Incremental Attribute Evaluation and Multiple Subtree Replacements

Stephen B. Peckham
Ph.D Thesis

TR 90-1093
February 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

INCREMENTAL ATTRIBUTE EVALUATION AND MULTIPLE SUBTREE REPLACEMENTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Stephen Bailey Peckham

May 1990

INCREMENTAL ATTRIBUTE EVALUATION AND
MULTIPLE SUBTREE REPLACEMENTS

Stephen Bailey Peckham, Ph.D.
Cornell University 1990

The standard model for incremental attribute evaluation allows single subtree replacements followed by attribute reevaluation to restore consistency to a derivation tree. This thesis advocates an extended model that allows multiple subtree replacements. A static (tree-walking) algorithm for performing incremental updating after such changes is developed. The algorithm cannot be used with all attribute grammars, but is restricted to grammars contained in the new class of "globally partitionable attribute grammars" (GPAGs). A test for determining whether an attribute grammar is GPAG is described.

The multiple subtree replacement algorithm (*GPAG-evaluate*) in this thesis improves on two shortcomings of existing algorithms. First, many evaluators have a running time that depends linearly on the size of the derivation tree or on the number of concurrent subtree replacements. *GPAG-evaluate* has a running time of $O(\log n \cdot |\text{AFFECTED}|)$, where n is the number of nodes in the derivation tree and AFFECTED is the set of attributes needing reevaluation. Second, experience with incremental, attribute grammar-based environments demonstrates that dynamic evaluators are noticeably slower than static evaluators because they require time-consuming data structure manipulations. Most existing algorithms for multiple subtree replacements are dynamic, but *GPAG-evaluate* is static.

A second problem treated in this thesis is asynchronous subtree replacements, that is, allowing changes to be made while propagation continues after previous changes. A method for analyzing the efficiency of asynchronous subtree replacement algorithms is presented. An asynchronous evaluator (*ASYNCH-evaluate*) is described that, like *GPAG-evaluate*, guarantees that no attributes will be evaluated unnecessarily. Under some restrictions, *ASYNCH-evaluate* is as efficient as *GPAG-evaluate*. In particular, propagation in trees containing dynamically-generated, nonlocal dependency edges can be supported.

Both *GPAG-evaluate* and *ASYNCH-evaluate* must find lowest common ancestors of nodes in a tree where subtree replacements were made. A simple technique performs this operation in time $O(n)$. To make the evaluators more efficient, this

thesis describes an algorithm that uses self-adjusting binary trees to perform the necessary operations in amortized $O(\log n)$ time. These operations are not restricted to attributed derivation trees, but can be used for any application using trees.

Biographical Sketch

Stephen Bailey Peckham was born on July 10, 1957 in Canton, N. Y. Aside from a year living in Ft. Collins, Colorado, he spent his entire formative years in Canton, graduating from Canton Central High School in 1975. He attended Carleton College in Northfield, Minnesota, graduating with a B.A. in mathematics in 1979. Upon graduation, he began working for IBM, first in Rochester, Minnesota, and then in Austin, Texas. In 1983, he took a leave of absence from IBM to attend Cornell, where he received an M.S. Degree in 1987 and a Ph.D. in 1990.

To Julie

Acknowledgements

First, I thank my advisor, Tim Teitelbaum, for his influence and assistance with the development of this thesis. His work with the Cornell Program Synthesizer attracted me to Cornell in the first place, and working with the Synthesizer Generator group was captivating. I would also like to thank Dexter Kozen and John Bowers for serving on my committee.

During my years in Ithaca, I benefitted greatly from my association with other members of the department, from the ping-pong days through the intramural hockey days. Two people were especially helpful, though. Roger Hoover explained many details about attribute grammars, and his wry sense of humor helped make my sojourn at Cornell more enjoyable. John Field, who shared an office with me for six-and-a-half years, made many different subjects more understandable for me.

The research reported in this thesis was supported in part by an NSF and ONR grant under NSF grant number CCR 8514862, and IBM granted me an educational leave of absence, allowing me to attend Cornell.

Finally, I thank Julie for her patience while I wrote this thesis. Being married to a graduate student is a hard job, and when the last few months stretched out to a year, her encouragement and support were essential to my success.

Table of Contents

List of Figures

Chapter 1

Introduction

Attribute grammars allow context-sensitive checking to be added to a context-free grammar. They have become a natural choice for implementing interactive editors, because their declarative nature eliminates the need to consider the sequence of modifications leading to a particular derivation tree; the values of consistent attributes depend only on the current derivation tree. The standard model for incremental attribute evaluation allows single subtree replacements followed by attribute reevaluation to restore consistency to a derivation tree. This thesis advocates an extended model that allows multiple subtree replacements. A static (tree-walking) algorithm for performing incremental updating after such changes is developed. The algorithm cannot be used with all attribute grammars, but is restricted to grammars contained in the new class of "globally partitionable attribute grammars" (GPAGs). A test for determining whether an attribute grammar is GPAG is described.

The multiple subtree replacement algorithm (*GPAG-evaluate*) in this thesis improves on two shortcomings of existing algorithms. First, many existing evaluators have a bound on running time that depends linearly on the size of the derivation tree or on the number of concurrent subtree replacements. *GPAG-evaluate* has a running time of $O(\log n \cdot |\text{AFFECTED}|)$, where n is the number of nodes in the derivation tree and AFFECTED is the set of attributes needing reevaluation. Second, experience with incremental, attribute grammar-based environments demonstrates that dynamic evaluators are noticeably slower than static evaluators because they require time-consuming data structure manipulations. Most existing algorithms for multiple subtree replacements are dynamic, including the Reps-Marceau-Teitelbaum algorithm [RMT86], the only existing evaluator with the same asymptotic time bound as *GPAG-evaluate*. *GPAG-evaluate*, on the other hand, is static.

A second problem treated in this thesis is asynchronous subtree replacements, that is, allowing additional subtree replacements to be made while propagation continues after previous changes. A method for analyzing the efficiency of asynchronous subtree replacement algorithms is presented. An asynchronous evaluator (*ASYNCH-evaluate*) is described that, like *GPAG-evaluate*, guarantees that no attributes will be evaluated unnecessarily. Under some restrictions, *ASYNCH-evaluate* is as efficient as *GPAG-evaluate*. In particular, propagation in trees containing dynamically-generated, nonlocal dependency edges can be supported.

Both *GPAG-evaluate* and *ASYNCH-evaluate* must find lowest common ancestors of two (or more) nodes in a tree quickly, and the nodes are used to define a "structure tree" (see Chapter 5). A simple technique performs the desired operations in time $O(\textit{depth of tree})$, which could be proportional to the size of the tree, since the derivation trees used in language-based editors are typically unbalanced. To make the evaluators more efficient, this thesis describes an algorithm that uses self-adjusting binary trees of Sleator and Tarjan [ST85] to perform the necessary operations in amortized $O(\log n)$ time. These operations are not restricted to attributed derivation trees, but can be used for any application using trees.

1.1 Background

Attribute grammars were introduced in 1968 by Knuth to assign "meaning" to a string in a context-free language [Knu68]. Since then, the use of attribute grammars has become widespread and much research has been done studying properties of attribute grammars and developing algorithms for evaluating attributed trees. The fundamental problem in using attribute grammars is finding an efficient algorithm for attribute evaluation. From the beginning, people have realized that an algorithm for evaluating the attributes of a derivation tree for an arbitrary attribute grammar can require a large amount of time and space. Therefore, attribute grammar classes have been defined that allow more efficient algorithms to be used.

The specification of an attribute grammar, along with an instance of a derivation tree, defines a dependency graph, i.e., a directed graph representing the dependencies between attribute values. If the dependency graph is acyclic, attributes can be evaluated in topological order. Most applications require that an attribute grammar be noncircular, that is, that all such graphs be acyclic. In fact, Knuth's original paper on attribute grammars included a test to detect whether an attribute grammar was circular.

Evaluation algorithms can be divided into two categories: static and dynamic. Static algorithms analyze all possible dependencies between attributes during evaluator construction, while dynamic evaluators must examine the dependencies in an actual derivation tree during evaluation. In general, static evaluators are faster, because much of the work of analyzing the attribute grammar is done once.

Dynamic evaluators work, in effect, by topologically sorting the dependency graph. Static evaluators, also called tree-walking evaluators, work by traversing the derivation tree in some order, evaluating attributes at nodes reached during the traversal. For example, Bochmann [Boc76] suggests using a fixed number of left-to-right passes through a tree and gives a test to see if a grammar can be evaluated in this way. Unfortunately, attribute grammars meeting this restriction are not very practical.

A generic tree-walking evaluator is described by Engelfriet and Filè [EF80]. It subsumes different static evaluators by giving a general, nondeterministic algorithm, then restricting the nondeterminism in various ways. All tree-walking evaluators are instances of the Engelfriet and Filè evaluator, although practical evaluators must be deterministic. The tree-walking evaluator of Cohen and Harry [CH79] comes closest to the most general algorithm of Engelfriet and Filè. It can be used with any attribute grammar, but is really a hybrid between a static and a dynamic evaluator, since it requires a preliminary preorder-traversal of the derivation tree to compute the proper order to visit the nodes when evaluations are actually performed.

Other tree-walking evaluators include one by Kennedy and Warren [KW76] that works for "absolutely-noncircular attribute grammars" and one by Kastens [Kas80] that works for "ordered attribute grammars." For the Kennedy and Warren evaluator, the order of visits to the children of the current node depends on the nodes between the current node and the root of the derivation tree. The Kastens evaluator, on the other hand, does not require any state information other than the current node. Kastens's evaluator actually works for "partitionable attribute grammars," a superclass of ordered grammars. Nevertheless, ordered attribute grammars are important because an efficient (polynomial-time) algorithm exists to test whether a grammar is ordered.

These early evaluators were algorithms for one-time evaluation, that is, algorithms that evaluate the attributes of an entire derivation tree, computing the value of a distinguished attribute value and then discarding the tree. For example, the GAG system [KHZ82] uses attribute grammars as a basis for a compiler-generator, and the value of the distinguished attribute is the generated code. For the one-time evaluators, other issues are important besides computing an evaluation order, such as optimizing the use of storage during evaluation. In some cases, different attributes can share the same storage, space can be freed after a value has been used for the last time, and useless attributes, i.e., ones whose values do not contribute to the distinguished attribute's value, can be identified and ignored.

In 1983, Reps, Teitelbaum, and Demers proposed using attribute grammars as a basis for allowing "incremental context-dependent analysis for language-based editors" [RTD83], and the problem of incremental attribute evaluation arose. The use of attribute grammars was pursued, and the work of Reps [Rep84] demonstrated the practicality of implementing an interactive editor using attribute grammars and incremental reevaluation to enforce context-sensitive constraints. Other interactive systems using attribute grammars include a system to maintain consistency of a modular system [Pfr86], a distributed editing system [KK86], and Poe [JF85], a language-based editor. These systems are not concerned so much with optimizing storage usage as they are with finding practical, incremental algorithms.

Reps's original algorithm works for any noncircular attribute grammar, but it must incrementally maintain partial dependency graphs (called characteristic graphs) at each node, and this overhead is expensive. Experience with the Synthesizer Generator [RT89a] shows that this first Reps algorithm is quite slow, and as an additional drawback, uses a large amount of storage for saving the characteristic graphs. Fortunately, a modified version of Kastens's evaluator can perform incremental updating efficiently without saving any state information in the nodes [Rep84,Yeh83,RT89a].

For the most part, attribute-grammar-based interactive systems allow a single object (or a collection of independent objects) to be edited. After each editing change, attribute values are recomputed using an incremental algorithm, providing immediate feedback about errors or inconsistencies in the object. The standard incremental evaluation algorithm provides for reestablishing consistency of a tree after a single subtree replacement. The goal of most incremental algorithms is to run in time $O(|\mathrm{AFFECTED}|)$, where AFFECTED is the set of attributes needing new values after a change, and such algorithms are called "optimal." Incremental versions exist for both static and dynamic evaluators, but once again, the static evaluators have much lower overhead, and so are preferred to dynamic evaluators.

In measuring the efficiency of incremental algorithms, one usually assumes that evaluating an attribute takes a constant amount of time. For some attribute grammars, this assumption is unrealistic, and improvements are possible to speed up the evaluation of individual attributes. For example, Pugh [Pug88] proposed saving the results of function calls, avoiding recomputation if the same function call is used again. In this thesis, I ignore the time needed to evaluate attributes and call an algorithm "optimal" if it evaluates $O(|\mathrm{AFFECTED}|)$ attributes and in addition, requires $O(|\mathrm{AFFECTED}|)$ bookkeeping time.

Hoover [Hoo87] studied ways to reduce the number of attribute values that need to be recomputed after a change and ways of reducing the number of dependencies between values. He proposed using finite functions to avoid some of the overhead of manipulating large attribute values, but a new evaluation technique (approximate topological order) was needed to perform propagation after changes. Hoover's algorithm is dynamic, but its space overhead is smaller than that of other dynamic algorithms, because it does not maintain characteristic graphs. Although Hoover's algorithm can be badly suboptimal in the worst case, since attributes can be evaluated more than once, he reports that it works well, in practice.

The incremental-evaluation model allows a single subtree replacement to be made at a time. This model can be cumbersome; it can be made more general by allowing multiple subtree replacements at a time. This extra flexibility can be useful for a number of reasons. First, a user editing a large derivation tree may wish to turn off attribution while making changes, invoking the evaluator manually from time to time. This feature would be beneficial when making a series of changes whose overall effect is slight, but for which each separate change might require evaluating a large number of attributes. Second, it is often desirable to replace an interior portion of a tree, whether done manually or under control of some transformation on the derivation tree. While such an operation can be simulated by making several single subtree replacements, it is more efficient to perform propagation once after all changes. Third, some systems support the creation of "nonlocal" edges, direct dependencies between attributes of widely separated nodes. One can model such a situation by treating a change at the source end of a nonlocal edge as an asynchronous subtree replacement at the defining node of the attribute at the destination end of the edge.

These reasons for extending the model suggest providing for efficient propagation after multiple subtree replacements. A few solutions have been suggested for solving this problem, but most of them have one of two drawbacks: either they evaluate attributes outside AFFECTED, or they are dynamic evaluators and require the maintenance of characteristic graphs. Reps suggested invalidating the attributes of all nodes connecting the multiple points of change [Rep84], but as a result attributes outside AFFECTED may be evaluated. Kaplan and Kaiser [KK86] treat the points of change independently, performing propagation in parallel at all points of change, but once again, this may require evaluating attributes outside AFFECTED, and furthermore, some attributes may be evaluated more than once. Hoover's algorithm can be extended to this new problem without change, but as before, it does not guarantee that attributes will be evaluated in topological order, so it can perform poorly for some attribute grammars. Finally, Kastens and Yeh describe a tree-walking evaluator that only evaluates attributes in AFFECTED, but can have overhead proportional to the size of the derivation tree [YK88].

One algorithm that does not evaluate attributes outside AFFECTED and has a running time of $O(\log n \cdot |\mathrm{AFFECTED}|)$ is the Reps-Marceau-Teitelbaum algorithm [RMT86], but it is a dynamic algorithm. To avoid the overhead of maintaining characteristic graphs essential to most dynamic algorithms, I develop a static evaluator, called *GPAG-evaluate*, that can be used to update attributes after multiple subtree replacements. *GPAG-evaluate* works for a new class of attribute grammars: "globally partitionable attribute grammars," a subclass of partitionable attribute grammars. Its overhead is $O(\log n \cdot |\mathrm{AFFECTED}|)$, and it will not evaluate attributes unnecessarily nor evaluate attributes outside AFFECTED.

A further extension of the editing model allows asynchronous subtree replacements, that is, changes to a derivation tree while propagation is still occurring as a result of a previous change. This model would allow multiple users to manipulate a single object, allowing them to edit an entire system, rather than a single file. Just as an interactive editor provides immediate feedback about errors as changes are being made while writing a single program, the same sort of feedback would help designers editing a large, modular system. Such a system editor would allow configuration and interface errors to be discovered sooner than is possible with current tools.

Therefore, I show how to extend *GPAG-evaluate* to create *ASYNCH-evaluate*, which supports asynchronous subtree replacements. I give a precise definition of AFFECTED in the context of asynchronous changes, and show that *ASYNCH-evaluate* will never evaluate attributes outside AFFECTED. At best, its overhead is $O(\log n \cdot |\text{AFFECTED}|)$, the same as for *GPAG-evaluate*, but depending on the changes being made, overhead can be higher. The Kaplan-Kaiser and Hoover algorithms can also perform propagation after asynchronous subtree replacements without being modified substantially. As usual, since these algorithms can evaluate attributes outside AFFECTED, and can evaluate attributes more than once, their worst case running times can be poor. Finally, Geitz [Gei86] describes an algorithm for asynchronous subtree replacements that is an extension of the Reps-Marceau-Teitelbaum algorithm. The efficiency of these algorithms is hard to judge, because they have time bounds that do not depend on AFFECTED, but instead depend on the attribute grammar used and the sequence of changes made to the derivation tree.

1.2 Thesis Outline

Chapter 2 reviews attribute grammars, discusses dependency graphs and attribute evaluation, and defines "subtree replacement," the principle editing operation. Measuring the complexity of an attribute evaluation algorithm is discussed, as well. Chapter 3 presents the two principal attribute evaluation methods: dynamic and static algorithms. Basic incremental algorithms are also discussed. The problem of multiple subtree replacements is introduced in Chapter 4, which also shows the general approach to extending the incremental algorithms of Chapter 3 to perform incremental attribute reevaluation after multiple subtree replacements. Chapter 5 can be read separately from the rest of this thesis. It discusses the operations used to maintain "structure trees," which are needed by both *GPAG-evaluate* and *ASYNCH-evaluate*. Chapter 6 defines "globally partitionable attribute grammars" and describes *GPAG-evaluate*, the incremental algorithm that works with GPAGs, and Chapter 7 shows how *GPAG-evaluate* can be extended to create *ASYNCH-evaluate*, an algorithm supporting asynchronous subtree replacements. The thesis concludes with Chapter 8, which includes suggestions for future research.

Chapter 2

Attribute Grammars

An *attribute grammar* G [Knu68,Rep84] is a context-free grammar G_0 that has been extended by associating a finite set of attributes with each nonterminal of G_0. Associated with each production of G_0 is a set of *attribution rules*, with each rule defining the value of an attribute occurrence of one of the nonterminals in the production in terms of a *function* applied to other attribute occurrences of the nonterminals in the production. The attributes of a nonterminal X are divided into two disjoint sets: a set of *synthesized* attributes, denoted $S(X)$, and a set of *inherited* attributes, denoted $I(X)$. We use $A(X)$ to denote $S(X) \cup I(X)$, and write $X.a$ to refer to attribute a of nonterminal X.

If p is a production of G_0, then the *output attributes* of p are the synthesized attribute occurrences of the left-hand-side nonterminal of p and the inherited attribute occurrences of the right-hand-side nonterminals. *Input attributes* of p are the remaining attribute occurrences. Each attribution rule associated with a production p defines a value for an output attribute of p. An attribute grammar is in *normal form* [Boc76] if no attribution function uses an output attribute as an argument.

G_0 is called the *underlying context-free grammar* of G. Usually, we use G in place of G_0, and talk of productions and nonterminals of G rather than of G_0. Because we are only concerned with attributes and their values, we ignore terminal symbols, for the most part. We treat all terminal symbols as if they were ϵ (the empty string).

A *derivation tree* is used to represent a sentence in the language of G_0 (or equivalently, of G). We assume that G_0 has no useless nonterminals [HU79], and hence, no useless productions. That is, for each nonterminal X of G_0, there exists a derivation tree T containing an instance of X. Likewise, for each production of G_0, there exists a derivation tree T containing an instance of p. For each derivation tree node α, we define $\tau(\alpha)$ to be the nonterminal instantiated at α, and $\rho(\alpha)$ to be the production applied at α. If α and β are two tree nodes such that α is the parent of β, we write $\alpha \overset{1}{\leadsto} \beta$. The relation $\leadsto$ is defined as the transitive closure of $\overset{1}{\leadsto}$, so if α is any ancestor of β ($\alpha \neq \beta$), then $\alpha \leadsto \beta$. The same relation is defined for nonterminals, so $X \rightsquigarrow Y$ if and only if there exists a derivation tree with nodes $\alpha \rightsquigarrow \beta$ such that $\tau(\alpha) = X$ and $\tau(\beta) = Y$.

Each tree node α defines a set of *attribute instances* corresponding to the attributes of $\tau(\alpha)$. If $a \in A(\tau(\alpha))$, then $\alpha.a$ is the corresponding attribute instance of α, and $A(\alpha) = \{\alpha.a \mid a \in A(\tau(\alpha))\}$. The entire tree, then, defines a set of attribute instances, each instance being associated with a nonterminal instance in the tree. For each attribute instance $\alpha.a$, the *defining node* of $\alpha.a$ is the node that is a left-hand-side nonterminal occurrence of the production for which a is an output attribute. Note that $\mathit{defining\_node}(\alpha.a)$ is α if a is synthesized, and is the parent of α if a is inherited.

In order to compute values for all attribute instances in a tree, a derivation tree should always be a complete parse tree for G. Yet in order to use attribute grammars to support incremental editing of programs and other structured objects, we would like the user to be able to manipulate partial derivation trees. To reconcile these views, we use *completing productions* [Rep84]. Each nonterminal in G has an associated ϵ-production that is automatically used where an unexpanded nonterminal has been left in the tree by the user. Hence, partial trees in the view of the user are complete trees in the view of the system.

An *attributed tree* T is a derivation tree together with an assignment of either a value or the special token **null** to each attribute instance in the tree. T is *fully attributed* if each of its attribute instances is *available*, i.e. non-null. The arguments of an attribute instance $\alpha.a$ are the attribute instances that are arguments to the attribution function for $\alpha.a$. An attribute instance $\alpha.a$ is *consistent* if its arguments are available and the value of $\alpha.a$ is equal to its attribution function applied to its arguments. Otherwise, $\alpha.a$ is *inconsistent.* If all attribute instances in T are consistent, then T is consistent. Conversely, if any attribute instance in T is inconsistent, T is inconsistent.

An *attributed subtree* T' of an attributed tree T is a subtree of T whose root is not necessarily an instance of the start nonterminal of G and whose leaves may be nonterminals of G. If T' has root α, then some attribute instances of α may be output attributes of a production with an occurrence of $\tau(\alpha)$ on the right-hand side. Similarly, attribute instances of an instance of a nonterminal X at a leaf of T' may be output attributes of a production with an occurrence of X on the left-hand side. Nevertheless, we call an attribute instance $\beta.a$ in T' consistent if there is an attributed tree T containing T' and its attributes, such that $\beta.a$ has the same value in both T and T', and $\beta.a$ is consistent in T. Likewise, we call an attributed subtree consistent if all its attributes instances are consistent. Consequently, a subtree of a consistent, attributed tree is also consistent.

An attribute grammar is *well-formed* when the terminal symbols of the grammar have no synthesized attributes, the start nonterminal of the grammar has no inherited attributes, and each production includes attribution rules for all output attributes of the production.[1] If an attribute grammar is well-formed, then each

[1]Some authors use "well-formed" as a synonym for "noncircular," which we define in the next section.

attribute instance in the derivation tree will be defined by exactly one attribution rule.

2.1 Dependency Relationships

In order to talk about evaluation orders, it is useful to model dependencies between attribute instances in a derivation tree with dependency graphs. A *dependency graph* is a directed graph that represents dependencies among the attribute instances of a tree.

A *production-graph* for a production $p : X_0 \rightarrow X_1, \ldots, X_m$, denoted $DP(p)$, has vertices $\{X_i.a \mid 0 \leq i \leq m \text{ and } a \in A(X_i)\}$ and directed edges $[X_i.a, X_j.b]$, if in production p, the attribution function defining $X_j.b$ uses $X_i.a$ as an argument. In this case, we say that $X_j.b$ *depends* on $X_i.a$. The dependency graph for a tree T, denoted $D(T)$, is constructed by "pasting" together the production-graphs for each production instance in the derivation tree. More precisely, $D(T)$ has vertices $\{\alpha_0.a \mid \forall\alpha_0 \in T \text{ such that } a \in A(\tau(\alpha_0))\}$. For each node α_0 in T, $D(T)$ has edges

$$\{[\alpha_i.a, \alpha_j.b] \mid [X_i.a, X_j.b] \text{ is an edge in } DP(p), \text{ and } \rho(\alpha_0) = p\},$$

where p is the production $X_0 \rightarrow X_1, \ldots, X_m$ and α_i are the children of α_0, for $1 \leq i \leq m$, so that $\tau(\alpha_i) = X_i$. Note that the $DP(p)$ graphs depend only on the attribute grammar itself, while $D(T)$ depends on the particular derivation tree T.

An attribute grammar G is *noncircular* when the dependency graph of every possible derivation tree of G is acyclic. In this thesis, we only consider attribute grammars that are well-formed and noncircular. (Nonetheless, work has been done in evaluating circular attribute grammars. For example, see [Wal88,JS86, Far86].) For any noncircular attribute grammar G, $DP(p)$ must be acyclic for all p in G, and so G can be transformed easily into normal form [Boc76]. Therefore, we assume that all attribute grammars are in normal form.

For a noncircular attribute grammar, the edges in $D(T)$ define a partial order over the attribute instances in T. If attribute instance $\alpha.a$ is an argument of attribute instance $\beta.b$, $D(T)$ contains a directed edge $[\alpha.a, \beta.b]$. A path from $\alpha.a$ to $\beta.b$ means that $\alpha.a$ is used to determine the value of $\beta.b$, and so $\alpha.a$ must be evaluated before we are able to assign the proper value to $\beta.b$. This order is important, because an efficient evaluator should avoid evaluating $\beta.b$ more than once. If a grammar is noncircular, we can evaluate attributes in topological order respecting $D(T)$, resulting in every attribute instance being consistent.

2.1.1 Derived Dependency Graphs

Dependency relationships can also be defined for nonterminals, and are represented by directed graphs, denoted generally by $DS(X)$, whose vertices are the attributes of nonterminal X. The edges of these graphs are defined in different ways, depending on our purposes. Some of these dependency graphs are defined by projecting a graph onto the attributes of a nonterminal.

Definition 2.1 *Let $G = (V, E)$ be a graph, with vertices V and edges E. Let N be a subset of the vertices, i.e. $N \subseteq V$. Then the* graph projection *of G onto N, denoted $G \backslash N$, is a graph with vertices N and edges*

$E' = \{[v_1, v_2] \mid$ *there is a path from v_1 to v_2 in G that contains no vertices in N other than v_1 and v_2*$\}$.

This definition assures us that if $v_1, v_2 \in N$ and v_1 is reachable from v_2 in G, then v_1 is reachable from v_2 in $G \backslash N$.

Let T be a derivation tree. A *subordinate characteristic graph* $c_\alpha(X)$ is a graph whose vertices are the attributes of X and whose edges represent the dependencies induced by the subtree rooted at node α (with $\tau(\alpha) = X$). More precisely, we remove all edges in $D(T)$ leaving synthesized attributes of α or entering inherited attributes of α. We then project the resulting graph onto the attributes of α, obtaining $c_\alpha(X)$. This graph defines a partial evaluation order for the attribute instances of α. Similarly, a *superior characteristic graph* $c^\alpha(X)$ is a graph whose vertices are the attributes of X and whose edges represent the dependencies induced by the part of T above α. That is, we remove all edges in $D(T)$ entering synthesized attributes of α or leaving inherited attributes of α, and project the resulting graph onto the attributes of α. Naturally, $c_\alpha(X) \cup c^\alpha(X)$ must be acyclic for all nodes in T if the grammar is noncircular. Other dependency graphs can be defined in terms of characteristic graphs and used to define various grammar classes.

2.2 Attribute Evaluation

Attribute evaluation is the process of computing values for the attribute instances in a derivation tree. The definition of an attribute grammar does not impose an order on evaluating attributes, which is one of the advantages of using attribute grammars. The values in a consistent tree depend only on the tree itself, not on the sequence of operations used to construct the tree, nor on the order in which the attribute instances are evaluated. Of course, finding an efficient evaluation order is crucial if attribute grammars are to be of practical importance. In particular, we are primarily concerned with algorithms that do not evaluate any attribute instance more than once. Obviously, to compute a fully-attributed tree from scratch, we must evaluate all attribute occurrences at least once.

Some authors define the "meaning" of a derivation tree to be the value of the instance of a distinguished synthesized attribute of the root of the tree. With this view, we only need to evaluate attribute instances that are needed to compute the distinguished attribute. A *useless attribute* is one whose value does not contribute to the distinguished attribute's value. In addition, we can discard values for attribute instances after they have been used for the last time in computing the value of other attribute instances. The GAG system [KHZ82] has an algorithm for eliminating the computation of useless attributes, and Reps [Rep84] presents two algorithms that save space by reusing memory when a value is no longer needed.

An editor, though, uses attributes to provide information to the user about the object being edited. For example, the Synthesizer Generator [RT89a] allows error messages to be displayed at the sites of type errors and other static semantic errors. Therefore, we take the view that all attribute values must be maintained consistently at all times. An attribute that is useless for one derivation tree might not be useless for a slightly changed tree. Therefore, we are not concerned with the optimizations mentioned in the previous paragraph.

2.3 Incremental Attribute Evaluation

Incremental evaluation (also called *incremental updating* or *attribute propagation*) is the process of taking a consistent, fully-attributed derivation tree, changing it slightly (in a way to be described), and then reestablishing consistency of the derivation tree. By using existing attribute values, we can avoid recomputing new values for all attribute instances. If we are fortunate, only a few attribute instances will require new values. The goal of incremental evaluation is to discover which attribute instances need new values without examining the entire tree.

An incremental algorithm saves work by reusing the results of previous computations, that is, by using old attribute values. If the values of all the predecessors of an attribute instance $\alpha.a$ are unchanged, then $\alpha.a$ will not have to be recomputed, and neither will any of its successors (unless they happen to depend on another changed attribute instance). Thus, any incremental algorithm must keep track of whether the value of an attribute instance has changed, avoiding reevaluation in a particular portion of the dependency graph if the value has not changed.

2.3.1 Subtree Replacements

In order to talk about incremental algorithms and efficiency, we need a way to describe how a tree can be modified. The basic editing operation will be a *subtree replacement.* Suppose we wish to replace a subtree T_1 rooted at α, with $\tau(\alpha) = X$. We must replace T_1 with another tree rooted at β such that $\tau(\beta) = X$. After making the replacement, some of the attribute instances of β may be replaced with corresponding attribute instances of α. A subtree replacement is shown in Figures 2.1 and 2.2. In this example, X has four attributes, shown next to the nodes. Inherited attributes are marked with $\downarrow$'s and synthesized attributes are marked with $\uparrow$'s. Note that we kept the inherited attribute instances of β, but replaced the synthesized attribute instances. As long as both T (including T_1) and T_2 are consistent to begin with, then immediately after making the subtree replacement, all inconsistent attribute instances are at node β, and it makes sense to begin by evaluating one of the four attribute instances at β.

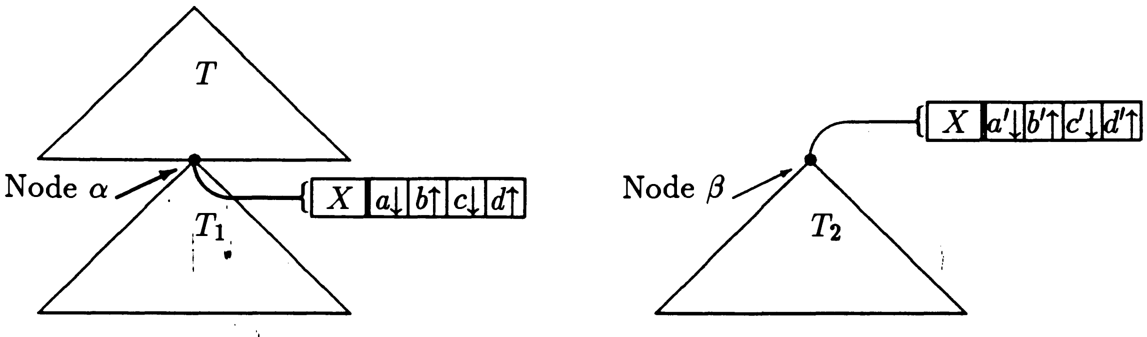


Figure 2.1: Situation before subtree replacement

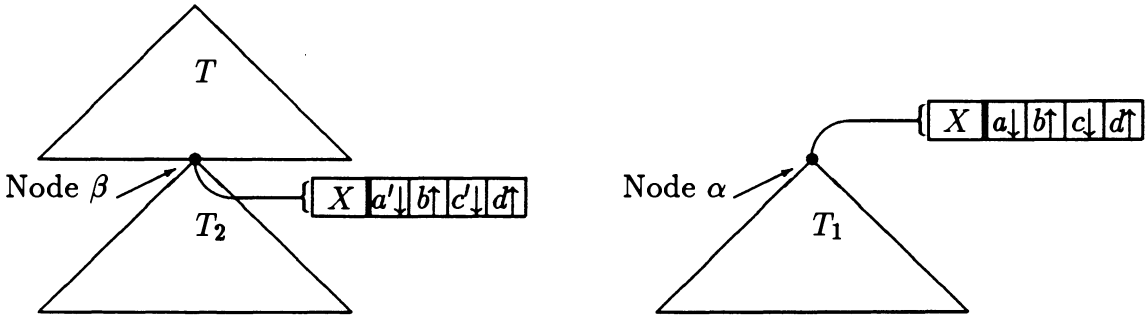


Figure 2.2: Situation after subtree replacement

Suppose we choose to reevaluate attribute instance $\beta.a$. If it turns out that the new value of $\beta.a$ is different from the old value, then the successors of $\beta.a$ might themselves be inconsistent, and so they would have to be reevaluated as well. Evaluation continues in this fashion until the entire tree is consistent. Of course, selecting the proper attribute instance to evaluate next is the crucial part of the algorithm.

2.3.2 Complexity of Incremental Algorithms

To discuss the efficiency of incremental algorithms in more detail, we need the following definitions:

Definition 2.2 *Let T be an attributed derivation tree, and T', an identical, but consistent derivation tree. Then*

$$\text{AFFECTED} = \{\alpha.a \mid \alpha.a \textit{ in } T \neq \alpha.a \textit{ in } T'\}.$$

That is, AFFECTED is the set of all attribute instances in a derivation tree that do not have the value they need for the tree to be consistent.

Definition 2.3 *Let T be an attributed derivation tree that has just undergone a subtree replacement at node β. Then*

$$\begin{aligned}\text{INFLUENCED} &= A(\beta) \cup \text{AFFECTED} \cup \\ &\quad \{\alpha.a \mid [\gamma.c, \alpha.a] \textit{ is an edge in } D(T) \textit{ and } \gamma.c \in \text{AFFECTED}\}.\end{aligned}$$

In other words, INFLUENCED is AFFECTED and any direct successors of attribute instances in AFFECTED, along with the attribute instances at the subtree replacement node. INFLUENCED contains the attributes that must be reevaluated to restore the consistency of the tree if nothing is known about the attribution functions by the evaluation algorithm. When an argument of an attribute instance changes, we must reevaluate the attribute instance to see whether its value has changed. Although AFFECTED and INFLUENCED are defined at the time a subtree replacement has been made, we can only discover their members by evaluating attribute instances. INFLUENCED includes all the attribute instances of the change point, because previous arguments to these attributes no longer exist in the tree, so there is no way to check whether recomputation is necessary.[2] Because we cannot determine AFFECTED *a priori*, we must evaluate attributes to determine membership in AFFECTED.

If an attribute instance $\alpha.a$ is in INFLUENCED, it will be reevaluated eventually. If $\alpha.a$ receives a new value during propagation, then all the successors of $\alpha.a$ will be in INFLUENCED as well, and we will know that $\alpha.a$ is in AFFECTED. If $\alpha.a$ keeps its original value, then $\alpha.a$ is not in AFFECTED, and we will not need to evaluate its successors (at least not on account of this attribute instance). Because the maximum outdegree of a vertex in the dependency graph is fixed for any given attribute grammar, then $O(|\text{AFFECTED}|) = O(|\text{INFLUENCED}|)$.[3] Therefore, we will generally talk about AFFECTED rather than INFLUENCED when discussing time bounds.

As mentioned in the previous section, we do not define AFFECTED as the set of all inconsistent attribute values in T. It is possible for a single attribute instance in T to be inconsistent, but for AFFECTED to be all attribute instances in T. Since we cannot find AFFECTED without actually recomputing its members, any algorithm for establishing the consistency of T must take time at least $O(|\text{AFFECTED}|)$. For an algorithm to achieve this bound, it must be able to compute AFFECTED by evaluating no more than $O(|\text{AFFECTED}|)$ attributes, giving us the following definition:

Definition 2.4 *An incremental algorithm is* optimal *if restoring consistency to a derivation tree takes $O(|AFFECTED|)$ bookkeeping time.*

This definition does not take into account the time to evaluate the attribution functions, because we make the usual assumption that functions can be evaluated in constant time.

In this thesis, the efficiency of an algorithm is measured relative to the efficiency of an optimal algorithm. Yet efficiency is not always defined the same

[2] Actually, if we save the old subtree, we can avoid including all the attributes of β in INFLUENCED. For example, if $a{\downarrow} = a'{\downarrow}$, then we will not have to reevaluate $\beta.a$, nor its successors in T_2, because they are already consistent.

[3] We assume that $|\text{AFFECTED}| > 0$.

way. In [YK88], for example, the measure of efficiency is just the number of attributes evaluated. But if overhead is ignored, we could compute and use $D(T)$ to determine an optimal evaluation order after any subtree replacement. Such an algorithm would take $O(|D(T)|)$ time to evaluate |INFLUENCED| attribute instances. Since $|D(T)|$ could be arbitrarily larger than |INFLUENCED|, this algorithm is not efficient.

Chapter 3

Attribute Evaluation Algorithms

There are two general approaches to algorithms for attribute evaluation. The first approach uses the dependency information of the current derivation tree to determine an evaluation order. Evaluators using this approach are called *dynamic* or *graph-based* evaluators. The second approach uses static information computed from the attribute grammar definition to determine an evaluation order. A *static* or *tree-walking* evaluator uses this second kind of approach. A tree-walking evaluator "moves" from node to node in the derivation tree, interleaving these moves with attribute evaluations. These approaches are not exclusive, nor really clear cut. Some evaluators are hybrids, using some static information coupled with dynamic examination of the existing dependencies.

Both approaches can be used to perform incremental evaluation. An incremental, dynamic evaluator is one of the main results of Reps's thesis [Rep84]. His algorithm works by maintaining characteristic graphs in the derivation tree. These graphs provide the necessary information to sequence attribute evaluations without reexamining the entire tree. A static evaluator performs incremental updating by only visiting the parts of the tree containing attributes that need new values. The remainder of the tree does not have to be examined at all.

Both static and dynamic evaluators can be optimal, because some versions of them have $O(|\text{AFFECTED}|)$ overhead. A dynamic algorithm is generally slower, though, because it takes a long time to maintain and sort dependency graphs. Static evaluators make use of a lot of information that can be computed once and for all from an attribute grammar, so they are usually much faster, but in general, they only work with subclasses of noncircular attribute grammars.

3.1 Dynamic Evaluators

A dynamic evaluator uses the dependency graph $\mathrm{D}(T)$ directly to determine the order of evaluation when attributing or reattributing a tree. In fact, if an evaluator is purely dynamic, it depends only on $D(T)$, and so it can be used for any problem that represents a collection of computations and dependencies by a directed, acyclic graph.

procedure evaluate(T : derivation tree)
S := {attribute instances of T that are ready for evaluation};
while $S \neq \emptyset$ **do**
Select and remove an attribute instance $b \in S$.
Evaluate b.
for each c a successor of b **do**
if c is ready for evaluation **then** $S := S \cup \{c\}$;

Figure 3.1: Worklist, attribute-evaluation algorithm

While we could compute a topological order for the entire dependency graph and evaluate the attributes in the order obtained, it is easier to use a worklist algorithm to sort the graph, interleaving sorting with evaluation [Rep84]. The worklist algorithm is simple. We call an attribute *ready* if all its arguments are consistent. Thus, constant attributes are always ready. As the arguments to other attribute instances are computed, those instances become ready as well. The worklist algorithm is shown in Figure 3.1.

If this algorithm is used to attribute a tree T, we make a separate pass over T to initialize the set S of constant (and hence ready) attribute instances. Afterwards, T can be ignored, except to find the attribution equations associated with each attribute instance. If we want to evaluate a dependency graph in some other setting, we need to find a way to initialize the set S, but afterwards, the same algorithm can be used. In other words, the worklist algorithm can be used to evaluate any graph-based problem.

3.1.1 Incremental, Dynamic Evaluators

To obtain an incremental, dynamic evaluator, we use a procedure that is similar to *evaluate*, but it uses a set INFLUENCED, instead of the set S, that is initialized with the attributes at the point of subtree replacement. The key step of the algorithm is saving the old value of an attribute instance before it is reevaluated, and comparing this old value to the new value. Only if the value has changed will successors of the attribute be added to INFLUENCED. This algorithm is shown in Figure 3.2. It is the "naive" evaluator of Reps [Rep84], because selecting b from INFLUENCED arbitrarily can result in some attribute instances being evaluated more than once. In fact, there are some attribute grammars such that if the set of attributes needing evaluation is treated as a queue, then any predefined selection order, such as FIFO or LIFO, could result in some attribute instances being evaluated an exponential number of times relative to the size of AFFECTED [Rep84].

An optimal, incremental, graph-based algorithm is described in Reps's thesis. The algorithm in Figure 3.2 is not optimal, because there is no way to determine when an attribute instance is ready without examining the entire dependency

procedure update(α : node)
/* Node α is the point of subtree replacement. */
INFLUENCED := $A(\alpha)$;
while INFLUENCED $\neq \emptyset$ **do**
Select and remove an attribute instance $b \in$ INFLUENCED.
old_value := b;
Reevaluate b.
if old_value $\neq b$ **then** add all successors of b to INFLUENCED.

Figure 3.2: Incremental, dynamic algorithm

graph, and so selecting an attribute b from INFLUENCED is nontrivial. What is needed is some way to see which attributes depend on which other attributes. In Reps's algorithm, this information is kept in superior and subordinate characteristic graphs.

The algorithm maintains a *model*, a projection of $D(T)$ onto a subset of the vertices of $D(T)$. The initial model is the union of the superior and subordinate characteristic graphs at the change point α. Its vertices are simply the attribute instances in $A(\alpha)$. Evaluation begins by selecting a vertex (i.e., an attribute instance) b in the model with in-degree 0 and deleting edges leaving this vertex. Then b is reevaluated, and its new value is compared to its old value. If the value of b has changed, the model is examined. If all of the successors of b are already in the model, then nothing further needs to be done, and another attribute instance is selected for reevaluation. Otherwise, the model is expanded to include vertices for the attribute instances of the production containing the newly-found inconsistent attribute instances. If the model is expanded upward, in order to include the attributes of the parent or a sibling of α, we remove edges from the model that came from $c^{\alpha}(\tau(\alpha))$ and add the edges of $DP(p)$, where $p = \rho(\beta)$ and β is the parent of α. Then we add $c_{\alpha_i}(X_i)$ to the graph, for all i such that α_i is a sibling of α and $\tau(\alpha_i) = X_i$, and we add $c^{\beta}(X)$, where $X = \tau(\beta)$. A similar operation is needed if expansion is made to a child of α.

Because arbitrarily many characteristic graphs can be affected by a subtree replacement, not all graphs can be updated in $O(|\text{AFFECTED}|)$ time. Instead, superior characteristic graphs are maintained at nodes on the path from α to the root of T, both graphs are maintained at α, and subordinate characteristic graphs are maintained at all other nodes. When this condition is met, the tree is called *prepared-to-propagate.* In order to maintain this condition, the user can only move through the tree a node at a time, and the appropriate characteristic graphs are recomputed as a side-effect of this motion.

Another incremental, dynamic evaluator is the "approximate topological order" (ATO) algorithm of Hoover [Hoo87]. All the attribute instances in an attributed tree are given priority numbers that are supposed to represent a topological order with respect to $D(T)$. The attribute instances that are ready for

```
procedure ATO_evaluate(α : node)
/* Node α is the point of subtree replacement. */
/* W is a priority queue of attribute instances */
    Initialize W with the attribute instances in A(α).
    while W ≠ ∅ do
        Remove the attribute instance b ∈ W with the highest priority.
        old_value := b;
        Reevaluate b.
        if old_value ≠ b then
            for each c a successor of b do
                if priority_number(b) < priority_number(c) then
                    swap_priority_numbers(b, c);
                W := W ∪ {c};
```

Figure 3.3: Approximate topological order algorithm

evaluation are kept in a priority queue, and the attribute instance that is evaluated next is the one with the highest priority. Hoover's algorithm is shown in Figure 3.3.

After a subtree replacement, the priority numbers are not changed, and so they are not necessarily correct. Therefore, at any time, the priority numbers only approximate a true topological order. As propagation proceeds, the priority numbers of attributes added to the priority queue W are checked. Whenever a dependency is found between two attribute instances whose priority numbers are out of order, the numbers are reversed.

Hoover's algorithm is not optimal, because attributes can be evaluated more than once if the priority numbers are not correct. Nevertheless, Hoover claims that in practice, the evaluator is fairly efficient, and only a small number of attributes are evaluated more than once [Hoo87].

3.2 Tree-Walking Evaluators

A tree-walking evaluator works by walking though the derivation tree, evaluating some attributes at each node visited. When an evaluator is "at" a derivation tree node α, it is evaluating output attributes of $\rho(\alpha)$. When an evaluator "moves" to an adjacent node, it modifies its internal state in order to evaluate output attributes of a new production instance. During evaluation, an evaluator alternates between recomputing some attribute instances and visiting an adjacent node.

In full generality, a tree-walking algorithm is nondeterministic and can be used with any attribute grammar. But, in order for a tree-walking evaluator to be efficient, we would like to compute the evaluation order ahead of time, based on a static analysis of the grammar, not of any particular derivation tree. In addition, we would like the algorithm to be deterministic.

procedure *V_evaluate*(**node** α_0);
/* Let $\alpha_1, \ldots, \alpha_m$ be the children of α_0. */
1. Guess a sequence $\ell = \langle \ell_1, \ldots, \ell_v \rangle$, such that $v \geq 0$ and for each $1 \leq i \leq v$, $1 \leq \ell_i \leq m$;
2. **for** $i := 1$ **to** v **do**
3. Compute *some* inherited attributes of α_{ℓ_i}.
4. $V\_evaluate(\alpha_{\ell_i})$;
5. **end**
6. Compute *some* synthesized attributes of α_0.

main program
for $i := 1$ **to** k **do** *V_evaluate*(root of the derivation tree T);
end

Figure 3.4: General, tree-walking attribute evaluator

A general tree-walking evaluator, described by Engelfriet and Filè in [EF81], is shown (slightly modified) in Figure 3.4. This evaluator is really a nondeterministic template for generating evaluators for various attribute grammar classes. The entire tree is evaluated by executing the main program.

The algorithm is nondeterministic for two reasons. First, lines 3 and 6 specify that *some* attributes should be evaluated. Second, line 1 specifies that a sequence of children should be guessed. A tree-walking evaluator may visit children in any order, and they may be visited more than once. Every time a node is visited, the algorithm guesses a new sequence of visits to the node's children. For some grammar classes, we can obtain a deterministic algorithm by always making the same guesses for each instance of a given production.[1]

[1] The way that guesses are made in *V_evaluate* gives rise to eight classes of evaluators. The names of these classes are of the form "x multi-y," where x is chosen from {simple, pure} and y is chosen from {pass, alternating-pass, sweep, visit}.

The distinction between "pure" and "simple" depends on the order in which attributes are evaluated for nodes of a given nonterminal. An evaluator is *simple* if we can assign a visit number to each attribute of all nonterminals such that for any derivation tree, the following property holds: For any instance α of X, if $a \in S(X)$ and i is the visit number assigned to a, then $\alpha.a$ is evaluated during the i-th visit to α. Similarly, if $b \in I(X)$ and j is the visit number assigned to b, then $\alpha.b$ is evaluated just before visiting α for the j-th time. An evaluator without this property is *pure*.

The choice for "y" is based the order in which visits to children are made. If ℓ is always chosen to be a permutation of the sequence $1, \ldots, m$, then the resulting class is "x multi-sweep," and each call to *V_evaluate* from the main program is called a "sweep." Note that a multi-sweep evaluator visits every node in a derivation tree the same number of times. If we always choose $\ell = \langle 1, \ldots, m \rangle$, we obtain an "$x$ multi-pass" evaluator, and if for a given sweep, we always choose $\ell = \langle 1, \ldots, m \rangle$ or $\ell = \langle m, \ldots, 1 \rangle$, we obtain an "$x$ multi-alternating-pass" evaluator. Finally, if the choice of ℓ is unrestricted, including sequences specifying multiple visits to some children and no visits to others, we have an "x multi-visit" evaluator. (For the multi-sweep and multi-visit evaluator classes, we assume that the same choice of ℓ is made during the i-th visit to any instance of a given nonterminal. No extra evaluation power is obtained by relaxing this assumption [EF80].)

For any evaluator class, we can replace "multi-" by "k-" for some integer k to obtain a specific

As written, this algorithm is not suitable for incremental updating. First, because it is written as a recursive procedure, it is hard to initialize the run-time stack at the start of an incremental update. Second, "guessing" is nondeterministic, and is required in three places. To overcome these problems, we use *plans*, instruction sequences that are interpreted by the evaluator.

A second version of the evaluator is a single routine that focuses on a single tree node at a time. As the evaluator walks through the tree, some local variables of the procedure must be saved in the tree nodes, but no run-time stack is needed. This implementation allows incremental computation to be done by restarting evaluation at any node in the tree, as long as we can reinitialize the local variables.

Plans are a sequence of instructions of three types. These instructions are interpreted when the evaluator is at a node α_0 with children $\alpha_1, \ldots \alpha_m$:

- EVAL(i, a): Evaluate attribute instance $\alpha_i.a$. An output attribute must be specified.[2]
- VISIT(i, s), $1 \leq i \leq m$: Move to node α_i and begin interpreting a new plan. The value s may be used in selecting a new plan.
- SUSPEND: Move to the parent of α_0 and resume interpreting the plan there.

Each plan must have one SUSPEND instruction, which must be last. Each production p of the grammar has an associated set of plans that determine how evaluation will be performed at any instance of p in a derivation tree. During evaluation, a sequence of plans available for each production will be interpreted at each production instance. This sequence must specify that each output attribute of the production be evaluated exactly once. In addition, the sequence of plans must specify the appropriate visits to ensure that all attributes instances in the tree are evaluated.

For some attribute grammars, it is possible that an exponential number of plans would need to be associated with some productions. In practice, a small number of plans usually suffices. In fact, for the so-called partitionable attribute grammars, a single sequence of plans for each production is enough to assure proper evaluation. Using plans does not mean that the evaluator is deterministic. In some cases, the proper selection of a plan when visiting a node will depend on descendants of the

evaluator in the class. Such an evaluator can never move to a given node more than k times during evaluation. Hence, a one-pass evaluator must be able to evaluates all attributes in a single left-to-right pass. A simple multi-visit evaluator, which we will discuss later, can visit children of a node in any order, but attributes of a given nonterminal are evaluated in the same order for any instance of the nonterminal.

Finally, we define an attribute grammar G to be "x k-y" if it can be evaluated by an "x k-y" evaluator. All attribute grammars are pure k_0-visit, where k_0 is the maximum number of attributes for any nonterminal [EF80].

[2] We are not concerned with the attribution functions themselves here. We assume that the EVAL instruction performs the appropriate computation.

procedure *P_evaluate*(**node** α_0);
/* Let $p = \rho(\alpha_0)$ and let $\alpha_1, \ldots, \alpha_m$ be the children of α_0. */

```
repeat forever
    case PLAN(α_0)[plan_index(α_0)] of
        EVAL(i, a):
            Evaluate α_i.a.
            plan_index(α_0) + +;
        VISIT(i, s):
            α_0 := α_i;                                        /* Visit child i. */
            PLAN(α_0) := select_new_plan(s, p, PLAN(α_0));
            plan_index(α_0) := 1;
        SUSPEND:
            if α_0 is the root of T then return;     /* Evaluation is done. */
            α_0 := parent(α_0);                      /* Return control to parent. */
            plan_index(α_0) + +;
```

Figure 3.5: Nondeterministic, tree-walking evaluator

node. Therefore, without some extra information or a preprocessing step, it will not always be possible to pick the proper plan deterministically.

By encoding both the visits to children and the attribute evaluations in a plan, we restrict nondeterminism in the algorithm to a single place: selecting a plan. An evaluator using plans is shown in Figure 3.5.Note that no recursive calls are made, so no run-time stack is needed, but each node must be able to save its current plan and its current plan index. Ordinarily, these values are initialized when a node is visited for the first time, so we can start evaluation at the root by giving appropriate values to *PLAN* and *plan_index*. The main program using this evaluator is:

$\alpha := root(T)$;
PLAN(α) := *select_initial_plan*($\rho(\alpha)$);
plan_index(α) := 1;
P_evaluate(α);

The call to *select_initial_plan* returns the proper plan to begin interpreting, based on the production applied at the root of the derivation tree.

In *P_evaluate*, the guessing steps have been replaced by a single step "selecting" a plan when a visit is made. This selection is based on the value s given in the VISIT instruction, the production p applied at the new node being visited, and the plan that was interpreted during the previous visit to the node. The *select_new_plan* routine must be nondeterministic, in general, but we are interested in classes of attribute grammars for which it is always possible to select a plan deterministically. Nevertheless, for all tree-walking evaluators, plans can be computed ahead of time, so tree-walking evaluators are more efficient than dynamic evaluators.

$\cdots$	Attributes outside T'	$I^{j'}_{\alpha_0}$	$I^{j}_{\alpha_i}$	Attributes inside T_i	$S^{j}_{\alpha_i}$	$\cdots$	$S^{j'}_{\alpha_0}$	$\cdots$

Visit to α_i (may be repeated) Other visits

Figure 3.6: A pictorial diagram of evaluation of attributes at a node α_0, with children $\alpha_1, \ldots, \alpha_m$

Two examples of deterministic, tree-walking evaluators are presented by Kennedy and Warren in [KW76] and by Kastens in [Kas80]. Kennedy and Warren describe a tree-walking evaluator that works for "absolutely noncircular" attribute grammars (ANCAGs), a fairly large attribute grammar subclass. Kastens describes an algorithm that works with "partitionable" attribute grammars, a smaller subclass of attribute grammars.[3]

3.2.1 Incremental, Tree-Walking Evaluators

To obtain an incremental algorithm from the evaluator in Figure 3.5, it helps to look at the order in which attribute instances are evaluated for a given attributed tree. For any tree-walking evaluator, execution of the evaluator establishes a total order on evaluations of the attribute instances in the tree, and this total order respects the partial order over attributes determined by $D(T)$. A subsequence of the total order is shown schematically in Figure 3.6 for a visit to a node α_0, the root of a subtree T'. If α_i are the children of α_0, then we let T_i be the subtrees rooted at α_i.

Each block represents the evaluation of some attribute instances. The notation I_α represents the evaluation of a set of inherited attributes of node α. Similarly, S_α represents the evaluation of some synthesized attributes. A superscript, as in $I^{j'}_{\alpha_0}$, distinguishes different sets of attributes of the same nonterminal instance.

In the figure, the boxes with a bar at the bottom represent evaluation of some of the output attributes of production $p = \rho(\alpha_0)$, and the boxes with a bar at the top represent evaluation of input attributes of p. The other boxes do not directly affect attributes of p. Noting this total order leads to a few observations. Suppose we have just made a subtree replacement at node α_0. Then all the attributes of α_0 are in INFLUENCED and must be reevaluated. But no attribute instances to the left (in Figure 3.6) of any inconsistent attribute will need to be reevaluated, because their values cannot be affected by the subtree replacement. Similarly, as soon as the rightmost attribute instance in INFLUENCED has been evaluated, propagation is complete.

[3]The class of partitionable attribute grammars is defined in Chapter 6. An attribute grammar is *absolutely noncircular* if for each nonterminal in the grammar, the union of all possible subordinate characteristic graphs is acyclic.

Furthermore, the inherited attributes of α_0 are the only predecessors of attributes inside T' that are evaluated outside T', so if no inherited attributes of α_0 change, the visit to α_0 does not have to be made. In other words, if the box $\boxed{I^{j'}_{\alpha_0}}$ contains no attributes whose values have changed, the visit to α_0 can be skipped. By skipping visits, we avoid an arbitrary amount of work, because a subtree can be arbitrarily large. To keep track of which subtrees must be visited, we mark a production instance if an input attribute of the production instance has a new value.[4] A visit to an unmarked production instance is skipped.

Based on these observations, we obtain the incremental evaluator shown in Figure 3.7. When we interpret a VISIT(i, s) instruction, we only visit the i-th child if it has been marked. Otherwise, we skip the visit instruction and continue with the next instruction. Similarly, SUSPEND instructions can also be skipped, but in this case, we must also select a new plan to interpret, since SUSPEND instructions are last in the plans.

The algorithm terminates when a SUSPEND instruction at the root of the tree is interpreted, or when we skip the SUSPEND instruction for the final plan of any production instance. SUSPEND instructions ending any final plan for a production are detected easily by marking them. When a SUSPEND instruction is not skipped at a node, the node is unmarked, because it will not be visited again until there is another subtree replacement.

The Kennedy-Warren and the Kastens evaluators both have optimal, incremental versions. In [Rep84], Reps describes how selecting a new plan for the Kennedy-Warren evaluator requires information derived from the productions applied at the nodes along the path from the current node to the root of the derivation tree. It turns out that this information can be represented by a single integer that can be stored as a state variable at each node. Since a subtree replacement can affect many of these state variables, the incremental version of the Kennedy-Warren evaluator must also maintain a prepared-to-propagate condition, which requires that the state variable be correct for each node along the path from the current node to the root of the derivation tree.

The Kastens evaluator, on the other hand, has an incremental version that does not need to maintain a prepared-to-propagate condition. Such an evaluator depends only on the *plan_index* of the VISIT or SUSPEND instruction being interpreted to select a new plan [Yeh83,Rep84]. With this evaluator, the user can move directly from one node to any other node in the derivation tree without being forced to follow edges in the tree.

[4]Actually, we mark nodes in a tree rather than production instances. We mark a production instance by marking its left-hand-side nonterminal instance.

```
procedure Incremental_P_evaluate(node α0);
/* Let p = ρ(α0) and let α1, ..., αm be the children of α0. */
begin
Mark α0 and the attribute instances in A(α0).
plan_index(α0) := 1;
repeat forever
    case PLAN(α0)[plan_index(α0)] of
        EVAL(i, a):
            if αi.a is marked then    /* Only evaluate marked attributes. */
                Unmark and reevaluate αi.a.
                if αi.a has a new value then
                    for each successor β.b of αi.a do
                        Mark attribute instance β.b.
                        Mark defining_node(β.b).
            end
            plan_index(α0) + +;
        VISIT(i, s):
            if αi is marked then
                α0 := αi;                                   /* Visit child i. */
                PLAN(α0) := select_new_plan(s, p, PLAN(α0));
                plan_index(α0) := 1;
            else
                plan_index(α0) + +;                         /* Skip VISIT. */
        SUSPEND:
            if parent(α0) is marked then
                if this is a final SUSPEND then unmark α0.
                α0 := parent(α0);                    /* Return to parent. */
                plan_index(α0) + +;
            else                                       /* Skip SUSPEND. */
                if this is a final SUSPEND then return;
                PLAN(α0) := select_new_plan_suspend(p, PLAN(α0));
                plan_index(α0) := 1;
            end
    end case
end repeat
end
```

Figure 3.7: Incremental version of a tree-walking evaluator

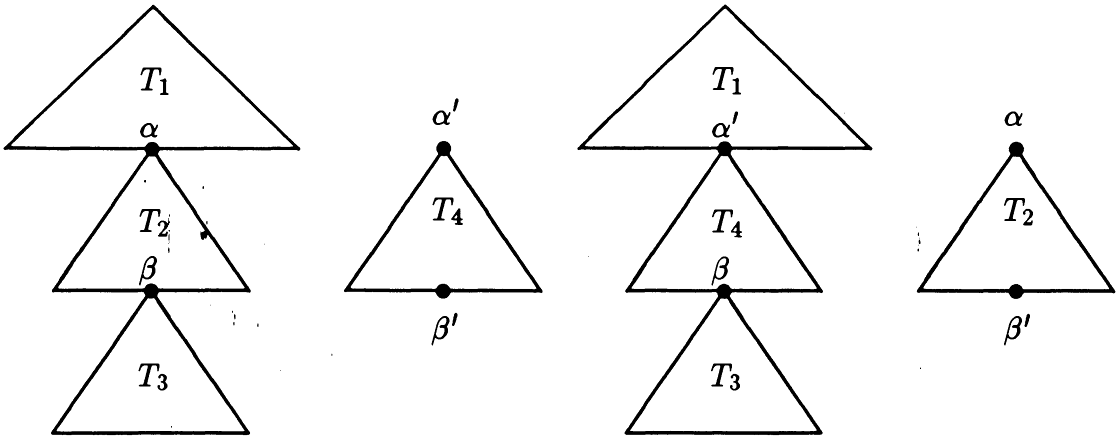
Chapter 4

Multiple Subtree Replacements

The definition of subtree replacement given in Chapter 2 only allows a single change to be made at a time. After a subtree replacement, an evaluator must reattribute the tree before another change can be made. By allowing multiple subtree replacements, we can support a more robust model of editing. Multiple subtree replacements are useful for a number of reasons. First, a user editing a large object may wish to turn off attribution while making changes, invoking the evaluator manually from time to time. Second, a subtree replacement is not always the most natural way to change a tree. Figure 4.1, for example, shows an interior part of a tree being replaced, resulting in two initial inconsistent points: α' and β. Third, some systems allow nonlocal edges between attributes, that is, edges between attributes that are not in the same production instance. The Poe system [JF85], for example, allows an attribute grammar writer to specify nonlocal dependencies and Hoover [Hoo87] describes an algorithm that creates nonlocal edges dynamically during evaluation. In both of these cases, if there is a nonlocal edge between $\alpha_1.a$ and $\alpha_2.b$ and the value of $\alpha_1.a$ changes, propagation can continue as if a new subtree replacement had been made at α_2. Finally, some systems allow multiple users to edit a single object (e.g., [KK86]), and we would like to maintain consistency of a derivation tree while allowing users to make changes to the tree independently.

The last two reasons for allowing multiple subtree replacements require that asynchronous changes be supported. In other words, an evaluator must allow a second subtree replacement to be made before updating has finished for the first change. In this chapter, however, we only consider algorithms to reestablish consistency of a derivation tree after synchronous multiple subtree replacements. In Chapter 7, we will consider asynchronous subtree replacements.

When multiple subtree replacements are made, it is possible that the effects of some changes will cancel out the effects of others. In deciding on the proper evaluation order, an efficient algorithm must take into account all subtree replacements that were made. That is, in order to perform efficient updating, the evaluator must coordinate the evaluation of attributes at all change points.



Initial derivation tree and subtree After replacing T_2 with T_4

Figure 4.1: Replacement of the interior part of a derivation tree

We define AFFECTED after multiple subtree replacements just as we do for single subtree replacements: AFFECTED is the set of attributes that need to be given new values to make the tree consistent. But AFFECTED is not the union of the AFFECTED sets that would exist for each change independent of the others. Figure 4.2 shows a tree containing two subtrees, T_1 and T_2. If we make a subtree replacement at β, then $AFFECTED_\beta$ is the affected set due to this change. Similarly, $AFFECTED_\gamma$ is the affected set that results from a subtree replacement at γ. If both replacements are made at the same time, then in general, $AFFECTED \neq AFFECTED_\beta \cup AFFECTED_\gamma$. For example, suppose there is an attribute $\alpha.a = v$ in the original tree. Replacing subtree T_1 could require that $\alpha.a = v'$ for the tree to be consistent again, but if we replace T_2 at the same time, then v could already be the consistent value for $\alpha.a$ in the resulting tree. So in

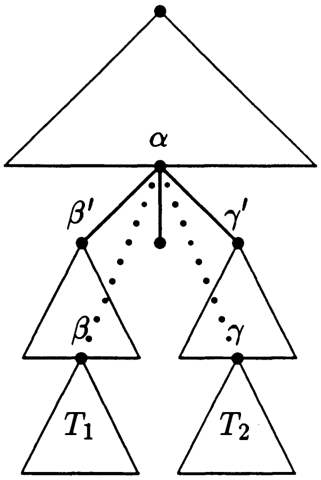


Figure 4.2: A tree T with two subtrees that may be replaced

$$\cdots\boxed{I_\alpha}\cdots\underbrace{\boxed{I^1_{\beta'}}\cdots\boxed{I^1_\beta}\cdots\boxed{S^1_\beta}\cdots\boxed{S^1_{\beta'}}}_{\text{First visit to } \beta'}\cdots\underbrace{\boxed{I_{\gamma'}}\cdots\boxed{I_\gamma}\cdots\boxed{S_\gamma}\cdots\boxed{S_{\gamma'}}}_{\text{Visit to } \gamma'}\cdots\underbrace{\boxed{I^2_{\beta'}}\cdots\boxed{I^2_\beta}\cdots\boxed{S^2_\beta}\cdots\boxed{S^2_{\beta'}}}_{\text{Second visit to } \beta'}\cdots$$

Figure 4.3: Total order of evaluation of attributes in tree of Figure 4.2

this case, $\alpha.a \in AFFECTED_\beta \cup AFFECTED_\gamma$, but $\alpha.a \notin AFFECTED$. Even if $\alpha.a$ needs a new value, we will save work by reevaluating $\alpha.a$ once, not twice, once as a result of replacing T_1 and again as the result of replacing T_2.

If it turns out that $AFFECTED_\beta \cap AFFECTED_\gamma$ is empty, then the subtree replacements are independent, and we could propagate in parallel from both β and γ. But if propagation from both points of change reaches α, the lowest common ancestor (LCA) of β and γ, then an evaluator must consider both changes in determining an efficient evaluation order. If more than two simultaneous changes are made, we have even more of a reason to coordinate evaluation among all the changes.

We will extend the algorithms in Figures 3.5 and 3.7 to produce algorithms that can deal with multiple subtree replacements. These new evaluators will make nonlocal visits, i.e., they will be able to move from a given node to any ancestor or descendant of the node. While such algorithms could be used for non-incremental evaluation with some attribute grammars, it makes much more sense to use them for a multiple subtree replacement evaluator. Therefore, we will not discuss a non-incremental multiple subtree replacement algorithm. But just as the operation of the non-incremental evaluator led us to the derivation of an incremental evaluator for single subtree replacements, we also use the non-incremental evaluator to help find an approach for deriving a multiple subtree replacement algorithm.

In Chapter 3, we saw how the total order imposed by a tree-walking evaluator on attribute evaluations has a pattern that can be used in deriving an incremental algorithm. Figure 4.3 shows in more detail the evaluations that occur in tree T of Figure 4.2 during a visit to α. Note that β' and γ' are children of α, with $\beta' \rightsquigarrow \beta$ and $\gamma' \rightsquigarrow \gamma$. Suppose β has two attributes, a and b, and γ has one attribute, c. If $a \in I^1_\beta$, $b \in I^2_\beta$, and $c \in I_\gamma$, then in a sense, the visits from α to β' and from α to γ' schedule the evaluation of the attribute instances a, b, and c. They must be evaluated in the order a, c, b.

With this observation in mind, we see that we can use the plans of α to coordinate the evaluation of attributes at β and γ, since α is the LCA of β and γ. (A similar idea is found in the graph-based algorithm of Reps, Marceau, and Teitelbaum [RMT86], which uses $DP(\rho(\alpha))$ to construct an initial model.) The new tree-walking evaluator, then, is similar to the evaluator in Figure 3.5, except that instead of visiting β' or γ' when interpreting VISIT instructions, we will visit β and γ by following the "dotted" paths from α.

These "dotted" paths can be arbitrarily long, so by making nonlocal visits, we can save an arbitrary amount of work compared to an evaluator that can only

VISIT(i, s):
    $\alpha_0 := \alpha_i$ or a descendant of α_i;
    PLAN(α_0) := *select_new_plan*(s, p, PLAN(α_0));
    plan_index(α_0) := 1;
SUSPEND:
    if α_0 is the root of T **then return**; /* Evaluation is complete. */
    α_0 := *ancestor*(α_0); /* Return control to an ancestor. */
    plan_index(α_0) + +;

Figure 4.4: New actions for VISIT and SUSPEND instructions

move to adjacent nodes. If it turns out that INFLUENCED does not include attributes of the nodes along the "dotted" paths, then we never visit those nodes. To allow nonlocal visits and suspensions, we extend the meaning of the VISIT and SUSPEND instructions. A nonlocal visit is a move from the current node to a descendant of the current node along a "dotted" path. Similarly, a nonlocal suspension allows moving from the current node to an ancestor. New subcases for these commands are shown in Figure 4.4.

These new actions cannot be implemented without a way of deciding which descendant or ancestor of a node to move to. We must attend to at least two details. First, we must be able to find the "dotted" paths. This is done by using "structure trees." Structure tree operations are described in Chapter 5, but briefly, the structure tree operations maintain a subtree of the derivation tree that contains marked nodes and the LCA of each pair of marked nodes. Henceforth, the "dotted" paths will be called structure tree edges. At all times, the structure tree contains all of the marked nodes, that is, nodes whose productions have been marked because an input attribute of the production has changed. As propagation proceeds, additional nodes can be marked, and so the structure tree changes during propagation.

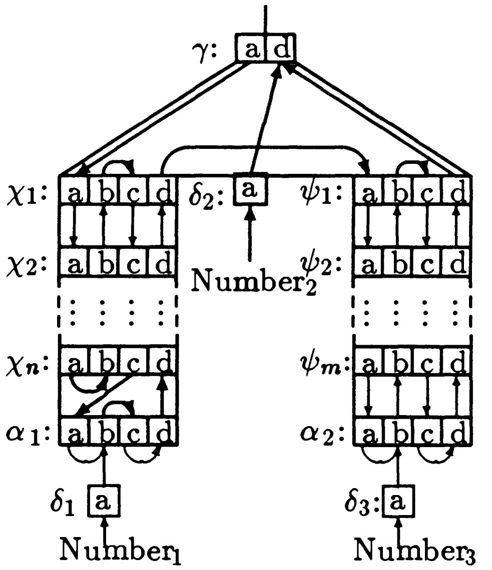
The second difficulty comes from the fact that selecting a plan to interpret when visiting a node may not be straightforward. With the ordinary evaluator, a node is always visited from its parent, and the parameter s in the VISIT instruction can be used to select the proper plan. When we make nonlocal visits, we must be able to select the proper plan by using the parameter of a VISIT instruction of an ancestor. This prevents deterministic multiple subtree algorithms from working with all attribute grammars.

4.1 Propagation After Multiple Subtree Replacements

To see how a multiple subtree replacement algorithm works, we consider an example. The attribute grammar shown in Figure 4.5 has a single basic derivation tree

1: $R \to C$	$C.a = 1$	
2: $C \to XYZ$	$X.a = C.a;$	$X.c = X.b$
	$Z.a = X.d;$	$Z.c = Z.b$
	$C.d = Y.a + Z.d$	
3: $X_1 \to X_2$	$X_2.a = X_1.a;$	$X_1.b = X_2.b$
	$X_2.c = X_1.c;$	$X_1.d = X_2.d$
4: $X \to A$	$A.a = X.c;$	$X.b = X.a$
	$A.c = A.b;$	$X.d = A.d$
5: $Z_1 \to Z_2$	$Z_2.a = Z_1.a;$	$Z_1.b = Z_2.b$
	$Z_2.c = Z_1.c;$	$Z_1.d = Z_2.d$
6: $Z \to A$	$A.a = Z.a;$	$Z.b = A.b$
	$A.c = Z.c;$	$Z.d = A.d$
7: $A \to Y$	$A.b = A.a + Y.a;$	
	$A.d = A.c$	
8: $Y \to$ Number	$Y.a =$ Number	

Figure 4.5: An attribute grammar



$\tau(\chi_i) = X;\ \tau(\delta_i) = Y;\ \tau(\psi_i) = Z;$
$\tau(\alpha_i) = A;\ \tau(\gamma) = C;$

Figure 4.6: A generic derivation tree

shown in Figure 4.6. For simplicity, the attribution functions represent dependencies only. In practice, the functions will be arbitrary, so that we must evaluate a function to see whether its value has changed.

Suppose that the user simultaneously replaces the subtrees rooted at α_1 and α_2. In order to start evaluation, we first mark nodes α_1 and α_2, the original points of change, and mark the attributes of α_1 and α_2 as well. As a result, a structure tree will be created containing α_1 and α_2. The structure tree will also contain γ, the LCA of α_1 and α_2.[1] Propagation begins by starting a plan for $\rho(\gamma)$, the root of the structure tree. A possible plan for the production $C \to XYZ$ is:

EVAL(X, a); VISIT(X, 1); EVAL($X.c$); VISIT(X, 2); EVAL(Z, a); VISIT(Z, 1); EVAL(Z, c); VISIT(Z, 2); VISIT(Y, 1); EVAL(C, d); SUSPEND;

The first instruction, EVAL(X, a), can be skipped, because $\chi_1.a$ is not marked. The second instruction, the first visit to X, can also be skipped, because we can see that the dependency path leaving $\chi_1.a$ does not reach node α_1, and so the subtree replacement at α_1 cannot affect any attributes that would be evaluated by a non-incremental evaluator during the first visit to χ_1. The second eval instruction, EVAL($X.c$), is also skipped, but when we interpret the VISIT(X, 2) instruction, we make a nonlocal visit to α_1, because a non-incremental evaluator would visit

[1] Actually, since attributes of any node α are defined not only in $\rho(\alpha)$ but also in the production applied at the parent of α, we should really include χ_n and ψ_m in the initial structure tree as well. We ignore that detail in this example.

α_1 during the second visit to χ_1. If the value of $\alpha_1.d$ does not change, we will never have to visit nodes $\chi_1, \ldots, \chi_n$. The visits to ψ_1 are handled in a similar fashion, except that both visits must be taken, because a non-incremental evaluator would visit α_2 during each of the visits to ψ_1.

Using the example for inspiration, we explain the actions for EVAL, VISIT, and SUSPEND instructions in more detail. EVAL instructions are interpreted in the usual manner. That is, if the attribute instance to be evaluated is not marked, the evaluation is skipped. Otherwise, the attribute instance, $\alpha.a$, for example, is reevaluated, and its new value is compared to its old value. If the value is unchanged, nothing further needs to be done. Otherwise, each successor of $\alpha.a$ is marked, along with the successor's defining node. So far, this is the same action that is performed by the single subtree replacement algorithm. The new wrinkle here is that the structure tree might have to be modified, because any node marked as a result of a change to an attribute value must be added to the structure tree.

Returning to our example, the original structure tree contained an edge between γ and α_1. Suppose the value of $\alpha_1.d$ changes. Then we mark χ_n, adding it to the structure tree by removing the structure tree edge between γ and α_1, and adding structure tree edges between γ and χ_n and between χ_n and α_1. The important invariant is that the structure tree always contains all marked nodes.

VISIT instructions are a little more complicated, as there are three possible actions for a VISIT(i, s) instruction. A visit to a marked child is always taken, just as with the single subtree replacement algorithm. On the other hand, if we are at node α and the i-th child is not marked, we see if there is a structure tree edge between α and some descendant of the i-th child of α. If such a structure tree edge exists, we may make a nonlocal visit, following the structure tree edge. If neither a local nor a nonlocal visit is made, the VISIT instruction is skipped.

SUSPEND instructions are handled analogously. If the parent of the current node is marked, the evaluator moves to the parent. If there is a structure tree edge to an ancestor of the current node, a nonlocal suspension may be made. Otherwise, the SUSPEND instruction is skipped.

By making nonlocal visits and suspensions, an evaluator is able to skip over the intervening nodes in constant time. Of course, if changed values propagate along the entire path between structure tree nodes, all the intervening nodes will have to be visited eventually, but if the changes have a local effect, propagation will terminate without visiting those intervening nodes.

The key part of a multiple subtree replacement algorithm, then, is deciding whether to move along a nonlocal edge. In the case of a multi-sweep attribute grammar, the decision is easy. Visits along nonlocal edges are always made, because during the visit to a node α by a non-incremental evaluator, every descendant of α will be visited. In general, though, the decision is more complicated. In the example, we skip one of the VISIT instructions from γ to ψ_1, but make a nonlocal visit when interpreting the second VISIT instruction. As an additional complication, after the second VISIT instruction, we begin interpreting the same plan at α_1 that would be interpreted if we were making the first visit directly from χ_n to α_1. Therefore, a modified *select_new_plan* function must be used when making a nonlocal visit.

4.2 A General, Multiple Subtree Replacement Algorithm

We can now give a general algorithm for propagation after multiple subtree replacements. We modify the main loop of the evaluator in Figure 3.5 to obtain a general, multiple subtree replacement evaluator, shown in Figure 4.7.

The Boolean functions *do_visit?* and *do_suspend?* are used to determine when to make nonlocal visits and suspensions. This evaluator is still not completely specified, because we provide no details about the *select_*$\cdots$ and *do_*$\cdots$ functions. In Chapter 6, we define an attribute grammar class for which these functions can be computed statically ahead of time. We also provide a complete algorithm for incremental attribute evaluation after multiple subtree replacements.

```
/* Let p = ρ(α_0) and let α_1, ..., α_m be the children of α_0. */
repeat forever
    case PLAN(α_0)[plan_index(α_0)] of
        EVAL(i, a):
            if α_i.a is marked then          /* Only evaluate marked attributes. */
                Unmark and reevaluate α_i.a.
                if α_i.a has a new value then
                    for each successor β.b of α_i.a do
                        Mark attribute instance β.b.
                        Mark defining_node(β.b).
                            (And add it to the structure tree, if necessary.)
            end
            plan_index(α_0) ++;
        VISIT(i, s):
            if α_i is marked then
                α_0 := α_i;                                      /* Visit child i. */
                PLAN(α_0) := select_new_plan(s, p, PLAN(α_0));
                plan_index(α_0) := 1;
            else if there is a structure tree edge from α_0 to a descendant σ_i of α_i
                and do_visit?(α_i, σ_i, s) then
                α_0 := σ_i;                                 /* Make nonlocal visit. */
                PLAN(α_0) := select_new_nonlocal_plan(s, p, PLAN(α_0));
                plan_index(α_0) := 1;
            else
                plan_index(α_0) ++;                                /* Skip VISIT. */
        SUSPEND:
            if parent(α_0) is marked then
                if this is a final SUSPEND then
                    unmark α_0 (possibly removing it from the structure tree).
                α_0 := parent(α_0);                          /* Return to parent. */
                plan_index(α_0) ++;
            else if there is a structure tree edge from α_0 to an ancestor σ of α_0
                and do_suspend?(α_0, σ) then
                α_0 := σ;                                 /* nonlocal suspension */
                plan_index(α_0) ++;
            else                                                /* Skip SUSPEND. */
                if this is a final SUSPEND then return;
                PLAN(α_0) := select_new_plan_suspend(p, PLAN(α_0));
                plan_index(α_0) := 1;
            end
    end case
end repeat
```

Figure 4.7: Static, nondeterministic, multiple subtree replacement evaluator

Chapter 5

Structure Trees

The multiple subtree replacement evaluators described in this thesis require that we be able to find lowest common ancestors of two or more nodes in a derivation tree quickly. In addition, since derivation trees are being modified continually by users, we must be able to find LCAs of nodes in trees that are being modified by subtree replacements. To be more precise, given a set of nodes in a tree, then whenever a new node α is added to the set, we want to be able to find the closest ancestor of α that is also an ancestor of another node in the set. These requirements are satisfied by the operations of a "structure tree," a projection of a set of marked nodes in a derivation tree. The term "structure tree" was introduced by Hoover [Hoo87], but the origin of the concept is unknown. Hoover's algorithm for maintaining structure trees takes $O(n)$ time per operation, where n is the number of nodes in the underlying tree. The algorithm presented here takes amortized $O(\log n)$ time per operation. It uses self-adjusting binary trees of Sleator and Tarjan [ST85] as an auxiliary data structure in order to achieve this bound.

5.1 Structure Trees

A structure tree is a "structure-preserving projection of a tree onto a vertex subset" of the tree [Hoo87].

Definition 5.1 *Let $T = (V, E)$ be a tree with nodes V and edges E, and let N be a set of nodes $N \subseteq V$. Let $\widehat{N}$ be the smallest subset of V containing N and closed under lowest common ancestor. That is*

- *if $v \in N$ then $v \in \widehat{N}$, and*
- *if $v_1, v_2 \in N$ and v is the lowest common ancestor of v_1 and v_2 in T, then $v \in \widehat{N}$.*

Let $T_N = T \backslash \widehat{N}$. *Then* T_N *is the* structure tree *of* T *with respect to* N. T *is the* underlying tree, N *is the set of* marked nodes, *and* $\widehat{N}$ *is the set of* structure tree nodes. *Edges in* T_N *are called* structure tree edges.[1]

Hoover [Hoo87] shows that this definition indeed defines a tree, and furthermore, each node in $\widehat{N}$ has at most as many children in T_N as the same node has in T. Finally, it is shown that $|\widehat{N}| < 2|N|$, if $N \neq \emptyset$, and for each new node v added to N, at most two new nodes must be added to $\widehat{N}$: v, and if necessary, the lowest common ancestor of v and another node in N.

5.2 Structure Tree Operations

We wish to maintain a forest of trees, for which each tree contains a set of marked nodes represented by a structure tree. The structure tree operations that we need are of two varieties. First, we must be able to create and modify underlying trees containing marked nodes. Second, we must be able to mark and unmark nodes in any underlying tree.

If two trees containing marked nodes are linked, a single underlying tree results, and a new structure tree is defined containing all the marked nodes originally contained in the two trees. Similarly, cutting a tree in two may cause an existing structure tree to be split into two structure trees.

All the structure tree operations must make sure that the structure tree nodes and edges for each underlying tree are correct according to the structure tree definition. Therefore, the structure tree associated with each underlying tree is maintained by each of the following operations:

maketree(v) Return a new tree containing the single unmarked node v.

link(T_1, T_2, w) Link trees T_1 and T_2 by adding an edge between the root of T_2 and node w, which must be in tree T_1. Tree T_1 will be updated, along with its associated structure tree, and tree T_2 will be destroyed.

cut(T, v) Split tree T into two trees by removing the edge between v and its parent. Node v must not be the root of T. Return the new tree whose root is v, along with its structure tree. Tree T and its structure tree will be updated.

mark_node(T, v) Add node v to the set of marked nodes for tree T, which must contain node v. Update the structure tree in T as needed.

unmark_node(T, v) Remove node v from the set of marked nodes for tree T, which must contain node v. Update the structure tree in T as needed.

In addition, for each node in the underlying tree, we will be able to determine whether the node is a structure tree node or is marked.

[1]Hoover adds the root of T to the structure tree whenever N is nonempty. We choose instead to associate with each tree a separate pointer to its structure tree root. Depending on the application, either option may be more appropriate.

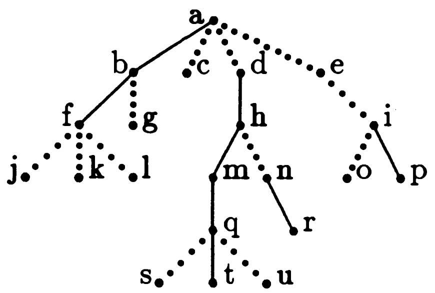


Figure 5.1: A path tree

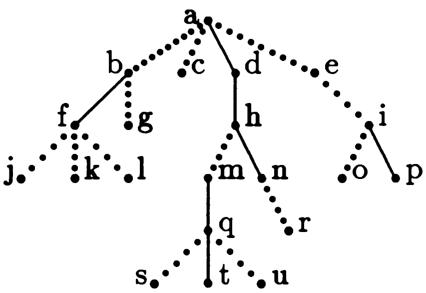


Figure 5.2: Path tree after *expose*(n)

5.3 Representing Trees With Paths

We will represent underlying trees as a collection of paths, the same representation used by Sleator and Tarjan in the implementation of link-cut trees [ST83]. Each path consists of one or more nodes, and each node in the tree is a member of exactly one path. Elements of each path are ordered relative to their depths in the tree. (We assume that edges are directed from nodes to their parents, for the purposes of this representation.) It is helpful to visualize trees using this representation by drawing them with *solid edges* between nodes that are in the same path, and *dotted edges* between the last node of a path and its parent. A *path tree*, that is, a tree represented by paths, is shown in Figure 5.1. The *successor* of the path whose last node is the root of the tree is **null**, so in the figure, the successor of the path containing b is **null**. For every other path, the *successor* is the parent of the last node in the path. E.g., the successor of the path containing p is e. Nodes with no solid edges incident to them (such as e or j) are the only members of one-node paths.

The basic operation used by Sleator and Tarjan to maintain link-cut trees is *expose*, which creates a single solid path starting at a node v and ending at the root of the tree. In the process, solid edges incident to this path are converted to dotted edges. For example, Figure 5.2 shows the tree in Figure 5.1 after we have executed *expose*(n). Although it is possible for a single *expose* to take $O(n)$ time, it will be seen that over a sequence of operations, and with an appropriate implementation for the path operations, all the tree operations can be performed in amortized $O(\log n)$ time.

5.4 Representing Paths With Self-Adjusting Binary Trees

In order to achieve the desired bound on running time, we use *self-adjusting binary trees* [ST85] to represent paths. The symmetric order of nodes in the self-adjusting binary tree is the same as the order of nodes in the path represented by the tree.

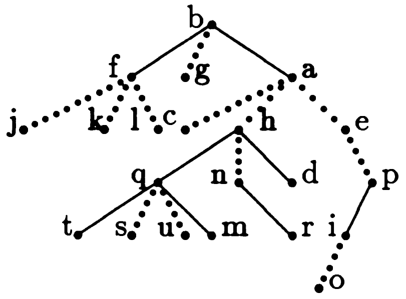


Figure 5.3: A virtual tree representing the path tree shown in Figure 5.1

Each binary tree representing a path is called a *solid tree.* Most of the path operations restructure solid trees by *splaying* [ST85], that is, moving a designated node to the root of the solid tree while preserving the symmetric order. The path operations that we need are:

makepath(v) Create and return a new path containing the single node v.

findpath(v) Return the path containing the node v. (As a side-effect, node v will become the root of its solid tree.)

findlast(p) Return v, the last node of path p. (As a side-effect, node v will become the root of its solid tree.)

join(p, v, q) Create and return a new path containing the nodes of path p, a one-node path v, and path q. The nodes of p will be before v in the new path, and the nodes of q will be after v. Either p or q can be **null**.

split(v) Split the path containing v into three parts: the part before v, a one-node path containing v, and the part after v. Return the "before" and "after" paths as a pair, either of which can be **null**. Node v must be the root of its solid tree.

Tree operations can restructure a solid tree, but at all times, a path is designated by the node at the current root of its solid tree. This means that we can treat a path as a node in the algorithm, but we can only use a node as a path if we are sure that the node is the root of its solid tree.

A *virtual tree* shows the internal, self-adjusting binary tree representation of a path tree. Figure 5.3 shows a virtual tree corresponding to the path tree of Figure 5.1. Because the binary tree representation for a path is not unique, the virtual tree corresponding to a given path tree is not unique either.

5.5 Representing Structure Trees With Path Trees

We use path trees to represent structure trees by letting paths represent structure tree edges in addition to representing the underlying tree. Each structure tree edge is represented by a single path. The paths must satisfy the following *structure tree invariant*, if any structure tree nodes exist:

- There is a solid path from the structure tree root to the underlying tree root.
- All edges entering structure tree nodes are dotted. That is, each structure tree node is the beginning of a path.
- Each structure tree edge from a structure tree node v to its structure tree parent w is represented by a single solid path from v to a child x of w and a dotted edge from x to w. If w has m children in the underlying tree, then we number the structure tree children of w from 1 to m, even though some of them may be **null**. Therefore, if x is the i-th child of w, then v will be the i-th structure tree child of w.

A node v in the underlying tree is *on-path* if it is part of a path representing a structure tree edge, but is not a structure tree node. Paths outside a structure tree are not constrained in any special way, but they are still modified by structure tree operations.

In order to maintain the structure tree invariant, we must modify the *expose* operation. Otherwise, a single path created by *expose* could contain many structure tree nodes, and reestablishing the structure tree invariant could take too much time. Instead, the *expose* operation used here will create a solid path from a node v to a child of the first-encountered structure tree node on the path from v to the root of the underlying tree. (If no structure tree nodes are encountered, then *expose* will create a solid path from v to the root, as usual.)

As structure tree operations are called, we need to be sure that the structure tree invariant is maintained. For example, we must be sure that *mark_node* splits paths when new structure tree edges are added, and that *unmark_node* joins paths whenever structure tree nodes are removed from the interior of a structure tree.

5.6 Data Structures Needed

In the underlying trees, we choose to provide direct pointers to the parent and children of each node. Each node in the underlying tree has the following fields:

node *parent* — The parent of the node (or **null** for the root node).

node *child*$[1 \ldots m]$ — The children of the node.

integer *child_no* — The child number of the node with respect to its parent. Children are numbered from left to right. This field is undefined for the root of the tree.

boolean *is_marked* — True if and only if this node is in the marked set.

boolean *is_structure_tree_node* — True if and only if this node is a structure tree node. Marked nodes are always structure tree nodes.

In addition, structure tree nodes must have additional fields *st_parent*, *st_child*, and *st_child_no*, which are defined just like *parent*, *child*, and *child_no*, respectively. The *st_parent* field is null for the root of the structure tree, and $st\_child[i]$ is null if and only if no descendant of $child[i]$ (including $child[i]$ itself) is a structure tree node. If $st\_child[i]$ is not **null**, then $st\_child\_no[st\_child[i]] = child\_no[child[i]] = i$.

In order to maintain the path representation for underlying trees, we must add two or three fields (depending on the implementation) to each node. Our algorithm uses the parent and child pointers and maintains the extra fields used to keep track of the paths. (We assume that a client program maintains the parent and child pointers for *link* and *cut* operations.)

Finally, so that we can find the roots of structure trees, each *tree* variable in the algorithm will have two fields: *root*, the node that is the root of the underlying tree, and *structure_tree_root*, the root of the current structure tree, if the tree contains any structure tree nodes.

5.7 The Algorithm

The routines in this section are shown as concisely as possible, so we do not show the path operations. They can be found in [Tar83]. First, we introduce two functions that modify a virtual tree without affecting the underlying path tree.

5.7.1 Splice

Splice converts a dotted edge connecting a path p to its successor v into a solid edge. The successor of p must be v, and v must be the root of its structure tree root. If necessary, an existing solid edge entering v is converted into a dotted edge by splitting the original path containing v. Path p may be **null** in which case the returned path will have v as its first node. Otherwise, p is joined to the path containing v. The returned path will still have v as the root of its new solid tree. In an actual implementation, *splice* would probably be expanded in-line, but it is shown separately here for clarity. The code for *splice* is in Figure 5.4.

```
path function splice(node v; path p);            /* successor(p) must be v. */
    path below, above;                           /* v must be a solid tree root. */
    [below, above] := split(v);
    if below ≠ null then successor(below) := v;
    return join(p, v, above);
end splice;
```

Figure 5.4: Splice Operation

5.7.2 Expose

The *expose* function is used by most of the structure tree operations. It creates a solid path beginning at a node v and ending at the root of the underlying tree, unless a structure tree node is encountered. A structure tree node is detected by the condition "*v.is_structure_tree_node*" in the **until** statement. That is, if node v, which is the successor of path p after each iteration of the **repeat** loop, is a structure tree node, then *expose* returns immediately, rather than continuing toward the root. The path returned by *expose* will be represented by the node where the last *splice* was performed. *Expose* does not maintain the structure tree invariant, so routines calling *expose* must make sure that the invariant is restored, if necessary. The code for *expose* is shown in Figure 5.5.

```
path function expose(node v);
    node w;
    path p := null;
    repeat
        w := successor(findpath(v));             /* v becomes a solid tree root. */
        p := splice(v, p);
        v := w;
    until v = null or v.is_structure_tree_node;
    successor(p) := v;
    return p;
end expose;
```

Figure 5.5: Expose Operation

5.7.3 Maketree

Maketree creates a new tree containing a single, unmarked node v. The code for *maketree* is shown in Figure 5.6.

5.7.4 Mark_node

The *mark_node* operation adds a node v to the set of marked nodes, ensuring that the structure tree definition is still satisfied in the resulting tree. It is the most

```
tree function maketree(node v);
    tree t;
    t.root := v;
    t.structure_tree_root := null;
    successor(makepath(v)) := null;
    return t;
end maketree;
```

Figure 5.6: Maketree Operation

complicated of the structure tree operations, because there are so many ways that a new structure tree node can be placed into the existing structure tree. There are three primary cases:

Case 1 Node v is already a structure tree node, although it was not previously marked.

Case 2 No ancestor of v is in the structure tree.

Case 3 Some ancestor of v is in the structure tree.

The code in Figure 5.7 distinguishes the three main cases. As can be seen, Case 1 is trivial, since the structure tree is unchanged.

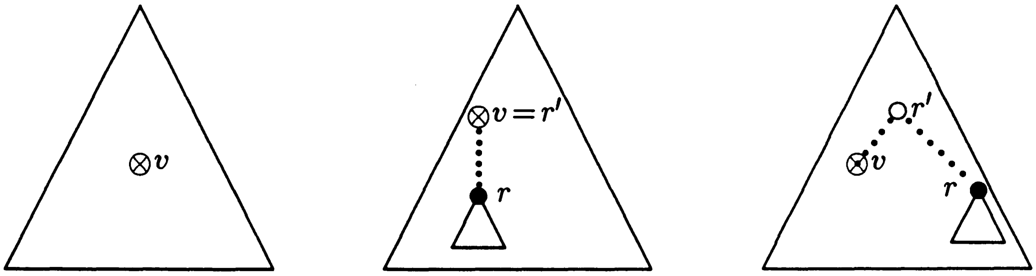
For the other two cases, we check the successor of the path returned by *expose*. If the successor is **null**, then *expose* reached the root, and no ancestor of v is in the structure tree. In this case, we have the three subcases shown in Figure 5.8 and described here:

Case 2.1 Node v is the first node to be added to the structure tree.

Case 2.2 Node v is an ancestor of the existing structure tree root r. Node v will be the new structure tree root, and a single structure tree edge will be added between v and r.

```
procedure mark_node(node v; tree t);
    path p;
    v.is_marked := true;
    if v.is_structure_tree_node then
        return;                            /* Case 1: v is already in structure tree */
    v.is_structure_tree_node := true;
    p := expose(v);
    if successor(p) = null then [Case 2 code]     /* Expose reached the root */
    else [Case 3 code]
end mark_node;
```

Figure 5.7: Skeleton code for *mark_node* operation



Case 2.1: The new node is first in the structure tree.

Case 2.2: The new node is an ancestor of the old root.

Case 2.3: An ancestor of the new node becomes the structure tree root.

Key: ⊗ = the node being marked
○ = a structure tree node added according to the structure tree definition
● = an existing structure tree node.
······ = a new structure tree edge

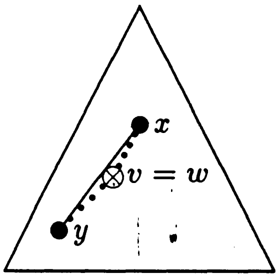
Figure 5.8: Subcases for case 2 of the Mark_node operation

Case 2.3 Node v is not an ancestor of the existing structure tree root r, so r', the lowest common ancestor of v and r, will have to be added to the structure tree, and two new structure tree edges will be added. Because a single solid path existed from r to the root of the underlying tree, then r' is found easily, since it will be the node returned by *expose*.

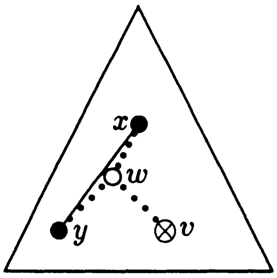
The code for Case 2 is shown in Figure 5.9. In all three subcases, the structure

```
/* Node v is outside structure tree. The structure tree acquires a new root. */
if t.structure_tree_root ≠ null then
    r := t.structure_tree_root;
    r' := p;                  /* r' is the root of the solid tree representing path p. */
    if r' = v then make_st_edge(v,r);                              /* Subcase 2.2 */
    else                                                           /* Subcase 2.3 */
        successor(splice(r', null)) := null;     /* Create path from r' to root. */
        make_st_edge(r', v);
        make_st_edge(r', r);
        r'.is_structure_tree_node := true;
    end
else r' = v;                                                       /* Subcase 2.1 */
t.structure_tree_root := r';                                       /* All subcases */
```

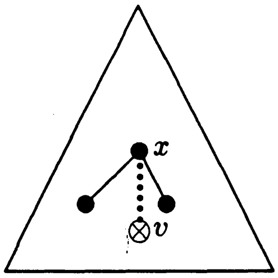
Figure 5.9: Code for case 2 of the Mark_node operation



Case 3.1: The new node is *on-path.*



Case 3.2: An ancestor of the new node is *on-path.*



Case 3.3: An ancestor of the new node is a structure tree node.

Key: $\otimes$ = the node being marked
$\circ$ = a structure tree node added according to the structure tree definition
$\bullet$ = an existing structure tree node.
$\cdots$ = a new structure tree edge
—— = an existing structure tree edge (which might be broken)

Figure 5.10: Subcases for case 3 of the Mark_node operation

tree will acquire a new root. New structure tree edges are made by calling one of two functions: *make_st_edge_m*(p, c, m), which makes c the m-th structure tree child of p, or *make_st_edge*(p, c), which makes c the structure tree child of p, but calls *findlast*(*findpath*(c)) to ascertain the proper child number.

If the successor of the path returned by *expose* is not **null**, then the successor must be an existing structure tree node, and we have the third main case. Again, we have three subcases, which are depicted in Figure 5.10:

Case 3.1 Node v is *on-path.* The existing structure tree edge containing v will be split into two edges.

Case 3.2 Node v has an ancestor w that is *on-path*, and no structure tree or *on-path* nodes occur between v and w. A structure tree edge will be added between v and w, and the existing structure tree edge containing w will be split into two edges.

Case 3.3 Node v has an ancestor x that is a structure tree node, and no structure tree or *on-path* nodes occur between v and x. A new structure tree edge will be added between v and x. No existing structure tree edges are affected.

The code for Case 3 is shown in Figure 5.11.

```
/* successor(p) is a structure tree node, and v is a descendant of it. */
node x := successor(p);
node w := p;            /* Save representative node of path returned by expose. */
node c := findlast(p);
if x.st_child[c.child_no] ≠ null then
    y := x.st_child[c.child_no];
    if w ≠ v then                                              /* Subcase 3.2 */
        w := findpath(w);          /* Make sure w is solid tree (path) root. */
        successor(splice(w, null)) := x;
        make_st_edge(w, v);
        w.is_structure_tree_node := true;
    end
    make_st_edge(w, y);                                 /* Subcases 3.1 & 3.2 */
    make_st_edge_m(x, w, c.child_no);
else
    make_st_edge_m(x, v, c.child_no);                          /* Subcase 3.3 */
```

Figure 5.11: Code for case 3 of the Mark_node operation

5.7.5 Path_compress

The *unmark_node* and *cut* operations will need to remove unmarked structure tree nodes that are no longer necessary according to the structure tree definition. An auxiliary procedure *path_compress* is used to check whether a node v, which must not be a structure tree leaf, should be removed from the structure tree. If v has more than one structure tree child, then it must remain in the tree. Otherwise, v is removed from the structure tree, while maintaining the structure tree invariant. We only need to deal with two cases. First, if v is the former root of the structure tree, then the only child of v will be the new root. Otherwise, v is removed from the structure tree, and a single structure tree edge replaces the two former structure tree edges incident to v. The *path_compress* function is shown in Figure 5.12. The function *only_st_child*(v) returns the structure tree child of v if the node has exactly one structure tree child. Otherwise, it returns **null**.

5.7.6 Unmark_node

The code for *unmark_node* is much simpler than that for *mark_node* because it uses *path_compress*. *Unmark_node* unmarks a node v in a structure tree, possibly removing v and another node from the structure tree in order to satisfy the structure tree definition. Two cases need to be distinguished for *unmark_node*. First, if v is not a leaf in the structure tree, then *path_compress* is called in case v is no longer needed as a structure tree node. Otherwise, the structure tree edge between v and its parent w is removed, and *path_compress* is called with w as its argument. Of course, if v is a leaf node and the structure tree root as well, then

```
procedure path_compress(node v; tree t);
    node child, parent;
    if v.is_marked then return;              /* Cannot remove marked node */
    child := only_st_child(v);
    if child = null then return;                    /* Node v still needed. */
    parent := v.st_parent;
    v.st_parent := null;
    v.st_child[child.st_child_no] := null;
    v.is_structure_tree_node := false;
    if parent ≠ null then make_st_edge_m(parent, child, v.st_child_no);
    else
        t.structure_tree_root := child;
        child.st_parent := null;
    end
    successor(splice(v, findpath(child))) := parent;
end path_compress;
```

Figure 5.12: Path_compress Operation

v was the only node in the structure tree, and the structure tree becomes empty. Code for *unmark_node* is in Figure 5.13. The *findpath* calls are needed, because the first argument to *path_compress* must be a solid tree root.

```
procedure unmark_node(node v; tree t);
    node w := v.st_parent;
    v.is_marked := false;
    v := findpath(v);                    /* v becomes a structure tree root. */
    if v has no structure tree child then      /* Removing structure tree leaf */
        if w = null then t.structure_tree_root := null;
        else
            w.st_child[v.st_child_no] := null;
            v.is_structure_tree_node := false;
            v.st_parent := null;
            path_compress(findpath(w), t);
        end
    else path_compress(v, t);
end unmark_node;
```

Figure 5.13: Unmark_node Operation

5.7.7 Link

The *link* operation links trees $t1$ and $t2$ by adding an edge from the root of $t2$ to node w, which must be in $t1$. To simplify *link*, we mark node w temporarily if it is not already a structure tree node. Once this has been done, the structure trees in $t1$ and $t2$ can be linked easily. Afterwards, if we marked node w, then we unmark the node, removing w from the structure tree if it is not needed. The code for the link operation is in Figure 5.14.

```
procedure link(tree t1, t2; node w);
    node v := t2.root;
    boolean remove := not w.is_structure_tree_node;
    if remove then mark_node(w, t1);
    successor(findpath(v)) := w;
    if t2.structure_tree_root ≠ null then
        make_st_edge_m(w, t2.structure_tree_root, v.child_no);
    if remove then unmark_node(w, t1);
end link;
```

Figure 5.14: Link Operation

5.7.8 Cut

The *cut* function removes a subtree rooted at a node v from an existing tree t by removing the edge from v to its parent. The removed subtree is returned. By marking node w, the parent of v, we can easily determine whether the subtree being cut contains any structure tree nodes. Afterwards, w is unmarked, if necessary. The code for *cut* is in Figure 5.15.

```
tree function cut(tree t; node v)
    node w := v.parent;                          /* Node v must have a parent. */
    tree t';
    boolean remove := not w.is_structure_tree_node;
    if remove then mark_node(w, t);
    t'.root := v;
    successor(findpath(v)) := null;
    t'.structure_tree_root := w.st_child[v.child_no];              /* may be null */
    if w.st_child[v.child_no] ≠ null then
        w.st_child[v.child_no].st_parent := null;
        w.st_child[v.child_no] := null;
    if remove then unmark_node(w, t) else path_compress(findpath(w), t);
    return t';
end cut;
```

Figure 5.15: Cut Operation

5.8 Analysis of Running Time

In analyzing the running time of these routines, much of the analysis in [Tar83] can be used. The main difference has to do with the *expose* operation. Because *expose* as described here does not always create a solid path all the way to the root, the analysis must be modified. First, we need a few definitions regarding nodes in the underlying tree.

Definition 5.2 *The* size *of a node v is the number of descendants of v in the underlying tree, including v itself.*

Definition 5.3 *An edge from v to its parent w is* heavy *if* $2 \cdot \mathrm{size}(v) > \mathrm{size}(w)$ *and* light *otherwise.*

Lemma 5.1 *If v is any node, there is at most one heavy edge entering v.*

Lemma 5.2 *There are at most $\lfloor \log n \rfloor$ light edges on the tree path from v to the tree root.*

Proof: The proof of Lemma 5.1 is obvious. For Lemma 5.2, we call a node *light* if the edge to its parent is light. If the size of a light node is x, then the size of its parent must be at least $2x$. If there are more than $\lfloor \log n \rfloor$ light edges on the tree path from v to the tree root, then the weight of the root must be at least $2^{\lfloor \log n \rfloor + 1} > n$, a contradiction. □

These lemmas allow us to prove the following theorem (*cf.* [Tar83]).

Theorem 5.1 *A sequence of m structure tree operations including n maketree operations requires $O(m)$ path operations and in addition, at most m calls to* expose. *The* expose *calls require $O(m \log n)$ splices, each of which requires $O(1)$ time.*

Proof: The *split* and *join* calls in *splice* are constant time operations, since no splaying is needed. Therefore, each *splice* takes constant time. Each structure tree operation requires a constant number of path operations, so m structure tree operations require $O(m)$ path operations. *Mark_node* can call *expose*, as can *link* and *cut* since they can call *mark_node*. *Unmark_node* and *maketree* do not call *expose* at all. So overall, m structure tree operations require at most $m - n < m$ *expose* calls.

Next, we prove that the m *expose* calls require $O(m \log n)$ *splice* calls. By Lemma 5.2, for any given *expose*, each of at most $\lfloor \log n \rfloor$ splices will turn a light, dotted edge into a light, solid edge. Each of the rest of the splices will convert a heavy, dotted edge into a heavy, solid edge, increasing the number of heavy, solid edges by one, since any edge converted to dotted must be light.

We bound the number of heavy, solid edges created by noting that after the m operations, the tree has at most $n-1$ edges. Therefore, the number of heavy, solid edges created by the m *expose* calls is at most $n - 1$ plus the number of heavy, solid edges destroyed by the operations.

Each call to *expose* destroys at most $\lfloor \log n \rfloor + 1$ heavy, solid edges, possibly one for the first *splice* call, and at most one for each light, solid edge created. In addition to the heavy, solid edges destroyed by *expose*, the other operations can destroy additional heavy, solid edges as follows:

maketree, unmark_node No edges are created or affected by *maketree*, and no solid edges are destroyed by *unmark_node*.

link *Link* (making a node w the parent of a node v) increases the size of all nodes on the path from w to the root of the underlying tree, possibly making those edges heavy, and making edges incident to the path light. At most $\lfloor \log n \rfloor$ edges incident to the path can be heavy, so *link* can destroy at most $\lfloor \log n \rfloor$ additional heavy, solid edges. But these heavy, solid edges would have been destroyed anyway if *expose* had reached the root, so at most $\lfloor \log n \rfloor$ solid, heavy edges are destroyed in total by the *expose* and by the addition of the edge from v to w.

cut *Cut* can convert edges on the path from the cut node to the root from heavy to light, but at most $\lfloor \log n \rfloor$ of them can be made light. In addition, the cut edge might have been solid and heavy, so *cut* can destroy as many as $\lfloor \log n \rfloor + 1$ heavy, solid edges.

mark_node *Mark_node* can call *splice*, splitting the path returned by its call to *expose*, so at most one heavy, solid edge can be destroyed.

When we combine these bounds, we see that at most $O(m\lfloor \log n \rfloor)$ solid, heavy edges are destroyed by the *expose* calls, and $O(m\lfloor \log n \rfloor)$ additional solid, heavy edges are destroyed by the other operations. So $O(n - 1 + m\lfloor \log n \rfloor)$ solid, heavy edges are created by the exposes. Adding the $O(m\lfloor \log n \rfloor)$ splices that do create solid, light edges, our theorem is proved. □

To prove an overall amortized bound on the operations, we use a modified definition of *size*:

$$\overline{size}(v) = \begin{cases} 1 & \text{if } v \text{ is a structure tree node,} \\ 1 + \sum_i \overline{size}(i), \text{for all children } i \text{ of } v & \text{otherwise.} \end{cases}$$

In other words, $\overline{size}(v)$ treats the underlying tree as if it were partitioned into a forest of trees by removing every edge between a structure tree node and its children. Then we use an accounting scheme that depends on the paths used to represent the tree. Each node in a path tree has an *individual weight*

$$iw(v) = \begin{cases} \overline{size}(v) - \overline{size}(u) & \text{if } [u, v] \text{ is a solid edge,} \\ \overline{size}(v) & \text{otherwise.} \end{cases}$$

When a node v becomes a structure tree node, the $\overline{size}$ of nodes above v can be affected, but except for v, no individual weights are affected. The *total weight*

$tw(v)$ of a node v is the sum of the individual weights of the descendants of v in the solid tree containing v. For any node v, $tw(v)$ is the number of descendants of v in the virtual tree containing it, not including any proper descendants of a structure tree node w that is a descendant of v. Finally, let $rank(v) = \lfloor \log tw(v) \rfloor$. We wish to maintain $rank(v)$ credits on each node at all times. Each structure tree operation will be assessed an amortized cost of $O(\log n)$ credits. Credits not used to pay for the actual operation will be used to maintain the invariant. Extra operations can be accounted for by credits released during structure tree operations.

We will prove that if we use self-adjusting binary trees, we obtain an $O(m \log n)$ time bound on a sequence of m structure tree operations. We state the following lemma from Tarjan without proof [Tar83].

Lemma 5.3 *Splaying a solid tree with root v at a node x while maintaining the credit invariant takes* $3(rank(v) - rank(x)) + 1$ *credits.*

Since $1 \leq tw(v) \leq n$ for all nodes v, then $0 \leq rank(v) \leq \log n$ for all nodes, and so a single solid path operation takes $O(\log n)$ credits.

Examining the *expose* operation, we see that the first *findpath*(v) operation takes $3(rank(u) - rank(v)) + 1$ credits where u is the root of the solid tree containing v. After the *findpath* call, v will be the root of its solid tree. The *split* and *join* operations in *splice* do not affect any total weights, so one additional credit pays for the rest of the splice. The successor of v, which is the next node that will be splayed, will have rank at least as great as that of u, unless it is a structure tree node, in which case *expose* returns.

Summing over all splices during an *expose*, we notice that the sum telescopes, and so the total number of credits needed is $3(rank(w) - rank(v))$ plus two per splice, where v is the node exposed and w is the root of the last solid tree reached by the *expose*. By Theorem 5.1, we conclude that the number of credits needed for all $O(m)$ *expose*s required by m tree operations is $O(m \log n)$.

Total weights can be affected by other operations, as follows:

maketree Node v, the only node in the new tree, will have $rank(v) = 0$.

mark_node This operation can only reduce total weights, since the individual weight of the marked node is reduced from its previous value to one.

link *Link* adds an edge from a subtree root v to an existing node w. Because w is marked first, $iw(w) = 1$ before and after the edge $[v, w]$ is added. Therefore, *link* does not affect any total weights or ranks, other than those affected by *mark_node* and *unmark_node*.

cut *Cut*, like *link*, marks w, the parent of v. Therefore, removing the subtree rooted at v doesn't affect any total weights or ranks.

path_compress *Path_compress*, which can unmark a node v, calls *findpath* on any node being unmarked, making it a solid tree root. After v is removed from the structure tree, its individual weight increases. But the only total weight affected is v's, because it is the root of its solid tree. Therefore, by adding $O(\log n)$ credits to v, we restore the credit invariant.

unmark_node *Unmark_node*, which can be called by *link* and *cut*, can modify the individual weight of a node, by removing it from the structure tree. If the node being removed is a leaf in the structure tree, then the *findpath*(v) call makes v the root of its solid tree. As with *path_compress*, an extra $O(\log n)$ credits takes care of any violation of the credit invariant. If we need to remove other nodes from the structure tree, we call *path_compress* to do the work, so an additional $O(\log n)$ credits might be needed to maintain the invariant.

Taking all these operations into account, we have the following theorem:

Theorem 5.2 *If we use self-adjusting binary trees to represent solid paths, a sequence of m structure tree operations including n maketree operations requires $O(m \log n)$ time.*

The algorithm presented here is an improvement over currently published methods for maintaining structure trees. While some additional storage overhead is incurred to maintain paths as self-adjusting binary trees, it is modest, and the improvement in execution time should outweigh the storage penalty. In addition, this algorithm allows efficient operation of multiple subtree replacement algorithms as described in this thesis.

Chapter 6

Globally Partitionable Attribute Grammars

In Chapters 3 and 4, we saw examples of incremental evaluators for single and multiple subtree replacements that were incomplete because they depended on unspecified functions such as *do_visit?* and *select_new_plan*. In this chapter, we present a new incremental algorithm for updating after multiple subtree replacements. Because this new algorithm is based on Kastens's algorithm, we describe Kastens's algorithm in detail in the first part of the chapter, as well as precisely defining the class of partitionable attribute grammars.

An incremental version of Kastens's tree-walking evaluator [Kas80], which can be used with "partitionable attribute grammars" (PAG), is described first.[1] Kastens defined his evaluator in terms of "ordered attribute grammars," a subclass of PAGs for which membership in the class can be determined in polynomial time. Nevertheless, his evaluator works for the larger class of PAGs.

The primary algorithm in this chapter, *GPAG-evaluate*, supports multiple subtree replacements and provides for incremental updating after such changes. The attribute grammar class associated with *GPAG-evaluate* is a new class called "globally partitionable attribute grammars" (GPAG), a subclass of PAGs. We develop *GPAG-evaluate* and the definition for GPAGs as an extension of the evaluator and definition for PAGs.

Each algorithm described in this chapter has an associated class of attribute grammars that it can be used with, and both grammar classes have the following characterization:

- Determining whether an attribute grammar is PAG (GPAG) is an NP-complete problem.
- There is a polynomial-time algorithm for finding a potentially "valid" partitioning (global partitioning).

[1] The term *partitionable* was introduced by Waite and Goos [WG84], but the class had been examined previously by Kastens [Kas80] and by Engelfriet and Filè [EF80]. Engelfriet and Filè call the class *simple multi-visit*.

- Given a partitioning (global partitioning) for an attribute grammar, there is a polynomial-time algorithm to determine whether the partitioning (global partitioning) is "valid."

In order to talk about these grammar classes in more detail, we will need to be more precise when talking about evaluation orders than we have been until now. So we begin this chapter by presenting the definitions of computation sequences and visit traces used by Engelfriet and Filè [EF80].

6.1 Computation Sequences

As we have seen before, the execution of a tree-walking evaluator establishes an order of evaluation on the attribute instances of a derivation tree. This evaluation order is not arbitrary, but must satisfy certain properties. Let T be a derivation tree. A *computation sequence* [EF80] is a list of sets of attribute instances of T in the order in which they are evaluated by a tree-walking evaluator, and reflects both the order of evaluation of the attribute instances and also the order in which the nodes of T are visited. Each set of attribute instances is represented by a basic-action symbol (*ba-symbol*). A ba-symbol is a pair (α, A) where α is a tree node and $A \subseteq I(\tau(\alpha))$ or $A \subseteq S(\tau(\alpha))$, and it indicates the evaluation of the instances of the attributes in A at node α. We also write (α, IA) or (α, SA) when a ba-symbol refers to inherited or synthesized attributes, respectively. A computation sequence $s(T)$ for T contains ba-symbols representing evaluation of all the attribute instances in the tree.

A ba-symbol is a set of attribute instances, that is, $(\alpha, A) = \{\alpha.a \mid a \in A\}$. Some of the ba-symbol sets may be empty, but they may not be omitted from the computation sequence because they are used to indicate the order in which nodes are visited in the tree. Because a tree-walking evaluator always moves to adjacent nodes in the derivation tree and must respect dependencies between attribute instances, a computation sequence $s(T)$ for a derivation tree T must meet the following four conditions.

1. **Start-end condition.** The first symbol must be $(\alpha, \emptyset)$ where α is the root of T, and $\emptyset$ is specified because the start nonterminal cannot have inherited attributes. The last symbol must be (α, SA), where $SA \subseteq S(\tau(\alpha))$.

2. **Sequentiality condition.** When α_1 and α_2 appear in consecutive ba-symbols, α_2 must be equal to α_1, or it must be the parent, a child, or a sibling of α_1. If $A_1 = \tau(\alpha_1)$ and $A_2 = \tau(\alpha_2)$, then consecutive ba-symbols must occur as follows:

   (a) If $\alpha_2 = \alpha_1$ then (α_1, IA_1) is followed by (α_2, SA_1), or (α_1, SA_1) is followed by (α_2, IA_1).

   (b) If α_2 is the parent of α_1, then (α_1, SA_1) is followed by (α_2, SA_2).

(c) If α_2 is a child of α_1, then (α_1, IA_1) is followed by (α_2, IA_2).

(d) If α_2 and α_1 are siblings, then (α_1, SA_1) is followed by (α_2, IA_2).

3. **Feasibility condition.** The order must be an evaluation order. In other words, an attribute instance in (α_i, A_i) cannot depend on another attribute instance in (α_j, A_j), if $j > i$.

4. **Completeness condition.** Each attribute instance in T must be a member of exactly one ba-symbol.

A computation sequence for a derivation tree is not necessarily unique, partly because children of a node can be visited in any order whenever there are no dependencies between attribute instances of the children. Furthermore, the sequentiality condition is more strict than necessary, but is based on the observation that the evaluation of some inherited attributes of a node, for example, can be postponed until just before the node is visited.

6.2 Visits and Visit Traces

Referring to Figure 3.5, we see how a tree-walking algorithm evaluates the attribute instances of a derivation tree T. Suppose T' is a subtree of T and α is the root of T'. At some time during evaluation, some inherited attributes of α are evaluated, and T' is visited, whereupon attributes of nodes inside T' are evaluated. Finally, some synthesized attributes of α are evaluated and the visit terminates.

Thus, each visit to a subtree made by the evaluator has an associated *visit trace*, which is a subsequence of the computation sequence for the entire tree. If $s(T)$ is a computation sequence for a derivation tree T, then the visit trace for the i-th visit to a subtree T' of T must have the form

$$v_i(T') = (\alpha, IA_i), \ldots, (\alpha, SA_i)$$

where the ellipsis denotes ba-symbols having descendants of α as their first component.

A *visit-trace tuple* $v(T')$ for a subtree T' is a string

$$v(T') = \#v_1(T')\#v_2(T')\cdots\#v_k(T')\#,$$

where each $v_i(T')$ is a visit trace for T'. (The # symbols could be omitted, but they serve as placeholders for ba-symbols containing attribute instances outside T'. We ignore # symbols when determining whether a string of ba-symbols meets the conditions for a computation sequence.)

6.3 Partitionable Attribute Grammars

We now begin a brief foray into the subject of partitionable attribute grammars, since the concepts used in defining PAGs are also used to define GPAGs. In addition, the evaluator for incremental updating that works with PAGs serves as a basis for *GPAG-evaluate*, the evaluator for incremental updating after multiple subtree replacements that works with GPAGs.

6.3.1 Partitions

Let G be an attribute grammar and let X be a nonterminal of G. A *partition* for X, denoted $\Pi(X)$, is a division of all the attributes of X into a sequence of mutually disjoint (potentially empty) subsets,

$$\Pi(X) = \langle A_1(X), \ldots, A_k(X) \rangle.$$

That is, $A(X) = \bigcup_{i=1}^{k} A_i(X)$ and $A_i(X) \bigcap A_j(X) = \emptyset$, if $i \neq j$. The value k depends on X; when we need to make this explicit, we write k_X. Each set $A_i(X)$ is the union of a set of inherited attributes, $I_i(X)$, and a set of synthesized attributes, $S_i(X)$, and so usually we write $\Pi(X)$ as

$$\Pi(X) = \langle I_1(X), S_1(X), \ldots, I_k(X), S_k(X) \rangle;$$

or, relabeling the elements, as

$$\Pi(X) = \langle p_1(X), \ldots, p_{2k}(X) \rangle.$$

We write $\Pi(G)$ or just Π for a complete set of partitions

$$\{\Pi(X) \mid X \text{ a nonterminal of } G\},$$

and we call Π a *partitioning* for G.

Although we can specify a partitioning for any attribute grammar, we are interested in partitions whose order can be used by a tree-walking evaluator. The significance of partition orders is captured by the following definitions:

Definition 6.1 *Let G be an attribute grammar, X a nonterminal of G, $\Pi(X)$ a partition for X, and T a complete derivation tree for G. A computation sequence $s(T)$ for T* respects $\Pi(X)$ *if $\forall \alpha \in T$ such that $\tau(\alpha) = X$, the ba-symbol $(\alpha, p_i(X))$ occurs before the ba-symbol $(\alpha, p_j(X))$ in $s(T)$, for all $1 \leq i < j \leq 2k_X$.*

Definition 6.2 *Let G be an attribute grammar, $\Pi(G)$ a complete set of partitions for G, and T a complete derivation tree for G. A computation sequence $s(T)$* respects Π *if for all nonterminals X of G, $s(T)$ respects $\Pi(X)$.*

Ideally, for a given attribute grammar, we would like to be able to generate plans that when interpreted by a tree-walking evaluator, always result in a computation sequence that respects Π for any derivation tree. We will be able to do this if we have a "valid" set of partitions.

Definition 6.3 *Let G be an attribute grammar and $\Pi(G)$ a complete set of partitions for G. Π is* valid, *if for all derivation trees T of G, there exists a computation sequence, $s(T)$, respecting Π.*[2]

Finally, we can define partitionable attribute grammars:

Definition 6.4 *Let G be an attribute grammar. G is* partitionable *(or PAG) if there exists a set of partitions, $\Pi(G)$, such that Π is valid.*

By virtue of these definitions and the definition of a computation sequence, we can see that if a computation sequence respects $\Pi(G)$ for an attribute grammar G, then for any subtree T' of a derivation tree, with α the root of T' and $X = \tau(\alpha)$, $s(T)$ contains k_X visit traces to T', and the i-th visit trace begins with $(\alpha, I_i(X))$ and ends with $(\alpha, S_i(X))$, where $I_i(X)$ and $S_i(X)$ are elements of $\Pi(X)$.

6.3.2 An Incremental Evaluator for Partitionable Attribute Grammars

If an attribute grammar G is partitionable, then there is a tree-walking evaluator that always visits each instance of a given nonterminal the same number of times, for any derivation tree of G. Furthermore, for each production p, the plans associated with p are always interpreted in the same order. Therefore, a simple version of the general tree-walking evaluator can be used. Kastens [Kas80] describes such an algorithm. His algorithm associates a single plan, rather than a set of plans, with each production p, but the plans can contain embedded SUSPEND instructions. In addition, because of the way that plans of neighboring productions interact, we no longer need the *select_new_plan* function, nor do we need to store the *plan_index* in a tree node. Instead, a single, global *plan_index* variable is used, and its proper value is computed after each VISIT or SUSPEND instruction by using new mapping functions *mapup* and *mapdown*.

Each of the mapping functions can be implemented as a simple table lookup. If we are at a node α_0 and we need to visit α_i, the i-th child of α_0, for the s-th time, the corresponding VISIT instruction will be VISIT(i, s). By looking up the value of $mapdown(\rho(\alpha_i), s)$, we find the plan index for starting (or resuming) interpretation of the plan at α_i. Similarly, if we are interpreting the s-th SUSPEND instruction at α_i, we use $mapup(\rho(\alpha_0), s, i)$ to determine where to resume interpretation of

[2] Engelfriet and Filè use the term *simple multi-visit* not only for the grammar class, but also where we use *valid* to describe a set of partitions.

the plan at α_0. We obtain the value s by making it a parameter of the SUSPEND instruction.

The *mapup* and *mapdown* functions are defined to ensure that the plans of neighboring productions "fit" together properly, so that every instruction in the plan for each node is interpreted exactly once. For example, for any production p, $mapdown(p, 1) = 1$, because we will always start interpreting a plan at the beginning when we visit a node for the first time. Also, if $\text{PLAN}(p)[i] = \text{VISIT}(j, s)$, then $mapup(p, s, j) = i + 1$.

Kastens's evaluator is a tree-walking evaluator that interprets these plans. His evaluator can be converted easily to an incremental algorithm [Yeh83,Rep84, RT89a], and an incremental version of the Kastens evaluator is shown in Figure 6.1. After a subtree replacement at α, we call *Incremental_K_evaluate*(α) to reestablish consistency to the derivation tree.

One part of this evaluator handles a problem that we have mentioned previously only in passing. For any node α in a derivation tree, the defining node for the synthesized attributes of α is α, but the defining node for the inherited attributes of α is the parent of α. If a subtree replacement is performed at α, then in order to make sure that all attributes of α are reevaluated, we must mark both α and its parent (unless α is the root) before starting the evaluator. This marking is performed in lines 2–4 of the algorithm.

The code for the incremental algorithm includes the code for Kastens's nonincremental evaluator, which can be obtained by omitting the lines marked with "†" from the algorithm in Figure 6.1. To evaluate the attributes of a tree from scratch, the nonincremental algorithm is called with the root of the derivation tree as its argument.

6.3.3 Efficiency of the Incremental Algorithm

Incremental_K_evaluate is optimal, because when it is used to restore consistency to a derivation tree after a single subtree replacement, the plan for a production instance is interpreted only if an input attribute of the production instance changed. For a given attribute grammar, the size of a plan is proportional to the maximum number of attributes for any nonterminal in the grammar. Therefore, for each attribute in INFLUENCED, a constant number of instructions is interpreted by the evaluator, and hence, a constant amount of overhead is needed.

6.3.4 Finding a Valid Partitioning for an Attribute Grammar

Kastens's evaluator and its incremental counterpart are useful only if we can find a valid set of partitions for an attribute grammar. Therefore, if we are given an attribute grammar G, two obvious questions come to mind: Is G partitionable? If G is partitionable, can we find a valid partitioning for G? Unfortunately, these

```
  procedure Incremental_K_evaluate(node α_0);
  /* Let p = ρ(α_0) and let α_1, ..., α_m be the children of α_0. */
  /* Each production p in the grammar has a plan PLAN(p). */
  begin
†   Mark α_0 and the attribute instances in A(α_0).
†   if α_0 is not the root of the tree then
†       α_0 := parent(α_0);
†       Mark α_0;
    plan_index := 1;
    repeat forever
        case PLAN(p)[plan_index] of
            EVAL(i, a):
†               if α_i.a is marked then    /* Only evaluate marked attributes.*/
                    Reevaluate α_i.a.
†                   if α_i.a has a new value then
†                       for each successor β.b of α_i.a do
†                           Mark β.b.
†                           Mark defining_node(β.b).
†                   Unmark α_i.a.
                plan_index++;
            VISIT(i, s):
†               if α_i is marked then
                    α_0 := α_i;                                /* Visit child i. */
                    plan_index := mapdown(p, s);
†               else
†                   plan_index++;                              /* Skip VISIT. */
            SUSPEND(s):
†               if this is the final SUSPEND in PLAN(p) then unmark α_0.
                if α_0 is the root of the tree then return;
†               if parent(α_0) is marked then
                    j := child_no(α_0);
                    α_0 := parent(α_0);                  /* Return to parent. */
                    plan_index := mapup(p, s, j);
†               else                                       /* Skip SUSPEND. */
†                   if this is the final SUSPEND in PLAN(p) then return;
†                   plan_index++;
                end
        end case
    end repeat
  end
```

†If these statements are removed, the resulting procedure is Kastens's (non-incremental) algorithm.

Figure 6.1: Incremental version of Kastens's tree-walking evaluator

questions are NP-complete problems [EF80] and there is no known practical algorithm for answering them. Instead, Kastens uses a polynomial-time algorithm that generates a partitioning and checks to see whether it is valid. An attribute grammar for which Kastens's algorithm succeeds is an *ordered attribute grammar.* If the algorithm fails, the grammar is not ordered, but in most cases, we cannot infer whether the grammar is partitionable. Ironically, it is usually easier to test whether a grammar is noncircular, an inherently exponential problem [JOR75], than it is to determine whether it is partitionable. This is because the test for noncircularity decomposes into a number of small problems for most attribute grammars [Bel88]. No such simplification is known to exist for determining whether an attribute grammar is partitionable.

6.3.5 Ordered Attribute Grammars

Kastens's algorithm for determining whether an attribute grammar is ordered consists of three steps, which are sketched here. Appendix A contains code for the algorithm and some additional explanation. For more details, [Kas80] can be consulted.

Step 1: Compute $DS^+(X)$ graphs

For each nonterminal X, we construct an induced DS graph (written as $DS^+(X)$), an instance of a $DS(X)$ graph. The graph contains an edge between two attributes a and b of X if for some derivation tree T and some instance α of X in T, there is a path from $\alpha.a$ to $\alpha.b$ in $D(T)$. Because the edges in $DS^+(X)$ result from potentially different derivation trees, some of the $DS^+(X)$ graphs may contain cycles even if G is noncircular. If any graph has a cycle, then the algorithm fails and the grammar is not partitionable. If some $DS^+(X)$ is acyclic, the attributes of any instance of X can always be evaluated in an order respecting the partial order in $DS^+(X)$. The procedures for constructing the $DS^+(X)$ graphs are in the appendix in Section A.1.2.

Step 2: Compute a complete set of partitions

If the $DS^+(X)$ graphs are acyclic for all nonterminals X in G, we can use them to generate a set of partitions. For each nonterminal X, attributes are assigned to partitions as follows: First, we find all synthesized attributes that have no successors in $DS^+(X)$, remove them from the graph, and assign them to $S_k(X)$. (The value k is determined after all attributes have been assigned to a partition, so usually, partitions are numbered in reverse order at first, and are renumbered after all attributes have been assigned.) Then we find all inherited attributes with no successors, remove them from the graph, and assign them to $I_k(X)$. Next we assign attributes to partitions $S_{k-1}(X)$, and then $I_{k-1}(X)$. Continuing in this fashion, we assign attributes alternately to synthesized and inherited partitions until all attributes are assigned. The code for this step is in the appendix in Section A.1.3.

One of the key properties of this step is that the assignment of attributes to partitions for a nonterminal X depends only on $DS^+(X)$, not on the order of definitions in the attribute grammar specification. This property is essential in using the algorithm to define a grammar class. In effect, this step completes the partial order in $DS^+(X)$ to a total order.

Step 3: Test the partitioning for validity

Once we have a set of partitions, we must check to see whether the set is valid. This test is described later, when we describe the test to see whether a global partitioning is valid.

Testing a set of partitions for validity is possible even if we do not use Kastens's algorithm to generate the partitions, but obtain them by some other means. Therefore, the algorithm is a heuristic for guessing a potential partitioning. If an attribute grammar is partitionable, an exhaustive search will find a valid partitioning, but it is prohibitively expensive to do so. Fortunately, the algorithm succeeds for many practical attribute grammars.

6.4 Global Partitionings

With the discussion of partitionable attribute grammars behind us, we can return to developing the definition for globally partitionable attribute grammars. In order to allow a multiple subtree replacement algorithm to work, it is not sufficient to require that a valid set of partitions exist. For each pair of nonterminals X and Y, we also need to discover a total order, or an "interleaving," for the elements of $\Pi(X)$ and $\Pi(Y)$ combined.

6.4.1 Interleavings

Let G be an attribute grammar and $\Pi(G)$ a complete set of partitions for G, and let X and Y be nonterminals of G such that $X \rightsquigarrow Y$. An *interleaving* for X and Y with respect to Π, denoted $\Gamma_\Pi(X,Y)$, is a sequence merging the elements of $\Pi(X)$ and $\Pi(Y)$,

$$\Gamma_\Pi(X,Y) = \langle I_1(X), \sigma_1, S_1(X), \ldots, I_{k_X}(X), \sigma_{k_X}, S_{k_X}(X)\rangle,$$

where each σ_i is a (possibly empty) sequence of some pairs $I_j(Y), S_j(Y), \ldots$, and

$$\langle \sigma_1, \ldots, \sigma_{k_X}\rangle = \Pi(Y).$$

When it is more convenient, we write $P_1(X,Y), \ldots, P_{2(k_X+k_Y)}(X,Y)$ for the elements of $\Gamma_\Pi(X,Y)$, and when the nonterminals are obvious from context, we will write P_i in place of $P_i(X,Y)$. If $X = Y$, then each $p_i(X)$ will appear twice in $\Gamma_\Pi(X,Y)$, and superscripts can be used to differentiate them, as in $p_i(X^1)$ and $p_i(X^2)$.

We write $\Gamma_\Pi(G)$ (or just Γ_Π) for a complete set of interleavings

$$\{\Gamma_\Pi(X,Y) \mid X,Y \text{ nonterminals of } G, X \rightsquigarrow Y\}.$$

A set of partitions $\Pi(G)$ along with a set of interleavings $\Gamma_\Pi(G)$ is called a *global partitioning* of an attribute grammar.

A set of interleavings can only be defined with respect to a complete set of partitions. Therefore, if we are given an interleaving for a pair of nonterminals, we must implicitly have a complete set of partitions for the grammar as well. Henceforth, we will drop the subscript Π for all uses of Γ.

Example 6.1 Let $\Pi(X) = \langle A_1(X), A_2(X)\rangle$ and $\Pi(Y) = \langle A_1(Y), A_2(Y)\rangle$. Two possibilities for $\Gamma(X,Y)$ are

$$\langle I_1(X), I_1(Y), S_1(Y), S_1(X), I_2(X), I_2(Y), S_2(Y), S_2(X)\rangle$$

and

$$\langle I_1(X), S_1(X), I_2(X), I_1(Y), S_1(Y), I_2(Y), S_2(Y), S_2(X)\rangle.$$

The order of elements in $\Gamma(X,Y)$ is chosen to match the order in which a nonincremental, tree-walking evaluator would evaluate the partitions of instances of X and Y. For example, suppose that α and β are nodes in a derivation tree T such that $\alpha \rightsquigarrow \beta$, $\tau(\alpha) = X$, and $\tau(\beta) = Y$. The element $I_1(X)$ is first in $\Gamma(X,Y)$, because the attribute instances in $(\alpha, I_1(X))$ are evaluated just before visiting α for the first time. Node β cannot be visited by a tree-walking evaluator without being in the middle of a visit to α, so during some visit (or visits) to α, the evaluator will visit β, and at least one pair of ba-symbols, $(\beta, I_i(Y))$ and $(\beta, S_i(Y))$, $1 \le i \le k_Y$, will occur in the computation sequence (possibly separated by ba-symbols for attribute instances inside the subtree rooted at β), so $I_i(Y)$ and $S_i(Y)$ are consecutive elements in $\Gamma(X,Y)$. If β is not visited during the i-th visit to α, then σ_i of $\Gamma(X,Y)$ will be empty. Finally $S_{k_X}(X)$ is last in $\Gamma(X,Y)$, because the attribute instances in $(\alpha, S_{k_X}(X))$ are evaluated just before leaving α for the last time.

Although we can specify an interleaving for any attribute grammar, we are interested in interleavings whose order can be followed by a tree-walking evaluator.

Definition 6.5 *Let G be an attribute grammar, $\Pi(G)$ a complete set of partitions for G, X and Y nonterminals of G such that $X \rightsquigarrow Y$, $\Gamma(X,Y)$ an interleaving for X and Y with respect to $\Pi(G)$, and T a derivation tree for G. A computation sequence $s(T)$ for T* respects $\Gamma(X,Y)$, *if $\forall \alpha, \beta \in T$ such that $\tau(\alpha) = X$, $\tau(\beta) = Y$, and $\alpha \rightsquigarrow \beta$, the ba-symbol ($\alpha$ or β, $P_i(X,Y)$) occurs before the ba-symbol (α or β, $P_j(X,Y)$) in $s(T)$, for all $1 \le i < j \le 2(k_X + k_Y)$.*[3]

[3]The locution "α or β" in (α or β, $P_i(X,Y)$) means to use α if $P_i(X,Y)$ is a partition of X and to use β otherwise.

Definition 6.6 *Let G be an attribute grammar, $\Pi(G)$ a complete set of partitions for G, $\Gamma(G)$ a complete set of interleavings for G, and T a complete derivation tree for G. A computation sequence $s(T)$* respects Γ *if $s(T)$ respects Π and for all nonterminals X and Y of G, $s(T)$ respects $\Gamma(X, Y)$.*

We state explicitly that $s(T)$ must respect Π in order to respect Γ. This condition is needed solely to deal with derivation trees containing a single node, because a computation sequence for a one-node tree satisfies all interleavings trivially. For other derivation trees, if a computation sequence s respects $\Gamma(X, Y)$, then it necessarily respects $\Pi(X)$ and $\Pi(Y)$, because if $p_{i1}(X)$ occurs before $p_{j1}(X)$ in $\Pi(X)$, then $P_{i2}(X, Y)$ occurs before $P_{j2}(X, Y)$ in $\Gamma(X, Y)$, where $p_{i1}(X) = P_{i2}(X, Y)$ and $p_{j1}(X) = P_{j2}(X, Y)$.

In our discussion of PAGs, we sought plans that would always result in a computation sequence respecting Π for any derivation tree. Now in addition, we would like plans to result in a computation sequence that respects Γ for any derivation tree. This is possible if we have a "valid" set of interleavings.

Definition 6.7 *Let G be an attribute grammar and $\Gamma(G)$ a complete set of interleavings for G. Γ is* valid, *if for all derivation trees T of G, there exists a computation sequence, $s(T)$, respecting Γ.*

Definition 6.8 *Let G be an attribute grammar. G is* globally partitionable *(or GPAG) if there exists a set of interleavings, $\Gamma(G)$, such that Γ is valid.*

6.4.2 A Multiple Subtree Replacement Evaluator for Globally Partitionable Attribute Grammars

If an attribute grammar is globally partitionable, then the grammar must be partitionable as well, and so there exists a tree-walking evaluator that always evaluates the attributes of a given nonterminal in the same order for any instance of the nonterminal. If we wish to use the same evaluator for propagation after multiple subtree replacements, we must be sure that the attributes of pairs of nonterminals are always interpreted in an order respecting the interleaving for the nonterminals. Therefore, we must be sure that nonlocal visits and suspensions respect the interleavings for all pairs of nonterminal instances in a derivation tree, and we must define the *do_visit?* and *do_suspend?* functions accordingly.

In the multiple subtree replacement algorithm in Figure 4.7, we can make nonlocal visits from time to time to structure tree children of the current node. If we are at node α_0 in a derivation tree interpreting a VISIT(i, s) instruction, then if α_i, the i-th child of α_0, is unmarked, we will sometimes need to make a nonlocal visit to σ_i, the i-th structure tree child of α_0. Yet, as we saw in Chapter 4, we should not always make the nonlocal visit. Furthermore, if we do make the visit, we must decide where to resume interpretation of the plan at σ_i. We cannot use the *mapdown* function directly, since we are not visiting σ_i from its parent.

These decisions are made by using two additional functions, *mapto* and *mapfrom*, that are similar in purpose to the *mapdown* and *mapup* functions. If we are interpreting a VISIT(i, s) instruction and $\tau(\alpha_i) = X$ and $\tau(\sigma_i) = Y$, then we look up the value of $mapto(X, Y, s)$. If the value is 0, the visit is skipped. Otherwise, the visit is made, and the value is used as the argument to the *mapdown* function to determine where to begin interpretation of the plan at σ_i. A similar function, $mapfrom(X, Y, s)$, is used if we need to decide whether to make a nonlocal suspension from σ_i to α_0. Again, if $mapfrom(X, Y, s) = 0$, the SUSPEND is skipped.

Generating the *mapfrom* and *mapto* functions

We use the interleaving $\Gamma(X, Y)$ to generate the functions $mapto(X, Y, s)$ and $mapfrom(X, Y, s)$. The functions are inverses of a sort, in the same way that *mapup* and *mapdown* are. That is, $mapto(X, Y, s)$ and $mapfrom(X, Y, s)$ have the same number of non-zero values so that the plans "fit" together even when we are making nonlocal visits and suspensions. We define the functions as follows:

$$mapto(X, Y, s) = \begin{cases} j & \text{if } I_j(Y) \text{ follows } I_s(X) \text{ in } \Gamma(X, Y), \\ 0 & \text{if } S_s(X) \text{ follows } I_s(X) \text{ in } \Gamma(X, Y); \end{cases}$$

and

$$mapfrom(X, Y, s) = \begin{cases} j & \text{if } S_j(X) \text{ follows } S_s(Y) \text{ in } \Gamma(X, Y), \\ 0 & \text{if } I_{s+1}(Y) \text{ follows } S_s(Y) \text{ in } \Gamma(X, Y). \end{cases}$$

These mapping functions are used in *GPAG-evaluate*, shown in Figure 6.2. The lines marked with "†" indicate the differences between *GPAG_evaluate* and *Incremental_K_evaluate*.

A careful examination of the algorithm will show that an implementation must take two precautions that are omitted here in order to simplify the code, both related to the SUSPEND instruction. First, as written, during interpretation of a SUSPEND instruction, we unmark a node and remove it from the structure tree before checking whether it has a structure tree parent. Second, suppose that the current node α has a structure tree parent σ that is unmarked, and σ is in the structure tree only because it is the LCA of two nodes, α and another node β. When we unmark α, then σ will be removed from the structure tree. After the SUSPEND instruction, we will need to continue evaluation either at β, or at the structure tree parent of σ. To surmount these problems, when we interpret a SUSPEND instruction, we simply save some of the structure tree parent and child pointers so that we can resume interpreting the proper plan after we have finished interpreting the current instruction.

```
  procedure GPAG_evaluate(set of nodes S);
  /* Set S contains the nodes where subtree replacements were performed. */
  /* Let p be the production X_0 → X_1 ... X_m, where p = ρ(α_0). */
† for each α ∈ S do
†     Mark the attribute instances in A(α).
†     Mark α, adding it to the structure tree.
†     if α is not the root of the tree then
†         Mark parent(α), adding it to the structure tree.
† α_0 := the root of the structure tree.
  plan_index := 1;
  repeat forever
      case PLAN(p)[plan_index] of
          EVAL(i, a):
              if α_i.a is marked then        /* Only evaluate marked attributes. */
                  Unmark and reevaluate α_i.a.
                  if α_i.a has a new value then
                      for each successor β.b of α_i.a do
                          Mark β.b.
                          Mark defining_node(β.b)
†                             (adding it to the structure tree, if necessary).
              plan_index++;
          VISIT(i, s):
              if α_i is marked then
                  α_0 := α_i;
                  plan_index := mapdown(p, s);                  /* Visit child i. */
†             else if there is an i-th structure tree child σ_i
†               and mapto(X_i, τ(σ_i), s) > 0 then
†                 plan_index := mapdown(ρ(α_i), mapto(X_i, τ(σ_i), s));
†                 α_0 := σ_i;                               /* Make nonlocal visit. */
              else plan_index++;                                /* Skip VISIT. */
          SUSPEND(s):
              if this is the final SUSPEND in PLAN(p) then unmark α_0;
†                 (removing α_0 from the structure tree, if necessary).
              if α_0 is the root of the tree then return;
              if parent(α_0) is marked then
                  j := child_no(α_0);
                  α_0 := parent(α_0);                          /* Return to parent. */
                  plan_index := mapup(p, s, j);
†             else if there is a structure tree parent σ
†               and mapfrom(τ(σ), X_0, s) > 0 then
†                 plan_index:= mapup(ρ(σ), mapfrom(τ(σ), X_0, s), st_child_no(α_0));
†                 α_0 := σ;                                 /* Nonlocal SUSPEND */
              else                                              /* Skip SUSPEND. */
                  if this is the final SUSPEND in PLAN(p) then return;
                  plan_index++;
              end
```

† These statements were added to *Incremental_K_evaluate* to create *GPAG_evaluate*.

Figure 6.2: Static, deterministic, multiple subtree replacement evaluator

6.4.3 Efficiency of *GPAG-evaluate*

If we ignore the time needed for the structure tree operations, *GPAG-evaluate* is optimal. For each attribute instance in INFLUENCED, its defining node is marked and added to the structure tree. Since marking a node can cause a second node to be added to the structure tree, we interpret at most two plans for any attribute instance in INFLUENCED. For a given attribute grammar, the maximum size of a plan is fixed, so the overhead of the algorithm is $O(|\text{AFFECTED}|)$.

Unfortunately, each structure tree operation takes amortized $O(\log n)$ time, where n is the number of nodes in the derivation tree. If k subtree replacements are made, then initializing the structure tree takes amortized $O(k \log n)$ time. Since at least one attribute is reevaluated at each point of change, then $|\text{AFFECTED}| > k$, and the overall bound for the algorithm is amortized $O(\log n \cdot |\text{AFFECTED}|)$.

It is natural to wonder how bad this time bound is. If n is large, the structure tree operations could be expensive. On the other hand, once the structure tree has been initialized, the virtual tree representation of the derivation tree has, in effect, been partitioned into a number of smaller, self-adjusting binary trees. Subsequent structure tree operations will only affect a single binary tree in the virtual tree representation, because any new nodes marked during the course of evaluation are neighbors of existing nodes. Therefore, the actual running time for these additional structure tree operations will probably be less than $O(\log n)$.

Suppose we make two structure tree replacements in a derivation tree at nodes α and β such that $\alpha \rightsquigarrow \beta$. Then the initial structure tree will contain a single edge between α and β. If during evaluation, propagation does not extend far from α or β, then we will only be marking a few additional nodes, and the actual cost of the structure tree operations will be slight. On the other hand, if we must mark all the nodes between α and β, then we will end up splitting the structure tree edge between α and β into a number of structure tree edges, each corresponding to a single edge in the underlying tree. But the nodes in the balanced tree representing the original structure tree edge (i.e. path) between α and β are accessed in order. According to a theorem of Tarjan, accessing the nodes of a self-adjusting binary tree in order takes linear time [Tar85]. In our case, the nodes are accessed in order, but they come from either end of the path representing the structure tree edge. Nonetheless, we conjecture that the theorem holds in this case as well, although this has not been proved formally.

So in general, if all the "on-path" nodes are eventually marked during evaluation, the running time reduces to amortized $O(k \log n + |\text{AFFECTED}|)$. Of course, the actual running time depends on the depths of the nodes where subtrees were replaced as well as the initial configuration of the virtual tree representing the derivation tree.

One way we can reduce the perceived running time of an interactive evaluator is to use idle processor time to rebalance the trees representing the structure tree, or to make solid paths as long as possible by making sure that there is always a solid path between a node and one of its children, whenever this can be done without violating the structure tree invariant.

1: root → type	type.t_in = 0;	
2: decl → field option	$$.t_out = option.t_out;	
	$$.type = field.type + option.type;	
	field.t_in = $$.t_in;	
	option.t_in = field.t_out;	
	option.all = field.type + option.type;	
3: field → ε	$$.t_out = $$.t_in;	$$.type = 0;
4: field → decl	$$.t_out = decl.t_out;	$$.type = decl.type;
	type.t_in = $$.t_in;	
5: option → variant	$$.t_out = variant.t_out;	$$.type = $$.t_in + variant.type;
	variant.t_in = $$.t_in;	variant.all = $$.all;
6: variant → field	$$.t_out = field.t_out;	$$.type = field.type;
	field.t_in = $$.t_in;	

Figure 6.3: Sample attribute grammar

6.4.4 Finding a Valid Global Partitioning for an Attribute Grammar

As is the case with Kastens's evaluator and partitions, *GPAG-evaluate* is useful only if we can find a valid set of interleavings for an attribute grammar. Since finding a valid partitioning is an NP-complete problem, finding a valid global partitioning is at least as hard a problem. Nonetheless, in this section we give a polynomial-time algorithm to find a potential global partitioning and check it for validity. If the constructed global partitioning is not valid, then the grammar may still be GPAG, but only for a global partitioning different from the one chosen by the algorithm. This situation is analogous to Kastens's algorithm, which can fail to find a valid partitioning, even if an attribute grammar is partitionable.

An algorithm for finding a global partitioning consists of five steps, and parallels the algorithm for finding an ordinary partitioning. Because we want to find a static characterization of an attribute grammar, we use the dependencies from the grammar to compute dependency graphs that represent the union of dependencies from all possible derivation trees. As long as these graphs are acyclic, we can use them to construct the mapping functions and plans needed by the evaluator.

In describing the algorithm, we will use the attribute grammar found in Figure 6.3.The attribution rules use the syntax of the Synthesizer Generator [RT89b]. The term "$$" denotes the left-hand-side nonterminal of each production. Code for the algorithm, which runs in polynomial time, is in Appendix B. The analysis is fairly straightforward, but is also in the appendix, because it depends on details of the algorithm not given here.

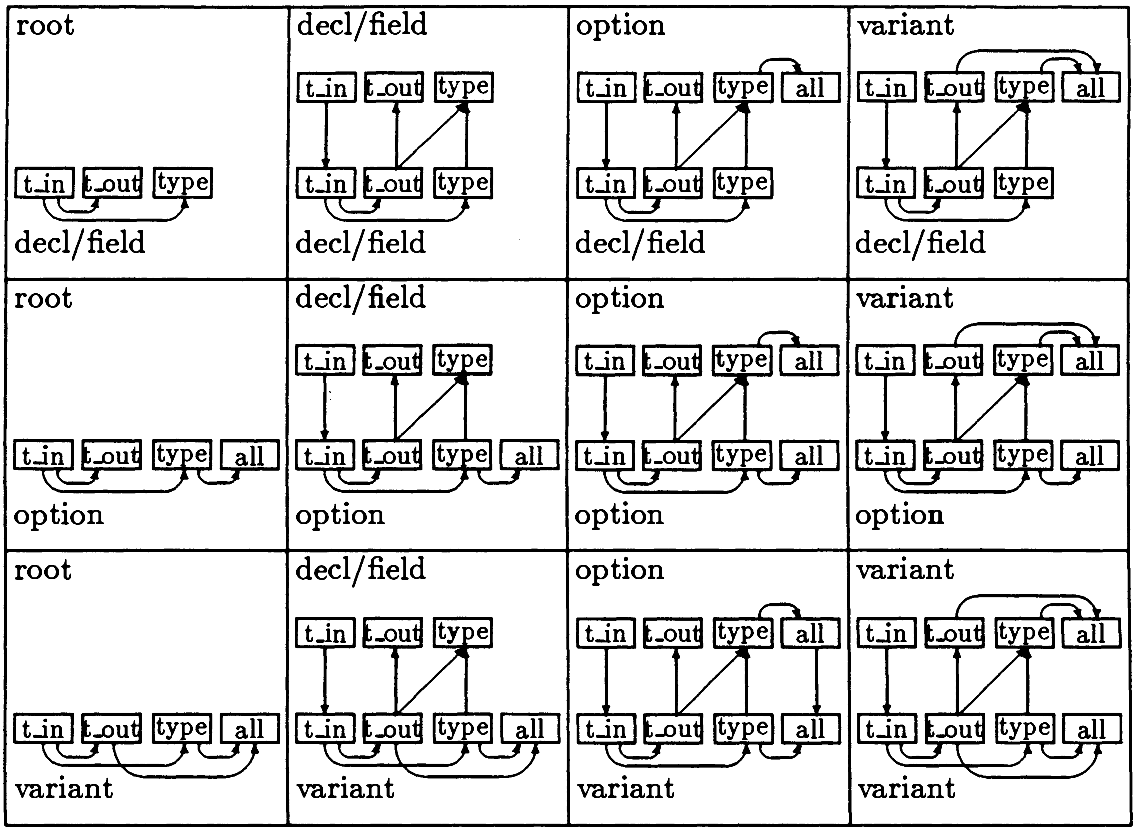


Figure 6.4: DSS and DS graphs after step 1

Step 1: Construct $DSS(X,Y)$ and $DS^+(X)$ graphs

For each pair of nonterminals X and Y such that $X \rightsquigarrow Y$, we construct a $DSS(X,Y)$ graph whose vertices are the attributes of X and Y, and for which there is an edge between two vertices a and b whenever there is a path in $D(T)$ for some derivation tree T between attributes corresponding to a and b. From the DSS graphs, we can compute DS^+ graphs by projecting the DSS graphs onto the attributes of a single nonterminal.

This step was originally specified in [RMT86]. Code for this step can be found in Section B.1.1. After completing the step, we have the DSS graphs shown in Figure 6.4. (It turns out that the graphs involving *decl* or *field* are isomorphic, so they are combined in the figure.) Note that we can obtain any $DS^+(X)$ graph by projecting one of the DSS graphs. For example, if we project the $DSS(option, variant)$ graph onto the attributes of *option*, we obtain $DS^+(option)$. In the example, since root does not have any attributes, we can easily read off the DS^+ graphs for the other nonterminals in the left column of Figure 6.4.

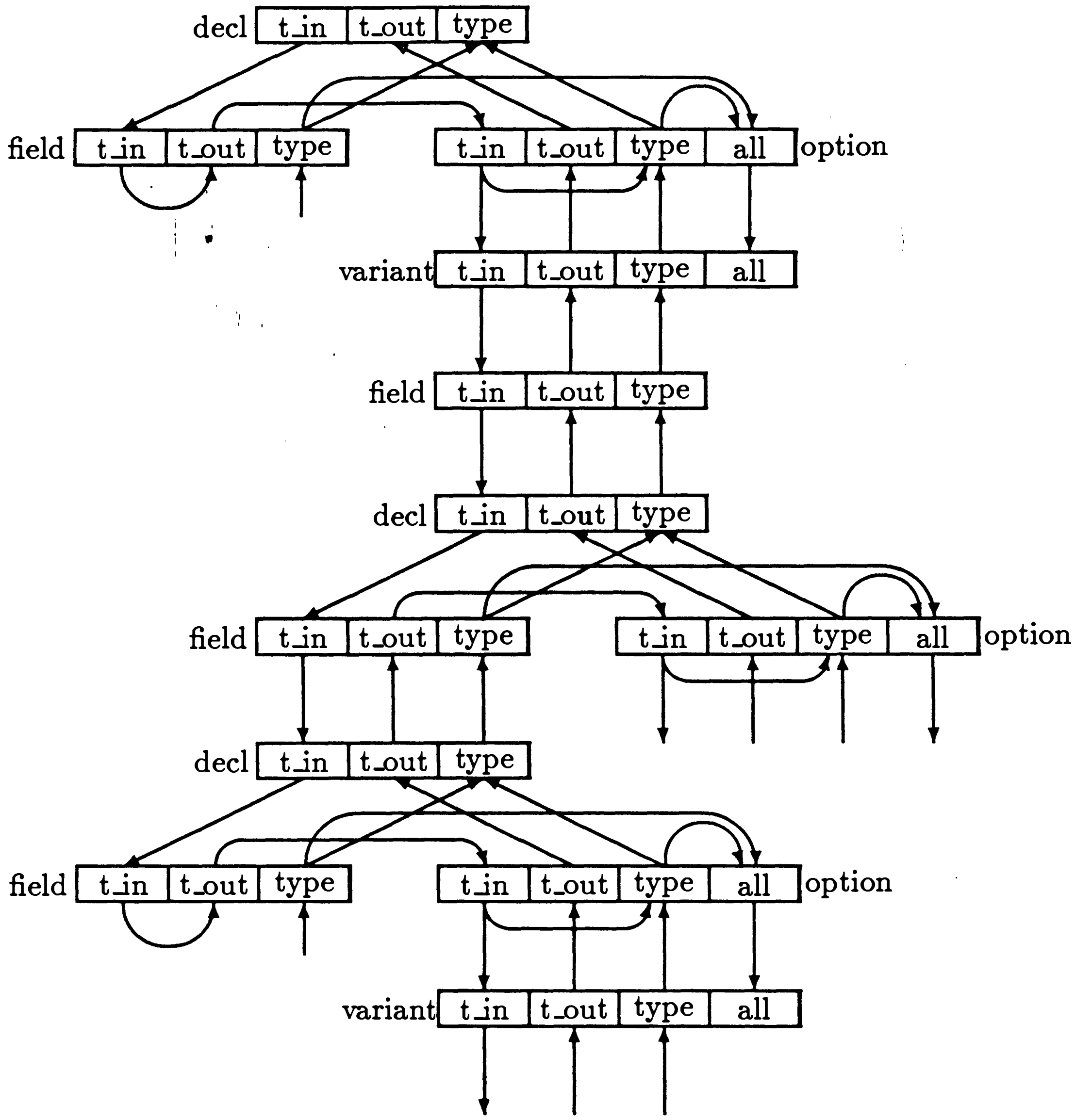


Figure 6.5: A sample derivation tree

Step 2: Compute a complete set of partitions

Computing the partitions proceeds exactly as in Kastens's algorithm, but we must use the $DS^+(X)$ graphs computed in the previous step, because for some grammars, there will be edges in the DS^+ graphs that would not exist if we used Kastens's algorithm. For example, $DS^+(\textit{variant})$ contains an edge from *t_out* to *all*, even though there is no derivation tree containing an instance of *variant* with a path from *t_out* to *all*. This edge does not exist when using Kastens's algorithm. Examining Figure 6.5, we can see where the edge comes from. The extra edge is a result of two dependency paths. First, there is a path from the bottom *variant.t_out* to the top *option.type*. Second, there is a path from *option.type* to *variant.all*. Thus, in the $DSS(option, variant)$ graph computed in Step 1, there will be an edge from *variant.t_out* to *option.type* and from *option.type* to *variant.all*. These edges create a path from *variant.t_out* to *variant.all*, and this edge will be induced in the $DS^+(variant)$ graph. This imposes an additional constraint on the order of evaluation of attributes of instances of *variant*.

The partitions that are computed in this step are as follows:

decl:	$I_1 = \{t\_in\}$	$S_1 = \{t\_out,\ type\}$		
field:	$I_1 = \{t\_in\}$	$S_1 = \{t\_out,\ type\}$		
option:	$I_1 = \{t\_in\}$	$S_1 = \{type\}$	$I_2 = \{all\}$	$S_2 = \{t\_out\}$
variant:	$I_1 = \{t\_in\}$	$S_1 = \{t\_out,\ type\}$	$I_2 = \{all\}$	$S_2 = \{\}$

Step 3: Construct $DSS_\Pi(X,Y)$ graphs

Each $DSS(X,Y)$ graph is reduced to a $DSS_\Pi(X,Y)$ graph, which has a vertex for each partition of X and Y, and an edge between partitions whenever an edge exists in the DSS(X,Y) graph between an attribute in one partition and any attribute in the other partition. Edges between attributes in a single partition are deleted. Examples of these reduced graphs are shown in Figure 6.6

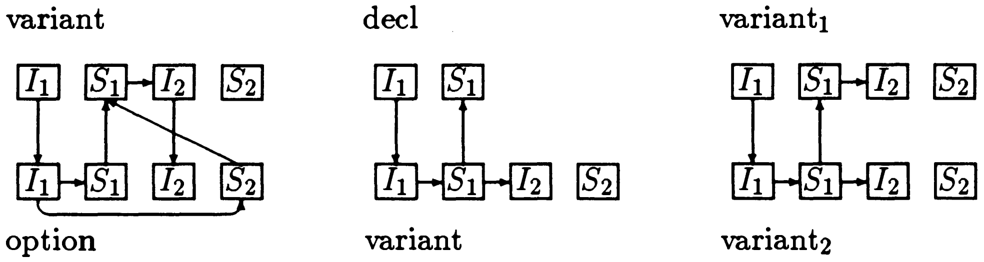


Figure 6.6: Some DSS_Π graphs

Once we have a set of partitions Π, we no longer need to be concerned with the individual attributes. In effect, each partition can be treated as a single attribute depending on all the predecessors of attributes contained in the partition. By using partitions instead of attributes, we reduce the effective size of the $DSS(X,Y)$ graphs. The result of this step is to give us a partial order on the partitions of pairs of nonterminals.

Step 4: Generate a complete set of interleavings

We use these reduced graphs to compute interleavings for all pairs of nonterminals X and Y such that $X \leadsto Y$. Each interleaving must start with the first partition of X, i.e., $I_1(X)$. Next we consider the pairs of partitions that can follow for a partitioning to be valid. We choose between the pair $S_1(X), I_2(X)$ or the pair $I_1(Y), S_1(Y)$. We must choose the pair that has no predecessor among the as-yet-unassigned partitions. If there is a choice, we arbitrarily choose the pair from X. Finally, the interleaving ends with $S_k(X)$.

Example 6.2 The reduced graphs shown in Figure 6.6 give rise to the following interleavings:

$$\begin{aligned}
\Gamma(\mathit{variant}, \mathit{option}) &= \langle I_1(\mathit{variant}), I_1(\mathit{option}), S_1(\mathit{option}), I_2(\mathit{option}), \\
&\qquad S_2(\mathit{option}), S_1(\mathit{variant}), I_2(\mathit{variant}), S_2(\mathit{variant})\rangle \\
\Gamma(\mathit{decl}, \mathit{option}) &= \langle I_1(\mathit{decl}), I_1(\mathit{variant}), S_1(\mathit{variant}), \\
&\qquad I_2(\mathit{variant}), S_2(\mathit{variant}), S_1(\mathit{decl})\rangle \\
\Gamma(\mathit{variant}_1, \mathit{variant}_2) &= \langle I_1(\mathit{variant}_1), I_1(\mathit{variant}_2), S_1(\mathit{variant}_2), S_1(\mathit{variant}_1), \\
&\qquad I_2(\mathit{variant}_1), I_2(\mathit{variant}_2), S_2(\mathit{variant}_2), \\
&\qquad S_2(\mathit{variant}_1)\rangle
\end{aligned}$$

Note that the first two interleavings are unique, but there are two possibilities for the third one, After the partition $S_1(\mathit{variant}_2)$, we could have chosen the pair $I_2(\mathit{variant}_2)$, $S_2(\mathit{variant}_2)$. Instead, we arbitrarily chose the pair from the first nonterminal, namely $\mathit{variant}_1$.

Step 5: Check the global partitioning for validity

Finally, we check the global partitioning for validity using the test described in the next section.

6.5 Determining Whether a Global Partitioning is Valid

In the rest of the chapter, we deal with the problem of determining whether a set of interleavings is valid. We describe a test and prove that the test works. Much of the material in this section is inspired by the approach of Engelfriet and Filè [EF80], who give an algorithm for testing a set of partitions for validity.

A set of interleavings cannot be valid unless the interleavings for all pairs of nonterminals can be respected simultaneously for any derivation tree.

Example 6.3 Let

$$\begin{aligned}
\Gamma(X,Y) &= \langle I_1(X), I_1(Y), S_1(Y), S_1(X), I_2(X), S_2(X)\rangle, \\
\Gamma(Y,Z) &= \langle I_1(Y), I_1(Z), S_1(Z), S_1(Y)\rangle, \text{and} \\
\Gamma(X,Z) &= \langle I_1(X), S_1(X), I_2(X), I_1(Z), S_1(Z), S_2(X)\rangle.
\end{aligned}$$

These interleavings cannot be valid. There is no way to list the elements of $\Pi(X)$, $\Pi(Y)$, and $\Pi(Z)$ (implied by the interleavings) such that all three interleavings are respected at the same time. That is, for any derivation tree containing at least three nodes α_1, α_2, and α_3, with $\tau(\alpha_1) = X$, $\tau(\alpha_2) = Y$, $\tau(\alpha_3) = Z$, and $\alpha_1 \rightsquigarrow \alpha_2 \rightsquigarrow \alpha_3$, no computation sequence can respect all three interleavings simultaneously.

In this example, we could determine that the interleavings were not valid without knowing the dependencies of the attribution equations of the grammar, because there are inherently inconsistent. Informally, a complete set of interleavings Γ is *consistent* if for any derivation tree T, interleavings for all pairs of nodes in T can be respected simultaneously, independent of the dependencies of the attribute grammar.

To see whether a set of interleavings is valid, we must construct certain graphs and check them for cycles. Two kinds of graphs are needed. One kind makes sure that the interleavings are consistent, and the other kind makes sure that the interleavings respect the dependencies in the attribute grammar. The first kind of graph is constructed from interleaving graphs.

Definition 6.9 *Let G be an attribute grammar, X and Y nonterminals of G with $X \rightsquigarrow Y$, and $\Gamma(X,Y) = \langle P_1, \ldots, P_n \rangle$ an interleaving for X and Y, where $n = 2(k_X + k_Y)$. The* interleaving graph *for X and Y, denoted $G\Gamma(X,Y)$, has vertices $P_1, \ldots, P_n$, and directed edges $[P_1, P_2], \ldots, [P_{n-1}, P_n]$.*

The P_i are sets of attributes of X or Y. Recall that $\Gamma(X,X)$ has elements $P_i(X^1)$ and $P_i(X^2)$ for $1 \le i \le 2k_X$. Similarly, $G\Gamma(X,X)$ has distinct vertices $P_i(X^1)$ and $P_i(X^2)$. If we use superscripts to distinguish between repeated nonterminals, then $G\Gamma(X^1,Y^2) \cup G\Gamma(X^1,Z^3)$ has $2(k_X + k_Y + k_Z)$ vertices and the vertices $P_i(X^1)$ occur in each constituent graph, while $G\Gamma(X^1,Y^2) \cup G\Gamma(X^4,Z^3)$ has $2(k_X + k_Y + k_X + k_Z)$ vertices and is disconnected. Normally, we will omit the superscripts, because they can be inferred from context.

The graphs that are used to determine whether the interleavings are consistent can now be defined.

Definition 6.10 *Let G be an attribute grammar and $\Gamma(G)$ a set of interleavings for G, and let X, Y, and Z be nonterminals with $X \overset{1}{\rightsquigarrow} Y \rightsquigarrow Z$. The* consistency graph *for X, Y, and Z is*

$$G\Gamma(X,Y,Z) = G\Gamma(X^1,Y^2) \cup G\Gamma(X^1,Z^3) \cup G\Gamma(Y^2,Z^3).$$

For a given attribute grammar, there is a finite number of consistency graphs. These graphs can be used to test whether a set of interleavings is consistent.

Definition 6.11 *Let G be an attribute grammar and $\Gamma(G)$ a set of interleavings for G. The* consistency test *for Γ requires constructing and examining graphs $G\Gamma(X,Y,Z)$ for all $X \overset{1}{\rightsquigarrow} Y \rightsquigarrow Z$. The consistency test succeeds if all such graphs are acyclic.*†

†Now we can define "consistency" more formally. A set of interleavings Γ is consistent if the consistency test passes. Since we will not need to discuss consistency, except when testing the validity of a set of partitions, we leave the definition as a footnote.

The second kind of graph that we need is a *production graph*, and is used to make sure that the interleavings respect the dependencies in the attribution equations. Production graphs contain three kinds of edges. One kind represents dependencies between attribute occurrences of a production, a second kind reflects the order of partitions for the nonterminals in a production, and the third kind is based on the order of partitions in the interleavings for pairs of nonterminals in a production.

Definition 6.12 *Let G be an attribute grammar, $\Gamma(G)$ a set of interleavings for G, and p a production of G with $p: X_0 \to X_1 \cdots X_m$. The* production graph, *denoted $G\Gamma(p)$, is a graph with vertices*

$$\{p_i(X_j) \mid 0 \leq j \leq m \text{ and } 1 \leq i \leq 2k_{X_j}\}$$

and directed edges:

- $[p_{i1}(X_{j1}), p_{i2}(X_{j2})]$, *if there is a dependency in an attribution equation of p between attribute occurrences v and w, where $v \in p_{i1}(X_{j1})$ and $w \in p_{i2}(X_{j2})$;*
- $[P_i(X_0, X_j), P_{i+1}(X_0, X_j)]$ *for $1 \leq j \leq m$ and $1 \leq i < 2(k_X + k_Y)$;*
- $[p_i(X_j), p_{i+1}(X_j)]$ *for $0 \leq j \leq m$ and $1 \leq i < 2k_X$.*

If we did not have any ϵ-productions, we could define $G\Gamma(p)$ by starting with $\bigcup_{i=1}^{m} G\Gamma(X_0^0, X_i^i)$ and adding edges for dependencies arising from the attribution equations, because the interleaving graphs already have edges respecting the partition orders of their constituent nonterminals. But for a production $p: X_0 \to \epsilon$, the graph $\bigcup_{i=1}^{m} G\Gamma(X_0^0, X_i^i)$ is empty, and the edges (and vertices) from the partitions of X_0 must be used to construct $\Gamma(p)$.

Definition 6.13 *Let G be an attribute grammar and $\Gamma(G)$ a set of interleavings for Γ. The* feasibility test *for Γ requires constructing and examining graphs $G\Gamma(p)$ for all productions p in G. The feasibility test succeeds if all such graphs are acyclic.*

We now have the machinery for testing the validity of a set of interleavings Γ for an attribute grammar. We will prove that Γ is valid if and only if both the consistency and feasibility tests succeed. We prove that the success of the two tests is sufficient for Γ to be valid by using the production graphs to construct a computation sequence for any derivation tree. We prove that the tests are necessary by showing that for any graph involved in either test, there is some derivation tree T containing an instance of the graph. If Γ is valid, then a computation sequence for T must exist, and we see that any such computation sequence ensures that the graph in question is acyclic.

In the proof, we will need to consider computation sequences for partial derivation trees, that is, trees whose roots are not the start nonterminal of G. The conditions defining a computation sequence only apply to complete derivation trees, but we can extend them to partial trees by noting that a visit-trace tuple for a subtree T' of T is a computation sequence for T'. We extend the start-end condition by requiring that the first and last ba-symbols of $v(T')$ be $(\alpha', I(\tau(\alpha')))$ and $(\alpha', S(\tau(\alpha')))$, respectively, where α' is the root of T'. The feasibility condition will hold trivially for inherited attribute instances of α', because their predecessors are not in T'. The sequentiality and completeness conditions apply to partial trees, without change.

We use the $G\Gamma(p)$ graphs to generate templates for constructing computation sequences for an entire derivation tree.

Definition 6.14 *Let G be an attribute grammar, $\Gamma(G)$ a set of interleavings for G, $p : X_0 \rightarrow X_1 \cdots X_m$ a production of G, and $G\Gamma(p)$ the production graph for p. If $G\Gamma(p)$ is acyclic, then a* visit-sequence *for p, V_p^Γ, is a list of the nodes in $G\Gamma(p)$ in topological order with # symbols inserted as placeholders in certain places:*

$$V_p^\Gamma = I_1(X_0), \sigma_1, \ldots, \sigma_n, S_k(X_0),$$

where each σ_i is either $I_{j1}(X_{j2}), S_{j1}(X_{j2})$ for $1 \leq j2 \leq m$ and $1 \leq j1 \leq k_{X_{j2}}$ or $S_{j3}(X_0), \#, I_{j3+1}(X_0)$ for $1 \leq j3 < k_{X_0}$.

This order is possible, because G is in normal form. Therefore, edges leading to $I_{i1}(X_0)$ or to $S_{i2}(X_j)$ for $j > 0$, are in $G\Gamma(p)$ only because of edges added between consecutive elements of $G\Gamma(X_0, X_i)$ and $\Pi(X_j)$. Specifically,

1. we can list $I_1(X_0)$ first in V_p^Γ, since it is the first element in all the interleavings included in $G\Gamma(p)$, and also the first element in $\Pi(X_0)$;

2. we can list both $I_i(X_j)$ and $S_i(X_j)$ as soon as all predecessors of $I_i(X_j)$ for $j > 0$ have been listed;

3. we can list both $S_i(X_0)$ and $I_{i+1}(X_0)$ as soon as all predecessors of $S_i(X_0)$ for $i < k_{X_0}$ have been listed; and

4. we can list $S_k(X_0)$ last in V_p^Γ, because it is the last element in all the interleavings included in $G\Gamma(p)$, and also the last element in $\Pi(X_0)$.

Lemma 6.1 *Let G be an attribute grammar and $\Gamma(G)$ a set of interleavings for G. If the feasibility test succeeds and the consistency test succeeds, then for any (complete or partial) derivation tree T of G, there exists a visit-trace tuple, $v(T)$, that respects Γ.*

Proof: This proof is similar to the proof in [EF80] showing that a slightly different feasibility test (one that doesn't mention interleavings) is sufficient for the existence of a visit-trace tuple respecting Π. Engelfriet and Filè only need to ensure that $v(T)$ respects Π, while we require that $v(T)$ respect Γ, as well, and so our proof depends on the success of the consistency test. The proof is by induction on the height of derivation trees.

Base case: (Trees of height 1). Let T be a derivation tree of height 1. It must be an instance of a production $p : X \rightarrow \epsilon$. The visit sequence for p is $V_p^\Gamma = I_1(X), S_1(X), \#, \ldots, \#, I_k(X), S_k(X)$. The i-th visit trace is $v_i(T) = (\alpha, I_i(X)), (\alpha, S_i(X))$, where α is the only node in T, and so the visit trace tuple for T is $v(T) = \#v_1(T)\# \cdots \#v_k(T)\#$. Rewriting $v(T)$ as $(\alpha, I_1), (\alpha, S_1), (\alpha, I_2), \ldots, (\alpha, S_k)$, we verify that $v(T)$ satisfies the four conditions for a computation sequence:

Start-end $v(T)$ starts with $(\alpha, I_1(X))$ and ends with $(\alpha, S_k(X))$.

Sequentiality All ba-symbols in $v(T)$ have α as the first component, and the second component alternates between inherited and synthesized attributes, as required.

Feasibility If an attribute instance in $(\alpha, S_i(X))$ depends on an attribute instance in $(\alpha, I_j(X))$, then $[I_j(X), S_i(X)]$ is an edge in $G\Gamma(p)$, so $I_j(X)$ must be listed before $S_i(X)$ in V_p^Γ, and $(\alpha, I_j(X))$ must precede $(\alpha, S_i(X))$ in $v(T)$.

Completeness Each $p_i(X)$ appears in exactly one ba-symbol in $v(T)$.

Finally, we must make sure that $v(T)$ respects Γ. Since T only has one node, we just need to check that $\Pi(X)$ is respected. But this is obviously true, because if $p_i(X)$ occurs before $p_j(X)$ in $\Pi(X)$, then there is an edge $[p_i(X), p_j(X)]$ in $G\Gamma(p)$, so $p_i(X)$ occurs before $p_j(X)$ in V_p^Γ, and $(\alpha, p_i(X))$ comes before $(\alpha, p_j(X))$ in $v(T)$.

Induction step: Assume the lemma holds for all trees of height less than n. Consider a tree T of height n, with an instance of $p : X_0 \rightarrow X_1, \ldots, X_m$ at the root and with root node α_0. Let α_j be the children of α_0, and let T_j be the subtree rooted at α_j, for each $1 \leq j \leq m$. By assumption, there is a visit-trace tuple $v(T_j)$ that respects Γ for each T_j. We construct $v(T)$ from V_p^Γ as follows:

- Replace $I_i(X_0)$ and $S_i(X_0)$ with $(\alpha_0, I_i(X_0))$ and $(\alpha_0, S_i(X_0))$, respectively, for $1 \leq i \leq k_{X_0}$.
- Replace the pair $I_i(X_j), S_i(X_j)$ with $v_i(T_j)$, for all $1 \leq j \leq m$ and $1 \leq i \leq k_{X_j}$.

Once again, we verify that $v(T)$ is a visit-trace tuple by checking that the conditions are satisfied.

Start-end Since V_p^Γ starts and ends with $I_1(X_0)$ and $S_k(X_0)$, respectively, $v(T)$ starts with $(\alpha_0, I_1(X_0))$ and ends with $(\alpha_0, S_k(X_0))$.

Sequentiality In V_p^Γ, element $I_i(X_0)$ is preceded by $S_{i-1}(X_0)$, if $i > 0$, and it is followed by $S_i(X_0)$ or by $I_{i'}(X_j)$ for $1 \leq j \leq m$ and $1 \leq i' \leq k_{X_j}$. Similarly, $S_i(X_0)$ is followed by $I_{i+1}(X_0)$, if $i < k_{X_0}$, and preceded by $S_{i'}(X_j)$, for $1 \leq j \leq m$ and $1 \leq i' \leq k_{X_j}$, or by $I_i(X_0)$. In all these cases, when the substitution is made, the resulting $v(T)$ will satisfy the sequentiality condition for all pairs of ba-symbols for which one of the ba-symbols has α_0 as its first component. By the induction hypothesis, all other consecutive pairs of ba-symbols in $v(T)$ also satisfy the sequentiality condition.

Feasibility If an attribute instance in $(\alpha_j, p_i(X_j))$ depends on an attribute instance in $(\alpha_{j'}, p_{i'}(X_{j'}))$, then $[p_{i'}(X_{j'}), p_i(X_j)]$ is an edge in $G\Gamma(p)$, so $p_{i'}(X_{j'})$ must come before $p_i(X_j)$ in V_p^Γ, and $(\alpha_{j'}, p_{i'}(X_{j'}))$ must precede $(\alpha_j, p_i(X_j))$ in $v(T)$.

Completeness From the method of construction, it is easy to see that this condition is satisfied.

Last, we show that $v(T)$ respects Γ. Recall that α_0 is the root of T. By the induction hypothesis, for all pairs of nodes β and β' such that $\beta \neq \alpha_0$ and $\beta \rightsquigarrow \beta'$, $v(T)$ respects $\Gamma(\tau(\beta), \tau(\beta'))$. Otherwise, for any node β in T such that $\beta \neq \alpha_0$, then $\alpha_0 \rightsquigarrow \beta$. To complete the proof, we must show that $v(T)$ respects $\Gamma(X_0, \tau(\beta))$, for all $\beta \neq \alpha_0$.

Because $v(T)$ is complete, then for each element P_j in $\Gamma(X_0, \tau(\beta))$, there is a corresponding ba-symbol (α_0, P_j) or (β, P_j) in $v(T)$, depending on whether P_j is a partition of X_0 or of $\tau(\beta)$. Suppose β is a child of α_0, that is, $\beta = \alpha_i$ for some $1 \leq i \leq m$. Then $\tau(\beta) = X_i$, and if P_{i1} occurs before P_{i2} in $\Gamma(X_0, X_i)$, then in $v(T)$, the ba-symbol corresponding to P_{i1} must come before the ba-symbol corresponding to P_{i2}, because there is a path in $G\Gamma(p)$ from P_{i1} to P_{i2}.

If β is not a child of α_0, then it must be a descendant of α_i for some $1 \leq i \leq m$. Let $\tau(\beta) = Y$, and let P_{j1} and P_{j2} be elements in $\Gamma(X_0, Y)$ such that P_{j1} comes before P_{j2}. If P_{j1} and P_{j2} are both partitions of X_0, then the ba-symbols in $v(T)$ corresponding to P_{j1} and P_{j2} are (α_0, P_{j1}) and (α_0, P_{j2}). But the method of construction for $G\Gamma(p)$ assures that there is a path in $G\Gamma(p)$ from P_{j1} to P_{j2}, so (α_0, P_{j1}) must occur before (α_0, P_{j2}) in $v(T)$. A similar argument holds if P_{j1} and P_{j2} are both partitions of Y.

Now we consider the case that P_{j1} is a partition of X_0 and P_{j2} is a partition of Y. We will use p_{X_0} and p_Y to refer to these partitions. The ba-symbols corresponding to P_{j1} and P_{j2} are (α_0, p_{X_0}) and (β, p_Y), respectively. Suppose that (β, p_Y) comes before (α_0, p_{X_0}) in $v(T)$, so that $v(T)$ does not respect $\Gamma(X_0, Y)$. Because of the way visit sequences are constructed, there must be a ba-symbol $(\alpha_i, p_{X_i}(X_i))$ between (β, p_Y) and (α_0, p_{X_0}) in $v(T)$, where $p_{X_i}(X_i)$ is a partition of X_i.

Among the vertices of $G\Gamma(X_0, X_i, Y)$ are p_{X_0}, p_{X_i}, and p_Y. Since $v(T_i)$ respects Γ, and since each interleaving defines a total order among the partitions of its nonterminals, there must be a path from p_Y to p_{X_i} in $G\Gamma(X_i, Y)$. We already saw that the partitions of α_0 and α_i respect $\Gamma(X_0, X_i)$, so there must be a path from p_{X_i} to p_{X_0} in $G\Gamma(X_0, X_i)$. Finally, there is a path from p_{X_0} to p_Y in $G\Gamma(X_0, Y)$, because P_{j1} comes before P_{j2} in $\Gamma(X_0, Y)$. But these three paths must also be in

$G\Gamma(X_0, X_i, Y)$, and so $G\Gamma(X_0, X_i, Y)$ must contain a cycle, a contradiction, since $G\Gamma(X_0, X_i, Y)$ is checked for cycles by the consistency test.

If P_{j1} is a partition of Y and P_{j2} is a partition of X_0, then a similar argument holds. Therefore, in $v(T)$, the ba-symbol corresponding to P_{j1} must come before the ba-symbol corresponding to P_{j2}. Since our choice of P_{j1} and P_{j2} and β was arbitrary, then we conclude that $\Gamma(X_0, \tau(\beta))$ must be respected for all $\beta \neq \alpha_0$.

Finally, we mentioned in Section 6.4.1 that when $\Gamma(X, Y)$ is respected by a computation sequence for a tree with at least one instance of X and Y, then $\Pi(X)$ and $\Pi(Y)$ are respected as well. Therefore, $v(T)$ must respect Π, and our proof is complete. □

This proof shows that the success of the consistency and feasibility tests is sufficient for determining whether a global partitioning is valid. The following theorem is now easy to prove.

Theorem 6.1 *Let G be an attribute grammar and $\Gamma(G)$ a set of interleavings for G. The consistency test succeeds and the feasibility test succeeds iff Γ is valid.*

Proof: ($\Rightarrow$) This is true by Lemma 6.1, because a visit-trace tuple for an entire tree is a computation sequence for the tree.

($\Leftarrow$) Since G has no useless nonterminals, then for any production p, there exists a derivation tree T containing an instance of p. Let p be the production $X_0 \rightarrow X_1 \cdots X_m$ and α_0 a node of T such that $\rho(\alpha_0) = p$. Let α_i be the children of α_0. Then $X_i = \tau(\alpha_i)$ for $0 \leq i \leq k$. Assuming that Γ is valid, there exists a computation sequence $s(T)$ respecting Γ. For each vertex $p_i(X_j)$ of $G\Gamma(p)$, there is a corresponding ba-symbol, $(\alpha_j, p_i(X_j))$ in $s(T)$. Let $[p_{i1}(X_{j1}), p_{i2}(X_{j2})]$ be an edge in $G\Gamma(p)$. This edge is in $G\Gamma(p)$ for one of three reasons:

- An attribute occurrence in $p_{i2}(X_{j2})$ depends on an attribute occurrence in $p_{i1}(X_{j1})$ for some attribution equation of p.
- Elements $p_{i1}(X_{j1})$ and $p_{i2}(X_{j2})$ occur consecutively in some $\Gamma(X_0, X_i)$ (so $i = j1$ or $i = j2$).
- Elements $p_{i1}(X_{j1})$ and $p_{i2}(X_{j2})$ occur consecutively in some $\Pi(X_i)$ (so $i = j1 = j2$ and $i2 = i1 + 1$).

In all three cases, $(\alpha_{j1}, p_{i1}(X_{j1}))$ must occur before $(\alpha_{j2}, p_{i2}(X_{j2}))$ in $s(T)$, in the first case, because $s(T)$ is feasible, and in the other two cases, because $s(T)$ respects Γ. Since $s(T)$ is a string of ba-symbols with a fixed order, $G\Gamma(p)$ cannot have a cycle, and so the feasibility test succeeds.

For any X_1, X_2, and X_3 with $X_1 \overset{1}{\rightsquigarrow} X_2$ and $X_2 \rightsquigarrow X_3$, there exists a derivation tree T with at least three nodes α_1, α_2, and α_3, such that $\alpha_1 \overset{1}{\rightsquigarrow} \alpha_2 \rightsquigarrow \alpha_3$, and $\tau(\alpha_i) = X_i$. Consider the graph $G\Gamma(X_1, X_2, X_3)$. For each vertex $p_i(X_j)$ in $G\Gamma(X_1, X_2, X_3)$, there is a corresponding ba-symbol $(\alpha_j, p_i(X_j))$ in $s(T)$. For each edge $[p_{i1}(X_{j1}), p_{i2}(X_{j2})]$ in $G\Gamma(X_1, X_2, X_3)$, the ba-symbol corresponding to

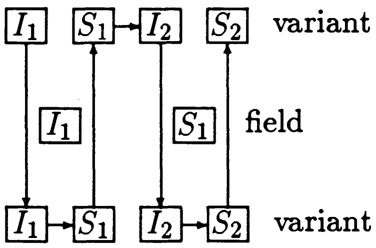


Figure 6.7: The $G\Gamma(\mathit{variant}, \mathit{field}, \mathit{variant})$ graph (with some missing edges)

$p_{i1}(X_{j1})$ must occur before the ba-symbol corresponding to $p_{i2}(X_{j2})$ because $s(T)$ respects Γ, and all edges in $G\Gamma(X_1, X_2, X_3)$ connect consecutive elements of some interleaving. Since $s(T)$ is a string of ba-symbols with a fixed order, and since the choice of X_1, X_2, and X_3 was arbitrary, then the consistency test succeeds. □

If we return to our example in section 6.4.4, we see that the consistency test fails for $G\Gamma(\mathit{variant}, \mathit{field}, \mathit{variant})$. This graph is shown in Figure 6.7 with some of its edges missing. We can make an immediate observation from the figure, even without adding the other edges. Suppose α and β are two nodes in a tree such that $\alpha \rightsquigarrow \beta$ and $\tau(\alpha) = \tau(\beta) = \mathit{variant}$. Then during each of the two visits to α, we must eventually visit β, and all the nodes between α and β must also be visited at least twice. But *field* only has two partitions, and so it is only visited once. Therefore, the graph will be inconsistent for any interleavings $\Gamma(\mathit{variant}, \mathit{field})$ or $\Gamma(\mathit{field}, \mathit{variant})$. In this example, we can modify $\Gamma(\mathit{variant}, \mathit{variant})$, and it turns out that with a different interleaving, the graph is acyclic. Unfortunately, this change causes another interleaving graph to be inconsistent.

We must add some empty partitions to *field* and *decl* so that they are visited twice. This results in transforming the grammar into a two-pass attribute grammar, and a multi-pass grammar is always GPAG.

6.6 Determining Whether a Partitioning is Valid

We also use a modified feasibility test to determine whether a set of partitions is valid. If p is a production, then let $G\Pi(p)$ to be the same graph as $G\Gamma(p)$, except that no edges are added to the graph between consecutive elements of interleavings for pairs of nonterminals in production p. We have the following result:

Corollary 6.1 *Let G be an attribute grammar and $\Pi(G)$ a set of partitions for G. Π is valid iff $\forall p$, $G\Pi(p)$ is acyclic.*

Proof: This follows from the proof for Theorem 6.1 if we omit all references to interleavings and the consistency test. □

This corollary is one of the theorems found in [EF80], and is the basis for the test used by Kastens in determining whether an attribute grammar is ordered.

Chapter 7

Asynchronous Subtree Replacements

Despite the use of an efficient algorithm for propagation after multiple subtree replacements, incremental evaluation may still take excessive time, say if AFFECTED is large or the attribution functions are complicated. Accordingly, we might wish to allow new subtree replacements before propagation has completed after a previous change. In addition, as we mentioned in Chapter 4, if we were to allow multiple users to edit a single object, we would want one user to be able to make changes without waiting for propagation to terminate after another user's change. Finally, for systems that create nonlocal dependencies during evaluation, if an attribute $\alpha.a$ affects a distant attribute $\beta.b$ (via a nonlocal dependency edge), we can handle the change as if a new, asynchronous subtree replacement had been made at the defining node of $\beta.b$.

In this chapter we consider trees undergoing asynchronous subtree replacements. As usual, our goal is to derive an efficient algorithm for restoring consistency to a derivation tree without ever evaluating attributes outside INFLUENCED, and without evaluating attributes more than once. Of course, without knowledge of future changes, we may need to reevaluate an attribute more than once if subsequent subtree replacements make the attribute inconsistent again. In addition, since each new subtree replacement can cause attributes to be added to and removed from the INFLUENCED and AFFECTED sets, our notion of efficiency will have to take into account these changing sets.

Some of the dynamic algorithms in Chapter 6 can be extended to support asynchronous subtree replacements. For the most part, they suffer from the usual drawbacks, evaluating attributes outside AFFECTED or evaluating attributes unnecessarily. An exception is the evaluator partially sketched by Geitz [Gei86] that extends the Reps-Marceau-Teitelbaum algorithm. In this chapter, though, we only consider tree-walking evaluators for asynchronous subtree replacements. We begin by describing the problem of asynchronous subtree replacements. Then we present a simple extension to *GPAG-evaluate* that handles asynchronous changes.

This algorithm will not be especially efficient, but will show a general approach. Afterwards, we discuss possible changes to improve the efficiency of the algorithm, developing a new algorithm, *ASYNCH-evaluate*. Finally, we note that the naive algorithm is as efficient as *ASYNCH_evaluate* when asynchronous changes result only from the use of nonlocal dependencies.

7.1 A Model for Asynchronous Subtree Replacements

The basic editing operation for asynchronous subtree replacements is still subtree replacement, as defined in Chapter 2, but we must assume that changes do not occur too frequently, or else propagation will never terminate and the tree will always be inconsistent. We must also deal with two additional concerns. First, if we allow subtree replacements when an evaluator is running, we must be careful about timing to be sure that the evaluator can be interrupted and restarted. Second, if editing and reattribution are concurrent, then the subtree that is being removed may contain inconsistent attributes.

We handle the first concern by defining a critical region within the code of an evaluator. A dynamic evaluator has a loop that selects an inconsistent attribute and reevaluates it. A static evaluator contains a loop that interprets instructions of a plan. In both cases, we treat the body of the loop as a critical region, only allowing a new subtree replacement to be made at the beginning of the loop.

We could relax this restriction somewhat: since attribution functions do not normally have side effects, we could abort the computation if we are in the middle of computing a new value for an attribute. But once a new value has been stored in a node, we must atomically mark the successors of the attribute, ensuring that the evaluator itself is in a consistent state before allowing another subtree replacement.

The second concern is handled by allowing free-standing subtrees to contain inconsistent attributes. These subtrees will result from cutting an inconsistent subtree from a larger derivation tree. Nevertheless, if the subtree is ever linked back into the derivation tree, we must be able to find these inconsistent attributes so that they can be reevaluated. To avoid having to search an entire subtree looking for marked attributes, we save information with each free-standing subtree that can lead us to the inconsistent attributes. For dynamic evaluators, a set of inconsistent but ready attributes can be saved with a subtree. For the tree-walking evaluators described in this chapter, we maintain the structure tree, which contains all the marked nodes in the subtree.

7.2 Measuring the Efficiency of an Asynchronous Subtree Replacement Algorithm

In order to measure the efficiency of an asynchronous algorithm, we must define AFFECTED so that it takes into account the changes made to a derivation tree by a sequence of subtree replacements. In Chapter 2, we defined AFFECTED as the set of inconsistent attributes at the time a subtree replacement (or replacements) has been made. With asynchronous subtree replacements, however, AFFECTED keeps changing. Let $r_1, r_2, \ldots$ be a sequence of subtree replacements, and let T_0 be an initial, consistent derivation tree. At time t_1, we make subtree replacement r_1, generating tree T_1. T_1 will not be consistent, so we begin propagation, continuing until just before time t_2, when the attributed tree will be T_1', the same derivation tree as T_1, but with some attribute instances having different values. In addition, T_1' may not necessarily be consistent if propagation has not completed. At time t_2, we make subtree replacement r_2, thereby creating tree T_2. We continue in this fashion, alternately making subtree replacements and reevaluating attributes. Because we require that changes not be made too often, then eventually, there will be enough time between t_m and t_{m+1} to restore consistency to T_m. Once consistency is restored to T_m, subsequent changes are not affected by the timing of the previous changes. Hence, the sequence of subtree replacements r_i (and replacement times t_i) can be split into subsequences such that consistency is restored to the tree after the final change of each subsequence, and so in the rest of the chapter, we assume a sequence of m asynchronous subtree replacements, $r_1, \ldots, r_m$.

Immediately after each subtree replacement r_i, there is a set of attributes, AFFECTED_i, whose values must change for T_i to be consistent. Immediately before time t_{i+1}, there is a set $\text{AFFECTED}_i'$ containing attributes in T_i' whose values must change for T_i' to be consistent. We let $A_i = |\text{AFFECTED}_i|$ and $A_i' = |\text{AFFECTED}_i'|$. Note that AFFECTED_i and $\text{AFFECTED}_i'$ depend not only on modification r_i, but also on previous modifications and evaluations. It might seem reasonable to hope that after modification r_m, an optimal algorithm could make T_m consistent in $O(A_m)$ time. Unfortunately, this is not possible, as shown by a simple example.

Suppose we have a derivation tree containing a path of nodes that have no synthesized attributes. If a subtree replacement is made affecting only the attributes of nodes along this path, we will need to visit all the nodes along the path until we reach its lower end. At this time, the tree will be consistent, but evaluation will not be complete, because the final SUSPEND instruction of the plans will not have been interpreted, and so the nodes along the path will still be marked. Now, let a second subtree replacement be made that affects a single attribute (i.e., $A_2 = 1$). We will not be able to finish propagation in $O(A_2)$ time, because we must still unmark all the nodes that were marked during propagation after the first subtree replacement, and this unmarking requires time proportional to the length of the path. Therefore, we cannot measure the efficiency of an algorithm relative to AFFECTED_i, but instead must consider the total time used by the evaluator to restore consistency to the derivation tree as it is modified by the entire sequence of changes.

The total time needed will depend not only on the subtree replacements but also on the elapsed time between consecutive subtree replacements. In order to measure the time between replacements, let c_i be the actual number of attributes in INFLUENCED that are reevaluated between t_i and t_{i+1}. Since T_m is restored to consistency, then $c_m = |\text{INFLUENCED}_m|$.[1] Let $C_i = \Sigma_{j=1}^{i} c_j$. Then the total number of evaluations performed is C_m.

Definition 7.1 *An algorithm for incremental propagation after multiple asynchronous subtree replacements is* optimal *if it requires $O(C_m)$ bookkeeping time. Furthermore, for any i, an optimal evaluator will require $O(C_i)$ bookkeeping time to evaluate C_i attributes.*

For example, suppose we make a subtree replacement and begin propagation. After a while, we will have evaluated c_1 attributes. Now, if a second change is made that restores the original derivation tree, then the c_1 attributes will have to be reevaluated, restoring them to their original values. Any remaining attributes in AFFECTED_1 will already be consistent, so they should not be evaluated at all. Therefore, an evaluator will be optimal if it requires $O(c_1 + c_2) = O(c_1)$ bookkeeping time.

A more informal way to look at efficiency, which lumps bookkeeping time and function evaluation time together, is to consider the effect a new subtree replacement has on the time needed to restore consistency to the tree, assuming no further change occurs until the tree is consistent again. At time t_i, just before subtree replacement r_i, let s_{i-1} be the time that would be needed to finish propagation if no additional changes were made. After change r_i, the minimum time needed by any evaluator to restore consistency is $s_{i-1} + O(\Delta A_i)$, where $\Delta A_i = A_i - A'_{i-1}$. Of course, ΔA_i can be negative, so a subsequent subtree replacement can result in propagation finishing sooner that it would have if the i-th change had not been made. Returning to the first example of this section, involving the tree containing a path of nodes with no synthesized attributes, once the tree is consistent, we still need time $s_1 = O(A_1)$ to unmark the nodes. After change r_2, which makes a single attribute inconsistent, $\Delta A_2 = A_2 - A'_1 = 1 - 0 = 1$. As we saw, we will need $s_1 + O(1)$ time to restore consistency to T_2.

The algorithms described in the rest of this chapter use structure trees, so they are not optimal, but require at least $O(\log n)$ overhead for each attribute evaluation. Nevertheless, no optimal evaluators for the asynchronous subtree replacement problem are known to exist, and in fact, most existing evaluators have even higher overhead. We return to the subject of running time later in the chapter.

[1] INFLUENCED_i and $\text{INFLUENCED}'_i$ are defined analogously to AFFECTED_i and $\text{AFFECTED}'_i$. Because $O(|\text{AFFECTED}|) = O(|\text{INFLUENCED}|)$, we will normally use AFFECTED, except in this case where we are counting individual attribute evaluations.

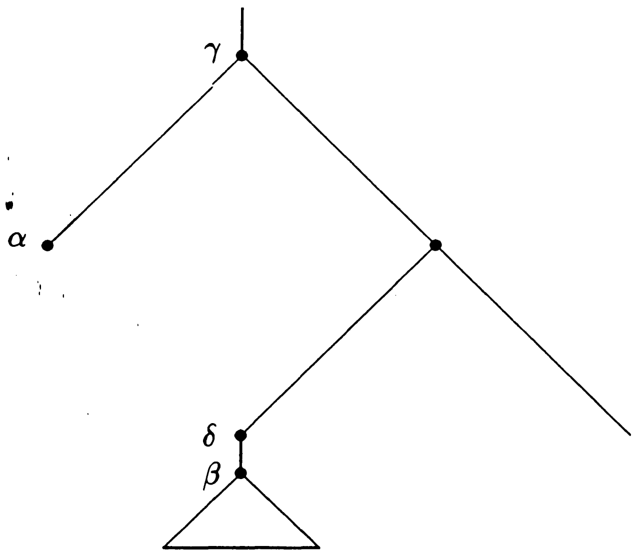


Figure 7.1: A skeleton of a possible derivation tree immediately after an asynchronous subtree replacement at node β

7.3 A Naive Algorithm

This section describes a naive algorithm, a simple extension to *GPAG-evaluate* that performs propagation after asynchronous subtree replacements. The naive algorithm is the basis for *ASYNCH_evaluate*, a more efficient algorithm for handling asynchronous subtree replacements which will be described in section 7.7. Suppose we are making a sequence of subtree replacements, and we wish to make replacement r_i at node β. Call the parent of β the *changed node*,[2] and call the node whose plan we are interpreting the *current node.* We will normally use α for the current node and δ for the changed node. For now, we assume that no attributes are inconsistent inside the subtrees being swapped, so α cannot be a descendant of δ. This assumption simplifies the discussion of both the naive algorithm and *ASYNCH_evaluate*, but is not a fundamental limitation. In section 7.9, we will see that the assumption is not required. The naive algorithm consists of the following steps:

1. Wait for interpretation of the current instruction to complete, since each instruction is an atomic action. Make the subtree replacement, marking the attributes at the point of subtree replacement and their defining nodes. Figure 7.1 shows part of a derivation tree after making the subtree replacement. In general, structure tree edges will exist in the tree, but the figure only suggests a potential arrangement of nodes.

[2]We do not call β the *changed node*, because *parent*(β) is the defining node for some attributes of β.

2. Find γ, the LCA of α and δ, and make γ the new current node. Restart *GPAG_evaluate* by interpreting the first instruction of the plan at γ, which will be an EVAL instruction for an instance of an attribute in the first inherited partition of $\tau(\gamma)$, i.e., $I_1(\tau(\gamma))$.

We claim that this algorithm will make the tree consistent again. After the first step, the resulting structure tree will contain the defining nodes for all inconsistent attributes. By restarting the evaluator at the beginning of PLAN(γ), we will eventually visit δ, so that attributes affected by change r_i can be reevaluated. Nodes marked before change r_i was made will still be visited as well, providing for the reevaluation of attributes that were already inconsistent. More importantly, the attribute instances in $I_1(\tau(\gamma))$ at γ are evaluated before any attribute instances of α or δ, so resuming evaluation at the beginning of PLAN(γ) is safe: we will be sure to reevaluate any inconsistent attributes in the proper order. Finally, the naive algorithm will not evaluate any attributes unnecessarily, because unmarked attributes are never reevaluated. As propagation continues, an attribute that was evaluated before change r_i may have to be evaluated again, but after change r_m, the final change in the sequence, the naive algorithm will only evaluate attributes in INFLUENCED$_m$.

The drawback of the naive algorithm, and the reason that it is not optimal, is that it visits a lot of nodes unnecessarily, and in the process, extra EVAL instructions are interpreted, even if only to check that the specified attribute is not marked. To understand why nodes are visited unnecessarily, note that during propagation after a given subtree replacement, a sequence of instructions $p_1, \ldots, p_n$ is interpreted by the naive algorithm, where p_1 is the first instruction interpreted after the subtree replacement and p_n is the final SUSPEND instruction for the last marked node.[3] Instruction p_1 is the first instruction of PLAN(γ). Let p_α be the instruction that would have been interpreted next at α (the current node) if change r_i had not been made, and let p_δ be PLAN(δ)[1], the first instruction that would be interpreted if r_i were the first change in the sequence. Let $\bar{p}$ be the earlier of p_α and p_δ in the instruction sequence. Naturally, instruction p_1 precedes $\bar{p}$, but the relative order of p_α and p_δ depends on the derivation tree and its dependencies.

By resuming the naive algorithm at instruction p_1, we reinterpret instructions between p_1 and $\bar{p}$. Unfortunately, no EVAL instruction in this range will encounter marked attributes, because these instructions were already interpreted before change r_i occurred, and the change cannot affect any attributes evaluated before p_δ. If p_δ comes before p_α, the naive algorithm may also reinterpret instructions between p_δ and p_α. These instructions will not perform attribute evaluations either, unless a changed value resulting from subtree replacement r_i propagates to attributes of ancestors of δ in the derivation tree.

[3] Of course, a new sequence of instructions will be defined if another subtree replacement is made.

For example, suppose two traversals are made along the path between γ and α in Figure 7.1. If p_α is interpreted during the second traversal, then making the first traversal requires moving from γ to α and back, interpreting instructions, but not evaluating any attributes. This path could be arbitrarily long, and so if m subtree replacements are made, nodes along the path between α and γ could be visited $O(m)$ times. In order to improve the efficiency of the naive algorithm, we should restart evaluation not at p_1, but at $\bar{p}$. We should also avoid reinterpreting unnecessary instructions between p_α and p_6.

7.4 Improving the Naive Algorithm

The naive algorithm is safe, but not efficient, because after every asynchronous subtree replacement, the evaluator "backs up" by changing the current node, ensuring that attributes are always evaluated in the proper order. In other words, the naive algorithm resumes propagation by reinterpreting some instructions, but too many instructions are reinterpreted. This section describes how *GPAG_evaluate* can be modified in a way that allows the naive algorithm to be more intelligent in determining whether and how far to back up. The modification is straightforward and only requires saving some additional information in marked nodes of the derivation tree. An example will show how this additional information can be used.

The first tree-walking evaluators described in this thesis maintained an instance of the *plan_index* variable with each node, either in the stack of a recursive function invocation or in the tree node itself. In Chapter 6, we saw that we could avoid saving individual *plan_index* values, because we could use *mapup* and *mapdown* functions to determine where to resume interpretation as we moved from node to node. Consequently, the only state maintained by *Incremental_K_evaluate* is whether a node is marked or not, which node is the current node, and what the *plan_index* is of the current node. Now, we once again choose to save *plan_index* values in the nodes, because these values will allow the naive algorithm to be more efficient.

During normal execution of *GPAG_evaluate* (that is, without the presence of asynchronous subtree replacements), the *plan_index* for a node α, written $pi(\alpha)$, is the index in PLAN(α) of the most recent VISIT or SUSPEND instruction to have been interpreted at α, or of the instruction about to be interpreted, if α is the current node.[4] The *plan_index* value for a node will be defined only if the node is marked, and it will always be initialized upon marking the node. Therefore, we do not increase the complexity of an incremental algorithm by saving the value (although we must use slightly more time and space).

For a given node α, PLAN(α)[$pi(\alpha)$] is the *current instruction* of α, which we write as $ci(\alpha)$. By examining the current instruction of a node β, we can infer

[4]Recall that the main loop of the evaluator is a critical section, and so we will only examine $pi(\alpha)$ at the beginning of the loop.

how evaluation is progressing with respect to β. For example, if $ci(\beta)$ is a VISIT instruction, the evaluator must be at a node inside the subtree rooted at β. If VISIT(i, s) occurs before $ci(\beta)$, we must have already completed (or skipped) the s-th visit to child i, and if VISIT$(j, 1)$ comes after $ci(\beta)$, then child j has not been visited yet.

At first glance, the saved *plan_index* values appear to be redundant, but the information is valuable. By updating the *plan_indexes* of some nodes, we can make it appear that the evaluator is in two or more different states at a time. To provide some intuition into how saved *plan_index* values can be used, we look at a running example. In section 7.5, we will define *plan_index* values more precisely.

7.4.1 Determining whether to backup

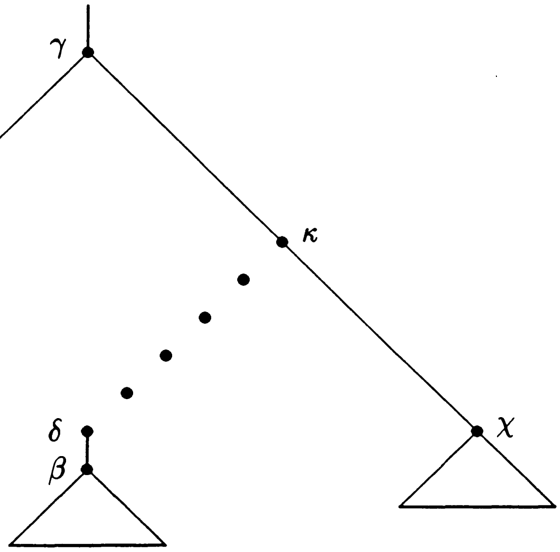
The example used in the rest of this section examines the first three changes in a sequence of asynchronous subtree replacements. The first change in any sequence does not need any special treatment, because it can be considered as a simple, synchronous subtree replacement. We only need the additional power of an asynchronous subtree replacement algorithm after the second subtree replacement has been made. Therefore, we assume that we started the modified *GPAG_evaluate* after the first subtree replacement, so *plan_index* values are being maintained in the marked nodes. Figure 7.2 shows a possible tree just after making subtree replacement r_2, the second change in a sequence of asynchronous subtree replacements, at node β. Instructions from a hypothetical PLAN(κ) are shown as well. The figure indicates that nodes between γ and α and between γ and χ are already marked. The dotted edge between κ and δ is a structure tree edge, added to make the replaced subtree part of the existing structure tree. We assume that node κ was already marked, so that $ci(\kappa)$ is defined.

To be efficient, we need a simple way to decide whether to back up after the subtree replacement. If p_α comes before p_δ, we can continue interpreting PLAN(α). Eventually, we will visit δ, and the attributes affected by change r_2 will be evaluated. On the other hand, if p_δ comes before p_α, we must make δ the new current node so that attributes affected by the new change will be reevaluated. Therefore, deciding whether to back up is equivalent to determining whether p_δ or p_α is first in the instruction sequence. We will use $ci(\kappa)$ to make this determination, by discovering where $ci(\kappa)$ fits into the sequence of instructions.

Since α is not a descendant of κ, the last instruction interpreted at κ (i.e., $ci(\kappa)$) must be a SUSPEND instruction. Furthermore, since α is the current node and p_α is $ci(\alpha)$, then $ci(\kappa) < p_\alpha$.[5] On the other hand, any instruction following $ci(\kappa)$ in PLAN(κ) must follow p_α in the instruction sequence.

Since δ is a structure tree child of κ, there must be a VISIT instruction in PLAN(κ) that precedes p_δ. If we were to interpret this VISIT instruction, we

[5]The notation $p_i < p_j$ means that instruction p_i precedes p_j in the instruction sequence.



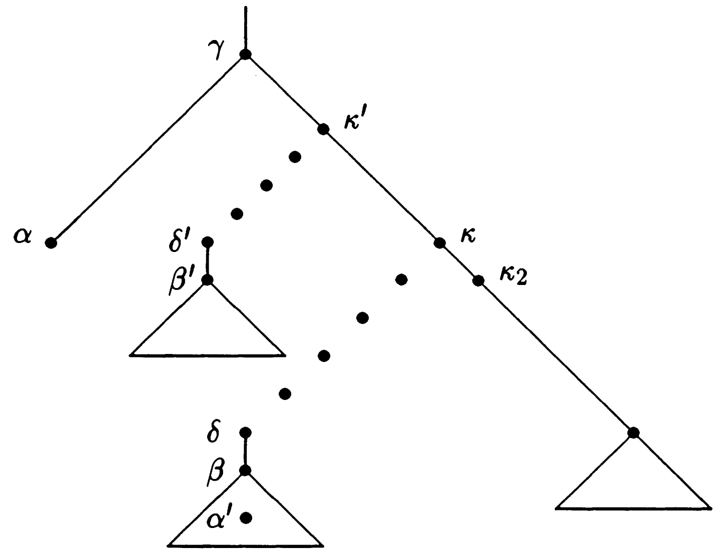
Some instructions from PLAN(κ):
...VISIT(2,1)...SUSPEND(1)...VISIT(1,1)...VISIT(2,2)...SUSPEND(2)...
VISIT(1,2)...VISIT(2,3)...SUSPEND(3)

Figure 7.2: Using *plan_index* values to determine whether to back up

would make a nonlocal visit to δ. Examining PLAN(κ) in Figure 7.2, there are two possible instructions that could result in a nonlocal visit to δ: VISIT(1,1) or VISIT(1,2). By checking the *mapto* function, we can see which VISIT instruction would make the first nonlocal visit to δ.

Suppose $ci(\kappa)$ is SUSPEND(2) and VISIT(1,2) is the only VISIT instruction in PLAN(κ) requiring a nonlocal visit to δ. Then $p_\alpha <$ SUSPEND(2) $<$ VISIT(1,2) $< p_\delta$, and we can see that we do not have to change the current node. Propagation can resume by interpreting p_α. For this example, however, we will assume that VISIT(1,1) requires making a nonlocal visit to δ. Therefore, VISIT(1,1) $< p_\delta <$ SUSPEND(2) $< p_\alpha$. In this case, we must back up, making δ the new current node and restarting the evaluator by interpreting p_δ.

Deciding whether to back up after the second change of any sequence of changes is easy, because we only have to examine the *plan_index* and the plan of a single node. Unfortunately, once we back up after any change, the *plan_index* values will no longer reflect the state of the evaluator. For example, since we backed up after change r_2, the new current node will be δ, a descendant of each node between γ and κ. Nevertheless, the current instructions of the nodes between γ and κ will still be SUSPEND instructions, and so for subsequent subtree replacements, we will not be able to use $ci(\kappa)$ directly to determine whether $p_\alpha < p_\delta$.



Some instructions from PLAN(κ):
...VISIT(2,1)...SUSPEND(1)...VISIT(1,1)...VISIT(2,2)...SUSPEND(2)...
VISIT(1,2)...VISIT(2,3)...SUSPEND(3)

Figure 7.3: Tree after third subtree replacement in sequence

To see the difficulty that this presents, assume that propagation continues after change r_2 and that the evaluator is visiting a descendant of δ when change r_3 occurs at node β', as illustrated by Figure 7.3. The current node is now α', but it is no longer true that $ci(\kappa') < p_{\alpha'}$. To accurately reflect the location of α', $ci(\kappa')$ should be a VISIT instruction, not a SUSPEND instruction. In section 7.7, we will see that we need to examine *plan_index* values of other nodes in order to decide whether to back up. For now, so that we can continue with the example, assume that $p_{\alpha'} < p_{\delta'}$, so propagation continues at $p_{\alpha'}$.

7.4.2 Skipping visits to marked nodes

We can also use the *plan_index* values to avoid visiting marked nodes under some circumstances. We already assumed that after change r_3, α' remains the current node. Eventually, the evaluator will move to δ, and if change r_2 does not require marking any nodes between δ and κ, then the evaluator will make a nonlocal suspend, moving from δ to κ. The next non-EVAL instruction in PLAN(κ) is VISIT(2, 2), specifying a visit to κ_2, the second child of κ. Normally, *GPAG_evaluate* visits marked nodes, but in this case, the visit will not perform any evaluations, because no attributes evaluated during the second visit to κ_2 can be inconsistent. (If change r_3 had affected an attribute evaluated during the second visit to κ_2, we would have backed up after change r_3, restarting the evaluator at δ', not at α'.)

The second visit to κ_2 must have been completed before change r_2 occurred, so $ci(\kappa_2)$ is SUSPEND(2). By checking $ci(\kappa_2)$ before making the visit, we can see that the visit has already been made and can be skipped this time. Since the subtree below κ_2 could be arbitrarily large, we avoid wasting time by skipping the visit to κ_2. Eventually, as propagation continues, VISIT(2, 3) in PLAN(κ) will be interpreted. Once again, $ci(\kappa_2)$ will be examined, but this time, it will indicate that the third visit to κ_2 has not been made, and so the evaluator will visit κ_2.

Of course, changed attribute values could propagate from δ up to κ, affecting attributes evaluated during the second visit to κ_2. If this is the case, we will have to modify $pi(\kappa_2)$ during propagation, ensuring that the second visit to κ_2 is repeated. In section 7.7, we will see how to modify *plan_index* values to ensure that visits are repeated if necessary.

7.5 Compatible *plan_index* Values

In the previous section, we used a version of *GPAG_evaluate* that maintains *plan_index* values in each derivation tree node. In this section, we use the same version of *GPAG_evaluate* to see how *plan_index* values of different nodes change during propagation after synchronous subtree replacements. These relationships are vital to the proper operation of *ASYNCH_evaluate*; we will be able to modify *plan_index* values in ways not normally done by *GPAG_evaluate*, while still using the values to determine how propagation is progressing.

During propagation by *GPAG_evaluate*, we can select two nodes $\alpha \overset{1}{\rightsquigarrow} \beta$ and use $ci(\alpha)$ to predict $ci(\beta)$ or *vice versa.* (Naturally, these predictions are only valid if both α and β are marked, because the current instruction of an unmarked node is undefined.) Since all marked nodes must have a meaningful *plan_index*, we assume that all plans have a dummy SUSPEND(0) as the 0-th instruction, and we set $pi(\alpha) = 0$ when α has been marked but has not been visited yet. If s visits to α have already been skipped when α is first marked, we assign a value to $pi(\alpha)$ so that $ci(\alpha)$ is SUSPEND(s), indicating that the s-th visit to α should not be made again, but the following visit will be needed.

Using the current instruction of one node to predict the current instruction of another node is guaranteed to succeed only when no asynchronous subtree replacements have occurred. More precisely, given i and j, we call PLAN(α)[i] *compatible* with respect to PLAN(β)[j] if *GPAG_evaluate* could reach a state during evaluation such that $pi(\alpha) = i$ and $pi(\beta) = j$. The possible relationships between the current instructions of any two adjacent nodes are shown in Figure 7.4. Similar predictions can be made for a node β that is a structure tree child or parent of α.

$ci(\alpha)$	Possible compatible values for $ci(\beta)$	
	α is the i-th child of β	β is the j-th child of α
VISIT(i', s)	$ci(\beta)$ = VISIT(i, s') where $s' - 1$ SUSPEND instructions precede $ci(\alpha)$	if $i' = j$, $ci(\beta)$ is between SUSPEND($s - 1$) and SUSPEND(s); otherwise, same as for SUSPEND(s)
SUSPEND(0)	$ci(\beta)$ precedes VISIT(i, 1)	$ci(\beta)$ precedes SUSPEND(1)
SUSPEND(s)	$ci(\beta)$ is between VISIT(i, s) and VISIT(i, $s + 1$)	$ci(\beta)$ = SUSPEND(s') where s' VISIT(j,?) instructions precede $ci(\alpha)$
EVAL(i'', a)	same as for VISIT(i', s)	same as for SUSPEND(s)

Figure 7.4: Predicted current instruction of a node adjacent to α

If two instructions are compatible, we write PLAN(α)[i] $\Longleftrightarrow$ PLAN(β)[j].[6] Examining the relationships in the figure closely, three observations are apparent. First, $x \Longleftrightarrow y$ implies that $y \Longleftrightarrow x$. Second, given i, there is not necessarily a unique j such that PLAN(α)[i] $\Longleftrightarrow$ PLAN(β)[j]; in some cases, any of several consecutive values of j may make PLAN(β)[j] and PLAN(α)[i] compatible. Third, the current instruction of any marked node between the current node and the root of the tree is a VISIT instruction, and the current instruction of all other nodes (except the current node) is a SUSPEND instruction. These properties allow *Incremental_K_evaluate* to work without saving multiple *plan_index* values, because whenever we move from a node α to an adjacent node β, we can use $pi(\alpha)$, along with *mapdown* or *mapup*, to initialize $pi(\beta)$.

Since a tree-walking evaluator interprets instructions in a fixed sequence, then in theory, for any nodes α and β, we should be able to determine whether PLAN(α)[i] comes before PLAN(β)[j]. An incremental evaluator does not interpret all instructions in the sequence, however, because visits to unmarked nodes are skipped.

Definition 7.2 *Let α and β be any two marked nodes in a derivation tree. We say that PLAN(α)[i]* reaches *PLAN(β)[j] if*

- *PLAN(α)[i] $\Longleftrightarrow$ PLAN(β)[j]; or*
- *an incremental evaluator interpreting PLAN(α)[i] will eventually interpret PLAN(β)[j], assuming that no additional nodes are marked.*[7]

[6] We will also read "PLAN(α)[i] $\Longleftrightarrow$ PLAN(β)[j]" as "PLAN(α)[i] and PLAN(β)[j] are compatible."

[7] No additional nodes will be marked if no attributes receive new values when EVAL instruction are interpreted.

If PLAN(α)[i] reaches PLAN(β)[j], we write PLAN(α)[i] $\Rightarrow$ PLAN(β)[j]. We also say that PLAN(β)[j] is *reachable* if the current instruction of the current node reaches PLAN(β)[j]. This definition allows us to specify an invariant for any tree-walking evaluator.

Tree-Walking Evaluator Invariant (TW-invariant): The EVAL instruction for any inconsistent attribute instance in a derivation tree must be reachable.

The definition for "reaches" is based on the current set of marked nodes. For example, if $\alpha \overset{1}{\rightsquigarrow} \beta \overset{1}{\rightsquigarrow} \gamma$, but β is not marked, then $ci(\alpha) \not\Rightarrow ci(\gamma)$ for any values of $pi(\alpha)$ and $pi(\gamma)$, since all VISITs from α to β will be skipped. In general, not all attributes in AFFECTED are reachable, because not all attributes in AFFECTED are inconsistent at any one time. Thus, when a tree-walking evaluator reevaluates an attribute and finds that its value has changed, it must mark the defining nodes of the changed attribute's successors, restoring the TW-invariant.

During execution of *GPAG_evaluate*, then by definition, the current instruction of each node is always compatible with adjacent nodes, and EVAL instructions of inconsistent attributes always follow the current instruction of the current node, so the TW-invariant is always satisfied. During execution of an asynchronous subtree replacement evaluator, however, current instructions of adjacent nodes may not necessarily be compatible, because we may have needed to back up. Therefore, when moving from α to β, we will use $pi(\beta)$, rather than *mapdown* or *mapup*, to decide which instruction to interpret next at β.

7.6 Incompatible *plan_index* Values

When *GPAG_evaluate* is used after synchronous subtree replacements, *plan_index* values are always compatible. An asynchronous subtree replacement, though, may require backing up, which can make some *plan_indexes* incompatible. An efficient evaluator supporting asynchronous subtree replacements will need to allow incompatible *plan_index* values.

Backing up is performed by *resetting* a node, that is, by reducing the value of its *plan_index*. If we reset α, then in all likelihood, $ci(\alpha)$ will no longer be compatible with its parent and children. While we could make the current instructions of all nodes compatible after each asynchronous change, to do so would take too much time. Instead, after each subtree replacement, we only reset some of the *plan_index* values, allowing other values to remain incompatible.

Plan_index values, whether compatible or not, are used to direct movement of the evaluator through the derivation tree, specifying which plan instructions should be interpreted. Naturally, we want to be sure that we interpret EVAL instructions for all inconsistent attributes. Therefore, whenever an attribute instance $\alpha.a$ is marked, we must reset α so that the EVAL instruction for $\alpha.a$ will be interpreted.

Plan_index values are increased when instructions are interpreted, but resetting always involves decreasing a *plan_index*. Although *ASYNCH_evaluate* will allow incompatible *plan_index* values, nodes cannot be reset arbitrarily; we require that the following invariant be maintained:

Asynchronous Evaluator Invariant (AE-invariant): If α is the current node and β is any other marked node then $ci(\alpha) \Rightarrow ci(\beta)$.

The AE-invariant is stronger than the TW-invariant, because the TW-invariant does not mention current instructions of nodes. An evaluator can maintain the TW-invariant while ignoring the *plan_index* values.

7.6.1 Using incompatible *plan_index* values

Incompatible *plan_index* values can arise whenever the current node is changed in preparation for resuming propagation after an asynchronous subtree replacement. After a change, we may have to restart propagation at a new current node α', possibly making $ci(\alpha)$ unreachable from $ci(\alpha')$. We must restore the AE-invariant by resetting other nodes.

At times, we will need to use the *plan_index* of a node to determine what the *plan_index* of an adjacent node would be in order to make the current instructions of the two nodes compatible. For a node α, *compatible_parent_index*(α, i) returns a value pi such that $\text{PLAN}(\alpha)[i] \Leftrightarrow \text{PLAN}(parent(\alpha))[pi]$. A similar function, *compatible_child_index*(α, i, j), returns the value pi such that $\text{PLAN}(\alpha)[i] \Leftrightarrow \text{PLAN}(\alpha_j)[pi]$, where α_j is the j-th child of α. These functions can be precomputed easily during analysis of the attribute grammar, and are derived from the relationships in Figure 7.4. Similar functions can also be defined relating the *plan_indexes* of structure tree parents and children.

7.7 An Algorithm for Asynchronous Subtree Replacements

We can now describe *ASYNCH_evaluate*, an efficient algorithm for propagation after asynchronous subtree replacements. Like the naive algorithm, this algorithm has two phases. During the first phase, the subtree replacement is performed, a potentially new current node is found, and the *plan_indexes* of some nodes may be reset. The second phase performs the reevaluations, skipping unnecessary visits.

7.7.1 ASYNCH_evaluate—Phase I

When we are ready to make a new asynchronous subtree replacement, we must wait until the current instruction has been completely interpreted by the evaluator, because we consider each instruction to be an atomic action. After completion of the instruction, we interrupt the evaluator and allow the subtree replacement to be made. The first phase of the algorithm makes the subtree replacement and determines whether the current node should be changed. If the current node changes, the AE-invariant is reestablished, if necessary. In describing the algorithm, we simplify matters by ignoring structure trees for the most part. Details of some boundary conditions are ignored as well.

Phase I is invoked to make a subtree replacement, and its operation is based on the following information:

T : the current attributed tree (containing a structure tree)
α : the current node (along with its *plan_index* and current instruction)
T_{OLD} : the subtree of T, rooted at β', that will be removed as part of the subtree replacement
T_{NEW} : the new subtree, rooted at β, that replaces T_{OLD}

When Phase I has completed, the derivation tree will be updated and the current node may have been changed. That is, we have the following output:

T : the updated tree, with T_{OLD} replaced by T_{NEW}, and with some *plan_index* values reset
α' : the (possibly new) current node (along with its updated *plan_index* and current instruction)

For now, we assume that no nodes in T_{OLD} or T_{NEW} are marked, including β and β'. In section 7.9, we will show that this assumption is not critical to the proper operation of the algorithm. The first phase consists of the following primary steps:

1. Make the subtree replacement. Mark the defining nodes of the attributes at the point of the subtree replacement.

2. Determine whether the current node should change; α' will be the new current node, whether it changes or not.

3. Reestablish the AE-invariant, by resetting nodes, if necessary.

When Phase I is complete, the TW- and AE-invariants will be satisfied, and Phase II will be able to perform propagation.

Even though three separate steps are listed for Phase I, the code, shown in Figure 7.5, overlaps the steps somewhat. Since we assume that the subtree being replaced has no marked nodes, we can perform the subtree replacement in the usual way. After the change, we replace the synthesized attributes of β with those of β', making the attributes of β the only attributes that are inconsistent, at least as a result of this new change. Then we mark the attributes of β, and also their defining nodes (i.e., β and the parent of β). Of course, marking nodes is no longer so simple, since we must initialize *plan_indexes* at the same time.

To determine whether we should change the current node, we must first find the *connecting node*, the nearest ancestor of β that was marked before we made

```
Mark attributes in A(β) and mark node β;
pi(β) := index of first EVAL in PLAN(β) for attributes in A(β);
δ := parent(β);
pi1 := min(compatible_parent_index(β, pi(β)),
            index of first EVAL in PLAN(δ) for attributes in A(β));
if δ is marked then
     κ := δ;
else
     Mark δ;
     pi(δ) := pi1;                                  /* Initialize pi(δ). */
     κ := parent(δ);                                /* or st_parent(δ) */
     pi1 := compatible_parent_index(δ, pi(δ));
end
if pi(κ) < pi1 then
     /* By the AE-invariant, ci(α) ⇒ ci(κ). If pi(κ) < pi1, then the current
          node does not have to be changed. We can continue with Phase II.
          */
else pi(κ) := pi1;               /* Reset κ, ensuring that ci(δ) ⇒ ci(κ). */
```

Figure 7.5: Phase I (Part 1)

the subtree replacement. Let κ denote this node, and let δ be the changed node, i.e., the parent of β. (This is the same notation used in Figure 7.2.) If δ is already marked, then $\kappa = \delta$. Otherwise, κ is the parent of δ, and we may have to reset κ.[8] Phase I will initialize or reset the *plan_indexes* of up to three nodes: β, δ, and κ. In the process, the algorithm can check whether the AE-invariant has been invalidated. If not, we will be able to proceed directly with Phase II.

When deciding whether to reset κ, we compare $pi(\kappa)$ to the value that would be needed to make $ci(\kappa)$ compatible with $ci(\delta)$. In other words, if $\delta \neq \kappa$, we compare $compatible\_parent\_index(\delta, pi(\delta))$ to $pi(\kappa)$. If $pi(\kappa)$ is smaller, then $ci(\kappa) \Rightarrow ci(\delta)$. By the AE-invariant, we know that $ci(\alpha) \Rightarrow ci(\kappa)$, and so $ci(\alpha) \Rightarrow ci(\delta)$. In this case, we will not have to back up, but can continue with Phase II immediately.

If $pi(\kappa) > compatible\_parent\_index(\delta, pi(\delta))$, we reset κ, so that $ci(\kappa)$ will be a visit to δ (or to β, if $\delta = \kappa$). If the current instructions of all nodes were compatible with each other before the change, we could let κ be the new current node; as we saw earlier, this will be the case for the second change in any sequence of asynchronous subtree replacements. Unfortunately, $ci(\alpha)$ and $ci(\kappa)$ may not be compatible, and so we cannot be sure that $ci(\delta) \Rightarrow ci(\alpha)$. We must check the *plan_index* values of more nodes to determine whether the current node should be changed.

[8] Actually, κ is the structure tree parent of δ, but we are ignoring structure trees for now. Furthermore, if this is the first subtree replacement in the sequence, κ is undefined, and we skip parts of the algorithm having to do with κ.

1. $\gamma := LCA(\alpha, \kappa)$;
2. $pi1 :=$ compatible index with respect to γ and α;†
3. $pi2 :=$ compatible index with respect to γ and κ;†
4. /* We use these values to determine whether $ci(\alpha) < ci(\kappa)$. */
5. **if** $pi1 < pi2$ **then**
6. $\quad\alpha' := \alpha$; /* No change to current node */
7. $\quad$/* By the AE invariant, $ci(\alpha) \Rightarrow ci(\gamma) \Rightarrow ci(parent(\kappa))$. Because we had to reset κ, we cannot be sure that $ci(parent(\kappa)) \Rightarrow ci(\kappa)$. So we make sure that $ci(\gamma) \Rightarrow ci(\kappa)$ by recursively resetting nodes $parent(\kappa)$, $parent(parent(\kappa)), \ldots$, until the invariant is restored. If we end up resetting γ, we repeat the process along the path from γ down to α. */
8. $\quad$Make $ci(\kappa)$ reachable from $ci(\gamma)$.
9. $\quad$**if** γ was reset **then**
10. $\quad\quad$Make $ci(\gamma)$ reachable from $ci(\alpha)$.
11. **else**
12. $\quad\alpha' := \kappa$; /* New current node */
13. $\quad$Make $ci(\alpha)$ reachable from $ci(\gamma)$. /* See comment on line 7. */
14. $\quad$**if** γ was reset **then**
15. $\quad\quad$Make $ci(\gamma)$ reachable from $ci(\kappa)$.
16. **end**

†These statements work as if there were structure tree edges from γ to both α and κ. By using the *mapfrom* function, we determine the value we would have to assign to $pi(\gamma)$ to make the nodes compatible.

Figure 7.6: Phase I (Part 2)

We use γ, the LCA of α and κ, to determine whether $ci(\alpha)$ precedes the new $ci(\kappa)$.[9] The *mapfrom* and *mapto* functions are used to find $pi1$ and $pi2$ such that $\text{PLAN}(\gamma)[pi1] \Leftrightarrow ci(\alpha)$ and $\text{PLAN}(\gamma)[pi2] \Leftrightarrow ci(\kappa)$. If $pi1 < pi2$, then the current node will not change, but we still must make sure that $ci(\alpha) \Rightarrow ci(\kappa)$, because κ was reset. On the other hand, if $pi2 < pi1$, we make κ the new current node, and we must ensure that $ci(\kappa) \Rightarrow ci(\alpha)$. Figure 7.6 shows the code for the rest of Phase I. Once we have determined the proper value for the current node and restored the AE-invariant, we can continue with Phase II.

7.7.2 ASYNCH_evaluate—Phase II

The second phase of *ASYNCH_evaluate* propagates changed attribute values. We restart the evaluator with $\alpha_0 = \alpha'$ and continue interpreting the plans, reevaluating marked attributes as necessary. Because we allow incompatible *plan_index* values, however, we must modify the actions used by *GPAG_evaluate*.

[9] If $\gamma = \alpha$ or $\gamma = \kappa$, this part of the algorithm can be simplified.

case PLAN(p)[plan_index(α_0)] **of**
EVAL(i, a):
  if $\alpha_i.a$ is marked **then** /* Only evaluate marked attributes.*/
    Unmark and reevaluate $\alpha_i.a$.
    if $\alpha_i.a$ has a new value **then**
      for each successor $\beta.b$ of $\alpha_i.a$ **do**
        Mark $\beta.b$.
        $\beta' :=$ *defining_node*($\beta.b$);
        /* Find index of EVAL($\beta.b$) instruction in PLAN(β'). */
        $pi1 :=$ *findEVAL*($\beta', \beta.b$);
        /* Make sure PLAN(β')[$pi1$] is reachable. */
        if β' is marked **then**
          $pi(\beta') :=$ **min**($pi(\beta'), pi1$); /* Reset β' if necessary. */
        else
          Mark β';
          $pi(\beta') := pi1$; /* Initialize $pi(\beta')$. */
        end
  plan_index(α_0) + +;

Figure 7.7: *ASYNCH-evaluate*: Action for EVAL instructions

EVAL instructions

The new action for EVAL instructions is shown in Figure 7.7. The primary feature distinguishing this action from previous EVAL actions is the fact that we not only must mark adjacent nodes when an attribute changes value, but must also update *plan_index* values for these nodes. When the value of an attribute changes, we mark all of its successors. To maintain the AE-invariant, we make sure that the EVAL instructions of these newly-marked attributes are reachable. So for each successor, if its EVAL instruction is not reachable, we reset its defining node's *plan_index*. This operation cannot invalidate the AE-invariant, because the EVAL instruction for any attribute b dependent on a changed attribute a must be reachable from the EVAL instruction for a.

VISIT and SUSPEND instructions

As usual, we skip VISITs and SUSPENDs to unmarked nodes. If a node is marked, however, its *plan_index* is already initialized, and so we do not need to use *mapdown* or *mapup* to determine where to start (or resume) interpreting the plan of the new node. Because of the new action for EVAL instructions, and if asynchronous changes are being made, then an adjacent node's *plan_index* will not necessarily be compatible. If we are interpreting a VISIT(i, s) instruction, but the s-th visit has already been completed, we can skip the visit, because no attributes would be evaluated during the visit. (If any attribute did need to be evaluated during

```
case PLAN(p)[plan_index(α_0)] of
VISIT(i, s):
    if α_i is marked then
        pi := plan_index(α_i);
        if pi = mapdown(ρ(α_i), s) − 1 then            /* ci(α_0) ⇐⇒ ci(α_i) */
            α_0 := α_i;                                   /* Move to child i. */
            plan_index(α_0) + +;              /* Move past SUSPEND(s − 1). */
        else if pi < mapdown(ρ(α_i), s) − 1 then
            abort;                 /* ci(α_0) ⇐/⇒ ci(α_i)—invariant violated */
        else if pi < mapdown(ρ(α_i), s + 1) − 1 then
            α_0 := α_i;                                   /* Move to child i. */
        else skip;              /* Visit s already complete—skip VISIT. */
    else skip;                     /* Node α_i not marked—skip VISIT. */
SUSPEND(s):
    if this is the final SUSPEND in PLAN(p) then
        Unmark α_0.
        if α_0 is the root of the tree or parent(α_0) is not marked
            then return;
    end;
    β := parent(α_0);                              /* α_0 cannot be the root. */
    if β is marked then
        pi := plan_index(β);
        j := child_no(α_0);
        if pi = mapup(ρ(β), j, s) − 1 then                /* ci(α_0) ⇐⇒ ci(β) */
            α_0 := β;                                     /* Move to parent. */
            plan_index(α_0) + +;                  /* Move past VISIT(j, s). */
        else if pi < mapup(ρ(β), j, s) − 1 then
            abort;                   /* ci(α_0) ⇐/⇒ ci(β)—invariant violated */
        else if pi < mapup(ρ(β), j, s + 1) − 1 then
            α_0 := β;                                     /* Move to parent. */
        else skip;   /* SUSPEND(s) already complete—skip SUSPEND. */
    else skip;                   /* Node β not marked—skip SUSPEND. */
```

Figure 7.8: *ASYNCH-evaluate*: Actions for VISIT and SUSPEND instructions

the s-th visit, the AE-invariant would not be satisfied.) If α_i is the i-th child of α and $ci(\alpha_i)$ indicates that the s-th visit to α_i was started but not completed, then we move to α_i, but we leave $pi(\alpha_i)$ alone. The next instruction to be interpreted at α_i will be $ci(\alpha_i)$. The new actions for VISIT and SUSPEND instructions are shown in Figure 7.8. To make the actions easier to understand, no checking for structure tree children or parents is shown. The complete algorithm will visit structure tree children as required, but nonlocal visits can also be skipped, if the *plan_index* values allow it.

7.8 Complexity of ASYNCH_evaluate

The complexity of *ASYNCH_evaluate* depends on the order and timing of the sequence of subtree replacements. If we never have to reset a node, then the algorithm runs as if all replacements had been made synchronously, and it is as efficient as *GPAG_evalute.* If we must reset nodes, however, then the time required by the algorithm can be separated according to the two phases.

Phase I requires finding γ, the LCA of nodes α and κ, in the structure tree. This can be done in amortized $O(\log n)$ time if we use the structure tree operations. Unfortunately, in order to restore the AE-invariant, then in the structure tree, we must also follow paths between γ and α and between γ and κ. If many underlying tree nodes are marked, then structure tree edges will be short, and this operation can be inefficient. On the other hand, if the structure tree is not very deep, this operation is quick.

Phase II requires moving through the tree, evaluating attributes as needed. We can skip visits to nodes that had been completed before the most recent subtree replacement. Unfortunately, there are some cases that require re-visiting a subtree, because a previous visit was started but was interrupted by a new subtree replacement. We may have to revisit nodes in a subtree in order to evaluate an attribute at the bottom of the subtree. In the worst case, we might have to visit these nodes once for every subtree replacement, adding an $O(m)$ factor to the complexity of the algorithm, where m is the number of subtree replacements in the sequence.

7.9 Removing the Simplifying Assumptions

In describing *ASYNCH_evaluate*, we made some non-fundamental assumptions in order to simplify the description. Here we briefly describe what remaining details must be addressed in a complete algorithm.

7.9.1 Structure tree exists

The main complication arising from using structure trees is that κ, the connecting node, may have been a second node added to the structure tree when we marked β, the point of subtree replacement. Whenever a node is marked, its *plan_index* must be initialized properly. In this case, κ will have one structure tree child and possibly a structure tree parent, but as long as the structure tree relatives of κ have reachable current instructions, we can use the *mapfrom* and *mapto* functions to initialize $pi(\kappa)$. This initialization must be included in the structure tree operations.

7.9.2 Current node is inside the subtree being replaced

If β', the root of T_{OLD}, is marked, it is possible that the current node is inside T_{OLD}. Before making the subtree replacement, we find the LCA of β' and α to determine whether $\beta' \rightsquigarrow \alpha$. If so, then we set the current node to δ, the structure tree parent of β', and set $pi(\delta)$ so that $ci(\delta)$ is a VISIT instruction compatible with $ci(\beta')$. We also reset nodes between α and δ to ensure that the new $ci(\delta) \Rightarrow ci(\alpha)$. This will ensure that the inconsistent attributes in T_{OLD} can be reevaluated if T_{OLD} is ever swapped back into the derivation tree.

After the subtree replacement, we must resume evaluation at δ, because the previous current node is no longer in the tree. Of course, if T_{NEW} also contains inconsistent attributes, we should have reset nodes inside the subtree when it was first removed from the derivation tree. If this was done properly, EVAL instructions for inconsistent attributes inside T_{NEW} will be reachable from $ci(\delta)$.

7.10 Asynchronous Changes as a Result of Nonlocal Edges

Although most of this chapter has been complicated, almost all of the complexity can be eliminated if we are only interested in the application of nonlocal dependency edges. Recall that we can treat a change at the source end of a nonlocal edge as an asynchronous subtree replacement at the defining node of the attribute at the destination end of the edge. In this case, evaluation of newly-marked attributes must follow evaluation of the current attribute, because the nonlocal edge is installed dynamically in order to bypass an existing path in $D(T)$. Under these circumstances, we can completely do away with saving the *plan_index*, because we never have to back up. We are guaranteed that each new node that is marked because of a nonlocal edge will be visited after the current node. If we do not need to support asynchronous changes, this simple implementation suffices.

Chapter 8

Conclusions

The main theme of this thesis has been the claim that tree-walking evaluators are preferable to dynamic evaluators, even though both kinds of evaluators may have the same asymptotic complexity for the same incremental problem. Experience has shown that systems using dynamic evaluators are too slow to be used in interactive applications [RT89a]. Tree-walking evaluators, such as *GPAG_evaluate* and *ASYNCH_evaluate*, are important because they use algorithms that do not require the maintenance of characteristic graphs.

On the other hand, tree-walking evaluators only work for restricted classes of attribute grammars. While this restriction could be a severe limitation, most useful attribute grammars appear to be contained in reasonable subclasses of attribute grammars. The class of ordered attribute grammars encompasses a wide range of applications, from proof-checking systems [Gri87] to syntax-directed editors. Even grammars that are not ordered are often partitionable, and by changing a small number of rules or adding augmenting dependencies, a valid set of partitions can be found.

Another way to enlarge the class of useful attribute grammars is to use an algorithm other than Kastens's to generate potential partitions. For example, Barbar [Bar84] describes algorithms that obtain different topological orders from the $DS^+(X)$ graphs. Barbar's algorithm finds valid partitions for some attribute grammars for which Kastens's algorithm fails.

The structure tree operations found in Chapter 5 are an integral part of the multiple subtree replacement algorithms described in this thesis. Despite their amortized $O(\log n)$ running time, a fairly compact version of the operations has been implemented in the Synthesizer Generator. It is hoped that other applications can use structure trees as well.

8.1 Future Research

Three primary areas of future research are suggested by this thesis. First, a characterization of the class of GPAGs is needed. Second, empirical tests that compare the efficiency of various incremental algorithms are needed, because quantitative analysis is difficult or impossible. Third, improvements to *ASYNCH_evaluate* could be explored.

Not much is currently known about the class of globally partitionable attribute grammars. I have used the Synthesizer Generator [RT89a] to analyze several attribute grammars, and I have observed that the algorithm for finding a potential global partitioning frequently makes poor choices when generating interleavings. By supplementing the algorithm described in this thesis with additional heuristics, some attribute grammars are shown to be GPAG, even though the unmodified algorithm fails to find a valid global partitioning. More complicated attribute grammars, such as one for Pascal, may need to be modified slightly in order to be GPAG. Nevertheless, I am confident that there is a large, useful class of globally partitionable attribute grammars.

Analyzing the efficiency of various incremental propagation algorithms described in this thesis and elsewhere is difficult for a few reasons. First, there has been little research concerning typical sequences of operations that are performed by a user of an incremental system. Second, some of the dynamic algorithms cannot be analyzed quantitatively, because in the worst case, their overhead is proportional to the size of the derivation tree, not to |AFFECTED|. Third, response time, rather than asymptotic complexity, is important in an interactive system. An empirical study of the performance of various incremental algorithms could settle claims of the superiority of one incremental algorithm over others.

Finally, more research is needed into the area of asynchronous subtree replacements. *ASYNCH_evaluate* is practical when used with nonlocal dependencies, but in general, a poor sequence of subtree replacements can cause the algorithm to have poor performance, because paths in the structure tree must be followed. In the worst case, the overhead of following these paths is proportional to the size of the derivation tree. A modification of the structure tree operations or new data structures are needed to allow an evaluator to avoid this unacceptable overhead. Improvements to *ASYNCH_evaluate* would make multiple-user, interactive programming environments possible.

Appendix A

Ordered Attribute Grammars

Ordered attribute grammars [Kas80] are a subclass of partitionable attribute grammars. An attribute grammar is ordered if and only if Kastens's algorithm succeeds in finding a valid partitioning for the grammar. Some grammars are not ordered, but are still partitionable. For them, it is sometimes possible for a user to specify a different partitioning for a nonterminal or two, or to add additional (augmenting) edges between some of the attribute occurrences to force a different partitioning. Kastens calls such an attribute grammar "arranged orderly." In this appendix, we give the code for Kastens's algorithm.

A.1 Kastens's Algorithm

The algorithm described here determines whether an attribute grammar is ordered. If the algorithm succeeds, a valid set of partitions is generated, and plans can be produced for each production in the grammar. If the algorithm fails either of the first two steps, the grammar is not partitionable, but if the third step fails, the grammar may still be partitionable. For each step of the algorithm, we show the running time of the step so that we can compare the running time of Kastens's algorithm with that of the algorithm for finding a global partitioning. For more details about Kastens's algorithm, see [Kas80].

We measure the running time of the algorithm in terms of the following parameters:

$$
\begin{aligned}
P &= \text{the number of productions} \\
R &= \text{the maximum number of nonterminals in a single production} \\
W &= \text{the maximum number of attributes of a single nonterminal} \\
V &= \text{the number of nonterminals} \\
G \leq PR &= \text{the length of the grammar} \\
D = RW &= \text{the maximum number of vertices in any } DP^{+}(p) \text{ graph}
\end{aligned}
$$

Kastens's algorithm is a heuristic for guessing a valid partition. To make a reasonable guess, we compute a set of augmented production graphs ($DP^+(p)$ graphs) and augmented nonterminal graphs ($DS^+(X)$ graphs). The $DP^+(p)$ graphs have the following inductive definition:

Definition A.1 *$DP^+(p) = DP(p) \cup \{[X_i.a, X_i.b] \mid [Y_j.a, Y_j.b] \in DP^+(q)$ where X_i is in production p, Y_j is in production q, and $X_i = Y_j\}$.*

The $DS^+(X)$ graphs can be obtained by projecting the augmented production graph for any production containing an occurrence of X onto the attributes of X. In the algorithm, however, both kinds of graphs are maintained explicitly.

A.1.1 Add_arc_induce

During execution of the algorithm, whenever we add an edge to a $DP^+(p)$ graph between two attributes of the same occurrence of a nonterminal X, we also add a corresponding edge to the $DS^+(X)$ graph. This operation is performed by the procedure *Add_arc_induce.* The edges added to the $DS^+(X)$ graphs are unmarked, so that a later step can induce each such edge in all $DP^+(q)$ graphs containing an occurrence of X. Once edges are induced, they are marked, so that the operation is only performed once per edge.

```
procedure Add_arc_induce(p, [Xi.a, Xj.b]);
  Add [Xi.a, Xj.b] to DP+(p);
  new_edges := {[Xi.a, Xj.b] and edges needed to keep DP+(p) closed};
  for each [Yi.a, Yj.b] ∈ new_edges do
       if i = j and [a, b] ∉ DS+(Yi) then
            Add an unmarked edge [a, b] to DS+(Yi).
```

Complexity: If we maintain each graph as an adjacency matrix, the cost of a call to this procedure is $O(D^2)$.

A.1.2 Step 1: Construct $DS^+(X)$ graphs

For each production p, we begin by creating a $DP^+(p)$ graph with no edges. Then we add the edges of $DP(p)$ to $DP^+(p)$. Each edge is added by calling *Add_arc_induce.* If any graph contains a cycle after execution of *Step1a*, the attribute grammar is circular.

```
procedure Step1a;
  for each production p do
       for each attribute occurrence Xj.b of p do
            for each argument Xi.a of Xj.b do
                 if [Xi.a, Xj.b] ∉ DP+(p) then
                      Add_arc_induce(p, [Xi.a, Xj.b]);
```

Complexity: *Step1a* makes $O(PD^2)$ calls to *Add_arc_induce*, so the step requires $O(PD^4)$ time.

We now take an unmarked edge in some $DS^+(X)$ graph, mark the edge, and add a corresponding edge to all $DP^+(p)$ graphs containing an occurrence of X. We repeat this process for all unmarked edges in all $DS^+(X)$ graphs. If *Step1b* creates a cycle in any graph, the grammar is not partitionable, but it is not necessarily circular.

```
procedure Step1b;
for each unmarked edge [a, b] in some DS+(X) graph do
    Mark [a, b] in DS+(X).
    for each occurrence X_i of X in some production p do
        if [X_i.a, X_i.b] ∉ DP+(p) then Add_arc_induce(p, [X_i.a, X_i.b]);
```

Complexity: There are $O(G)$ nonterminal occurrences in the grammar, and for each occurrence, we can induce $O(W^2)$ edges found in the $DS^+(X)$ graphs. Therefore, the complexity of the step is $O(GW^2D^2)$.

A.1.3 Step 2: Compute a complete set of partitions

If none of the resulting $DS^+(X)$ graphs contains a cycle, the graphs can be used to assign attributes of each nonterminal to a partition. Note that *Step2* assigns partitions in reverse order. If desired, the partition assignments for each attribute can be renumbered after all attributes have been assigned. *Step2* is called for each nonterminal in the grammar.

```
procedure Step2(X);
/* Note: Partition numbers are assigned in reverse order */
k := 1;
unassigned := A(X);
while unassigned ≠ ∅ do
    found_one := false;
    if ∃b ∈ unassigned
        and no successor of b ∈ unassigned
        and (b is synthesized and k is odd or b is inherited and k is even)
    then
        partition_no(b) := k;
        unassigned := unassigned − {b};
        found_one := true;
    end
    if found_one = false and unassigned ≠ ∅ then k := k + 1;
end
if k is odd then k := k + 1;
k_X := k;
```

Complexity: Each call to *Step2* takes $O(W^3)$ time and the procedure must be called for each of the V nonterminals, so the time needed by the step is $O(VW^3)$.

A.1.4 Step 3: Test the partitioning for validity

Now we can check the generated partitions for validity. To do this, we add edges to each $DP^+(p)$ graph between attributes of consecutive partitions for each nonterminal occurrence. Even though the test described in Chapter 6 starts with the unaugmented $DP(p)$ graphs, the extra edges added to $DP^+(p)$ by Steps 1 and 2 have no effect on the validity test.

If the test fails, then the attribute grammar is not ordered, but it may still be partitionable. There are slightly different algorithms that will find a different partitioning, but still run in polynomial time. For example, Barbar [Bar84] describes algorithms that "guess" different partitions. For some attribute grammars, some of these algorithms will succeed where Kastens's algorithm fails.

Complexity: Although no code is shown for this step, we must add $O(W)$ edges to each of the $O(G)$ nonterminal occurrences in the grammar. Determining whether the resulting $DP^+(p)$ graph contains a cycle requires $O(D^2)$ time if we use a depth-first search. Therefore, the complexity for the step is $O(GWD^2)$.

A.1.5 Generating plans

The $DP^+(p)$ graphs are used to generate plans for each production in the grammar. The code for this step is not shown either, but basically, it involves sorting the graphs topologically, and the complexity of the step does not add to the overall complexity of the algorithm.

A.2 Complexity of the Algorithm

Adding the times for all the steps of the algorithm, we see that the complexity is

$$O(PD^4) + O(GW^2D^2) + O(VW^3) + O(GWD^2).$$

Since $V \leq G$ and $D = RW$, this reduces to

$$O(PR^4W^4 + GW^2R^2W^2) = O(W^4GR^3).$$

Kastens points out that the time depends only linearly on the length of the attribute grammar, and as long as no nonterminal has too many attributes, the W^4 factor does not cause the algorithm to be too inefficient.

Appendix B

Algorithm for Finding a Global Partitioning

In this appendix, we present the algorithm for finding a potential global partitioning. It begins by running the Reps-Marceau-Teitelbaum algorithm [RMT86], which generates the $DS^+(X)$ and $DSS(X,Y)$ graphs. The $DSS(X,Y)$ graphs are used to "guess" a set of interleavings. Thus, as with Kastens's algorithm, this algorithm can be thought of as a heuristic, since it does not discover a valid global partitioning for all globally partitionable attribute grammars. By manually adding dependencies or explicitly specifying interleavings for some pairs of nonterminals, we can cause some grammars to pass the validity test.

B.1 The Reps-Marceau-Teitelbaum Algorithm

This algorithm is similar to Kastens's algorithm. Its goal is to construct the $DSS(X,Y)$ graphs that are used to generate interleavings.

B.1.1 Construct $DSS(X,Y)$ and $DS^+(X)$ graphs

Let G be an attribute grammar. We wish to construct $DS^+(X)$ graphs as well as $DSS(X,Y)$ graphs for all $X \rightsquigarrow Y$. Each edge in any $DSS(X,Y)$ graph represents an actual dependency in some derivation tree. More precisely, let T be a derivation tree of G, and let α and β be nodes in T such that $\tau(\alpha) = X$, $\tau(\beta) = Y$, and $\alpha \rightsquigarrow \beta$. If there is a directed path from $\alpha.a$ to $\beta.b$ in $D(T)$, then there is an edge $[X.a, Y.b]$ in $DSS(X,Y)$.

We also need an *Add_arc_induce* procedure for this algorithm, but it is a little more complicated, because it is used to construct a set of $DPC(X,p)$ graphs, where X is a nonterminal and p is a production $p : X_0 \rightarrow X_1 \cdots X_n$ such that $X \rightsquigarrow X_0$. The vertices of a $DPC(X,p)$ graph are the attributes of X and of each nonterminal $X_0, \ldots, X_n$. These graphs are used to induce edges between attributes of X and X_i (for $i > 0$) in the $DSS(X,X_i)$ graph. For example, suppose $X \overset{1}{\rightsquigarrow} X_0$. Then there is a production $q : X \to \cdots X_0 \cdots$, and any edges between attributes of X and X_0 in some $DPC(Y, q)$ graph will also be added between the same attributes of X and X_0 in the $DPC(X, p)$ graph.

```
procedure Add_arc_induce(DPC(X,p), [c,d]);
begin
    Add [c,d] to DPC(X,p).
    new_edges := {[c,d] and edges needed to keep DPC(X,p) closed};
    /* Note: X_{-1} denotes the occurrence of X in DPC(X,p) */
    for each [Y_i.a, Y_j.b] ∈ new_edges do
        case [Y_i.a, Y_j.b] of
            [X_i.m, X_i.n] :
                if [m,n] ∉ DS(X_i) then
                    Add an unmarked edge [m,n] to DS+(X_i).
            [X_i.m, X_j.n] for i = −1 and j ≥ 0 or i = 0 and j > 0:
                if [X_i.m, X_j.n] ∉ DSS(X_i, X_j) then
                    Add an unmarked edge [X_i.m, X_j.n] to DSS(X_i, X_j).
            [X_i.m, X_j.n] for j = −1 and i ≥ 0 or j = 0 and i > 0:
                if [X_j.m, X_i.n] ∉ DSS(X_i, X_j) then
                    Add an unmarked edge [X_j.m, X_i.n] to DSS(X_i, X_j).
```

Complexity: Since the maximum number of nodes in a DPC graph is $(R+1)W$ and $D = RW$, each call to *Add_arc_induce* takes $O(D^2)$ time.

Step 1

The first step is the initialization step. It enters the edges of the attribute grammar into the $DPC(X, p)$ and $DS^+(X)$ graphs. *Step1a* is almost identical to the same step in Kastens's algorithm; the only difference is that for each production p, we must add edges to all of the $DPC(X, p)$ graphs.

```
procedure Step1a;
for each production p do
    for each attribute occurrence X_j.b of p do
        for each argument X_i.a of X_j.b do
            for each DPC(Y,p) do
                if [X_i.a, X_j.b] ∉ DPC+(Y,p) then
                    Add_arc_induce(DPC(Y,p), [X_i.a, X_j.b]);
```

Complexity: There are $O(V)$ DPC graphs for each production. Hence, the time needed by *Step1a* is $O(PVD^4)$.

Step1b is again similar to the same step in Kastens's algorithm. We must take unmarked edges in either the $DS^+(X)$ graphs or the $DSS(X, Y)$ graphs and induce corresponding edges in the $DPC(Y, p)$ graphs. We use *Add_arc_induce* to add the edges.

```
procedure Step1b;
while there is an unmarked edge in any DS+(X) or DSS(X,Y) graph do
     for each unmarked edge [X.a, X.b] ∈ DS+(X) do
          Mark [X.a, X.b].
          for each occurrence X_i of X in any DPC(X_-1, p) do
               if [X_i.a, X_i.b] ∉ DPC(X_-1, p) then
                    Add_arc_induce(DPC(X_-1, p), (X_i.a, X_i.b));
               for each Y such that X ⇝ Y do
                    Add [X.a, X.b] to DSS(X, Y).
               for each Y such that Y ⇝ X do
                    Add [X.a, X.b] to DSS(Y, X).
     end
     for each unmarked edge [X_i.a, X_j.b] ∈ DSS(X_1, X_2) do
          Mark [X_i.a, X_j.b].
          for each DPC(X, p) where p is X_2 → Y_1 ··· Y_m do
               if [X_i.a, X_j.b] ∉ DPC(X_1, p) then
                    Add_arc_induce(DPC(X_1, p), (X_i.a, X_j.b))
     end
end
```

Complexity: The portion of this step dealing with unmarked edges in $DSS(X,Y)$ graphs has complexity $O(4W^2PVD^2)$, since each $DSS(X,Y)$ graph has $O(4W^2)$ edges. For each such edge, there are $O(PV)$ graphs where the edge can be induced by calling *Add_arc_induce*. For the part of *Step1b* dealing with edges from the $DS^+(X)$ graphs, we need to examine $O(PV(R+1))$ nonterminal occurrences in the $DPC(X,p)$ graphs, as well as examine $O(V)$ $DSS(X,Y)$ graphs. Adding an edge to the $DSS(X,Y)$ graphs takes constant time. Hence, we have a running time of $O(W^2(PVRD^2+V))$ for *Step1b*.

B.2 Computing A Global Partitioning

Once we have the graphs created by the Reps-Marceau-Teitelbaum algorithm, we can continue with the computation of a global partitioning.

B.2.1 Compute a complete set of partitions

This step is identical to the same step in Kastens's algorithm, and so has a complexity of $O(VW^3)$.

B.2.2 Construct DSS_Π graphs

This step is simple in concept, but the code is cumbersome, so it is not included. For each edge $[a,b]$ in a $DSS(X,Y)$ graph, we add an edge $[P_i(X,Y), P_j(X,Y)]$ to $DSS_\Pi(X,Y)$, as long as $i \neq j$. We must examine every edge in each $DSS(X,Y)$ graph once. Thus, the complexity of the step is $O(V^2W^2)$.

B.2.3 Generate a complete set of interleavings

This operation is similar to generating partitions from $DS^+(X)$ graphs, and has the same complexity. Because there are $O(V^2)$ DSS_{Π} graphs, the complexity of the step is $O(V^2W^3)$.

B.2.4 Check the global partitioning for validity

As described in Chapter 6, the validity test consists of two parts. The feasibility test has the same complexity as the feasibility test used by Kastens: $O(PRWD^2)$. For the consistency test, we must construct $O(V^3)$ graphs. We add $O(W)$ edges to each graph and test them for cycles in $O(W^2)$ time. So the consistency test needs $O(PRWD^2 + V^3W^2)$ time.

B.3 Complexity of the Algorithm

Adding the times for all the steps, we get

$$O(PVD^4) + O(W^2(PVRD^2 + V)) + O(VW^3) + O(V^2W^2) + O(V^2W^3)$$
$$+O(PRWD^2 + V^3W^3)$$

time. This reduces to $O(GVR^3W^4 + V^3W^2)$.

This result compares to Kastens's algorithm as follows: The first term has an additional factor V. More importantly, the second term contains the factor V^3. This factor appears to be the most expensive part of the algorithm. Nevertheless, as implemented on a Sun 3/60, the algorithm runs fast enough to be practical.

Bibliography

[Bar84] Kablan Barbar. Classification des grammaires d'attributs ordonnés. Technical Report NR. 8412, Université de Bordeaux I., April 1984.

[Bel88] Matthew Belmonte. A practical attribute grammar circularity test. Technical Report 88–920, Cornell University, June 1988.

[Boc76] G. V. Bochmann. Semantic evaluation from left to right. *CACM*, 19(2):55–62, February 1976.

[CH79] Rina Cohen and Eli Harry. Automatic generation of near-optimal linear-time translators for non-circular attribute grammars. *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 121–134, 1979.

[EF80] Joost Engelfriet and Gilberto Filè. Simple multi-visit attribute grammars. Technical Report NR. 314, Technische Hogeschool Twente, August 1980.

[EF81] Joost Engelfriet and Gilberto Filè. Passes, sweeps and visits. *Automata, Languages and Programming*, August 1981. Lecture Notes in Computer Science 115.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 85–98, 1986.

[Gei86] Bob Geitz. Asynchronous subtree replacement for language-based editors. Oberlin College (Unpublished Paper), 1986.

[Gri87] Tim Griffin. An environment for formal systems. Technical Report TR 87-846, Cornell University, 1987.

[Hoo87] Roger Hoover. *Incremental Graph Evaluation.* PhD Thesis, Cornell University, 1987.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[JF85] Gregory F. Johnson and Charles N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 141–151, January 1985.

[JOR75] M. Jazayeri, W. F. Ogden, and W. C. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Communications of the ACM*, 18(12):697–706, December 1975.

[JS86] Larry G. Jones and Janos Simon. Hierarchical VLSI design systems based on attribute grammars. *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 58–69, 1986.

[Kas80] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[KHZ82] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. *GAG: A Practical Compiler Generator.* Lecture Notes in Computer Science 141. Springer-Verlag, New York, 1982.

[KK86] Simon Kaplan and Gail Kaiser. Incremental attribute evaluation in distributed language-based environments. *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1986.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[KW76] Ken Kennedy and Scott Warren. Automatic generation of efficient evaluators for attribute grammars. *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 32–49, 1976.

[Pfr86] Mary Pfreundschuh. *A Model for Building Modular Systems Based on Attribute Grammars.* PhD Thesis, The University of Iowa, 1986.

[Pug88] William Pugh. *Incremental Computation and the Incremental Evaluation of Function Programs.* PhD Thesis, Cornell University, 1988.

[Rep84] Thomas Reps. *Generating Language-Based Environments.* MIT Press, 1984.

[RMT86] Thomas Reps, Carla Marceau, and Tim Teitelbaum. Remote attribute updating for language-based editors. *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[RT89a] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator.* Springer-Verlag, New York, 1989.

[RT89b] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer-Verlag, New York, third edition, 1989.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *TOPLAS*, 5(3):449–477, July 1983.

[ST83] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[Tar85] Robert E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 4(5):367–378, 1985.

[Wal88] Janet Walz. *Extending Attribute Grammar and Type Inference Algorithms.* PhD Thesis, Cornell University, 1988.

[WG84] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

[Yeh83] D. Yeh. On incremental evaluation of ordered attributed grammars. *BIT*, 23:308–320, 1983.

[YK88] D. Yeh and U. Kastens. Improvements of an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN Notices*, 23(12):45–50, December 1988.