INCREMENTAL EVALUATION FOR ATTRIBUTE GRAMMARS WITH UNRESTRICTED MOVEMENT BETWEEN TREE MODIFICATIONS

by

Thomas Reps

Computer Sciences Technical Report #671

October 1986

Incremental Evaluation for Attribute Grammars with Unrestricted Movement between Tree Modifications

THOMAS REPS
University of Wisconsin

This paper concerns the design of editors that perform checks on a language's context-dependent constraints. Our particular concern is the design of an efficient, incremental analysis algorithm for systems based on the attribute-grammar model of editing.

With previous incremental evaluation algorithms for arbitrary noncircular attribute grammars, the editing model required there to be a restriction on the operation that moves the editing cursor: moving the cursor was limited to just a single step in the tree – either to the parent node or to one of the child nodes of the current cursor location. This paper describes a new updating algorithm that can be used when an arbitrary movement of the cursor in the tree is permitted. After an operation that restructures the tree, the tree's attributes can be updated with a cost of $O((1+|\text{AFFECTED}|)\cdot\sqrt{m})$, where m is the size of the tree and AFFECTED is the subset of the tree's attributes that require new values, when the cost is amortized over a sequence of tree modifications. The editing cursor may be moved from its current location to any other node of the tree in a single, unit-cost operation.

CR Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding -- *program editors*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.1 [**Programming Languages**]: Formal Definitions and Theory -- *semantics, syntax* D.3.4 [**Programming Languages**]: Processors -- *translator writing systems and compiler generators* F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *denotational semantics*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Attribute grammar, incremental attribute evaluation, incremental context-dependent analysis, language-based editor, editor generator

---

1. INTRODUCTION

Because of their ability to declaratively express a wide variety of context-dependent relationships in different languages, attribute grammars [Knuth 1968] are a good basis for specifying and generating language-based editors. Such editors represent programs and documents as (consistently attributed) derivation trees of an underlying grammar. At each stage during editing, the *editing cursor* is positioned at a node of the derivation tree. An editing session is viewed as a succession of cursor motions and operations that restructure the tree, such as subtree pruning and grafting [Demers et al. 1981].

One example of an editor-generating system that is based on the attribute-grammar model is the Synthesizer Generator [Reps & Teitelbaum 1984, 1985, 1987]. With this system, a language's context-dependent constraints are incorporated in an editor by defining an attribute grammar in which certain attributes receive values that indicate whether or not the constraints are satisfied. In addition to the attribute-

---

This work was supported in part by the National Science Foundation under grant DCR-8552602, by an IBM Faculty Development Award, and by grants from Digital Equipment Corporation, Siemans, and Xerox.

Author's address: Thomas Reps, Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.

grammar component of an editor specification, the editor designer also furnishes an *unparsing* specification that defines how programs are formatted on the screen. Attributes used in the unparsing specification cause the screen to be annotated with values of attribute instances. In particular, the attributes that indicate satisfaction or violation of context-dependent constraints can be used to annotate the display to indicate the presence or absence of errors.

When a derivation tree is restructured, the values of the attributes at the modification point, which were previously consistent, may no longer have consistent values. Incremental analysis of context-dependent language features is performed by updating attribute values throughout the tree in response to modifications. If an editing operation modifies a program in such a way that formerly satisfied constraints are now violated (alternatively, formerly violated constraints are now satisfied), the attributes that indicate satisfaction of constraints will receive new values; the changed image of these attributes on the screen provides the user with feedback about new errors introduced and old errors corrected.

Fundamental to this approach is the idea of an *incremental attribute evaluator*, an algorithm to produce a consistently attributed tree after each restructuring operation. After each modification to the tree, only a subset of the tree's attributes, denoted by AFFECTED, requires new values. When updating begins, it is not known which attributes are members of AFFECTED; AFFECTED is determined as a result of the updating process itself. Nevertheless, there exists an attribute updating algorithm that uses $O(|\text{AFFECTED}|)$ steps, where an application of an attribute definition function is counted as an atomic step [Reps 1982, Reps et al. 1983, Reps 1984]. Because $O(|\text{AFFECTED}|)$ is the minimal amount of work required to update a derivation tree after a modification, this algorithm is asymptotically optimal. (Throughout the rest of the paper, the editing model described above will be referred to as the *standard editing model* and the updating algorithm described in [Reps 1982, Reps et al. 1983, Reps 1984] as the *standard updating algorithm*).

After the tree has been modified, an incremental attribute evaluator traverses parts of the derivation tree to establish a consistent set of values, reevaluating some of the tree's attributes as it progresses. To choose the next attribute to reevaluate during this process, the algorithm may make use of auxiliary information. For instance, the standard updating algorithm makes use of information about the dependencies that exist among the tree's attributes; part of this information is stored at the individual nodes of the tree.

We must also consider the cost imposed on each editing operation for keeping this auxiliary information up-to-date. For example, besides making the value of one or more attributes inconsistent, a restructuring operation may also invalidate some of the auxiliary information that is used by the incremental attribute evaluator. Consequently, we can think of there being two phases to updating a tree, although in some algorithms they may actually be interleaved. First comes the phase for *overhead-updating*, during which the auxiliary information used to choose the correct order for reevaluating attributes is updated. This is followed by the phase for *attribute-updating*, when the tree or the tree's attribute dependency graph is traversed and attributes are reevaluated. (The auxiliary information may also have to be updated after a cursor motion operation, so the term *overhead-updating* will also refer to any updating carried out after a single operation that moves the editing cursor).

With the standard algorithm for attribute updating, the limitation on cursor-movement operations was introduced to prevent the overhead-updating costs from dominating the attribute-updating costs. If arbitrary cursor movements are permitted, the overhead-updating cost associated with moving the cursor is proportional to the length of the path in the tree from the cursor's former site to its new site. In contrast, when the cursor's movement is restricted to a single step in the tree, overhead-updating costs can never be more than a constant amount.

This paper describes a new algorithm for updating attributes when arbitrary movements of the cursor in the tree are permitted; if this algorithm is used as an editor's incremental attribute evaluator, then the editing cursor can be moved from its current node to any other node of the tree in a single, unit-cost operation. After an operation that restructures the tree, the tree's attributes can be updated with a cost of $O((1+|\text{AFFECTED}|)\cdot\sqrt{m})$, where m is the size of the tree, when the cost is amortized over a sequence of tree modifications.

One novelty of the result is that the algorithm's behavior is analyzed by an amortized-cost measure. One part of the algorithm involves a delayed-update strategy of the kind used in [Tsakalidis 1984]. (One previous paper on attribute updating reports an amortized-cost result [Reps et al. 1986]; however, in that algorithm, the amortized cost arises only because it makes use of *link-cut* trees, a data structure for which a good amortized-cost complexity result is known [Tarjan 1983]).

A major drawback to the use of attribute grammars in language-based editors has been that attributes can only depend on neighboring attributes in a program's syntax tree. There have been several proposals to overcome this limitation, including [Demers et al. 1985], [Johnson & Fischer 1982, 1985], [Hoover 1986], and [Reps et al. 1986]. In trying to design a better algorithm for this sort of generalized updating problem, a colleague conjectured that for some sequences of changes to some graphs, each change requires an amount of overhead-updating that is linear in the size of the graph. He conjectured further that this situation could occur with attribute dependency graphs of (ordinary) attribute grammars. The algorithm presented in this paper rebuts the latter conjecture.

It should be pointed out that the algorithm described in this paper is probably only of theoretical interest because arbitrary movements of the cursor are already permitted if, as is the case for nearly all examples that arise in practice, the editor specification falls into the *ordered* subclass of attribute grammars, defined in [Kastens 1980]. For ordered grammars, all auxiliary information used to update the tree is static and therefore unchanged by either a cursor motion or a tree restructuring; therefore, the updating algorithms described in [Yeh 1983] and [Reps & Teitelbaum 1987] are optimal, even in the presence of arbitrary cursor motions.

After a brief introduction of terminology and notation in Section 2, the new attribute updating method is described in Section 3.

2. TERMINOLOGY AND NOTATION

An attribute grammar is a context-free grammar extended by attaching *attributes* to the terminal and nonterminal symbols of the grammar, and by supplying *attribute equations* to define attribute values [Knuth 1968]. In every production $p: X_0 \rightarrow X_1, ..., X_k$, each X_i denotes an *occurrence* of one of the grammar symbols; associated with each such symbol occurrence is a set of *attribute occurrences* corresponding to the symbol's attributes.

Each production has a set of attribute equations; each equation defines one of the production's attribute occurrences as the value of an *attribute-definition function* applied to other attribute occurrences in the production. The attributes of a symbol X, denoted $A(X)$, are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes. Each attribute equation defines a value for a synthesized attribute occurrence of the left-hand side nonterminal or an inherited attribute occurrence of a right-hand side symbol.

This paper deals only with attribute grammars that are *well formed*: An attribute grammar is well formed when the terminal symbols of the grammar have no synthesized attributes, the root symbol of the grammar has no inherited attributes, and each production has exactly one attribute equation for each of the left-hand

side nonterminal's synthesized attribute occurrences and for each of the right-hand side symbols' inherited attribute occurrences.

Example. As a running example to illustrate the reevaluation algorithm, we will use a language of arithmetic expressions that is extended with a *let* construct for binding identifiers to values. The abstract syntax of the language is defined by the following (ambiguous) context-free grammar:

ROOT $\rightarrow$ *exp*
exp $\rightarrow$ *exp* + *exp*
exp $\rightarrow$ *exp* − *exp*
exp $\rightarrow$ *exp* * *exp*
exp $\rightarrow$ *exp* ** *exp*
exp $\rightarrow$ **let** *id* = *exp* **in** *exp* **ni**
exp $\rightarrow$ *id*
exp $\rightarrow$ *integer*

We define a scheme to compute an expression's value by attaching an attribute *val* to each *exp* nonterminal to represent the subexpression's value. To define this attribute, we must also know the bindings of identifiers established in enclosing let clauses; this is represented by the attribute *env* of *exp*. The equations defining these attributes are:

ROOT $\rightarrow$ *exp*
$exp.env = \varnothing$
exp $\rightarrow$ *exp* + *exp*
$exp_1.val = exp_2.val + exp_3.val$
$exp_2.env = exp_1.env$
$exp_3.env = exp_1.env$
exp $\rightarrow$ *exp* − *exp*
$exp_1.val = exp_2.val - exp_3.val$
$exp_2.env = exp_1.env$
$exp_3.env = exp_1.env$
exp $\rightarrow$ *exp* * *exp*
$exp_1.val = exp_2.val * exp_3.val$
$exp_2.env = exp_1.env$
$exp_3.env = exp_1.env$
exp $\rightarrow$ *exp* ** *exp*
$exp_1.val = exp_2.val ** exp_3.val$
$exp_2.env = exp_1.env$
$exp_3.env = exp_1.env$
exp $\rightarrow$ **let** *id* = *exp* **in** *exp* **ni**
$exp_1.val = exp_3.val$
$exp_2.env = exp_1.env$
$exp_3.env = exp_1.env \cup \{<id, exp_2.val>\}$
exp $\rightarrow$ *id*
$exp.val = \text{LookUp}(id, exp.env)$
exp $\rightarrow$ *integer*
$exp.val = \text{ValueOf}(integer)$

The value of the entire expression is represented by the *val* attribute of the *exp* nonterminal derived from the tree's root.

A derivation tree node that is an instance of symbol X has an associated set of *attribute instances* corresponding to the attributes of X. (We shall sometimes shorten "attribute instances" and "attribute occurrences" to "attributes"; however, the intended meaning should be clear from the context). An *attributed tree* is a derivation tree together with an assignment of either a value or the special token **null** to each attribute instance of the tree. To analyze a string according to its attribute-grammar specification, first construct its derivation tree with an assignment of **null** to each attribute instance, and then evaluate as many

attribute instances as possible, using the appropriate attribute equation as an assignment statement. The latter process is termed *attribute evaluation*. An attribute instance in an attributed derivation tree is said to be *consistent* if its value is equal to the value obtained by evaluating the right-hand side of its defining attribute equation. An attributed derivation tree is *consistently attributed* if all of its attribute instances are consistent.

Functional dependencies among attribute occurrences in a production p (or attribute instances in a tree T) can be represented by a directed graph[1], called a *dependency graph*, denoted by $D(p)$ (respectively, $D(T)$) and defined as follows:

a) For each attribute occurrence (instance) b, the graph contains a vertex b'.

b) If attribute occurrence (instance) b appears on the right-hand side of the attribute equation that defines attribute occurrence (instance) c, the graph contains an edge (b', c'), directed from b' to c'.

An attribute grammar that has a derivation tree whose dependency graph contains a cycles is called a *circular* attribute grammar. This paper deals only with non-circular grammars; that is, grammars for which the dependency graph of any derivation tree is acyclic.

Example. Figure 1 shows a derivation tree for the expression "**let** a = 2 **in let** b = 3 **in let** c = 1 **in** b ** 2 – 4 * a * c **ni ni ni**". Nonterminals are connected by dashed lines; the dependency graph consists of the instances of the attributes *env* and *val*, linked by their functional dependencies, shown as solid arrows. (The solid arrows emanating from the constant $\varnothing$ and the tree's *id* and *integer* leaves indicate dependencies on constants and tree components, respectively; strictly speaking, they are not part of the dependency graph). The subscripts on the interior nodes will be used in subsequent examples that refer to Figure 1.

Creating a file using a language-based editor entails growing a derivation tree. During development, a file tree is a partial derivation tree that may contain unexpanded nonterminals. This is potentially a problem because at an unexpanded nonterminal X, we have no means for giving values to the synthesized attributes of X or to any of their successors. This conflicts with our desire to maintain values for every attribute of the tree.

To avoid this problem, we require that the grammar include a *completing production*, $X \rightarrow \perp$, for each nonterminal symbol X. The symbol $\perp$ denotes "unexpanded", and the attribute equations of the completing production define values for the synthesized attributes of X. By convention, an occurrence of an unexpanded nonterminal is considered to have derived $\perp$. By this device, all partial derivation trees (from the user's viewpoint) are considered complete derivation trees (from the editor's viewpoint), and so a program tree may be fully attributed at any stage of development.

3. AN INCREMENTAL EVALUATOR WITH COST O(|AFFECTED| · $\sqrt{m}$)

This section describes a new incremental attribute evaluation algorithm for arbitrary noncircular grammars. Although it is more expensive than the standard updating algorithm if cursor motions are restricted to single steps in the tree, the new algorithm performs better if arbitrary movement of the cursor is permitted. When the cost of attribute updating is amortized over a sequence of editing operations, the cost is $O((1 + |\text{AFFECTED}|) \cdot \sqrt{m})$, where m is the size of the tree. (The constant of proportionality depends on the quantities *MaxAttrs*, the maximum number of attributes of any grammar symbol, and *MaxSons*, the

[1]A *directed graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* E, where $E \subseteq V \times V$. Each edge $(b, c) \in E$ is directed from b to c. Throughout the paper, the term "vertex" is used to refer to elements of dependency graphs, whereas the term "node" refers to elements of derivation trees.

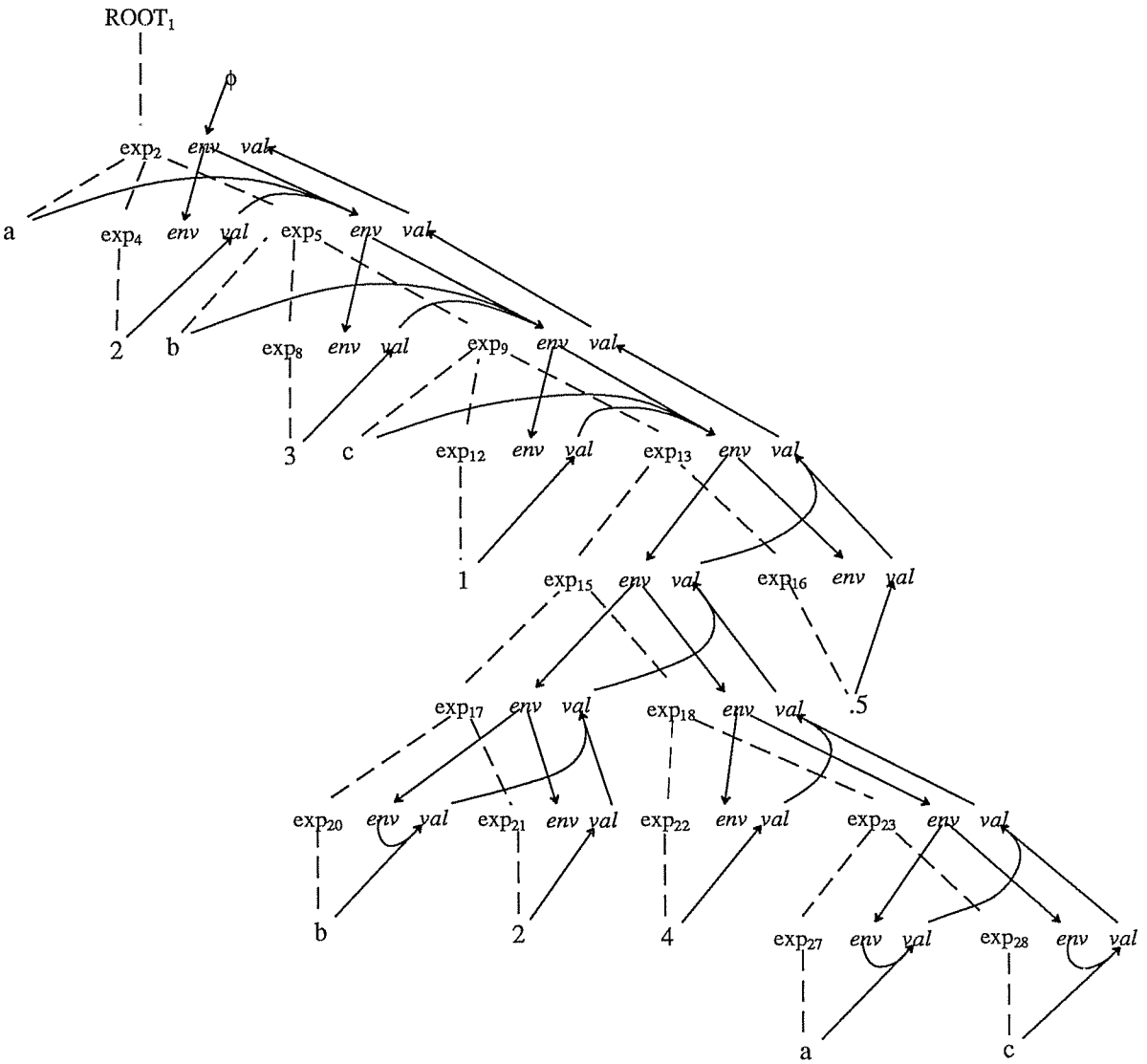


Figure 1. A derivation tree and its dependency graph.

maximum number of symbols on the right-hand side of any production, both of which are constants for a given attribute grammar).

Throughout the discussion of our algorithm, we make the following assumptions:

a) We assume that any of the children of a tree node can be accessed in constant time, that each node is labeled with a descriptor that can be used to determine the node's arity, and that a constant number of bits are associated with each node so that insertion, deletion, and membership operations on a set of tree nodes can be implemented as unit-time operations.

b) We assume that all operations introduce at most a constant number of new nodes into the tree. This assumption is made purely to simplify the presentation of the method. As stated, it forbids cut and paste operations on subtrees; however, the assumption can be relaxed by modifying the algorithm so that it is applied to the entire "forest" of editable objects rather than to just a single tree. If this step is taken, we must still retain the weaker assumption that each editing operation creates at most a constant number of new nodes in the *forest*.

c) We assume that some of the nodes in the tree are labeled with their *subordinate* and *superior characteristic graphs*. A node's characteristic graphs provide a convenient representation of transitive dependencies among the node's attributes. The vertices of the characteristic graphs at node r correspond to the attributes of r; the edges of the characteristic graphs at r correspond to transitive dependencies among r's attributes.

The subordinate characteristic graph at r is the projection of the dependencies of the subtree rooted at r onto the attributes of r. To form the superior characteristic graph at node r, we imagine that the subtree rooted at r has been pruned from the derivation tree, and project the dependency graph of the remaining tree onto the attributes of r. To define the characteristic graphs precisely, we make the following definitions:

a) Given directed graphs $A = (V_A, E_A)$ and $B = (V_B, E_B)$, that may or may not be disjoint, the *union* of A and B is defined as:

$$A \cup B = (V_A \cup V_B, E_A \cup E_B)$$

b) The *deletion* of B from A is defined as:

$$A - B = (V_A, E_A - E_B)$$

Note that deletion deletes only edges.

c) Given a directed graph $A = (V, E)$, a *path* from vertex a to vertex b is a sequence of vertices, $[v_1, v_2, ..., v_k]$, such that: $a = v_1$, $b = v_k$, and $\{ (v_i, v_{i+1}) \mid i = 1, ..., k-1 \} \subseteq E$.

d) Given a directed graph $A = (V, E)$ and a set of vertices $V' \subseteq V$, the *projection* of A onto V' is defined as:

$$A / V' = (V', E')$$

where $E' = \{(v, w) \mid v, w \in V'$ and there exists a path from v to w in A that does not pass through any other elements of $V'\}$.

The subordinate and superior characteristic graphs of a node r, denoted $r.C$ and $r.\overline{C}$, respectively, are defined formally as follows: Let r be a node in tree T, let the subtree rooted at r be denoted T_r, and let the attribute instances at r be denoted $A(r)$, then the subordinate and superior characteristic graphs at r satisfy:

$$r.C = D(T_r)/A(r)$$

$$r.\overline{C} = (D(T) - D(T_r))/A(r)$$

A characteristic graph represents the projection of attribute dependencies onto the attributes of a single tree node; consequently, for a given grammar, each graph is bounded in size by some constant. For example, if a characteristic graph is represented with a dependency matrix, it requires no more than MaxAttrs^2 bits.

The characteristic graphs are employed by Propagate, the attribute updating procedure, to determine the order in which attributes get reevaluated. However, a single subtree replacement can radically alter transitive dependencies among attributes; as a consequence, a subtree replacement at node r can invalidate characteristic graphs arbitrarily far away from r. Maintaining *every* characteristic graph in the tree would make subtree replacements far too expensive – the overhead-updating cost to update the tree's characteristic graphs would dominate the attribute-updating costs.

One of the key ideas used in the standard updating algorithm is to only make use of a subset of all of the tree's characteristic graphs. This idea is also used in our new updating algorithm, but the subset of the tree's characteristic graphs that is used in the new algorithm is quite different from the subset that would be maintained by the standard algorithm. In the new algorithm, the nodes of the tree at which characteristic graphs are maintained are chosen to that they partition the tree in a balanced fashion. In particular, they are chosen so that an m-node tree is partitioned into $O(\sqrt{m})$ components, with each component no larger than $O(\sqrt{m})$.

The characteristic graphs of partition-set nodes get created when the tree is partitioned, and they are updated each time the tree is modified. The invariant that is maintained by all editing operations is that after each operation, the subordinate and superior characteristic graphs are up-to-date for each tree node in the partition set. Although there are $O(\sqrt{m})$ such graphs, they can each be created in time $O(\sqrt{m})$; hence, their total contribution to the cost of the partitioning step is $O(m)$. After the tree is modified, each of the $O(\sqrt{m})$ graphs can be updated in unit time; thus, the incremental attribute evaluator's overhead-updating cost is $O(\sqrt{m})$.

The actual updating of attribute values is carried out in a fashion very similar to the way updating is performed by the standard algorithm. The chief difference is that expansions of the graph structure used to schedule attribute reevaluations are done on a per-component basis in the new algorithm, rather than on a per-production basis, as is done in the standard algorithm. Because an expansion is performed only when a member of AFFECTED has been found and each component is of size $O(\sqrt{m})$, the total cost of the attribute-updating phase amounts to $O(|\text{AFFECTED}| \cdot \sqrt{m})$.

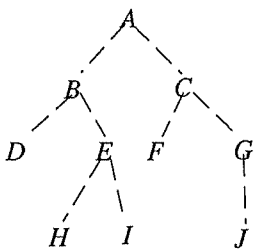
The second key idea used in the new updating algorithm is that the partition of the tree is not static: from time to time the tree is repartitioned. In particular, if the tree contains m nodes when partitioned, the partition is used for the next $\sqrt{m}$ steps, at which point the tree is repartitioned according to its current size and shape. Because of the assumption that each editing transaction introduces at most a constant number of new nodes, $\sqrt{m}$ steps can introduce at most $O(\sqrt{m})$ nodes; thus, throughout the sequence of $\sqrt{m}$ steps, each component never contains more than $O(\sqrt{m})$ nodes. Although the cost of repartitioning is $O(m)$, this cost is amortized over the previous $\sqrt{m}$ operations, which just adds an additional $O(\sqrt{m})$ to the cost of each operation; the extra cost gets absorbed in the constant of proportionality of the term $O((1+|\text{AFFECTED}|) \cdot \sqrt{m})$.

The ideas that have been outlined above are presented in detail in the next three sections.

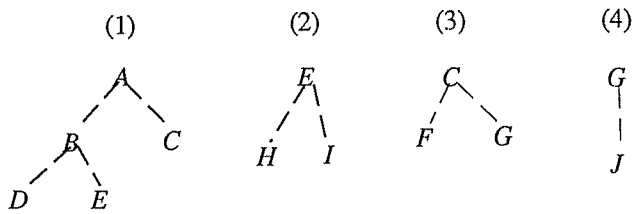
3.1. Partitioning[2]

A set of nodes P partitions a tree T into *components*, where a component is a maximal-size, connected region of T in which none of the interior nodes are elements of P. For example, the set $\{E, C, G\}$ partitions the tree:

[2]The presentation in this section borrows heavily from that given in [Reps & Demers 1987].



into the four components:



We refer to A, E, C, and G as the *roots* of the respective components, and $\{D, E, C\}$, $\{H, I\}$, $\{F, G\}$, and $\{J\}$ as their respective *leaves*. A leaf that is also a partition-set node is referred to as an *interface* node. For example, E and C are the interface nodes of component (1).

The goal of the partitioning step is to divide the derivation tree into $O(\sqrt{m})$ components, each of which (a) is no larger than $O(\sqrt{m})$ nodes, and (b) adjoins at most a constant number of other components. This is accomplished in two phases: the first phase partitions the derivation tree into components of maximum size $O(\sqrt{m})$; the second phase further subdivides the components found during the first phase so that each component adjoins at most MaxSons + 1 other components. Because both phases of partitioning carry out a postorder traversal of the tree, in practice one would probably combine the two phases and partition the tree during a single traversal.

The third phase of the tree-partitioning operation determines characteristic graphs for each of the nodes in the partition set found by the first two phases.

A fourth phase of partitioning is also carried out. The motivation for this phase, as well as its description, is put off until Section 3.3.

Phase 1. The first phase of partitioning is carried out using the function Partition, stated in Figure 2. Partition performs a scan of the tree, starting from the root, partitioning the tree into components that are each no larger than $O(\sqrt{m})$ nodes. Partition is parameterized by an arbitrary constant SplittingFactor; Partition constructs the set PartitionSet bottom-up, by traversing the nodes of T in postorder with the recursive function PartitionBySize. PartitionSet is initially empty, and PartitionBySize adds a new node to PartitionSet whenever a node is found that is at the root of a component larger than $\sqrt{m}$ /SplittingFactor.

```
function Partition(T) returns a set of tree nodes
declare
   T: a derivation tree
   m: an integer
   PartitionSet: a set of tree nodes
begin
   PartitionSet := ∅
   m := the number of nodes in T
   PartitionBySize(root(T))
   return(PartitionSet)
end

function PartitionBySize(Y) returns an integer
declare
   Y: a tree node
   Y_i: the i^th child of Y
   L, i: integers
   PartitionSet: a set of tree nodes (global)
   m: an integer (global)
begin
   L := 1
   for i := 1 to NumberOfChildren(Y) do
      L := L + PartitionBySize(Y_i)
   od
   if L > √m /SplittingFactor then
      Insert Y into PartitionSet
      L := 1
   fi
   return(L)
end
```

Figure 2. Partition T into $O(\sqrt{m})$ components of size $O(\sqrt{m})$, where m is the number of nodes in T.

For each node Y, PartitionBySize computes how large a component there would be if Y were the root of a component. Each recursive call on PartitionBySize(Y_i), where Y_i is the ith child of Y, either returns 1, if Y_i is a leaf, or it returns how large a component there would be if Y_i were the root of a component; thus, the quantity returned from each call ranges from 1 to $\sqrt{m}$/SplittingFactor (assuming that $m \geq$ SplittingFactor^2)[3]. The size of the component rooted at Y is computed by adding 1 (for Y itself) to the sum of the values returned by the calls to PartitionBySize with Y's children. Because Y has at most MaxSons children, the number of nodes in the component can be no larger than ($\sqrt{m}$ MaxSons/SplittingFactor) + 1. Because each node is visited once during the postorder traversal of T, the total running time of Partition is $O(m)$.

[3]We are interested in the asymptotic behavior of the updating algorithm; for small m, the tree may be reevaluated exhaustively.

Except for the component containing the root of the tree, the root of each component is shared by exactly one other component; thus, for the set of components C_P, defined by a partition P, the sum of the components' sizes is related to the total number of nodes m by:

$$m = 1 + \sum_{c \in C_P} (|c| - 1)$$

Except possibly for the component containing the root of the tree, each component defined by the set PartitionSet contains at least ($\sqrt{m}$ /SplittingFactor) + 1 nodes, so

$$m \geq 1 + (|C_{\text{PartitionSet}}| - 1)(\sqrt{m}\ /\text{SplittingFactor})$$

Solving for $|C_{\text{PartitionSet}}|$, we get:

$$|C_{\text{PartitionSet}}| \leq \frac{(m-1)\text{SplittingFactor}}{\sqrt{m}} + 1$$

$$< (\text{SplittingFactor})(\sqrt{m}) + 1$$

Thus, the first phase of partitioning finds no more than (SplittingFactor)($\sqrt{m}$) + 1 components, each containing no more than ($\sqrt{m}$ MaxSons/SplittingFactor) + 1 nodes.

The only operations performed on PartitionSet are insertions and, in the next phase, membership tests, so PartitionSet can be implemented using a single bit at each tree node and no other additional storage.

Example. The tree shown in Figure 1 has 30 nodes. If the splitting factor is chosen to be $\sqrt{30}/7$, so that $\sqrt{m}$ /SplittingFactor = 7, Partition returns the set $\{exp_5, exp_{13}, exp_{18}\}$, which partitions the tree into four components that have 6, 8, 9, and 10 nodes.

Phase 2. The components found during the first phase of partitioning may be adjacent to as many as $O(\sqrt{m})$ other components. The second phase of partitioning further subdivides the components found in the first phase so that each component is adjacent to no more than MaxSons + 1 other components. This latter property is needed by both the attribute updating step described in Section 3.2 and to reduce the space and time requirements of the projection step that is described in Section 3.3. The second phase of partitioning at most doubles the total number of components.

The second phase of partitioning is carried out using the function AddAncestors, stated in Figure 3. AddAncestors constructs the set NewPartition bottom-up, by traversing the nodes of T in postorder with the recursive function AddAncestorsInComponent. NewPartition is initially empty, and AddAncestorsInComponent adds a new node to NewPartition whenever a node is found that is the lowest common ancestor of some pair of nodes that belong to OldPartition. Note that because for each node X, LowestCommonAncestor(X, X) = X, every node in OldPartition is put in NewPartition.

For each node Y, AddAncestorsInComponent returns **true** if the subtree rooted at Y contains a member of OldPartition. The loop over the children of Y generates a recursive call on AddAncestorsInComponent for each child Y_i, and counts how many children contain a member of OldPartition. If the count is two or more, (or if Y itself is a member of OldPartition), then Y is the lowest common ancestor of some pair of nodes in OldPartition, and is added to NewPartition. Because each node is visited once during the postorder traversal of T, the total running time of AddAncestors is $O(m)$.

The partition set computed by AddAncestors has two important properties:

a) Each component defined by NewPartition is adjacent to at most MaxSons + 1 other components.

b) The number of elements in NewPartition is at most 2 |OldPartition| – 1.

```
function AddAncestors(T, OldPartition) returns a set of tree nodes
declare
   T: a derivation tree
   OldPartition, NewPartition: sets of tree nodes
begin
   NewPartition := ∅
   AddAncestorsInComponent(root(T))
   return(NewPartition)
end

function AddAncestorsInComponent(Y) returns boolean
declare
   Y: a tree node
   Y_i: the i^th child of Y
   i, SubordinatePartitionNodes: integers
   OldPartition, NewPartition: sets of tree nodes (global)
begin
   SubordinatePartitionNodes := 0
   for i := 1 to NumberOfChildren(Y) do
      if AddAncestorsInComponent(Y_i) = true then
         SubordinatePartitionNodes := SubordinatePartitionNodes + 1
      fi
   od
   if Y ∈ OldPartition then
      Insert Y into NewPartition
      return(true)
   else if SubordinatePartitionNodes = 0 then return(false)
   else if SubordinatePartitionNodes = 1 then return(true)
   else
      Insert Y into NewPartition
      return(true)
   fi
end
```

Figure 3. Given OldPartition, a set of nodes in T, build the set NewPartition, where NewPartition = $\{Z \mid Z = \text{LowestCommonAncestor}(X, Y), \text{ where } X, Y \in \text{OldPartition}\}$.

To see that property a) holds, consider the value of the variable SubordinatePartitionNodes at a node Y that is a member of NewPartition. SubordinatePartitionNodes contains the number of leaves of the component rooted at Y that connect to other components; this number can be no larger that MaxSons. In addition, there is a connection to one more component at Y itself.

Property b) is shown by induction on the height of the tree: Assume that b) holds when AddAncestorsIn-Component is used to repartition trees of height less than height(T). In particular, it holds for the subtrees of root(T), denoted T_i, and the partition sets OldPartition_i, where OldPartition_i denotes the subset of Old-Partition whose elements are nodes of T_i. The recursive calls on AddAncestorsInComponent generate the sets $\text{NewPartition}_i \subseteq \text{NewPartition}$, for which, by the inductive assumption,

$$|\text{NewPartition}_i| \le 2|\text{OldPartition}_i| - 1$$

If the root of T has more than one child, then

$$\begin{aligned}
|\text{NewPartition}| &\le 1 + \sum_{i=1}^{\text{NumberOfChildren(root}(T))} |\text{NewPartition}_i| \\
&\le 1 + \sum_{i=1}^{\text{NumberOfChildren(root}(T))} (2|\text{OldPartition}_i| - 1) \\
&\le 1 + 2|\text{OldPartition}| - 2 \\
&= 2|\text{OldPartition}| - 1
\end{aligned}$$

as was to be shown. A similar argument handles the case where the root has only a single child.

Example. AddAncestors happens to add no additional nodes to the partition set $\{exp_5, exp_{13}, exp_{18}\}$ that partitions the tree of Figure 1.

Phase 3. The third phase of partitioning determines the characteristic graphs for each of the nodes in the partition-set found by the first two phases. This phase is carried out using the procedure BuildCharacteristicGraphs, stated in Figure 4. BuildCharacteristicGraphs constructs the tree's subordinate characteristic

---

procedure BuildCharacteristicGraphs(T, P)
declare
  T: a derivation tree
  P: a set of tree nodes
  X, Y: tree nodes
  i: an integer
  X_i: the ith child of X
  p: a production
begin
  for each node X in a postorder traversal of T **do**
    p := the production that applies at X
    $X.C := (D(p) \cup X_1.C \cup \cdots \cup X_{\text{NumberOfSons}(X)}.C)/A(X)$
  od
  for each node X in a preorder traversal of T **do**
    Y := parent(X)
    i := the son number of X with respect to Y
    p := the production that applies at Y
    $X.\overline{C} := (D(p) \cup Y.\overline{C} \cup Y_1.C \cup \cdots \cup Y_{i-1}.C \cup Y_{i+1}.C \cup \cdots \cup Y_{\text{NumberOfSons}(Y)}.C)/A(X)$
  od
  Discard $X.C$ and $X.\overline{C}$ for all $X \notin P$
end

Figure 4. For each node X in P, determine the graphs $X.C$ and $X.\overline{C}$.

graphs bottom-up, during a postorder traversal of the tree. By properties of the projection operation, if X is the parent node of a production instance p that consists of X and X's children $X_1, ..., X_k$, we have:

$$X.C = (D(p) \cup X_1.C \cup \cdots \cup X_{\text{NumberOfSons}(X)}.C)/A(X)$$

The corresponding assignment statement can be evaluated in constant time using a constant amount of space; thus, the space and time cost of the postorder traversal is linear in the size of the tree.

The superior characteristic graphs are constructed during a second traversal of the tree, this time in preorder. The property used during this pass is that if X is the ith child in a production instance p that consists of parent node Y and Y's children $Y_1, ..., Y_{i-1}, X, Y_{i+1}, ..., Y_{\text{NumberOfSons}(Y)}$, we have:

$$X.\overline{C} = (D(p) \cup Y.\overline{C} \cup Y_1.C \cup \cdots \cup Y_{i-1}.C \cup Y_{i+1}.C \cup \cdots \cup Y_{\text{NumberOfSons}(Y)}.C)/A(X)$$

As with the traversal that builds the subordinate characteristic graphs, the cost of the traversal that builds the superior characteristic graphs is linear in the size of the tree.

The final step of BuildCharacteristicGraphs is to discard all graphs of nodes that are not members of the set P. (In our case, P is a partition set created by the first two phases of the partitioning operation).

3.2. Updating Attribute Values

The standard algorithm for incremental attribute evaluation achieves optimal behavior by obeying the following design heuristic:

> If, in the course of propagating new values, an attribute is ever (temporarily) reassigned a value other than its correct final value, spurious changes are apt to propagate arbitrarily far beyond the boundaries of AFFECTED, leading to suboptimal running time. To avoid this possibility, a change propagator should schedule attribute reevaluations such that any new value computed is necessarily the correct final value. That is, an attribute should not be reevaluated until all of its arguments are known to have their correct final values.
>
> [Reps et al. 1983]

In the standard updating algorithm, correct choices are made by taking into account both direct and transitive dependencies among attributes. The standard algorithm can be understood as a generalization of Knuth's topological sorting algorithm. Like topological sorting, it keeps a work-list of attributes that are ready for reevaluation (enumeration); an attribute is placed on the work-list when its in-degree is reduced to zero in a scheduling graph whose edges reflect dependencies among attributes that have not yet been reevaluated (enumerated). Whereas in topological sorting the vertices of the scheduling graph are known *a priori*, in the attribute updating algorithm, the set of vertices of the scheduling graph is generated dynamically at the same time as it is being enumerated.

The scheduling graph maintained by the algorithm is of crucial importance in finding an optimal reevaluation order. The initial scheduling graph represents dependencies among the attribute instances of the point of subtree replacement. The initial work-list consists of all vertices of the graph with in-degree 0, as these represent the attributes whose arguments are guaranteed to have consistent values. As updating progresses, the scheduling graph expands when changes propagate to attributes that are arguments of attributes outside the current graph.

During this process, the presence of edges representing transitive dependencies is particularly important in determining the optimal evaluation order. The presence of such edges ensures that an attribute is never updated until all of its ancestors have their correct, final values. Conversely, removing such an edge allows the algorithm to dispense in unit time with areas of the tree in which attribute values do not change.

The new attribute updating algorithm, called Propagate, is presented in Figure 5. Propagate is quite similar to the standard updating algorithm in the way it combines a topological-sort operation that removes edges from the scheduling graph with an expansion operation that causes new vertices and edges to be added to the scheduling graph. As long as attributes do not change value, the algorithm acts like topological sort applied to the scheduling graph; however, an expansion is triggered as soon as the current scheduling graph does not cover region of the tree in which attributes have changed value.

Expansions of the scheduling graph are handled by the calls on ExpandIfNecessary, given in Figure 6, performed when an attribute of a partition-set node changes value. In ExpandIfNecessary, this attribute is tested to determine whether it has successors that are not among the attributes represented by the vertices

---

```
procedure Propagate(T, r)
declare
   T: a fully attributed derivation tree
   r: a nonterminal node of T containing any inconsistent attributes of T
   S: a set of attribute instances
   M: a directed graph
   b, c: attribute instances
   OldValue, NewValue: attribute values
   K: a component of T
   k: a tree node
begin
   if r ∈ T.PartitionSet then
      M := r.C ∪ r.C̄
   else
      K := the component of T that contains r
      M := D(K) ∪ root(K).C̄ ∪ ⋃_{k ∈ interface nodes of K} k.C
   fi
   S := the set of vertices of M with in-degree 0 in M
   while S ≠ ∅ do
      Select and remove a vertex b from S
      OldValue := value of b
      evaluate b
      NewValue := value of b
      if OldValue ≠ NewValue and TreeNode(b) ∈ T.PartitionSet then
         ExpandIfNecessary(T, M, b, S)
      fi
      while there exists c, a successor of b in M do
         Remove edge (b, c) from M
         if in-degree_M(c) = 0 then Insert c into S fi
      od
   od
end
```

Figure 5. Incremental attribute evaluation.

```
procedure ExpandIfNecessary(T, M, b, S)
declare
   T: a fully attributed derivation tree
   M, Append, Discard: a directed graph
   b, c: attribute instances
   S: a set of attribute instances
   K: a component of the tree
   k: a tree node
begin
   if there exists c, a successor of b in D(T) that is not in M then
      if TreeNode(c) is a child of TreeNode(b) then
         K := the component of T whose root is TreeNode(b)
         Append := D(K) ∪ ⋃_{k ∈ interface nodes of K} k.C
         Discard := TreeNode(b).C
      else /* TreeNode(c) is the parent or is a sibling of TreeNode(b) */
         K := the component of T in which TreeNode(b) is a leaf
         Append := D(K) ∪ root(K).C̄ ∪ ⋃_{k ∈ interface nodes of K, k≠TreeNode(b)} k.C
         Discard := TreeNode(b).C̄
      fi
      M := (M − Discard) ∪ Append
      Insert into S all vertices of Append whose in-degree in M is 0
   fi
end
```

Figure 6. Expanding the scheduling graph.

of the scheduling graph. If such successors exist, the graph is expanded to take into account these attributes and their functional dependencies. Note that in ExpandIfNecessary an expansion introduces all the vertices and edges corresponding to attributes and dependencies in an entire *component*.

When Propagate terminates, the nodes of scheduling graph M consist of the attribute instances that are part of any component in which an attribute changed value. On termination, M has no edges at all; all dependency graph edges inserted into M by the calls on ExpandIfNecessary have been removed from M by the topological evaluation process.

Note that, as updating progresses, direct dependencies among attributes in a new component of the tree are included in the scheduling graph only if the component contains an attribute instance that changed value. The number of vertices and edges introduced into M by an expansion is bounded by $O(\sqrt{m})$, the maximum size of a component. M is only enlarged when a new member of AFFECTED has been identified; consequently, the maximum size of M is $O(|\text{AFFECTED}| \cdot \sqrt{m})$.

The cost of considering a vertex is one semantic function application and a constant amount of bookkeeping work. The cost of the expansion operation is $O(\sqrt{m})$. Consequently, the total cost of Propagate is $O(|\text{AFFECTED}| \cdot \sqrt{m})$.

It may be helpful to think of Propagate as simply being the standard updating algorithm operating on a

(compressed) tree whose nodes are the elements of *T.PartitionSet*[4] and whose productions correspond to the T's components. In fact, in the next section, it will be helpful to have an actual data structure to represent such a tree; this structure will be denoted as T/P. Clearly, T/P can be constructed during the second phase of partitioning, and its nodes can be given pointers to their corresponding node in T.

3.3. Overhead Updating

What remains to be described are the parts of the algorithm that handle overhead updating. The overhead-updating costs of the algorithm originate from two sources: the operation of updating the characteristic graphs, and the operation of repartitioning the tree. As will be shown, both kinds of operations contribute $O(\sqrt{m})$ to the cost of each tree-modification.

Updating characteristic graphs

Each time the derivation tree is modified, the characteristic graphs associated with the nodes of the current partition-set are updated. Because the number of graphs is $O(\sqrt{m})$, each characteristic graph must be updated in unit time if the cost of this operation is to be $O(\sqrt{m})$.

This is managed by having the partitioning step creates some additional auxiliary information for use when the characteristic graphs are updated. The information generated is the projection of each component's dependency graph on the attributes of the component's partition-set nodes and is stored at the root of each component. If X is the root of a component, the projected graph is denoted by $X.D$. This information is used by the procedure UpdateCharacteristicGraphs, shown in Figure 7, to recompute the characteristic graphs associated with each partition-set node. The parameter r represents the location at which tree T was modified. The first step of UpdateCharacteristicGraphs is to recompute the $X.D$ graph for the component that contains r, information which may have been invalidated by whatever editing operation was performed. When the size of the component is bounded by $O(\sqrt{m})$, so this step runs in time $O(\sqrt{m})$.

The two loops in UpdateCharacteristicGraphs recompute the subordinate and superior characteristic graphs of partition-set nodes. Note that this part is essentially the same as the operation BuildCharacteristicGraphs except that it operates on the tree of partition nodes, T/P, where $X.D$, the projected dependency graph for a component plays the role of $D(p)$, the dependency graph of an individual production. Each of the $O(\sqrt{m})$ graphs is updated in unit time; thus, the total cost of UpdateCharacteristicGraphs is $O(\sqrt{m})$.

The $X.D$ graphs are computed by the procedure Project, given in Figure 8. This procedure is the fourth partitioning phase that was referred to in Section 3.1. The projection operation that Project performs on each component is done by a depth-first search from each inherited attribute of the root of the component and each synthesized attribute of each interface node of the component. Each of the other three phases of partitioning runs in linear time; the reason that Project also runs in linear time is that it is only ever applied to the partition set generated by applying both Partition and AddAncestors to the tree – the second argument to Project has the value AddAncestors(T, Partition(T)). In this case, the number of nodes in P is no more than $O(\sqrt{m})$, and, by the properties of AddAncestors, each component is adjacent to at most MaxSons + 1 other components.

The components created by the nodes found by the first phase of partitioning may be adjacent to as many as $O(\sqrt{m})$ other components. This condition could cause Project to run in $O(m^{3/2})$ time, and to use $O(m)$

[4]For the remainder of the paper, we assume that certain structures are stored with a tree, such as the current partition set, denoted *T.PartitionSet*.

---

procedure UpdateCharacteristicGraphs(T, P, r)
declare
  T: derivation trees
  P, Q: a set of tree nodes
  r: a node of T containing any inconsistent attributes of T
  X, Y: tree nodes
  K: a component of the tree
  k: a tree node
begin
  K := the component of T that contains r
  Q := $\{\text{root}(K)\} \cup \{Y \mid Y \text{ is an interface node of } K\}$
  $\text{root}(K).D := D(K)/A(Q)$
  for each node $X \in P$ in the order of position in a postorder traversal of T/P **do**
    K := the component of T whose root corresponds to X
    $X.C := (X.D \cup \bigcup_{k \in \text{interface nodes of } K} k.C)/A(X)$
  od
  for each node $X \in P$ in the order of position in a preorder traversal of T/P **do**
    Y := parent(X)
    K := the component of T in which the node corresponding to X is a leaf
    $X.\overline{C} := (Y.D \cup \text{root}(K).\overline{C} \cup \bigcup_{k \in \text{interface nodes of } K, k \neq \text{TreeNode}(b)} k.C)/A(X)$
  od
end

Figure 7. For each node X of P, recompute $X.C$ and $X.\overline{C}$.

```
procedure Project(T, P)
declare
   T: a derivation tree
   P, Q: sets of tree nodes
   X: a tree node
   K_X: the component rooted at X
begin
   for each node X in a postorder traversal of T do
      if X ∈ P or X = root(T) then
         Q := {X} ∪ {Y | Y is an interface node of K_X}
         X.D := D(K_X)/A(Q)
      fi
   od
end
```

Figure 8. For each component defined by P, determine the projection of the component's dependency graph on the attributes of the component's partition-set nodes. This graph is denoted by $X.D$, where X is the root of the component.

space; however, the second phase of partitioning further subdivides the components from the first phase by adding to the partition set every node that is a lowest common ancestor of some pair of partition nodes. This means that each vertex of an $X.D$ graph can be connected to at most (MaxSons + 1)(MaxAttrs) other vertices. Each component is of size $O(\sqrt{m})$ and contains no more than MaxSons + 1 partition nodes, so the running time of the projection operation on the component is no more than $O(\sqrt{m})$. Since there are $O(\sqrt{m})$ components, the total running time of Project is $O(m)$.

Example. Figure 9 illustrates the four graphs created by Project when the set that partitions the tree from Figure 1 is $\{exp_5, exp_{13}, exp_{18}\}$. In this case, Project creates the graphs $ROOT_1.D$, $\exp_5.D$, $\exp_{15}.D$, and $\exp_{18}.D$.

Repartitioning the tree

The step of repartitioning the tree also contributes $O(\sqrt{m})$ in overhead-updating cost to each tree-modification operation. Repartitioning is handled by the procedure RepartitionIfNecessary, given below in Figure 10. Each editing transaction increments *T.OperationsSinceLastPartition*; if enough editing transactions have been performed since the last time the tree was partitioned, RepartitionIfNecessary calls Partition, AddAncestors, BuildCharacteristicGraphs, and Project and assigns *T.PartitionSet*, *T.OperationsSinceLastPartition*, and *T.SizeWhenLastPartitioned* appropriate values.

If a tree contains m nodes at the time it is partitioned, that partition is used by Propagate for the next $\sqrt{m}$ editing transactions. Because of the restriction that each modification of the tree can introduce at most a constant number of new nodes, at most $O(\sqrt{m})$ nodes are introduced by these transactions. Consequently, throughout the sequence of steps, every component contains at most $O(\sqrt{m})$ nodes. The $O(m)$ cost of repartitioning is amortized over the previous $\sqrt{m}$ editing transactions, which amounts to $O(\sqrt{m})$ per transaction.

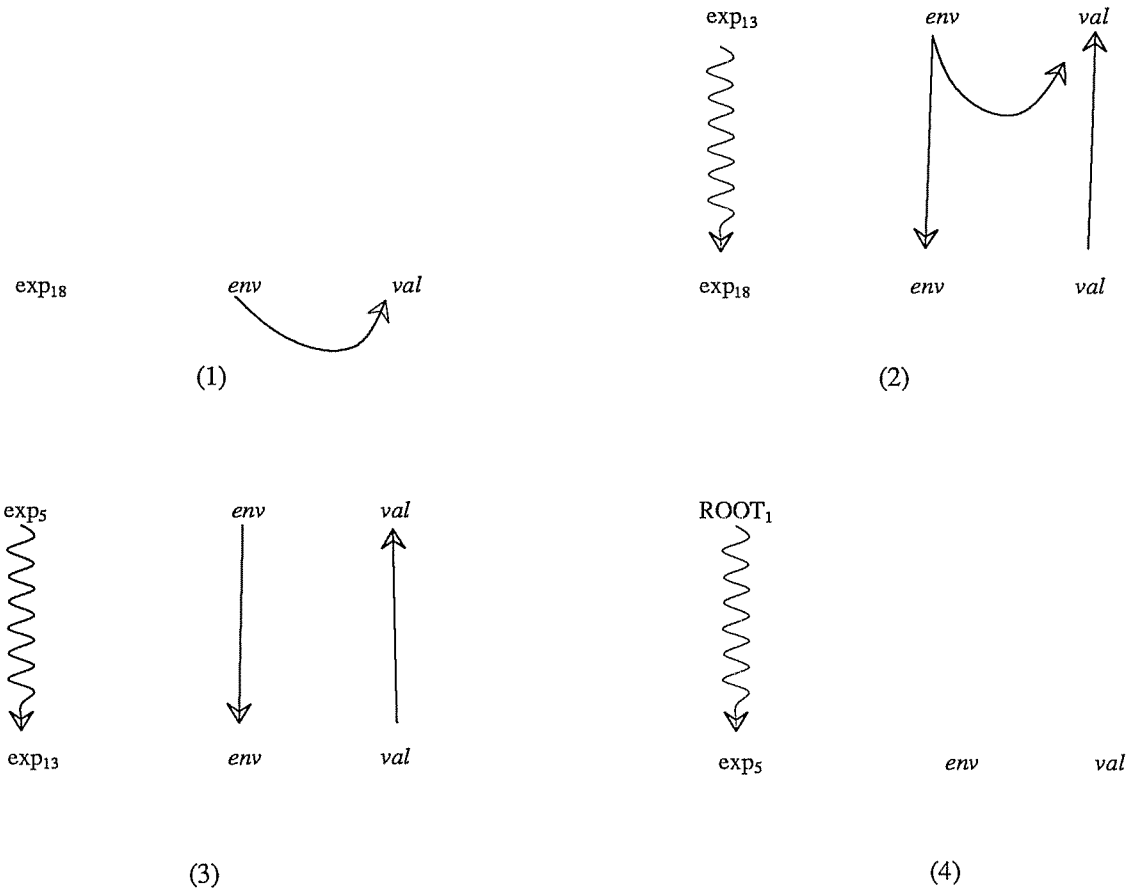


Figure 9. The four graphs created by Project when the set that partitions the tree shown in Figure 1 is $\{exp_5, exp_{13}, exp_{18}\}$. The graphs are created in the order shown: (1) $\text{exp}_{18}.D$, (2) $\text{exp}_{15}.D$, (3) $\text{exp}_5.D$, and (4) $ROOT_1.D$.

```
procedure RepartitionIfNecessary(T)
declare
   T: a consistent, fully attributed derivation tree
begin
   if T.OperationsSinceLastPartition ≥ √(T.SizeWhenLastPartitioned) then
      T.PartitionSet := AddAncestors(T, Partition(T))
      BuildCharacteristicGraphs(T, T.PartitionSet)
      Project(T, T.PartitionSet)
      T.OperationsSinceLastPartition := 0
      T.SizeWhenLastPartitioned := the number of nodes in T
   fi
end
```

Figure 10. Repartitioning a derivation tree.

3.4. Summary

The procedures defined in the previous three sections give us the tools for updating a tree after an editing operation modifies the tree. The first step is to call UpdateCharacteristicGraphs; this updates the characteristic graphs for the nodes in the tree's current partition set. The second step is to call RepartitionIfNecessary, which repartitions the tree if enough editing operations have been performed since the last time the tree was partitioned. The final step is to call Propagate so that all the tree's attribute values are given consistent values. When the cost of updating the tree is amortized over a sequence of tree modifications, the cost per modification amounts to $O((1 + |\text{AFFECTED}|) \cdot \sqrt{m})$, where m is the size of the tree.

The updating algorithm described above extends the standard attribute-grammar model of editing so that the editing cursor may be moved from one location to any other location as a single operation. Moving the cursor neither modifies the tree nor invalidates any of the tree's characteristic graphs, so no overhead-updating operations are necessary; consequently, cursor motion is a unit-cost operation.

REFERENCES

[Demers et al. 1981]

Demers, A., Reps, T., and Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In Conference Record of the Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, Va., Jan. 26-28, 1981, pp. 105-116.

[Demers et al. 1985]

Demers, A., Rogers, A., and Zadeck, F.K. Attribute propagation by message passing. In Conference Record of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, Seattle, WA, June 25-28, 1985. Appeared as: *SIGPLAN Notices 20*, 7 (July 1985), 43-59.

[Hoover 1986]

Hoover, R. Dynamically bypassing copy rule chains in attribute grammars. In Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, FL, Jan. 13-15, 1986, pp. 14-25.

[Johnson & Fischer 1982]

Johnson, G.F. and Fischer, C.N. Non-syntactic attribute flow in language based editors. In Confer-

ence Record of the Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., Jan. 25-27, 1982, pp. 185-195.

[Johnson & Fischer 1985]
Johnson, G.F. and Fischer, C.N. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages, New Orleans, La., Jan. 14-16, 1985, pp. 141-151.

[Kastens 1980]
Kastens, U. Ordered attribute grammars. *Acta Inf. 13*, 3 (1980), 229-256.

[Knuth 1968]
Knuth, D.E. Semantics of context-free languages. *Math. Sys. Theory 2*, 2 (June 1968), 127-145. Correction. *ibid. 5*, 1 (Mar. 1971), 95-96.

[Reps 1982]
Reps, T. Optimal-time incremental semantic analysis for syntax-directed editors. In Conference Record of the Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., Jan. 25-27, 1982, pp. 169-176.

[Reps 1984]
Reps, T. *Generating language-based environments*. M.I.T. Press, Cambridge, Mass., 1984.

[Reps et al. 1983]
Reps, T., Teitelbaum, T., and Demers, A. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 449-477.

[Reps & Demers 1987]
Reps, T. and Demers, A. Sublinear-space evaluation algorithms for attribute grammars. To appear in *ACM Trans. Program. Lang. Syst.*

[Reps & Teitelbaum 1984]
Reps, T. and Teitelbaum, T. The Synthesizer Generator. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn., Apr. 23-25, 1984. Appeared as joint issue: *SIGPLAN Notices* (ACM) *19*, 5 (May 1984), and *Soft. Eng. Notes* (ACM) *9*, 3 (May 1984), 42-48.

[Reps & Teitelbaum 1985]
Reps, T. and Teitelbaum, T. The Synthesizer Generator reference manual. Tech. Rep. 84-619, Dept. of Computer Science, Cornell Univ., Ithaca, NY, Aug. 1985.

[Reps & Teitelbaum 1987]
Reps, T. and Teitelbaum, T. *The Synthesizer Generator*. In preparation.

[Tarjan 1983]
Tarjan, R.E. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Tsakalidis 1984]
Tsakalidis, A.K. Maintaining order in a generalized linked list. *Acta Informatica 21* (1984), 101-112.

[Yeh 1983]
Yeh, D. On incremental evaluation of ordered attributed grammars. *BIT 23*, (1983), 308-320.