining which of several classes a given attribute grammar falls into, we will examine the problem of computing possible dependencies among the attributes of a given grammar. If a given attribute grammar has two productions m and n, and the productions possess attributes alpha and beta respectively, we will provide a method for determining the answer to the question, "Is there any conceivable parse tree derivable from this grammar such that an instance of alpha is an ancestor of an instance of beta?" We will then refine the question a bit, and ask, "Can we construct a parse tree with alpha an ancestor of beta such that the node containing alpha precedes the node containing beta in a pre-order traversal of the parse tree?" Answers to the above questions can be used to form a classification scheme for attribute grammars. It will be found that many attribute grammars exhibit a regularity property which permits the use of a particularly simple re-evaluation scheme with the desired strong optimality property that no unnecessary production visits take place.

A more general class of attribute grammars will then be identified which requires the imposition of a (finite) ordering on production states. By the "state" of a given production instance, we roughly mean the number of attributes which have been evaluated in that production at some point midway through the evaluation process.

Typically, the cardinality of the ordering will be small. A grammar will "almost" have the regularity property mentioned above, but there will be a few attributes whose inter-dependence structure requires that they be given different evaluation priorities. By way of analogy, many context-free grammars are "almost" SLR, but have a few states which require recourse to a more powerful parser generation method such as LALR.

Another yet more general class of attribute grammars requires that productions within certain evaluation priority classes be retrieved in the order in which they would be visited in a pre-order traversal of the parse tree. Except in some special cases, this leads to O(nlog n) algorithms, since sorting operations are required. Nevertheless, the cross-over point between a fast O(nlog n) algorithm and a slower linear algorithm may be large enough that the O(nlog n) algorithm is deemed preferable. Moreover, in many cases of practical importance, the "n" in the O(nlog n) algorithm will be far smaller than the "n" in the linear algorithm.

We begin by presenting an example to illustrate the sorts of situations that will be the focus of our attention. Figure 3.1 contains an attribute grammar for a small example language. The notation "A.atr1 := B.atr2" means that the attribute named "atr1" that is associated with symbol A in the production should be given the value possessed by the attribute named "atr2" that is associated with symbol B in the production. If a symbol appears more than once in a production, then the occurrences will be distinguished by numerical subscripts.

S ::= A	A.down := A.up mod 2
A ::= a A	A_1.up := A_2.up A_2.down := A_1.down
A ::= b B	A.join := A.down + B.seed A.up := B.seed
B ::= c	B.seed := 2
B ::= d	B.seed := 10
B ::= e	B.seed := 47

Figure 3.1.

The BNF of this attribute grammar describes the language {a* b (c|d|e) }. We will consider a string "a*n b c" (n occurrences of "a", where n is assumed to be large, followed by the symbols "b" and "c") and change it to "a*n b d." After the

change has been made, it will be necessary to perform an incremental evaluation in order to restore the values of attributes in the attributed tree representing the string. Using techniques of the previous sections, the re-evaluation will require approximately 2n visits to production instances. In our grammar, the parse tree of the string "a^n b c" will contain n+3 productions; n+2 of those productions contain nodes labeled A, and each occurrence of A's attribute "up" receives a new value. The evaluator thus visits n+2 productions going up the tree giving new values to the various instances of the attribute "up." In the production "S ::= A", we evaluate the attribute "down"; note that although its inputs change, "down" itself retains its former value. In the parlance of the previous chapter, it is a passive attribute. The "join" attribute has an active input (the attribute "seed"), and thus requires re-evaluation, but "join" also depends on the attributes "down." The evaluators of the previous chapter will re-visit each of the production instances "A ::= a A", update their states to indicate that their copies of "down" are up-to-date, and finally reach the production "A ::= b B", update its state to indicate availability of "A.down", and finally re-evaluate "A.join."

We can reduce re-evaluation time by a factor of two by immediately classifying "join" to be "provisionally evaluable" after evaluating "seed", and arranging that any attributes which may have to be evaluated before "join" is evaluated will be selected by the evaluator before it attempts to evaluate "join." In the new evaluation scheme we visit a production instance only if it has an unavailable output attribute that has at least one active input attribute. Thus, the n visits to productions to update their states indicating that the "down" attributes are available are not performed.

Notice that in the example "join" could have been evaluated immediately after evaluation of "seed," since it turns out that its other input retains its old

value. On the other hand, if we had changed the string from "a^n b c" to "a^n b e", we would not be able to evaluate "join" immediately, since its other input would eventually receive a new value. We account for both possibilities by classifying "join" as evaluable immediately after "seed" has been evaluated, but by giving it a lower "evaluation priority" than any instance of an "up" or "down" attribute: if instances of both "join" and "up" or "down" attributes are evaluable, then we require that an "up" or "down" attribute gets selected for evaluation. In general, an evaluator has at any given time a set of production instances with evaluable attributes, and it non-deterministically selects an element of that set for evaluation. We will partition the evaluability set into an ordered finite collection of subsets. If one production instance is in a subset that is lower in the ordering than another production instance, we will say that the first production instance has higher evaluation priority than the second production instance. The evaluator will be required to choose a production instance from the non-empty subset that has highest priority. In the preceding example, when the high-priority evaluation set becomes empty, that constitutes a proof that all inputs to "join" have their correct, final values, and that "join" may safely be evaluated.

In the example given at the beginning of this section, we reduced re-evaluation time by a factor of two by overcoming the need to update the status of production instances to reflect availability of passive attributes. We could extend the example so that passive attributes wind up and down the tree an arbitrary number of times before reaching the "join" attribute, and thus increase the speed-up factor obtained by ignoring passive attributes. For instance, Figure 3.2 shows an attribute grammar for which propagation of passive attributes constitutes three fourths of the work of the re-evaluation process.

S ::= A	A.down1 := A.up1 mod 2 A.down2 := A.up2
A ::= a A	A_1.up1 := A_2.up1 A_2.down1 := A_1.down1 A_1.up2 := A_2.up2 A_2.down2 := A_1.down2
A ::= b B	A.join := A.down2 + B.seed A.up2 := A.down1 A.up1 := B.seed
B ::= c	B.seed := 2
B ::= d	B.seed := 10

Figure 3.2.

The strategy outlined above depends on the ability to create a partial order among production instances in various evaluability states. In the next section we will present and prove the correctness of an algorithm which will serve that purpose. It will take as input an attribute grammar and produce the necessary partial order on its attributes. As noted in the introductory section of this chapter, it is possible to construct an attribute grammar for which creation of such a partial order is not possible; the algorithm will halt with an error indication if it is given an attribute grammar for which the desired partial order does not exist. The example presented at the beginning of this section presents the key situation which must be allowed for: a particular production instance in a parse tree has an attribute (in this case, the attribute "seed") which is an ancestor of another attribute in the production instance ("join") via more than one path. Attributes such as "join" will be referred to as rendezvous attributes. Every attribute along each path from the source attribute to the rendezvous attribute must be given higher priority than the rendezvous attribute. These higher-priority attributes will be called rendezvous ancestors of the rendezvous attribute to distinguish them from the source attribute and its ancestors, which need not be given higher evaluation priority than

the rendezvous attribute.

3.2. An algorithm for computing simple priority relations

We now state formally the the "simple priority" condition which the partial order constructed by the algorithm must satisfy: For attributes alpha(P) and beta(Q) (where P and Q are productions of the grammar with which alpha and beta are respectively associated), the algorithm will give alpha(P) a higher priority than beta(Q) if and only if there is some parse tree derivable from the attribute grammar under consideration such that beta(Q) is a rendezvous attribute and alpha(P) is one of beta(Q)'s rendezvous predecessors.

A classification which satisfies the above condition will be called a simple priority order.

The algorithm is skeletal, and annotations will be presented after the statement of the algorithm.

```
Algorithm 3.1.
Input:  a simple, well-formed attribute grammar.
Output: a simple priority order relation among the attributes
        or an indication that no such order exists.

compute IO and OI graphs for the grammar;

for each production P in the grammar,
  (a) append to its dependency graph the characteristic graphs of its
      non-terminals;

  (b) determine whether any pair of attributes (alpha, beta) in
      the augmented production has more than one path from alpha to beta.
      If not, look at the next production in the "for" loop.

  (c) For every attribute gamma on every path from alpha to beta
      (not including alpha or beta), put (gamma, beta) in the relation.

    Each characteristic graph used in (a) has a sub-graph whose nodes
    are rendezvous ancestors of beta.  Put each (characteristic graph,
    rendezvous ancestor sub-graph) pair for which the sub-graph is
    non-empty into a set "to be examined."

    while the "to be examined" set is non-empty,
      pick a (characteristic graph, sub-graph) pair C.

      for each "prod"/"position" creator of C (see Note 1), do
        (*) append to the dependency graph of "prod" characteristic
            graphs at all non-terminals except "position" (see Note 2.)

        (**) for each attribute alpha' in "prod" such that there is
            an arc i->j in the sub-graph with the property that
            i->* alpha' ->* j is in the appended dependency graph,
            insert (alpha', beta) in the overall relation.

            for each characteristic graph used in (*) that has
            not appeared in the "to be examined" set, if the
            rendezvous ancestor subgraph (see Note 3) is non-empty
            put the characteristic graph and the sub-graph
            in "to be examined."
    end { "to be examined" non-empty }
end { for each production P ... }
```

Comments on the algorithm are presented in the following paragraphs:

Note 1: A "creator" of a given characteristic graph C is a production, position, and collection of characteristic graphs selected at the top of the main loop of Algorithm 3.1 which results in creation by that algorithm of characteristic graph C.

A given characteristic graph may be created several times in the algorithm using different productions and sets of augmenting characteristic graphs. For example, in the attribute grammar of Figure 3.3, the OI graph for symbol B,

UI -> foo; bar -> glarch,

can be computed either using production (1) and no characteristic graphs or using production (2) and the IO graph of symbol A (UI -> foo).

(1) S ::= B	B.foo := 29 B.glarch := B.bar
(2) S ::= A B	B.foo := A.foo B.glarch := B.bar
(3) A ::= a	A.foo := 47
(4) B ::= b	B.bar := B.foo

Figure 3.3.

Note 2: we are performing here an operation identical to the one used in the characteristic graph computation. The purpose here is different, however; we intend to find all possible attributes which can be rendezvous ancestors of beta. In a sense, this algorithm is a reverse execution of the characteristic graph computation algorithm; in that algorithm we go from the creators of a characteristic graph to the characteristic graph, whereas in the present algorithm we go from a characteristic graph to its creators.

Note 3: by "rendezvous ancestor sub-graph" we mean the sub-graph of a characteristic graph all of whose nodes are known to be rendezvous ancestors of beta. As will be seen in the proof, these are precisely the attributes alpha' identified in the part on the algorithm marked (**).

We now prove the desired result about Algorithm 3.1:

Theorem 3.1. The attribute pair (alpha(P), beta(Q)) (where P and Q are the

productions with which alpha and beta are respectively associated) will be inserted in the relation by Algorithm 3.1 if and only if there exists a parse tree derivable from the attribute grammar such that beta(Q) is a rendezvous attribute and alpha(P) is a rendezvous ancestor of beta(Q).

Proof. We first show (by induction) that if the algorithm inserts a pair (alpha(P), beta(Q)) of attributes in the relation, then there exists a parse tree with the desired relationship among the attributes. First, there is indeed a parse tree for which beta(Q) is a rendezvous attribute. This follows from the fact that by Theorem 2.5 there corresponds to each characteristic graph of a grammar a near-complete parse tree from the grammar. If an attribute beta is recognized in step (b) as being a rendezvous attribute, a parse tree can be constructed by appending to an instance of production Q the parse-tree fragments that correspond to the characteristic graphs used in step (a). Attributes alpha(Q) asserted to be rendezvous ancestors of beta(Q) in step (c) of the algorithm will indeed be rendezvous ancestors in the same parse tree constructed to show that beta(Q) is a rendezvous attribute.

Now, we establish the invariant for the "while" loop that every pair (alpha(P), beta(Q)) inserted in the relation has a parse tree satisfying the rendezvous relationship, and that for every (characteristic graph, sub-graph) pair in the "to be examined" set the nodes of each sub-graph consist entirely of rendezvous ancestors. The first action of an iteration of the loop is to select a creator for a (characteristic graph, rendezvous sub-graph) pair. By assumption, the attributes in the sub-graph are rendezvous ancestors. We can construct a parse tree corresponding to the selected characteristic graph by creating an instance of the initial production Q and appending to it an instance of each production that is examined by the algorithm between production Q and the creator production of the characteristic graph selected by the iteration being examined. Now we know that

attributes "i" and "j" are rendezvous ancestors by assumption; by definition of the term, this means that alpha(Q) $\rightarrow^{*}$ i (and j) $\rightarrow^{*}$ beta(Q); another attribute alpha' which satisfies i $\rightarrow^{*}$ alpha' $\rightarrow^{*}$ j will then also satisfy alpha(Q) $\rightarrow^{*}$ alpha' $\rightarrow^{*}$ beta(Q), and will hence be a rendezvous ancestor. So, (alpha', beta(Q)) is seen to satisfy the rendezvous relationship. Further, since sub-graphs of additions to the "to be examined" set in the current iteration of the "while" loop consist entirely of rendezvous ancestors, the second clause of the loop invariant still holds after the current iteration completes.

Now, assume that a parse tree exists for which attributes alpha and beta are in the rendezvous relationship to each other. Since for every set of dependencies among attributes induced at a node by a parse tree there exists a corresponding characteristic graph, the production containing beta will be noticed during execution of step (b) in the algorithm. So, beta will be included as a rendezvous attribute.

Now we show that alpha will be included as a rendezvous ancestor of beta. There is a path in the parse tree from the node that contains beta to the one that contains alpha. The algorithm will follow creator productions that match this path since at each step we examine all creators of a given characteristic graph. Eventually, the production containing alpha will be visited by the algorithm, and included in the global relation.

□

The following sub-algorithm assigns evaluation priorities to attributes based on the relation computed by Algorithm 3.1:

Algorithm 3.2.
If the relation constructed is anti-symmetric (for no x, y is it true that x R y and y R x), then make priority classes based on the relation: (otherwise, the grammar is not SP.)

```
i := 0;
while the relation is not empty, do
  1) for all prod/pos pairs x such that there exists no y such
     that y R x, put x in priority class i;
  2) remove all elements x R z from the relation, where x satisfies
     condition 1);
  3) increment i.
end.
```

3.3. Priority calculation for join attributes

A less general priority scheme than that presented in the previous section which allows quite general re-evaluation schemes (re-evaluation after editing operations more complex than single subtree replacements) is presented in this section. Attribute grammars for which this algorithm will produce a priority ordering constitute a strict subset of those for which the algorithm of the previous section will produce an ordering. The algorithm which follows will establish an ordering between attributes alpha and beta if in any parse tree derived from the grammar alpha is an ancestor of beta.

Algorithm 3.3.
Input: a simple, non-circular, well-formed attribute grammar.
Output: a classification of attributes into evaluation priority classes
or an indication that the grammar has no join priority order

compute IO and OI graphs for the grammar;

for each production P in the grammar,
for every attribute that has more than one input, create
entries in the relation:

append to its dependency graph the characteristic graphs of its
non-terminals;

for every attribute alpha in the augmented dependency graph that
is an ancestor of beta, put (alpha, beta) in the join relation.

Each characteristic graph has a set of nodes whose corresponding
attributes are ancestors of beta. Put each (characteristic
graph, ancestor set) pair for which the ancestor set is non-empty
into a set "to be examined."

while the "to be examined" set is non-empty,
pick a (characteristic graph, ancestor set) pair C.

for each "prod"/"position" creator of C (see Note 1), do
(*) append to the dependency graph of "prod" characteristic
graphs at all non-terminals except "position" (see Note 2.)

(**) for each attribute alpha in "prod" which is an ancestor
of an attribute in the ancestor set, insert (alpha, beta)
in the join relation.

for each characteristic graph used in (*) that has not
appeared in the "to be examined" set, if the ancestor
set of that characteristic graph is non-empty put the
characteristic graph and the ancestor set in "to be
examined."
end { "to be examined" non-empty }
end { for each production P ... }

Notes 1 and 2 are the same as those for algorithm 3.1 above.

We now state and prove the desired result about the algorithm.

Theorem 3.2. Given an attribute grammar AG and an attributed parse tree derived from it, every attribute in the tree that has more than one input has strictly lower priority than any of its ancestors.

Proof. The proof of this algorithm is almost identical to the proof of Algorithm 3.1. The differences are in fact simplifications. Instead of identifying rendezvous attributes, we must simply identify attributes whose evaluation functions take more than one argument. Similarly, instead of identifying rendezvous ancestors we must simply look for ancestors. Therefore we will not repeat the details of the proof here.

3.4. An algorithm to compute left and right priorities

It turns out that the simple priority condition is fairly restrictive. Some common language constructs do not permit simple priority relations. Therefore, we now examine a more general class of priority relations, the "oriented priority relations." An oriented priority relation is one in which production instances in certain priority groups must be visited in some pre-determined tree order, such as depth-first left-to-right or right-to-left. After presenting an attribute grammar that has an oriented priority relation but no simple priority relation, we will define and present an algorithm for computing oriented priority relations.

In Figure 3.4 below we present an attribute grammar fragment for expressions for which there exists no simple priority relation.

(0) ... ::= E	E.symtab := (symbol table built in declaration section)
(1) E ::= E + T	E_2.symtab := E_1.symtab T.symtab := E_1.symtab E_1.type := TypeCalc(E_2.type, T.type)
(2) E ::= T	T.symtab := E.symtab E.type := T.type
(3) T ::= id	T.type := LookUp(T.symtab)

Figure 3.4.
An attribute grammar for which there exists no simple priority relation.

In the production "E ::= E + T", the attribute E_1.type is a rendezvous attribute, since it depends on E_1.symtab via two paths. However, one of its rendezvous ancestors turns out to be E_1.type, since another instance of "E ::= E + T" can be used to expand the "E" in the right-hand side. So, the relation is not anti-symmetric, and consequently it is not a simple priority relation. On the other hand, if we require that instances of E.type be evaluated in depth-first search order at low priority, then the difficulty is resolved. To see this, assume that a parse tree has been built and fully attributed, and that the user then makes a change to the declaration section, which changes the "symtab" attribute. Since E.value attributes of instances of production (1) have low priority, all attributes in an expression will be evaluated until the only remaining unevaluated attributes are instances of E.value. At that point, the lower-leftmost instance of E.value will be evaluated, then the next lowest, and so on until the uppermost E.value finally gets evaluated. It is illustrative to consider what could have happened if the low-priority evaluation class is not ordered. At the point when only instances of E.value are left to be evaluated, the evaluator is free to make a non-deterministic choice. If it picks any instance of E.value other than the lower-leftmost one, then at some later time it must evaluate the lower-left instance of E.value, and re-evaluate all of its descendants, including the one it had previously selected and evaluated. In worst case this can lead to quadratic evaluation behavior in the expression.

In order to handle situations like the one illustrated in the previous example, the priority computation algorithm keeps track not only of which attributes may depend on which other attributes, as in the previous section, but also on whether a particular ancestor attribute must be associated with a node of the parse tree to the left (or right) of the node containing the rendezvous attribute of interest.

In general, we may construct relations "Pre" and "Post" among the symbols of a reduced BNF, where "x Pre y" implies that there exists a parse tree derived from the BNF for which a node labeled "x" is visited before another node labeled "y". The relation may be stated as follows:

> "x Pre y" iff either x ==>+ y or there exist grammar symbols B and C such that there is a production A ::= .. B .. C .. in the grammar with B ==>* x and C ==>* y.

The relation "x Post y" is almost identical:

> "x Post y" iff either y ==>+ x or there exist grammar symbols B and C such that there is a production A ::= .. B .. C .. in the grammar with B ==>* x and C ==>* y.

The algorithm computes the "x Pre y" or "x Post y" relation, but with the added requirement that an attribute of "x" be a rendezvous ancestor of an attribute of "y".

Different attribute grammars may require different styles of traversal. Pre-order and post-order traversal are the two which seem to be the most common. Presumably attribute grammars can be constructed which require other methods of traversal, and there are doubtless attribute grammars for which no pre-determined traversal order can be used to form an oriented priority relation, although I am not aware of one. The algorithm will be written so that those parts which are dependent on the particular traversal scheme are contained in procedures. The procedures which correspond to pre-order traversal will be provided after the algorithm. The algorithm which computes oriented priority relations follows:

```
Algorithm 3.4.
Input:  a simple, non-circular, well-formed attribute grammar.
Output: an oriented priority order relation among the attributes
        or an indication that no such order exists.

compute IO and OI graphs for all symbols;

for each production P,
    append to its dependency graph the characteristic graphs of its
    non-terminals;

    determine whether any pair of attributes (alpha, beta) in
    the augmented production has more than one path from alpha to beta.
    If not, look at the next production in the "for" loop.

    For every attribute gamma on every path from alpha to beta
    (not including alpha or beta), put (gamma, beta) in a relation:

      { "pos( a )" is the position in a production, starting with zero
        for the left-hand side, with which "a" is associated }
      InitInsert( alpha, beta, pos(alpha), pos(beta) );

    For each characteristic graph C used in (a) which has a non-empty
    rendezvous ancestor sub-graph, put into a set "to be examined"
    the following record.  See Note 1 for an explanation of the fields.
      (C,
       rendezvous ancestor sub-graph,
       BetaNode (a boolean variable) := pos(C) = pos(beta),
       Direction := InitDirection( pos(C), pos(beta) ) );

    while the "to be examined" set is non-empty,
      pick a (characteristic graph, sub-graph, Direction, BetaNode) record C.

      if not C.BetaNode (see Note 2)
      then mark C.characteristic_graph "examined."

      for each "prod"/"position" creator of C, do
         (*) append to the dependency graph of "prod" characteristic
             graphs at all non-terminals except "position"

        (**) for each attribute alpha' in "prod" such that there is
             an arc i->j in the sub-graph with the property that
             i->* alpha' ->* j is in the appended dependency graph,
             insert (alpha', beta) in a relation:

                LoopInsert( alpha', beta, pos(alpha'), position,
                            C.Direction, C.BetaNode );

             for each characteristic graph D used in (*) that is not
             marked "examined," if the rendezvous ancestor subgraph
             is non-empty put the following record in "to be examined:"
             (D,
```

```
                rendezvous ancestor sub-graph of D,
                BetaNode := false;
                Direction := LoopDirection( C.Direction,
                                            pos(char. graph C),
                                            pos(char. graph D) );
        end { "to be examined" non-empty }
    end { for each production P ... }
```

Note 1: Elements of the "to be examined" set are records with four fields. The first field is a characteristic graph, and the second field is a sub-graph of the characteristic graph all of whose nodes are rendezvous ancestors of beta, the rendezvous attribute whose ancestors we are trying to find. The third field is a boolean variable indicating whether beta is associated with the node of its production which is in the same direction as the one we intend to explore. To illustrate the problem, consider the grammar

B ::= C
A ::= B

B ::= A.

Say we start from node "B" of the production "B ::= C." We next examine production "A ::= B", and we want to avoid the mistake of trying to conclude that B ==>+ B. The problem is that the B's of the two productions actually refer to the same parse tree node. The "BetaNode" field contains the value "true" if this situation must be checked for, and the value "false" otherwise.

Finally, the "Direction" field gives a summary of Algorithm 3.4's execution history. If we start at a particular node of a production and trace down from it, then we will always be in the "x ==>+ y" clause of the "Pre" or "Post" relation, and we indicate that in the Direction field. If we start from a node, travel up from it, and then start traveling down a left (or right) sibling of one of the nodes reached during the upward movement, then all nodes subsequently reached will come before (respectively after) the original node, and again we indicate whether

we are going down to the left or down to the right using the Direction field.

Note 2: We do not consider attributes of a node to be left or right ancestors of other attributes of the same node. On the other hand, the characteristic graph of C may be applicable at some other node of the tree if the symbol of the node can derive itself, and for that reason we might want attributes of the node to be left or right ancestors. In order to be able to re-use the characteristic graph if necessary, we do not mark it as having been examined.

The proof of Algorithm 3.4 is similar to the proof of Algorithm 3.1; the only difference is in maintenance of the "Pre" or "Post" ordering relation among grammar symbols. It will not be detailed.

We present in the following figures definitions of the procedures InitDirection, InitInsert, LoopDirection, and LoopInsert, which define a pre-order traversal of the parse tree.

```
procedure InitInsert( alpha, beta, pos(alpha), pos(beta) );

  if pos(alpha) < pos(beta), put (alpha, beta) in LeftRel;
  if pos(alpha) > pos(beta), put (alpha, beta) in RightRel;

  { if pos(alpha) = pos(beta), then the attributes are associated
    with the same parse tree node and belong in neither relation. }


function InitDirection( pos(C), pos(beta) ) returns a direction;
  if pos(C) = 0 then Direction := GoingUp
  else
    if pos(C) < pos(beta) then Direction := GoingDownLeft;
    if pos(C) >= pos(beta) then Direction := GoingDownRight;
```

```
procedure LoopInsert( alpha', beta, pos(alpha'),
                      position, Direction, BetaNode );
  if Direction = GoingDownLeft,
    put (alpha', beta) in LeftRel;

  if Direction = GoingDownRight,
    if ( not BetaNode ) or position <> pos(alpha')
    then put (alpha', beta) in RightRel;

  if Direction = GoingUp,
    if ( not BetaNode ) or position <> pos(alpha')
    then
          if pos(alpha') <= position,
            put (alpha', beta) in LeftRel;
          if pos(alpha') > position,
            put (alpha', beta) in RightRel;


function LoopDirection( Direction, oldpos, newpos ) returns a direction;
  if Direction in [ GoingDownLeft, GoingDownRight ]
  then return( Direction )
  else
    if oldpos = 0 then return( GoingUp )
    else if newpos < oldpos  then return( GoingDownLeft )
    else if newpos > oldpos  then return( GoingDownRight );
```

3.5. Priority-based incremental update for a new in-degree zero attribute

The algorithm which uses the priorities created by the algorithms of the last few sections to perform incremental evaluation when a single in-degree zero attribute alpha is changed follows:

```
Input:  attributed tree and a new value for an in-degree zero attribute
          alpha of the tree
Output: a consistent attributed tree that has the new value for alpha

  put all successors of alpha in their evaluation priority classes.

  while the evaluation set is not empty, do
      select and remove a prod/attribute from the highest non-empty
      priority class;

      evaluate the attribute.  If it has a value that is different
      from its previous value, place all of its successors in
      evaluation classes.
  end {while};
```

In order to demonstrate the validity of the above algorithm, we formulate the following invariant for the "while" loop: for every attribute in the highest non-empty priority class, each of its input attributes has its final value. We begin by demonstrating that the invariant is true after initialization:

Lemma: If gamma is dependent on beta, where gamma and beta are directly dependent on alpha, the new in-degree zero attribute given to algorithm x.y, then after 1) in the algorithm gamma is not in the highest non-empty priority class.

Proof. The production containing alpha as an input and beta and gamma as outputs will satisfy the initial condition of the poegen test, and hence beta R gamma, implying that beta is in a higher priority class.

□

Now, we infer from the lemma by way of contraposition that if gamma is in the highest priority class then it depends on no direct dependents of alpha. Now by assumption all attributes not dependent on alpha have their final values, and alpha has its final value, so all predecessors of gamma are thus seen to have their final values.

We now demonstrate that after execution of the loop the reputed invariant still remains true. In the loop, an attribute has been selected and evaluated; we consider two cases: the attribute assumed a new value, and it retained its old value.

Assume first that the attribute retained its old value. In this case it may have been the last attribute in its evaluation class or it may not have been. If it was not the only element in its evaluation class, then the invariant holds, since by assumption everything in the highest class was evaluable, and removing and evaluating one of those attributes will neither make an evaluable attribute unevaluable nor add new attributes to the evaluation set.

So, assume that the passive attribute was the last element in its evaluation class. When it is removed, an evaluation class which previously had not been highest is now highest (of course, if there had been no non-empty evaluation classes, the algorithm would have terminated). We claim that all inputs for each attribute in the new highest evaluation class are known to be available. First, since these attributes are not maximum priority, they are join attributes. They have at least one active input (since they are in the evaluation set); some inputs depend on alpha, and some do not; the ones that do not depend on alpha are available by assumption. Those that do are also known to be available; all paths from alpha to gamma will be made up of attributes which have higher priority than gamma, and the only way all elements of each of those paths could be absent from the evaluation set is if at some point they went passive.

Assume now that the evaluated attribute was active. Then each of its immediate descendants will be put in the appropriate evaluation class (if the attribute has no descendants, then the net impact of evaluating it will be the same as if it had been passive). After evaluating and removing gamma, if it should turn out that all of gamma's successors have lower priority than the highest non-empty priority class, then by the passivity argument above the invariant is maintained. So assume that one of gamma's successors is in the new highest priority set. The same argument as above applies: if it is a non-join, then its single input, gamma, is known to be available; if it is a join, then its alpha paths are of higher priority, and must have gone passive.

□

3.6. A subtree replacement incremental evaluator

In this section we present the algorithm that will use the priorities created by algorithms of previous sections to perform incremental re-evaluation after a tree replacement. The key difference between this algorithm and the one which re-evaluates after change of an in-degree zero attribute is that some descendants of the dependent intersection attributes must be "locked." Use of priorities allows us to "leap over" chains of passive attributes. In the case of subtree replacement, however, we cannot leap over the intersection node. Even if the inputs to an intersection attribute are all passive, the attribute itself may become active, because the attribute may have a new and different evaluation function after the subtree replacement.

In the previous chapter we needed to maintain the sets "active attributes" and "passive attributes" for each production. Here we must keep track of "active attributes" and "unavailable" attributes. An unavailable attribute will be one which is a descendant of an as-yet unexamined dependent intersection attribute whose versions in the two intersection nodes differ. An attribute which depends on an unavailable attribute will be considered not to be evaluable by the algorithm. The algorithm will start with the independent attributes of the intersection node, and evaluate all attributes in the tree that depend on them and not on any of the unavailable attributes. When the collection of evaluable sets becomes empty (we have done as much work as possible without touching the unavailable attributes), the algorithm selects a set S of unavailable intersection attributes which have no other unavailable intersection attributes as ancestors, and declares them to be available. It then goes through the parse tree making available for evaluation any attributes which had only been prevented from being evaluable by elements of S. After this second process has been completed, the algorithm starts working on the

Evaluable set again. The actual statement of the algorithm appears below.

Algorithm 3.5.
Input: an attributed tree and a new subtree to be spliced in
Output: a consistent attributed tree that has the new subtree

```
for each independent attribute of the intersection node,
classify it active or passive. Classify as unavailable every
attribute in the two productions that depends on a dependent
intersection attribute for which the values possessed by the
two productions differ. Put in evaluation sets of appropriate
priority classes all non-intersection attributes that depend on
active attributes and not on unavailable attributes.

while not done do begin

  while there is a non-empty evaluation set do begin
      select and remove a production instance and attribute from
      the highest non-empty priority class;

      evaluate the attribute. If it has a value that is different
      from its previous value, place all of its successors that
      do not depend of unavailable attributes in evaluation
      classes.

      If a production has attributes included in evaluation
      classes for the first time, initialize its unavailable set
      to contain all of its attributes that are descendants of
      unavailable attributes of the production instance from
      which it receives an active input attribute.
    end;

  end;

(*) if there are no unavailable input attributes,
    then done := true

    else begin
      Make all unavailable intersection attributes that depend on no
      other unavailable intersection attributes "available." Put
      productions with newly available attributes in "became available."

      while the "became available" set is not empty, do begin

        Select a production from the "became available" set.
        Remove from the production's "unavailable" set any attributes
        none of whose predecessors are unavailable.

        For each attribute that becomes available, include the
        production for which it is an input in the "became available"
        set.

        Place in the appropriate evaluation set any attribute which
        has active inputs and no unavailable inputs.
```

```
        end;
      end;

    end;
```

The proof of the algorithm of the previous section holds for the subtree replacement algorithm, with the exception of the behavior of unavailable attributes. The key lemma which allows the proof of the algorithm of this section to follow from the proof of that of the previous section is stated and proved below.

Lemma. When the Evaluable set becomes empty (at the point marked (*) in the algorithm), those elements of the set U of unavailable intersection attributes with no predecessors in U can be inferred to have their correct values. The values possessed by the donor productions are the correct values.

Proof. Let us refer to the set of newly available elements of U as NA. Since it is a dependent attribute, an element of NA is a descendant of other intersection attributes. The set of intersection attributes which it depends on, call it A, have their correct values by assumption. All attributes which depend on elements of A and no unavailable intersection attributes have been given their correct, final values. This is due to the fact that the algorithm performs exactly like the algorithm of the previous section in propagating the affects of changes in the set A to the rest of the tree. That there are no elements in the Evaluable set implies that all chains of attribute dependencies starting from elements of A terminated in passive attributes or attributes with unavailable predecessors. Since by assumption elements of U do not depend on other unavailable intersection attributes, all paths from elements of A to elements of NA must have terminated with passive attributes. So, the inputs to elements of NA may be inferred to be passive.

□

After the algorithm infers the availability of elements of NA, it propagates that information to all productions which have descendants which are elements of NA, and puts in an Evaluable set attributes which have active inputs but which were kept from being evaluable by a descendant of NA. Because the set U can be topologically sorted, each iteration of the algorithm will reduce it by at least one element, and so the algorithm will eventually terminate, after having looked at all attributes in the tree with active predecessors.

Chapter 4 - Non-local productions

4.1. Introduction

In Knuth's original formulation of attribute grammars, he stipulated that attributes may directly depend only on other attributes that appear in the same production instance. This assumption has turned out to be quite powerful; because of it a wide variety of analyses on attribute grammars can be neatly broken down and performed one production at a time. From an applied standpoint, however, the requirement of locality of dependencies can be burdensome. In this chapter the concept of non-local productions will be introduced. Use of non-local productions in the specification of languages will allow us to maintain some of the theoretical power of being able to conduct analyses on a production-by-production basis, while at the same time permitting more freedom in selecting the attributes upon which another attribute may depend.

After this introductory section, a definition of attribute grammars that contain non-local productions will be presented together with a few examples. Then an extension of the concept of characteristic graphs to non-local productions will be discussed. This extension will involve generalization of the concepts of IO and OI graphs, and of synthesized and inherited attributes. An algorithm for determining characteristic graphs for a certain class of non-local productions will be presented. Two particularly significant classes of non-local productions, those involving definitions and uses of identifiers, and those describing the abstract syntax of expressions, will be described in some detail, and extensions of the results of the previous chapters involving incremental attribute update will be presented. Finally, results involving "strongly optimal" attribute update in the presence of attribute grammars involving non-local productions will be presented.

4.2. Definition of Attribute Grammars Augmented by Non-local Productions

The standard definition of a context-free grammar is as a 4-tuple (N, T, P, S), where N is a finite set of "non-terminals," T is a finite set of "terminals," none of which are in N, P is a set of "productions" of the form

A ::= alpha,

where A is an element of N and alpha is in (N union T)*, and S is an element of N known as the "start symbol." For an extensive and clear discussion of context-free grammars, see Chapter 4 of [20]. We will add to the 4-tuple two more sets: a finite set of "interface symbols" A none of which are in N or T, and a finite set of non-local productions C. Also, we will modify the definition of P slightly. The elements of C are in A+, and will be written

alpha - beta - ... - delta.

Symbols in local and non-local productions may be subscripted by collections of interface symbols. (We call elements of the set P "local productions" where it is necessary to distinguish them from non-local productions.) If a symbol in a production is subscripted by an interface symbol, then the corresponding node in a parse tree must participate in an instance of a non-local production. For instance, if a grammar has a production

"<CONST ID> ::= ID_{defn}

and a non-local production

"defn - use",

a parse tree may have an instance of the first production, and the "ID" node may participate in an instance of the non-local production. The "ID" node will correspond to the "defn" interface symbol in the non-local production.

Interface symbol subscripts may optionally be labeled with either "+" or "*", indicating corresponding parse tree nodes may participate in one (respectively zero) or more instances of appropriate non-local productions. This capability is necessary, for instance, if one wishes a definition occurrence of an identifier to participate in a separate "defn - use" non-local production with each use occurrence of the same identifier.

We require that for each production instance in which a node in a parse tree participates, if the corresponding symbol of the production is subscripted by an interface symbol alpha, then the node must participate in an instance of a non-local production that has alpha as one of its elements. The node is associated with a position of the non-local production which has alpha as its element.

The purpose of introducing non-local productions is to have a vehicle to support non-local flow of attributes. We therefore permit specification of attribute evaluation rules in conjunction with non-local productions. Just as in the conventional attribute-grammar setting symbols have characteristic graphs for each way they can participate in productions, we will associate characteristic graphs with interface symbols.

Construction of non-local productions (a process somewhat analogous to parsing) will be automated through use of attribute grammars. The evaluation rules describing a language will be broken up into two parts: rules for describing construction of non-local productions, and rules for actual flow of information. A complete non-local attribute grammar is presented in the following figure.

S ::= Defs ; Uses	Uses.symtab := Defs.symtab;
Defs ::= INT id_{def} Defs	def.type := integer; $Defs_1$.symtab := AddId($Defs_2$.symtab, id.STRING_REP);
Defs ::= CHAR id_{def} Defs	def.type := character; $Defs_1$.symtab := AddId($Defs_2$.symtab, id.STRING_REP);
Defs ::= epsilon	Defs.symtab := empty;
Uses ::= id_{use} Uses	use.error := use.type ≠ integer; $Uses_2$.symtab := $Uses_1$.symtab; id.use := MakeLink($Uses_1$.symtab, id.STRING_REP);
Uses ::= epsilon	
def - use	use.type := def.type; def.error := use.error;

Figure 4.1.

A parse tree based on this grammar (before creation of non-local productions or attribute evaluation has taken place) will consist of a right-recursive tree of definition identifiers followed by a right-recursive tree of use identifiers. The "symtab" (symbol table) attribute starts empty from the lower right-most definition, and moves up the list of definition identifiers; at each step information necessary for creation of a non-local production involving the current definition identifier is concatenated to the symbol table attribute. In a Pascal implementation, the nodes might be heap objects, and the symbol table might consist of pointers to the definition identifier nodes together with some sort of hash-table encoding of the character string representations of the identifiers. When "symtab" has reached the top of the definitions list, it is passed to the uses list, where it works its way from the upper left-most down to the lower right-most use production. In each "Uses" production, the symbol table attribute is combined with the character representation of the use identifier to form an instance of the

"def - use" non-local production. At any point in the evaluation process, an evaluation function may give an error indication. The "AddId" and "MakeLink" routines could check for undeclared use identifiers, multiple definitions of the same identifier, and any other error or warning indications deemed appropriate. After creation of a non-local production, production instances that are linked together can be examined to determine which attributes of the newly linked nodes are available. If appropriate, the newly created production instance can be scheduled to have attributes evaluated.

The approach of unifying construction of non-local productions and flow of information will be seen to have several advantages. One can start with a conventional attribute grammar for a language, run performance tests on software based on it, and incrementally modify the grammar where it becomes evident that direct transmission of semantic information via non-local productions will result in time or space savings. Another advantage to the unified approach to non-local production building and information flow is that for some purposes it is helpful to be able to view a language description containing non-local productions as a conventional attribute grammar. We can always fall back to the view that information being conveyed via non-local productions was actually transmitted along the attribute flow paths used to build the non-local production. It is possible to use this technique, for instance, to create a non-circularity testing algorithm of non-local attribute grammars. One simply takes a non-local attribute grammar, assumes that information is flowing via production-building paths rather than via the non-local productions that could be built using those paths, and uses the standard non-circularity test of Knuth. Furthermore, in incremental evaluation there are some cases in which it is appropriate to ignore the presence of non-local productions in a parse tree and perform semantic update using the context-free re-evaluation techniques discussed in previous chapters.

Interface symbols can be thought of as a generalization of the concept of non-terminal symbols. An occurrence of a non-terminal in a production can be viewed as a specification of an interface between that production and a set of other productions in the grammar. Parse trees exist in which instances of productions P and Q can communicate attributes back and forth directly if the left-hand non-terminal of one production appears in the right-hand side of the other. Similarly, instances of productions P and Q can communicate directly with one another if a symbol of one production is subscripted with an interface symbol and the same interface symbol appears as one of the elements of the other production. Necessarily, the latter production must be non-local, since non-local productions have interface symbols as their elements, whereas normal (context-free) productions have grammar symbols as their elements. As noted above, we allow elements of an non-local production to be subscripted by interface symbols, but these subscripts will not be called elements of the the non-local production. The distinction between (unsubscripted) elements of a non-local production and subscripts of that production is roughly analogous to the distinction between left-hand side and right-hand side elements of a normal production.

Intuitively, we may think of each element of the interface symbol set as defining a unique plug and socket. If a symbol is subscripted by interface symbol alpha, then it can be thought of as having a socket of type alpha. If the interface symbol alpha appears as an element in a non-local production, then that non-local production can be thought of as having a plug which will fit into symbols with sockets of type alpha.

We illustrate the above with a few examples.

Example 1.
(1) S ::= A B_{α}
(2) S ::= A C_{β}
(5) A ::= a A
(6) A ::= D_{γ}
(7) A ::= E_{δ}

(8) $\alpha - \gamma$
(9) $\beta - \delta$
B ::= b; C ::= c; D ::= d; E ::= e

Figure 4.2.

The strings in the language of Figure 4.2 are { a^* d b } and { a^* e c }. Their derivation trees are shown in the figure below.

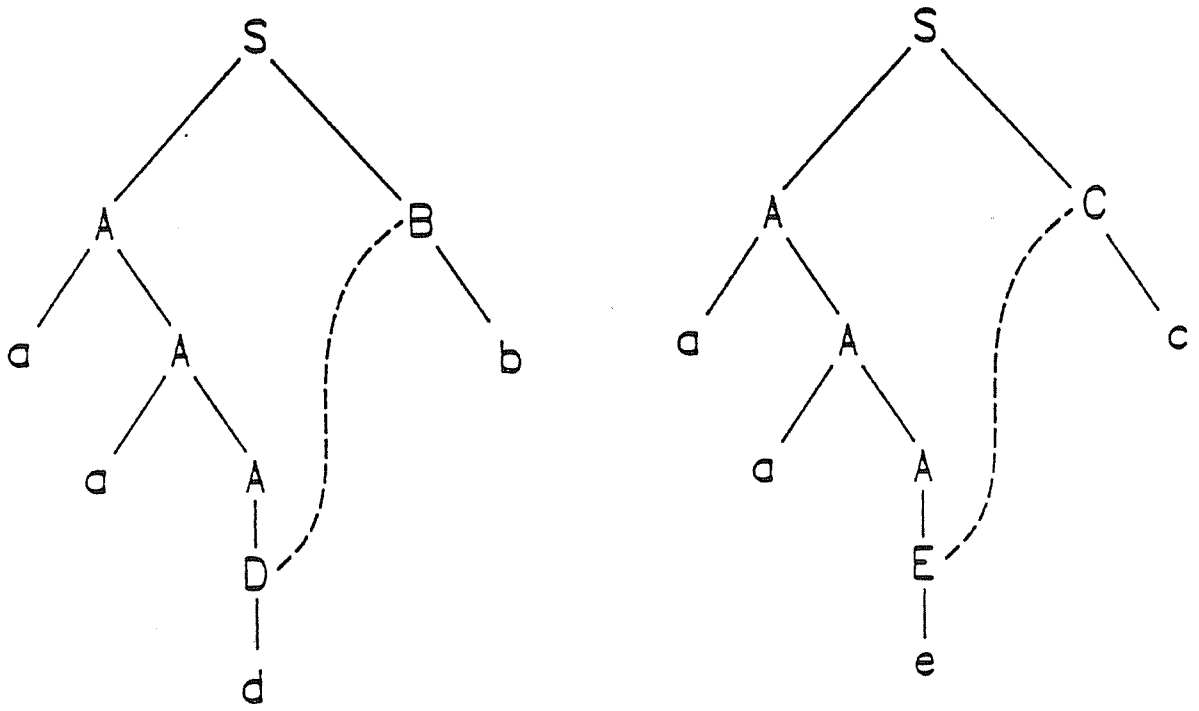

Figure 4.3.
Normal productions are shown with solid lines
Non-local productions are shown with dotted lines

If we tried to apply the production "A ::= E" to the right-most "A" in the left tree, then neither of the non-local productions would be applicable, and we would have nodes with subscripted grammar symbols that do not participate in non-local productions. That is why neither the string { a^2 b e } nor the string { a^2 c d } is in the language. A context-free grammar for the language of the example is easily derivable; on the other hand attribute evaluation rules could be associated with the non-local productions, and that would permit direct exchange of semantic information between nodes B and D (or C and E) without having to resort to possibly extensive collections of copy rules.

Example 2: We can describe the language { $a^n b^n c^n \mid n >= 0$ } as follows:

S ::= A B C

A ::= a_α

A ::= ε

B ::= b_β

B ::= ε

C ::= c_γ

C ::= ε

$\alpha - \beta - \gamma$

Figure 4.4.

For a particular string { $a^l b^m c^n$ }, we will have min(l, m, n) instances of the non-local production, and if l, m, and n are not all equal then we will have a node with a subscripted grammar symbol that does not participate in a non-local production, implying that the string is not in the language specified in the figure.

4.3. Characteristic Graphs for Interface Symbols

It is possible to use non-local productions to drastically change the attribute flow characteristics of a parse tree. For instance, one could form non-local productions which involve definitions and uses of identifiers. Then one could have

type information flow directly from the definition occurrence of an identifier to each of its uses, rather than requiring the type information to flow up and down the parse tree. We will rely on characteristic graphs in the non-local setting to guide incremental re-evaluation just as we did in previous chapters for context-free re-evaluation. Furthermore, evaluation plans can be created for non-local productions using characteristic graphs just as they were for context-free productions.

In the case of context-free connections between productions, we have two characteristic graphs, the IO graph and the OI graph, depending on whether we focus attention on the production below or the production above the parse tree node they have in common. In defining characteristic graphs for interface symbols, we will have an analogous situation. An interface symbol of a parse tree node represents a path for communication between two different production instances which have that node in common. Thus an interface symbol will have two characteristic graphs, and they will correspond to the two productions which are communicating via the interface symbol. In one of the productions, the interface symbol is a subscript, and we will (arbitrarily) refer to that characteristic graph as the OI graph. The other production must be a non-local production, and must have the interface symbol as one of its elements. We will call the characteristic graph of the element of the non-local production an IO graph.

In addition to the extension to the concept of characteristic graphs, we will extend the customary classifications of synthesized and inherited attributes of parse tree nodes. In the case of attributes that are inputs and outputs to grammar symbols, we retain the standard definition of synthesized and inherited attributes. In the case of attributes which are inputs or outputs to interface symbols, we make an arbitrary decision which is consistent with our choice of how

to classify IO and OI graphs for interface symbols. An attribute is defined to be an inherited attribute of an interface symbol if it is an output to a subscripted occurrence of the interface symbol. Conversely, an attribute is a synthesized attribute of interface symbol if it is an output to an occurrence of the interface symbol as an element of a non-local production. The example below shows a simple attribute grammar together with the inherited/synthesized classifications of attributes and the characteristic graphs of symbols.

$S ::= A_{foo}$	$A.\beta := foo.\alpha$ $foo.\delta := A.\gamma$
$A ::= a$	$A.\gamma := A.\beta$
foo	$foo.\alpha := 47$ $foo.\varepsilon := foo.\delta$

Figure 4.5.
A non-local attribute grammar.

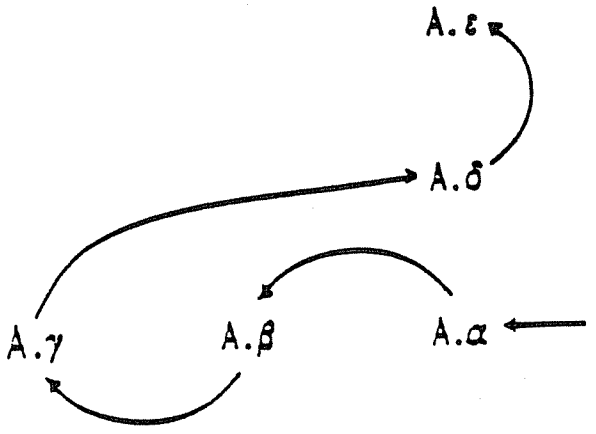

Figure 4.6.
Attribute dependencies at node A for a parse tree derived from the attribute grammar of Figure 4.5.

	Synthesized	Inherited
A	γ	β
foo	α, ε	δ

Figure 4.7.
Synthesized and inherited attributes of symbols of the attribute grammar in Figure 4.5.

We define define characteristic graphs for grammar and interface symbols in the non-local setting in a manner analogous to the method used for their definition in the context-free setting. We start by constructing a parse tree with a node labeled by the symbol for which we desire to find a characteristic graph. (In all that follows we will engage in a slight abuse of terminology and refer to a parse tree together with all its non-local productions as simply a parse tree. We might more properly refer to such objects as "parse graphs" since arcs corresponding to non-local productions will cause them no longer to be trees, as can be seen, for instance, in Figure 4.3.) Then, we prune out of the compound dependency graph of the parse tree the attribute dependencies that come from the production instance opposite the one for which we are trying to find a characteristic graph. If we are computing the characteristic graph of a subscripted interface symbol, for instance, we trim out the dependencies from the non-local production instance which has that interface symbol as an element. We then form the projection of of the reduced compound dependency graph on the synthesized and inherited attributes of the symbol, and this is the desired characteristic graph. Characteristic graphs for the parse tree of Figure 4.4 are presented in the figure below.

IO graph	OI graph
foo UI -> α; δ -> ε A β -> γ	ε; α -> δ γ; UI -> β

Figure 4.8.
Compound dependency graphs for symbols of the attribute grammar of Figure 4.5.

As a further example of determination of characteristic graphs for interface symbols, consider the following the following grammar involving definitions and uses of identifiers.

S ::= Defs ; Uses	Uses.symtab := Defs.symtab;
Defs ::= INT id_{def} Defs	def.type := integer; $Defs_1$.symtab := AddId($Defs_2$.symtab, id.STRING_REP);
Defs ::= CHAR id_{def} Defs	def.type := character; $Defs_1$.symtab := AddId($Defs_2$.symtab, id.STRING_REP);
Defs ::= epsilon	Defs.symtab := empty;
Uses ::= id_{use} Uses	use.error := use.type ≠ integer; $Uses_2$.symtab := $Uses_1$.symtab; id.use := MakeLink($Uses_1$.symtab, id.STRING_REP);
Uses ::= epsilon	
def - use	use.type := def.type; def.error := use.error;

Figure 4.9.

We intend that each definition of an identifier bind to exactly one use of that identifier, and that there be no duplicate definitions. These rules must be enforced by the functions "AddId" and "MakeLink". So, "INT a; a" is a valid string, whereas none of "INT a; b", "INT a; a a", or "INT a; INT a; a a" is valid. The parse tree for "INT a; a" is shown below.

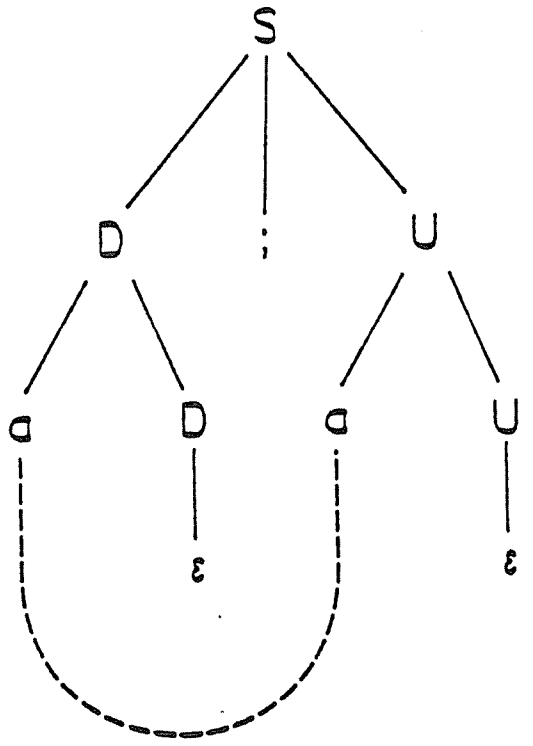

Figure 4.10.
Context-free links are solid lines.
The non-local production is a dotted line.

The corresponding compound dependency graph for the tree is given in the following figure.

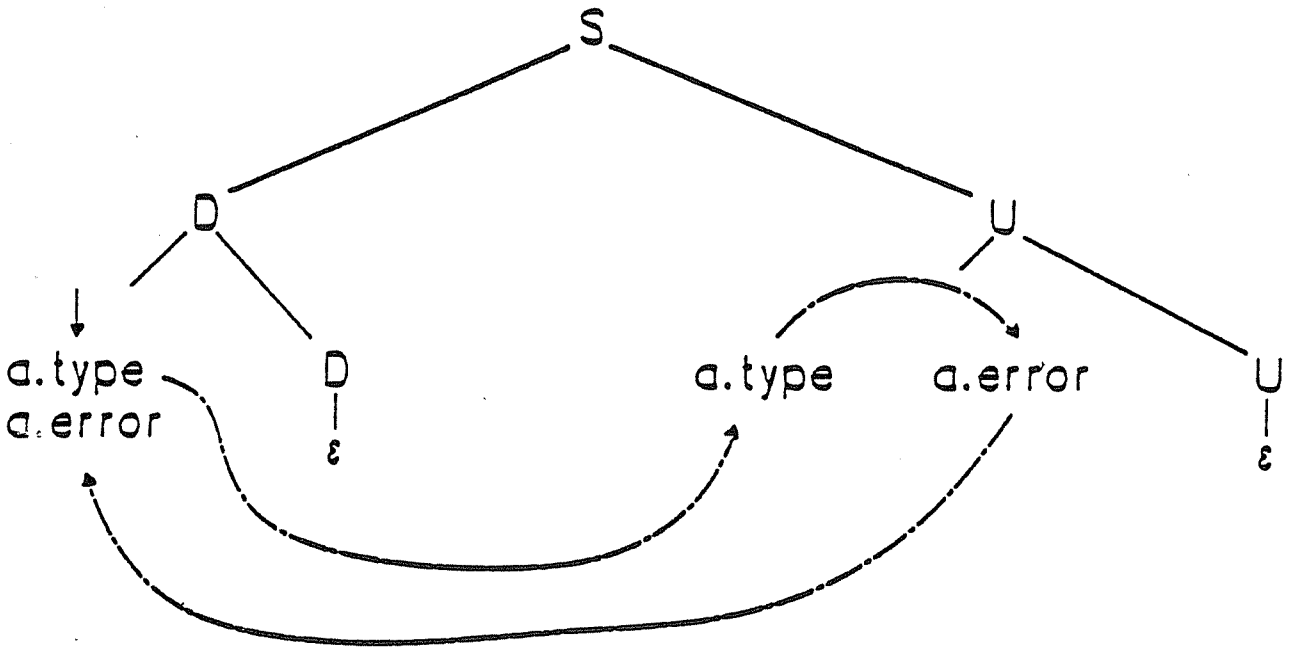

Figure 4.11.
Compound dependency graph for the parse tree of Figure 4.10.

To obtain a complete set of characteristic graphs for this grammar, we must compute the characteristic graphs for the interface symbols "def" and "use" in addition to the IO and OI graphs for the various grammar symbols. Note that IO/OI graph computation Algorithm 2.1 cannot be used in this augmented setting; that algorithm depends on the assumption that no attribute information can be introduced into a production instance from "below" a leaf node, whereas the terminal "asu*buse*" receives information through the non-local link.

First, we determine the OI graph of the symbol "def" for the parse tree by deleting the dependency graph of the other production instance in which the node participates, "def - use", and forming the projection on "def". This results in the graph

UI -> def.type; def.error

Next, we determine the characteristic graph for the interface symbol "use": deleting the dependency graph for the production "U ::= id_{use} U" and projecting results in the graph

UI -> use.type; use.error.

Repeating the procedure gives

use.type -> use.error

as the OI graph for "use", and

def.type -> def.error

as the characteristic graph for the interface symbol "def".

We will present below an extension of Algorithm 2.1 which computes characteristic graphs for a class of attribute grammars that contain non-local productions. It should be noted first, however, that there are some non-local attribute grammars to which the algorithm will not be applicable; moreover there are some grammars involving non-local productions for which the very concept of a characteristic graph is of questionable meaning. However, our intended use of characteristic graphs is for construction of incremental evaluation algorithms and evaluation plans for Pascal-like languages. The subset of non-local attribute grammars for which the algorithm of this section will be meaningful is large enough to encompass the non-local techniques we wish to employ for program-oriented editors for conventional languages.

We will define a class of non-local attribute grammars for which characteristic graphs are meaningful, and to which an extension to Algorithm 2.1 will be applicable. In order to motivate the definition, we present an example of

the sort of anomaly we wish to rule out. The attribute grammar is presented in Figure 4.12 below, and a parse tree and corresponding compound dependency graph are shown after that.

$S ::= A_{\alpha}$	A.a := 1; A.b := $_{\alpha}$.a;
A ::= B	B.a := A.a;
B ::= b_{β}	β.a := B.a;
$\alpha = \beta$	α.a := β.a;

Figure 4.12.
A non-local attribute grammar with node-level circularity.

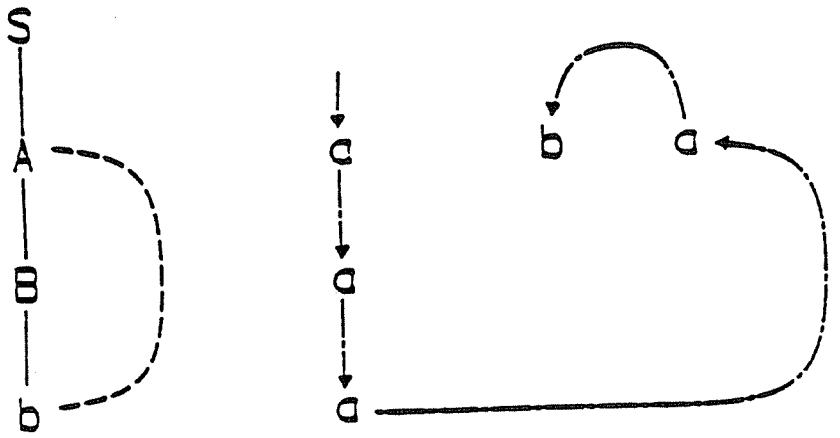

Figure 4.13.
A parse tree and compound dependency graph for the grammar of Figure 4.12.

Node A in the example receives attribute "b" as an output of the production "S ::= A", and "b" depends indirectly on attribute "A.a", which is an input to the production "A ::= B". In this example we have lost the production-by-production modularity which is so powerful for various analyses. The attribute dependency behavior of part of the tree cannot be summarized by a characteristic graph which

is completely independent of the rest of the tree. The outputs of part of the tree no longer depend exclusively on inputs to that part of the tree. In order to restrict ourselves to attribute grammars in which the notion of characteristic graphs is meaningful, we will require that the attributes of a node which are outputs of a given production instance depend only on inputs to the same production.

In order to construct an algorithm analogous to Algorithm 2.1 for the non-local setting, we will formalize a notion of "node-wise non-circularity" on non-local attribute grammars which eliminates situations like that presented in the previous paragraph. Unfortunately, we do not have a general algorithm which tests a non-local attribute grammar for this property, since the property depends on behavior of evaluation functions, which are assumed to be "black boxes." We start from a symbol N in a parse tree, pick some attribute N.alpha, and follow a dependency arc backwards to one of N.alpha's predecessors. Say the predecessor we pick is attribute beta, associated with symbol N'. We then pick an attribute of N' which is an output of the same production as beta, and follow one of its predecessor arcs. We continue the process until we either return to symbol N or come to an attribute with no predecessors. (If it should happen, for instance, that N equals N', then we stop after one step.) If it happens that we stop because we come back to N, we will require that the first and last dependency arcs in our chain be associated with the same production instance. If all parse trees derived from an attribute grammar have this property, then we will say that the attribute grammar is "node-wise non-circular." Notice that the above grammar is not node-wise non-circular, since we can get from "A.b" to "A.a" going backward along dependency arcs, and the first and last arcs are not associated with the same production instance. The algorithm for computing characteristic graphs for non-local attribute grammars will depend on the assumption that input grammars satisfy the node-wise non-

circularity assumption.

In computation of characteristic graphs for non-local attribute grammars, we will require a notion of the symbols that are "reachable" from a given symbol in a production. At each iteration of the main loop of algorithm , we will compute a characteristic graph of a symbol of a production. The reachable symbols of the production are the ones for which we require previously computed characteristic graphs in order to compute a new characteristic graph. We start with an output attribute of the given symbol, and move backward along an arc of the dependency graph of the production to an attribute of another symbol. We then select an input attribute of that symbol and repeat the process, moving back along a dependency graph arc. The reachable symbols are those which can be reached by this process.

```
Input:  a node-wise non-circular attribute grammar
Output: the set of IO and OI graphs for each symbol in the grammar

{initialization}
for each production P which has a symbol "I" that is not reachable
  from any other symbols in P,
do begin

  G := the projection of P's dependency graph on position "I";

  if position "I" is a grammar symbol, then
    if i = 0
    then add G to the IO graph set of symbol "I"
    else add G to the OI graph set of symbol "I";

  else { position "I" is an interface symbol }
    if position "I" is a subscript
    then add G to the IO graph set of symbol "I"
    else add G to the OI graph set of symbol "I";

end for;

{main loop}
while there remains an unexamined 3-tuple
    ( production P,
      symbol "I" in P,
      {characteristic graphs C_j for reachable positions
      of P other than the symbol at position "I"}
    )
do begin

  let D be the dependency graph for P augmented by the selected
  characteristic graphs;

  G := the projection of D on position "I";

  if position "I" is a grammar symbol, then
    if i = 0
    then add G to the IO graph set of symbol "I"
    else add G to the OI graph set of symbol "I";

  else { position "I" is an interface symbol }
    if position "I" is a subscript
    then add G to the IO graph set of symbol "I"
    else add G to the OI graph set of symbol "I";

end while;
```

In Algorithm 2.1, of which Algorithm is an extension, we were able to prove that the algorithm would generate a characteristic graph if and only if there was a

parse tree with that characteristic graph. In the current setting we will lose implication in one direction; the algorithm can be given a non-local attribute grammar that will cause it to produce graphs which are not characteristic graphs of any parse tree derivable from the grammar. Fortunately, we are still able to prove that for any parse tree the algorithm will find all possible characteristic graphs. In practice, attribute grammars which violate the implication in the one direction have not arisen, but even if they did, the only cost would be that the evaluator generator would invest time and table size planning for situations that could never arise at evaluation time.

Theorem 4.1. Let "I" be a symbol of some parse tree derived from a node-wise non-circular non-local attribute grammar. Then the characteristic graph of "I" will have been computed by the algorithm.

Proof. If the contrary is true, then one of the reachable interfaces of the production to which N interfaces via "I" must also have a characteristic graph not generated by the algorithm. Since by the assumption of node-wise non-circularity there are no circular chains of reachable predecessors, the two characteristic graphs are distinct. We continue tracing through reachable predecessors, and are guaranteed (by assumption) that we will not form a circular chain. Consequently, by the finiteness of the tree, we must reach a predecessor which itself has no predecessors, but for which the algorithm found no characteristic graph. But this is impossible, since during initialization the algorithm found characteristic graphs for all interfaces with no reachable predecessors.

□

Chapter 5 - Attribute Evaluation for Non-local Attribute Grammars

We now address the issues of initial and incremental attribute evaluation in the presence of non-local productions. In the first section after this introduction the related issues of building non-local productions and initial attribute evaluation will be discussed. After discussing initial evaluation, we will turn to the problem of incremental evaluation. An incremental evaluator for an example language that has non-local productions for definitions and uses of identifiers will be presented. The incremental evaluator will use evaluation priorities as discussed in chapter 3. In the next section an algorithm will be presented for translating a non-local attribute grammar into an equivalent local attribute grammar; non-local evaluation priorities will be obtained from the priorities of the equivalent local attribute grammar. In the final section, a condition under which incremental evaluation can use non-local productions will be stated, and an incremental evaluator which builds and uses non-local productions will be presented.

5.1. Initial evaluation and creation of non-local productions

As mentioned in the previous section, non-local attribute grammars will have two components, that dealing with construction of non-local productions, and that dealing with flow of data. We present in Figure 5.1 a re-capitulation of the attribute grammar involving definitions and uses of identifiers of Figure 4.1.

S ::= Defs ; Uses	Uses.symtab := Defs.symtab;
Defs ::= INT id_{def} Defs	def.type := integer; $Defs_1$.symtab := AddId($Defs_2$.symtab, id.STRING_REP);
Defs ::= CHAR id_{def} Defs	def.type := character; $Defs_1$.symtab := AddId($Defs_2$.symtab, id.STRING_REP);
Defs ::= epsilon	Defs.symtab := empty;
Uses ::= id_{use} Uses	use.error := use.type ≠ integer; $Uses_2$.symtab := $Uses_1$.symtab; id.use := MakeLink($Uses_1$.symtab, id.STRING_REP);
Uses ::= epsilon	
def - use	use.type := def.type; def.error := use.error;

Figure 5.1.

It might be noted that evaluation of "type" attributes for definition identifiers can occur before, during, or after evaluation of the "symtab" attributes. Specifically, an id_{def} node may receive type information before a "def - use" non-local production involving that node is created. This situation is analogous to evaluation of independent inherited attributes of a parse tree node before the subtree of the node has been created. In general, attributes which are inputs to a section of a parse tree may be evaluated before the section has been built, provided, of course, that they do not depend on outputs of the new section. The point to be emphasized is that the node receiving the "type" attribute exists and is able to receive attributes even though the evaluation function might make it appear that some new node which springs into existence on creation of the "def - use" production instance receives the attribute. Non-local productions do not cause new parse tree nodes to be created; rather, they link existing parse

tree nodes together in new ways.

The algorithm which implements the initial evaluation and creation of non-local productions discussed in the previous paragraphs appears below.

Algorithm 5.1.
Input: An unevaluated parse tree from a well-formed non-local attribute grammar.
Output: The tree with non-local productions built and attributes evaluated.

Put production instances with attributes of in-degree zero in Evaluable.

while Evaluable is not empty,
  select and remove an element P from Evaluable;
  Evaluate the attributes of Plan(P);

  if one of the evaluated attributes is actually an interface symbol,
  create the non-local production and initialize the attribute
  availability set by looking at the status of the other productions
  in which the nodes of the new production participate. Put the new
  production in Evaluable if its Plan is non-empty.

  Update the status of neighbors and put them in Evaluable if they
  have evaluable attributes.

The algorithm of the above figure is identical to Algorithm 2.2, except for the possibility of creation of non-local productions. The proof of correctness is identical to that of Algorithm 2.2 after we show that newly created non-local productions are in the Evaluable set if and only if they have evaluable attributes. The latter follows from the fact that newly created non-local productions have their attribute availability vectors initialized after they are created; it is possible at that time to determine whether a new non-local production has any unavailable output attributes all of whose inputs are available. Since no attributes are actually evaluated after creation of a non-local production, the evaluability states of other productions are not affected.

5.2. Incremental evaluation in the presence of non-local attributes

In priority-based incremental evaluation, we impose a structure on the pool of attributes to be evaluated, and are thus able to infer in some cases that all predecessors of a given attribute have their final values without going to the expense of explicitly verifying them. For instance, say some attribute A.α in a parse tree has an input(A.β_1 which has changed value, together with several other inputs A.β_2, ..., A.β_n about which we know nothing. If in analyzing the attribute grammar we have noticed that of the various attributes in the tree which have received new values none of them can possibly be ancestors of any of A.β_2, ..., A.β_n, we can evaluate A.α with confidence that all of its inputs have their final, correct values. In the context-free setting of previous chapters, this sort of analysis was not strictly necessary; we could always explicitly verify every input to an attribute if required to do so. This verification process depends on the fact that in context-free grammars the only way the parts of a parse tree that are connected together by a given production instance can communicate is through that production instance. For instance, say we have a language which allows sub-range types, but the bounds may only be manifest integer constants. Part of the grammar might be

```
<subrange type> ::= <int constant> .. <int constant>
    <subrange type>.type := CheckSubRng( <int constant>1.value,
                                         <int constant>2.value )

<int constant> ::= <optional sign> INT_TOKEN
    <int constant>.value := MakeNum( <optional sign>.sign,
                                     INT_TOKEN.StringRep )
```

Say an editing operation is performed which changes "1 .. 100" to "0 .. 100". The left <int constant> non-terminal will have a new value attribute, requiring inclusion of the '<subrange type>' production instance in the interior set. From the facts

that

(1) the right <int constant> receives no inputs from the <subrange type> production and

(2) the only way inputs may be presented to that subtree is from the <subrange type> production

we may infer that the outputs remain the same, i.e. that the value attribute of the right <int constant> node has its correct, final value.

Now let us change the situation somewhat and say that sub-range types may have bounds that are declared constants, and further that the declared constants participate in "def - use" non-local productions. Say we have a program fragment such as

```
const a = 96;
...
type HashIndex = a .. b;
```

If the user changes "a" from 96 to 97, eventually the <const> non-terminal above "a" will have an active "value" input, but we cannot infer that the <const> above "b" has a correct "value" attribute, since there is a source of input to that subtree about which we can learn nothing by looking at the <subrange type> production instance. That source of input is the non-local production involving the identifier "b". If the definition of "b" does not depend on "a", then <const>.value is independent of any active attributes, and we must infer that it is available for use in the computation of other attributes which depend on it. On the other hand, there might be some intervening declaration of the form

```
const b = -a;
```

in which case the "value" attribute of the <const> node above "b" does depend on active attributes, and we must infer that it is unavailable until it is re-computed

or becomes dominated by passive attributes.

The above discussion makes the difficulty of trying to rely on explicit verification of input attributes apparent. On the other hand, it is possible to extend the priority-based approach to the non-local case. That approach relies on a determination of priority classes for attributes, and in some cases a requirement that certain classes be evaluated in some fixed order. The "def - use" non-local production will fit into this scenario naturally, since in several high-level languages such as Pascal and Ada an identifier must be defined lexically to the left of any uses. We will allow order relations among nodes which participate in non-local productions to be specified to the evaluator generator, which will incorporate this information into its priority calculations.

As an example of a priority-based incremental evaluator for a grammar with non-local productions, we will examine a small language which allows declarations of constants, subrange types, and variables, and which has assignment statements. Subsequently we will add array declarations as an example of a more complex data structure. The first (fairly extensive) example appears below.

```
S ::= Defs $ Uses                    Defs.symtab := empty;
                                     Uses.symtab := Defs.newtab;

Defs ::= Def ; Defs                  Def.symtab := Defs1.symtab;
                                     Defs2.symtab := Def.newtab;
                                     Defs1.newtab := Defs2.newtab;

Defs := epsilon                      Defs.newtab := Defs.symtab;

Def ::= CONST id_def = <int or id>
                                     Def.newtab := AddId( Def.symtab,
                                                          id.STRING_REP );
                                     <int or id>.symtab := Def.symtab;
                                     def.type := <int or id>.type;

<int or id> ::= id_use               id.use := MakeLink( <int or id>.symtab,
                                                         id.STRING_REP );
                                     <int or id>.type := use.type;

<int or id> ::= INT_TOKEN            <int or id>.type := Int_Type;

Def ::= TYPE id_def = <type>         Def.newtab := AddId( Def.symtab,
                                                          id.STRING_REP );
                                     <type>.symtab := Def.symtab;
                                     def.type := <type>.type;

Def ::= VAR id_def = <type>          Def.newtab := AddId( Def.symtab,
                                                          id.STRING_REP );
                                     <type>.symtab := Def.symtab;
                                     def.type := <type>.type;

<type> ::= <int or id> <.. int or id>
                                     <type>.type := SubRng( <int or id>.type,
                                                            <.. int or id>.type );
                                     <int or id>.symtab := <type>.symtab;
                                     <.. int or id>.symtab := <type>.symtab;

<.. int or id> ::= .. <int or id>
                                     <int or id>.symtab := <.. int or id>.symtab;
                                     <.. int or id>.type := <int or id>.type;

<.. int or id> ::= epsilon           <.. int or id>.type := NoType;

Uses ::= Use ; Uses                  Use.symtab := Uses1.symtab;
                                     Uses2.symtab := Uses1.symtab;

Uses ::= epsilon

Use ::= id_use <= id_use             id1.use := MakeLink( Use.symtab,
                                                          id1.STRING_REP );
```

id_2.use := MakeLink(Use.symtab,
id_2.STRING_REP);
"<=".error := id_1.type
<> id_2.type;

def - use

use.type := def.type;

Figure 5.2

Subtree replacements on strings from this language fall into two categories that will be handled differently: those that insert or delete "id_{def}" nodes, and those that do not. A change of a definition occurrence of an identifier affects the entire program string to its right, possibly causing radical changes both in non-local productions and information flow. Handling a change of this sort using non-local incremental update is quite complex, and the changes made are basically the ones that would have to be made using context-free incremental update. Consequently, changes to definition identifiers will be handled using context-free re-evaluation techniques. Other changes will be handled using non-local techniques. Changes that do not involve definition identifiers can be handled in a relatively straightforward way using non-local productions, and in some cases the number of nodes that must be visited is far smaller than in the context-free setting. Specifically, changes to right-hand sides of declarations will be propagated to locations they affect via non-local productions.

There are only two productions in the example which have attributes with more than one input:

<type> ::= <int or id> <.. int or id>,

and

Use ::= id_{use} <= id_{use}.

The join attribute of the second production potentially depends on every other attribute in the grammar, so it must have the lowest evaluation priority. Since no instance of the attribute can depend on other instances of the same attribute (the attribute has no successors), the priority class can be unordered.

The join attribute of the subrange type production is a bit more interesting. It may depend on all attributes in the grammar except those associated with the Use productions. Furthermore, <type>.type may depend on other instances of <type>.type, implying that its priority class must be ordered. We may infer from the grammar that an instance of <type>.type may depend only on another instance of <type>.type to its left in the following way: first, <type>.type depends on attributes of nodes "id_{use}" that are children of the <type> node. Second, the "id_{use}" nodes receive attributes from "id_{def}" nodes that appear to the left of the <Def> node which is the parent of <type>. This latter fact is obtained from the part of the grammar used to build "def - use" non-local productions. The "symtab" attribute which flows down through the <type> node to "id_{use}" nodes below it does not depend on the "id_{def}" in the same right-hand side as the <type> node. It depends only on id_{def} nodes further to the left. Each of those nodes in turn depends on a <type> node immediately to its right.

The net result of the above discussion is that attributes "<-".error have lowest priority, and that attributes "<type>.type" have next lowest priority. Elements of the latter priority class must be retrieved in depth-first, left-to-right order. All other attributes have highest priority.

As an extension of the above example, we may allow arrays in the language by adding a production

<type> ::= ARRAY [<type>] of <type>.

We will require that the <type> inside brackets be a subrange type, and will allow the right-most type to be either a subrange or an array declaration, so that multi-dimensional arrays can be declared. The attribute evaluation rule for the above production is

$$\langle type\rangle_1.type := ArrayType(\ \langle type\rangle_2.type,\ \langle type\rangle_3.type\ ).$$

As it stands, this definition of array declarations presents a problem: the $\langle type\rangle_1$.type attribute depends on occurrences of <type>.type to its left as it did in the preceding example, but now it also depends on <type>.type attributes of nodes to its right, namely those in its subtree. This requires a slight re-definition of our node ordering scheme. We simply use a post-order traversal, in which a node is considered to have been visited after, rather than before, its children in a depth-first, left-to-right traversal of a parse tree. Then, once again, <type>.type depends on other occurrences of <type>.type to its left in the parse tree.

5.3. Translation of non-local attribute grammars into local attribute grammars

In this section we present a method for translating a non-local attribute grammar into an equivalent attribute grammar that has no non-local productions. Our primary purpose for this translation is to use context-free priority creation algorithms; priorities for non-local incremental re-evaluation can be gotten by applying priority computation schemes of previous chapters to translated non-local attribute grammars. Of secondary importance (from a practical standpoint) is the ability to perform non-circularity tests on non-local attribute grammars. If we can translate a non-local attribute grammar into an equivalent local attribute grammars, then we can apply the non-circularity test of Knuth [8].

The algorithm works iteratively, removing one non-local production at a time. It first finds a non-local production whose construction in a parse tree does not depend on the presence of any other non-local productions in that parse tree. This can be done by finding a subscript in an output position which does not depend on any attributes in the grammar which are outputs of non-local productions.

When a non-local production has been selected, we consider every attribute which participates in creation of that non-local production. Such an attribute must have an element of the non-local production as an ancestor and another element as a descendant. For each such attribute, we build a set of copy rules into the productions which possess it. Attributes such as "SymTab" contain information necessary to construct a set of non-local productions; we will construct a corresponding set of attributes which contain the information that would have been communicated via those non-local productions.

When we have performed the above operation on all non-local productions whose existence in a tree does not depend on the presence of other non-local productions, then we take the modified attribute grammar and repeat the procedure. The procedure is applied iteratively until all non-local productions have been removed, and information that had been communicated using them is now communicated along the local productions of the parse tree.

If at some point in the above program we should encounter two non-local productions the construction of each of which depends on the prior presence of the other, then we conclude that the non-local attribute grammar is ill-formed.

The algorithm described in last few paragraphs is summarized below.

Algorithm 5.2.
Input: A non-local attribute grammar.
Output: An equivalent local attribute grammar.

while there is still a non-local production in the grammar,
do begin

Pick a non-local production P whose construction attributes depend on no attributes output by non-local productions.

For each evaluation rule in P, create a set of copy rules which parallel the construction attributes of P and take the arguments of the evaluation rule to the node which is to receive the output of the evaluation rule.

Make the evaluation function of P take as inputs the results of the copy rules and produce as output the appropriate attribute of the node which received the attribute in the non-local production. Tag the attribute name with the name of the interface symbol of the non-local production.

Remove P from the grammar.

end.

In order to establish evaluation priority classes for non-local attribute grammars, we transform the non-local attribute grammar into a local attribute grammar and apply the priority computation algorithm to the transformed version.

6. Incremental evaluation in the presence of multiple intersection nodes

The final issue which must be resolved is the determination of which subtree replacements will be handled using context-free incremental evaluation and which will be handled using non-local techniques. After a change is made which results in modification of non-local productions, we are faced with a situation in which there are several intersection nodes, rather than the situation of a single intersection node studied in previous chapters. A certain class of tree modifications which require re-evaluation in the presence of several intersection nodes can be handled with the same set of priorities computed for the case of a single intersection node. We will illustrate with an example.

(1) E ::= E + T E_1.type := DeriveType(E_2.type, T.type);

(2) E ::= T E.type := T.type;

(3) T ::= INT T.type := int_type;

(4) T ::= CHAR T.type := char_type;

The "+" function is assumed to be overloaded. If applied to integers, it means integer addition, and if it is applied to characters or strings it means string concatenation. If an integer and a character are given as arguments to "+", then the "type" attribute is given the value "error_type." The attribute E_1.type has two inputs, so it must be put in a priority class below the attributes that it can depend on, and since it can depend on another instance of itself its priority class must be oriented.

Now, say that we have a collection of several attributed tree fragments of the form

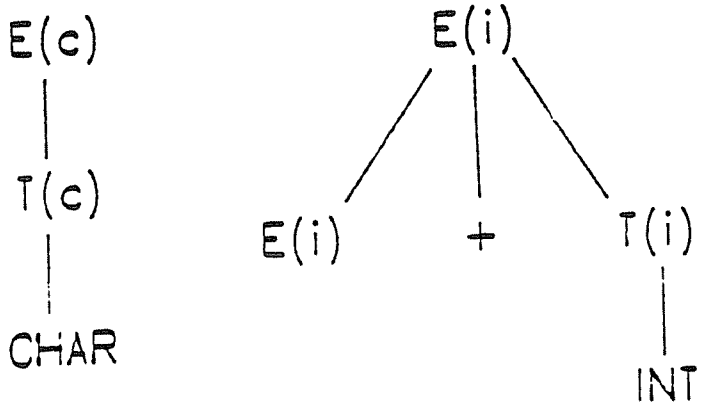

Figure 6.3.

The value of the 'type' attribute ("i" for int_type, 'c' for char_type) is indicated

in parentheses next to each symbol.

We paste the left fragment above together with three instances of the right fragment to obtain the tree

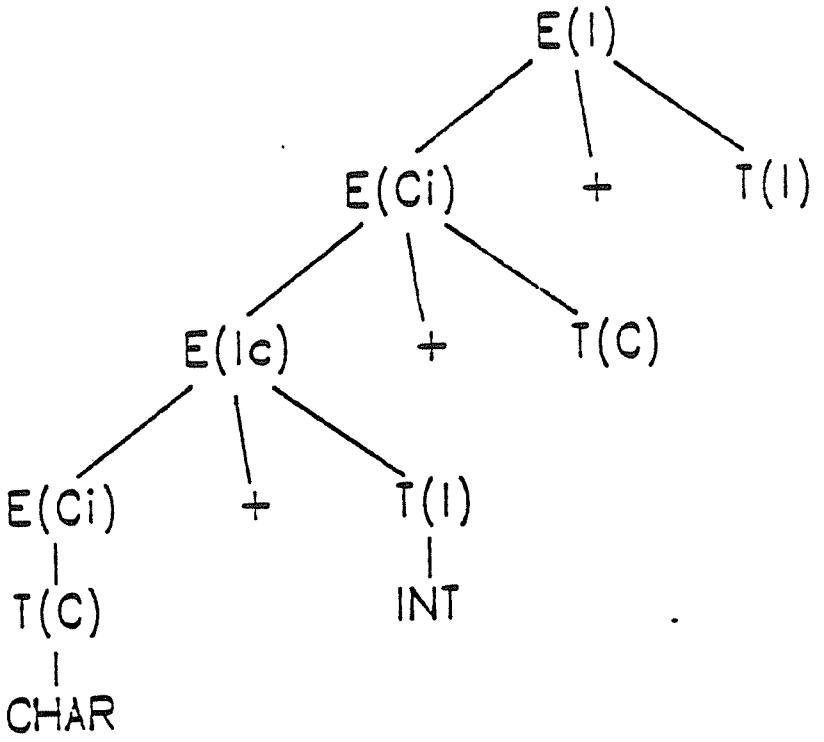

Figure 6.4.

The intersection nodes have two versions of the 'type' attribute, as is indicated in Figure 6.4. The version that supersedes is capitalized. We take independent attributes of the intersection nodes, determine whether they are active or passive, and place the active ones in Evaluation sets according to their priorities. In this example all attributes are independent, since we have no chains of attributes which go through a set of nodes and then turn around and go back through the same set of nodes. The intersection nodes are the three lower 'E' nodes, and all three of them have active 'type' attributes. The evaluation loop

(get a high-priority attribute, evaluate it, place dependents in evaluation sets if it changes value) is then iterated until all evaluation sets are empty.

It is possible that an independent intersection attribute may receive a new value in this re-evaluation process, unlike those of previous chapters in which we had only one intersection node. In this example, all of the the "E" nodes except the bottom one receive new versions of the "type" attribute, namely the value "error_type." Even so, in this case we are not in danger of using an attribute as an input to an evaluation function and then having it change at a later time in the same evaluation process. This is because the "E.type" attributes are in an oriented priority class. Even though several independent intersection attributes of different nodes are in the evaluation set, they will be retrieved in the pre-determined order.

On the other hand, it is easy to construct examples in which the priority classification will not keep us from using an attribute before it has its final value in the presence of multiple intersection nodes. Such an example appears below.

S ::= A	
A ::= a A	A_1.count := A_2.count + 1;
A ::= a	A.count := 1;

In this example we simply have a string of a's, and the "count" attribute counts the number of a's in the string. There are no join attributes, and so there is a single unordered priority class. If we take several internally consistent fragments from this grammar, paste them together, and try to apply the evaluation scheme outlined above, we will very likely run into problems. All of the "count" attributes are independent, so whenever two versions of "count" in an intersection node differ, that attribute instance will be placed in the evaluable set. Attribute values travel up the tree here as they did in the previous example, but in this case the priority class is unordered, so the evaluator is free to evaluate a "count" attribute

high in the tree. If an attribute below that is also active, then we will give a new value to an attribute which was used as an input to an evaluation function. This is the situation which gives rise to non-linear behavior in the evaluation process, and the which we are trying to avoid.

We are still restricting the user to making changes which consist of replacement of one subtree by another, but each such change may give rise to a modification of non-local production instances. We will perform re-evaluation after tree modification using multiple intersection nodes and non-local productions if the following condition is met:

(*) the priorities established for evaluation in the attribute grammar by the method of the previous section will ensure that no attribute is used as an input to an evaluation function and subsequently given a new value.

In POE (and in the extensive example given at the beginning of this chapter) changes that do not insert or delete definition identifiers satisfy the condition of the previous paragraph, whereas changes that affect definition identifiers do not. Consequently, we handle all of the former using non-local productions, whereas changes of the latter variety will be handled using the equivalent local attribute grammar and context-free re-evaluation techniques discussed in previous chapters. Included in the set of changes which will be handled with non-local productions are changes which affect the abstract syntax of expressions. The attribute flow rules for expressions in POE are essentially those presented in the example of expressions given at the beginning of this section.

At this point, classes of changes must be looked at on a case-by-case basis to determine whether they satisfy the above property. It is hoped that further research will uncover an algorithm which automatically determines which classes of changes must be handled using context-free re-evaluation and which can make

use of non-local productions.

The re-evaluation algorithm for non-local productions is given below.

Input: A fully attributed tree possibly containing non-local productions, a designated subtree which is to replaced, and an internally consistent subtree.
Output: A fully attribute, consistent tree with the new subtree inserted in place of the old one.

Evaluate that part of the attribute grammar which builds non-local productions. For every new non-local production that is built, mark the nodes it combines as new intersection nodes.

For each intersection node, compare new and old versions of independent attributes and place active attributes in appropriate evaluation sets. Mark dependent attributes whose old and new versions differ as being unavailable.

Perform the evaluation loop of Algorithm 3.5.

That the algorithm works correctly follows immediately from the assumption that the change which necessitates the re-evaluation satisfies condition (^{2}).

Chapter 7 - Conclusion

In this thesis we have examined the problem of making updates to the static semantics of a program in response to small, incremental changes to the program. We have proposed an extension of the attribute grammar formalism in which semantic information can be exchanged between semantically related objects. An attribute evaluation process somewhat different from those discussed in the literature has been presented. That evaluation process has been extended to allow for incremental updates and non-local productions. The method provides a new approach to the problem of incremental semantic evaluation studied, for instance, in [9] and [7] in that attributes may flow directly to where they are needed, rather than being restricted in their flow to paths in the parse tree of a program. The links take up space in memory and require time to construct, so their use involves trade-offs. In the case of linking definitions and uses of identifiers, the advantages out-weigh the costs.

7.1. Directions for future research

Attribute grammars suffer from the limitation that an attribute may only receive one value in the course of evaluation. Skedzeleski [10] has addressed this problem, and I feel that this is an area worth more research. The restriction limits a system like POE to consideration of static program semantics. The approach of denotational semantics gives more flexibility in addressing issues of static (compile-time) and dynamic (run-time) program meaning. This approach might lead to automated interpreters and program testing and debugging systems that can be produced automatically from language descriptions.

Short-term research goals include development of an algorithm which can verify that some attribute grammars possess the node-wise non-circularity

property defined in Chapter 4 and development of general tests for classes of tree modifications that can be re-evaluated using non-local productions. As automated systems for the software development process become increasingly widespread, and various components such as compiling, editing, and debugging become more integrated, it is hoped that research will provide models that will be of assistance to designers and implementors of those systems.

References

[1] Alberga, C. N., A. L. Brown, G. B. Leeman Jr., M. Mikelsons, and M. N. Wegman, "A program development tool," *8th POPL Conference*, pp. 92-104 (1981).

[2] Archer, James and Richard Conway, "COPE: a cooperative programming environment," TR 81-459, Cornell University (1981).

[3] Celentano, A., P.D. Vigna, C. Ghezzi, and D. Mandrioli, "SIMPLE: a program development system," *Computer Languages 5*, pp. 103-114 (1980).

[4] Donzeau-Gouge, V., G. Huet, G. Kahn, B. Lang , and J. Levy, "A structure oriented program editor: a first step towards computer assisted programming," Technical Report 114, IRIA Laboratories (1975).

[5] Habermann, A. N., "The gandalf research project," *Carnegie-Mellon University Computer Science Research Review - 1979*, pp. 28-35 (1979).

[6] Teitelbaum, T. and T. Reps, "The Cornell program synthesizer: a syntax-directed programming environment," *Communications of the ACM 24*, 9, pp. 563-573 (September, 1981).

[7] Demers, Allan, Thomas Reps, and Tim Teitelbaum, "Incremental evaluation for attribute grammars with application to syntax-directed editors.," *8th ACM Symposium on Principles of Programming Languages*, pp. 105-116 (1981).

[8] Knuth, Donald E., "Semantics of Context Free Languages," *Math. Systems Theory Journal 2*, 2, pp. 127-145 (June 1968).

[9] Reps, T., "Optimal-time Incremental Semantic Analysis for Syntax-directed Editors," *9th ACM Symposium on Principles of Programming Languages*, pp. 169-176 (1982).

[10] Skedzeleski, Stephen Karl, "Definition and Use of Attribute Reevaluation in Attributed Grammars," Computer Sciences Technical Report #340, University of Wisconsin-Madison (October 1978.) PhD thesis.

[11] Cohen, R. and E. Harry, "Automatic generation of near-optimal linear-time translators for non-circular attribute grammars," *6th Symposium on Principles of Programming Languages*, pp. 121-134 (January 1979).

[12] Bochmann, Gregor V., "Semantic evaluation from left to right," *Communications of the ACM 19*, 2, pp. 55-62 (February 1976).

[13] Jazayeri, M., "On attribute grammars and the semantic specification of programming languages," PhD thesis, Comp. and Inf. Sci. Dept., Case Western Reserve University (October 1973).

[14] Kennedy, K. and S.K. Warren, "Automatic generation of efficient evaluators for attribute grammars," *Third ACM Symposium on Principles of Programming Languages*, pp. 32-49 (January 1976).

[15] Kennedy, K. and J. Ramanathan, "A deterministic attribute grammar evaluator based on dynamic sequencing," *ACM Transactions on Programming Languages and Systems 1*, 1, pp. 142-160 (July 1979).

[16] R., Rowland, Bruce, "Combining Parsing and the Evaluation of Attributed Grammars," Ph.D. Thesis, Computer Sciences Department, University of

Wisconsin (June 1977).

[17] Knuth, Donald E., "Semantics of Context Free Languages: Correction," *Math. Systems Theory Journal* 5, 1, pp. 95-96 (1971).

[18] Fischer, Charles N., Greg Johnson, and Jon Mauney, "An Introduction to Editor Allan Poe," Tech. report #451, University of Wisconsin-Madison (1981).

[19] Katayama, Takuya, "Translation of Attribute Grammar into Procedures," Tech. Report CS-K8001, Tokyo Institute of Technology (July 1976).

[20] Hopcroft, John E. and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass. (1979).

Appendix 1 - Correction to Section 3.2

In this Appendix, we present an expanded version of algorithm 3.1. The version given in section 3.2 depends on the assumption that a rendezvous ancestor appears in the same production as the join attribute, and it is possible to construct attribute grammars in which this assumption is violated. For instance, the attribute grammar given in Figure A.1 will cause Algorithm 3.1 to fail.

S ::= A	A.down := A.up
A ::= B	A.up := B.up1 A.join := A.down + B.up2
B ::= C	B.up1 := C.seed B.up2 := C.seed
C ::= c	C.seed := 2

Figure A.1.

The attribute A.join depends on C.seed via two distinct paths in the only parse tree derivable from the grammar, but when Algorithm 3.1 is applied to the grammar it does not detect this multi-path dependency. The problem is in step (b) of the algorithm: In examining production "A ::= B", the fact that attributes C.up1 and C.up2 have a common ancestor in a parse tree derivable from the grammar is not taken into account.

In order to remedy the situation, we must identify for each characteristic graph which of its independent output attributes (hereafter referred to as i.o. attributes) may have a common ancestor in some parse tree. Then, instead of testing whether an attribute β has more than one dependency path from some other attribute α in the production, we test whether β has more than one dependency path from some set of attributes which may have a common ancestor. The next step in the algorithm is to determine which attributes can appear along one of the paths from an ancestor to β. In Algorithm 3.1 we needed only to look at

attributes which could appear along an arc of one of the characteristic graphs from α to β. This entailed tracing back through the creation of characteristic graphs. In the new setting, we must trace back through both the creation of characteristic graphs and through the creation of common ancestor sets.

Algorithm A.1 will associate with each characteristic graph a collection of subsets of its i.o. attributes (i.e., a subset of the power set of the i.o. attributes). A subset will appear in the collection if and only if there exists a near-complete parse tree in which the elements of that set share a common ancestor α, and none of the other i.o. attributes depend on α. For convenience we will refer to a set of i.o. attributes that satisfies our desired condition as a "family", and we will refer to a characteristic graph's "family set".

Algorithm A.1.
Input: an attribute grammar
Output: Characteristic graphs together with their family sets

```
for those productions P which have only one non-extremum, do begin

  {say the non-extremum is at position "1" of the production}

  G := the projection of P's dependency graph on symbol "1";

  if i = 0
  then add G to the IO graph set of symbol "1"
  else add G to the OI graph set of symbol "1";

  for each i.o. attribute α of G, add {α} to the family set of G;

end for;

{main loop}
while there remains an unexamined 3-tuple
    ( production P,
      position "1" in P,
      {characteristic graphs C_j for symbols of P
      other than the symbol at position "1"}
    )
do begin

  let D be the dependency graph for P augmented by the selected
  characteristic graphs;

  G := the projection of D on position "1";

  if i = 0
  then add G to the IO graph set of symbol "1"
  else add G to the OI graph set of symbol "1";

  for each family {β_k} of the family set of
  each C_j, add to the family set of G the set of
  all i.o. attributes of G which depend on an element of {β_k}.

end while;
```

We can state a theorem which is an extension of Theorem 2.1:

Theorem A.1. A family $\{\beta_k\}$ is included in the family set of a characteristic graph C if and only if there exists some near-complete parse tree corresponding to C in which the elements of $\{\beta_k\}$ have a common ancestor α, and no other i.o. attribute of C depends on α.

Proof. The proof is virtually identical to that of Theorem 2.1: For a given family $\{\beta_k\}$, we can construct a near-complete parse tree by simply following a sequence of operations the algorithm takes which leads to inclusion of $\{\beta_k\}$ in the family set. Conversely, for a given near-complete parse tree, it can easily be shown that if a family is not in the family set of the characteristic graph then we could keep "clipping" production instances off the near-complete parse tree to find successively smaller near-complete parse trees each having a family not in the appropriate family set. Eventually (by the finiteness of all parse trees) we must obtain a near-complete parse tree which is a single production instance. At this point we arrive at a contradiction, since all such near-complete parse trees will have been accounted for properly in the initialization step of the algorithm. q.e.d.

We now present the corrected version of Algorithm 3.1. The basic idea of Algorithm A.2 is to identify rendezvous attributes (referred to as "β") by determining which productions have an attribute which can be gotten to via two distinct paths from some family. After a family and a "descendant" rendezvous attribute have been identified, it is necessary to trace back through the ancestors of the family to discover all rendezvous ancestors of β.

Algorithm A.2
Input: A well-formed attribute grammar
Output: A simple priority order among the attributes

compute the characteristic graphs and family sets for the grammar;

for each production P in the grammar,
(a) append to its dependency graph characteristic graphs C_i at each of its non-terminals;

(b) determine whether there exists a family f' associated with some C_1 and an attribute β such that there is more than one path from f' to β. If not, look at the next production in the "for" loop.

(c) insert into a "families to be examined" set all sextuples
(P, pos (position in P of β), $\{C_1\}$, f, $\{\beta\}$, f')
where f contains β and is in the family set of the characteristic graph of the symbol at "pos" that is created by the characteristic graphs $\{C_i\}$, and f' is a family of some C_1 such that there is more than one path from f' to β.

while "families to be examined" is not empty,

select and remove a sextuple (P, pos, $\{C_1\}$, f, g, f');

append to the dependency graph of P the characteristic graphs $\{C_i\}$;

for every attribute γ on every path from an element f_i of f' to an element of g (including the elements of g but excluding the elements of f'), put (γ, β) in the relation; (This will typically entail a trace through creators of characteristic graphs to identify attributes that are represented by arcs of characteristic graphs.)

if it is not the case that all paths start from a single element of f', then

i) add (f_i, β) to the relation;

ii) add all creators (P', pos', C_j, f', $\{f_i\}$, f'') of f' for which no sextuple (P', pos', C_j, f', g, f'') in which g is a superset of $\{f_i\}$ has already been processed in the "families to be examined" set.

end;
end.

We now state and prove the desired result about Algorithm A.2.

Theorem A.2. If there exists a parse tree derivable from the grammar for which β is a rendezvous attribute and γ is a rendezvous ancestor of β, then a pair (γ, β) of attribute occurrences will be inserted into the relation.

Proof. First, we know that β will be identified as a rendezvous attribute, since by Theorem A.1 the family of which γ is an ancestor will have been identified. In the parse tree, we can trace attribute dependencies backward from β to α, the ancestor of β via perhaps several paths. The path in the parse tree from the node containing β to the node containing α might be called the stem of the chain of dependencies. Attributes on the stem will have been included in the relation, since each step along the stem is reflected in the algorithm by the inclusion of a "creator" family for a family. Attributes not on the stem will be included in the part of the algorithm which includes attributes along all paths in a particular production.

q.e.d.