

Incremental Context-Dependent Analysis for Language-Based Editors

THOMAS REPS, TIM TEITELBAUM, and ALAN DEMERS

Cornell University

Knowledge of a programming language's grammar allows language-based editors to enforce syntactic correctness at all times during development by restricting editing operations to legitimate modifications of the program's context-free derivation tree; however, not all language constraints can be enforced in this way because not all features can be described by the context-free formalism. Attribute grammars permit context-dependent language features to be expressed in a modular, declarative fashion and thus are a good basis for specifying language-based editors. Such editors represent programs as attributed trees, which are modified by operations such as subtree pruning and grafting. Incremental analysis is performed by updating attribute values after every modification. This paper discusses how updating can be carried out and presents several algorithms for the task, including one that is asymptotically optimal in time.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding—*program editors*; D.2.6 [Software Engineering]: Programming Environments; D.3.1 [Programming Languages]: Formal Definitions and Theory—*semantics*; *syntax*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Attribute grammars, incremental attribute evaluation, incremental semantic analysis, editor generators

1. INTRODUCTION

Our concern is the design and implementation of interactive environments for computer programming; our goal is the development of powerful language-specific tools that support incremental program development and testing and that exploit state-of-the-art personal computing hardware. We began in May 1978 with the design of the Cornell Program Synthesizer [38], an interactive language-based programming environment with syntax-directed facilities to edit, execute, and debug programs. The example set by the Synthesizer and other language-based systems such as Emily [12], MENTOR [6, 7], PDE1L [27], and Gandalf [25] has encouraged us to develop a tool for generating such systems from language

A preliminary version of this paper appeared in the Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, January 1982 [35].

This work was supported in part by the National Science Foundation under grants MCS80-04218 and MCS82-02677.

Authors' address: Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0164-0925/83/0700-0449 \$00.75

descriptions. Our current goal is the development of the Synthesizer Generator [36], a processor that enables language-based environments for different languages to be created easily from formal specifications of the syntax, the display format, and the semantics.

Simple language-based editors can easily be generated from a context-free grammar that covers the given language; the editor generator builds tables encoding the grammar in the form used by the language-independent kernel of the system. However, the capabilities of such a system are limited by the descriptive power of context-free grammars; the system would not be able to build an editor with facilities that require widely separated parts of a program to be interrelated or constrained in ways that vary depending on the context given by the rest of the program.

Attribute grammars extend the descriptive power of context-free grammars. First introduced by Knuth to assign semantics to context-free languages [19], they have subsequently been used to describe translations [22], code optimizations [30], correctness-preserving transformations [11], data-flow analysis [3, 9], and program anomalies [2]. Furthermore, attribute grammars have been used as the specification language for several compiler-writing systems, including FOLDS [8], DELTA [23], MUG2 [10], HLP [33], APARSE [28], and GAG [16].¹

Because of their utility in these applications, attribute grammars appeared to us to be a good basis for specifying and generating language-based editors. Accordingly, we have implemented a prototype Synthesizer Generator and have used the system to build experimental editors in which attributes control pretty-printing and code generation and detect program anomalies, type violations, and errors in program proofs.

The subject of this paper is the algorithmic foundations of the attribute-grammar approach to constructing language-based editors. Each editor represents a program as an attributed tree, and programs are modified by derivation-tree operations such as pruning, grafting, and deriving. A derivation-tree modification directly affects the values of the attributes of the modification point; incremental analysis is performed by updating attribute values throughout the tree in response to modifications.

After each modification to a program tree, only a subset of attributes, denoted by *AFFECTED*, require new values. It should be understood that, when updating begins, it is not known which attributes are members of *AFFECTED*; *AFFECTED* is determined as a result of the updating process itself. This paper presents algorithms that identify attributes in *AFFECTED* and recompute their values. One of these algorithms has cost proportional to the size of *AFFECTED*, which is asymptotically optimal in time because the work needed to update the tree can be no less than $|AFFECTED|$. Another of our algorithms, although suboptimal, may be preferable depending on the particular attribute grammar. The paper compares our algorithms to one another and contrasts them with alternative approaches to providing non-context-free facilities in language-based editors. We consider here only arbitrary noncircular attribute grammars; optimal algorithms for the restricted classes of *L*-attributed, ordered, and absolutely noncircular grammars are presented in [34].

¹ Reference [31] is an extensive bibliography of the attribute grammar literature.

2. ATTRIBUTE GRAMMARS

An attribute grammar is a context-free grammar extended by attaching attributes to the symbols of the grammar. Associated with each production of the grammar is a set of *semantic equations*; each equation defines one attribute as the value of a *semantic function* applied to other attributes in the production. Attributes are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes. Each semantic equation defines a value for a synthesized attribute of the left-side nonterminal or an inherited attribute of a right-side symbol. For brevity, the arguments of the semantic function defining the value of attribute b are referred to as the arguments of b .

Every attribute grammar can be put into a *normal form*, in which every semantic equation defines a value for a synthesized attribute of the left-side nonterminal, or an inherited attribute of a right-side symbol, in terms of zero or more inherited attributes of the left-side nonterminal and synthesized attributes of the right-side symbols.

Example. The ambiguous context-free grammar

$$\begin{aligned} \text{ROOT} &\rightarrow S \\ S &\rightarrow S _ S \\ S &\rightarrow \text{word}_1 \\ &\vdots \\ S &\rightarrow \text{word}_n \end{aligned}$$

generates sentences of one or more words separated by single blanks ($_$), selected from a vocabulary $\text{word}_1, \dots, \text{word}_n$. Suppose sentences are to be displayed on a screen of bounded width W such that each line contains as many words as possible and no word is split across a line. Assume that columns are numbered one to W and that no word is longer than W . Then we may wish to associate with each phrase S a synthesized attribute $S.\text{last}$ designating the column on the display screen of the last character of that phrase. To define this attribute, we must also know the column of the last character of the word immediately preceding phrase S , which we can represent in inherited attributed $S.\text{previous}$. The rules of the normal-form grammar defining these attributes are

$$\begin{aligned} \text{ROOT} &\rightarrow S \\ &S.\text{previous} = -1 \\ S_1 &\rightarrow S_2 _ S_3 \\ &S_2.\text{previous} = S_1.\text{previous} \\ &S_3.\text{previous} = S_2.\text{last} \\ &S_1.\text{last} = S_3.\text{last} \\ S &\rightarrow \text{word}_1 \\ &S.\text{last} = \text{if } S.\text{previous} + 1 + \text{length}(\text{word}_1) \leq W \\ &\quad \text{then } S.\text{previous} + 1 + \text{length}(\text{word}_1) \\ &\quad \text{else } \text{length}(\text{word}_1) \end{aligned}$$

etc., where subscripts distinguish among multiple occurrences of the same non-terminal in the second production.

A derivation-tree node labeled X defines a set of *attribute instances* corresponding to the attributes of X . A *semantic tree* is a derivation tree together with an assignment of either a value or the special token **null** to each attribute instance of the tree. We assume that **null** is a value outside the domain of every attribute.

Functional dependencies among attributes in a production p or a semantic tree T can be represented by a *dependency graph*, denoted $D(p)$ or $D(T)$, respectively, which is a directed graph defined as follows:

- (1) For each attribute b , the graph contains a vertex b' .
- (2) If attribute b is an argument of attribute c , the graph contains a directed edge (b', c') .

An edge from b' to c' has this meaning: b' is used to determine the value of c' . Although closely related, an attribute instance b in T and the vertex b' in $D(T)$ are different objects. When this distinction is not made explicitly clear, the intended meaning should be clear from the context. The notation $TreeNode(b')$ denotes the node of T with the attribute instance b corresponding to b' . Vertices of $D(T)$ with no incoming edges correspond to attribute instances defined by zeroary semantic functions, that is, constants.

A semantic tree is *fully attributed* if each of its attribute instances is *available*, that is, non-**null**. When all the arguments of an unavailable attribute instance are available, we say it is *ready for evaluation*.

Example. Continuing our formatting example, fix the width of the display screen at $W = 13$ and consider the displayed sentence

| |
|---|
| 1234567890123 |
| Candy is dandy but liquor is quicker |

One of the possible derivation trees for this sentence is shown fully attributed and together with its dependency graph in Figure 1. The semantic tree consists of the instances of nonterminals **ROOT** and **S**, together with their respective attributes shown in adjacent boxes. Nonterminals of the same production are connected by dashed lines. The dependency graph consists of the attribute instances, shown in boxes, linked by their functional dependencies, shown as solid arrows. The constant -1 , strictly speaking, is neither a part of the semantic tree nor a part of the dependency graph.

To further characterize semantic trees we introduce the notion of consistency. An attribute instance b is *consistent* if

- (1) the arguments of b are available and
- (2) the value of b is equal to its semantic function applied to the arguments.

In all other cases, we say b is *inconsistent*. We extend the definition of consistency to cover related concepts in semantic trees and dependency graphs; a semantic tree or a dependency graph is consistent if all of its attribute instances are consistent.

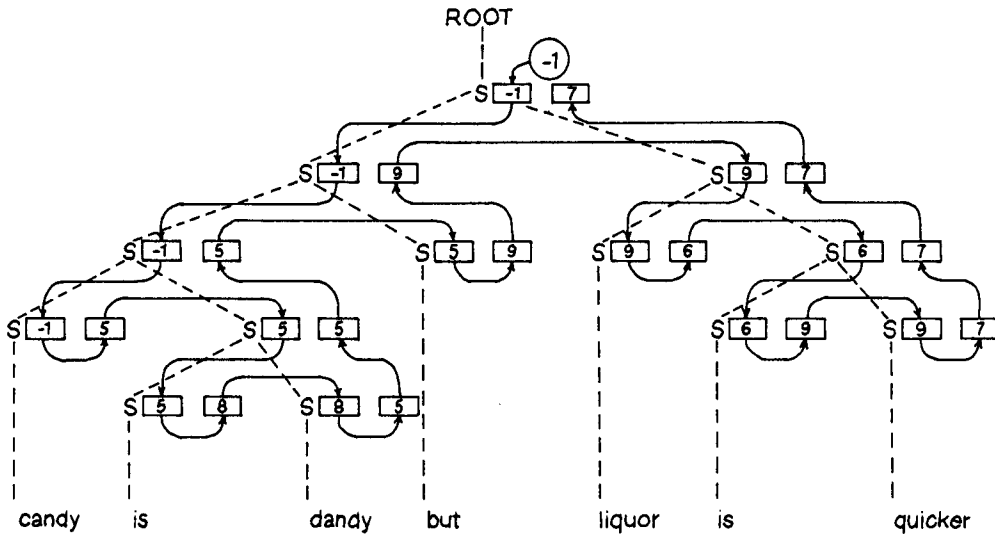


Fig. 1. An attributed tree.

An attribute grammar is *noncircular* when the dependency graph of every possible derivation tree is acyclic. Noncircularity is decidable [18] and, though of inherently exponential complexity [14], is feasible to test in practice. An attribute grammar is *well formed* when the terminal symbols of the grammar have no synthesized attributes, the root symbol of the grammar has no inherited attributes, and each production includes a semantic equation for all the synthesized attributes of the left-side nonterminal and all the inherited attributes of the right-side symbols. This paper deals only with attribute grammars that are well formed and noncircular; unless stated otherwise, we also assume that the grammar is in normal form.

3. A SIMPLE MODEL OF EDITING

This section opens the discussion of how attribute grammars can be used in a system for generating language-based editors. We are initially concerned with the simple model of editing described below [5]; later, we extend this basic model. The basic idea is for each editor to represent a file as an attributed tree of the attribute grammar. When editing operations modify the tree, analysis is carried out by reestablishing consistent attribute values throughout the tree. Any display formats and translation semantics that are defined in terms of attributes are thereby updated. Attribute values indicating violation of context-sensitive language constraints may be used to annotate the program display (if only error detection is desired) or to initiate undoing the structural modification (if error prevention is desired).

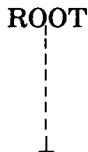
Creating a file using a language-based editor entails growing a semantic tree. During development, a file tree is a partial derivation tree; that is, it contains unexpanded nonterminals. This is potentially a problem because, at an unexpanded nonterminal X , we have no means for giving values to the synthesized

attributes of X or to any of their successors. This conflicts with our desire to maintain values for every attribute of the tree.

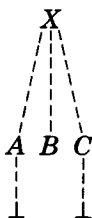
To avoid this problem, we require that the grammar include a *completing production* $X \rightarrow \perp$ for each nonterminal symbol X . The symbol \perp denotes “unexpanded,” and the semantic equations of the completing production define values for the synthesized attributes of X . By convention,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

Modifying a program entails restructuring a derivation tree by pruning and grafting subtrees. Let T be a semantic tree and U be a subtree of T with root node r labeled X . U is *pruned* from T by removing the subtree rooted at r . Let U' be a semantic tree with root r' also labeled X . U' is *grafted* onto T at leaf r labeled X by assigning the synthesized attribute values of r to the synthesized attribute instances of r' and then replacing r by U' in T .² We define *subtree replacement* of U by U' as the pruning of U followed by the grafting of U' in its place.

At each stage during editing, the *editing cursor* is positioned at an interior node of the semantic tree. An editing session is viewed as a succession of replacement operations and cursor motions starting from the complete, fully attributed semantic tree



with the cursor positioned at ROOT. Each insertion at an unexpanded nonterminal labeled X is viewed as the replacement of an instance of the completing production of X by a freestanding tree U' with root X . For example, when a derivation is made according to the production $X \rightarrow A B C$ where A and C are nonterminals, U' is



Each deletion is viewed as the replacement of a subtree U (with root X) by an instance of the completing production of X .

² The decision to save the *synthesized* attributes of r and the *inherited* attributes of r' may at first seem counterintuitive. The choice is somewhat arbitrary since, no matter which attributes are saved, inconsistencies may be introduced that must subsequently be eliminated. Our definition has the advantage of simplifying the presentation of our algorithms, as is explained in Section 4.1. More refined selection criteria, for example, minimizing the number of initially inconsistent attributes, are possible optimizations that do not influence the asymptotic worst case running times of our algorithms.

Because modifications may be made at any location in the program, the system must deal with freestanding trees derived from any of the nonterminals of the grammar, not just ones derived from the root symbol. For example, a subtree removed at X becomes a freestanding tree with root X . Such trees are retained so that they can be inserted into the program elsewhere.

The task of an *incremental attribute evaluator* is to produce a consistent, fully attributed tree after each subtree replacement. Of course, any nonincremental attribute evaluator could be applied to completely reevaluate the tree, but our goal is to minimize work by confining the scope of reevaluation required after each subtree replacement. The incremental viewpoint, and consequent concern with how to update attributes, sets our work apart from earlier work on attribute evaluation.

4. NAIVE INCREMENTAL ATTRIBUTE EVALUATORS

In this section, two simple approaches to incremental attribute evaluation are presented: change propagation and nullification/reevaluation. The deficiencies of the two naive algorithms are discussed and motivate the development, in Section 5, of an optimal-time change propagation algorithm. Aspects of the nullification/reevaluation approach reappear later when the basic model of editing is extended to include demand attributes.

4.1 Change Propagation

One approach to incremental attribute evaluation, called *change propagation*, involves propagating changes of attribute values through a fully attributed tree. Throughout the process, each attribute is available, although possibly inconsistent. When the value of an attribute instance is changed to make it consistent, its successors may become inconsistent; however, if reevaluating an attribute instance yields a value equal to its old value, changes need not be propagated further. Thus, change propagation can be accomplished by following attribute dependencies and maintaining a work-list of possibly inconsistent attributes that must be reevaluated because one of their arguments has changed value.

The algorithm for subtree replacement given as procedure REPLACE of Algorithm 1 uses the change propagation procedure PROPAGATE to reestablish consistent attribute values. REPLACE assumes that the freestanding tree U' to be grafted into T at r is consistent and fully attributed. Note, however, that in the freestanding tree U' the arguments of the inherited attributes of its root are not in U' . Therefore, we extend the definition of consistent to allow inherited attributes of the root of a freestanding tree to have arbitrary values. In general, the tree resulting from the grafting of U' will not be consistent. However, by virtue of our definition of grafting and the assumption that the grammar is in normal form, the initial inconsistencies are confined to the attributes of the modification point r .³

Using change propagation as part of subtree replacement makes certain editing operations inexpensive, such as inserting a subtree at a location where the attributes are identical to the attributes of the root of the subtree. Unfortunately,

³ In Section 5.5, we show how the normal-form assumption can easily be dropped and the definition of subtree replacement generalized.

```

REPLACE( $T, r, U'$ ):
  let
     $T$  = a consistent, fully attributed semantic tree
     $r$  = a node of  $T$ 
     $U'$  = a consistent, fully attributed semantic tree in which the syntactic label
          of the root matches the syntactic label of  $r$ 
  in
    prune the subtree at  $r$  from  $T$ 
    graft  $U'$  onto  $T$  at  $r$ 
    PROPAGATE( $T, r$ )
PROPAGATE( $T, r$ ):
  let
     $T$  = a fully attributed semantic tree
     $r$  = a nonterminal node of  $T$  containing all inconsistent attributes of  $T$ 
     $S$  = a set of attribute instances
     $b$  = an attribute instance
    Oldvalue, Newvalue = attribute values
  in
     $S :=$  the set of inconsistent attribute instances of  $r$ 
    while  $S \neq \emptyset$  do
      Select and remove a vertex  $b$  from  $S$ 
      Oldvalue := value of  $b$ 
      evaluate  $b$ 
      Newvalue := value of  $b$ 
      if Oldvalue  $\neq$  Newvalue then Insert all successors of  $b$  into  $S$ 
    od

```

Algorithm 1. Subtree replacement using naive change propagation.

the behavior of change propagation is sensitive to the order in which attributes are chosen for evaluation, and, if attribute dependencies are followed blindly, as in Algorithm 1, its behavior can be nonlinear in the number of attributes reevaluated. This sort of nonlinear behavior occurs when updating a tree that has a dependency graph in which attributes are connected by more than one path. In the appendix, we give a simple attribute grammar for which Algorithm 1 requires quadratic time if PROPAGATE manipulates its work-list S in FIFO (first in, first out) order, and exponential time if it manipulates S in LIFO (last in, first out) order.

4.2 Nullification/Reevaluation

The nonlinear behavior of subtree replacement that can result from using the change propagation algorithm described above can be avoided by using an alternative method for subtree replacement called *nullification/reevaluation*, given below as Algorithm 2. After U has been replaced by U' in T , the procedure NULLIFY is used to set all attributes of the modification point to the special value **null** and then to propagate **null** to all attributes that depend on them. Then the procedure EVALUATE is used to propagate consistent new values throughout the tree. Both NULLIFY and EVALUATE are essentially traversals of the portion of the DAG $D(T)$ reachable from r . As a “mark” indicating that an attribute has already been visited in the traversal, NULLIFY uses the presence of a **null** value whereas EVALUATE uses the presence of a non-**null** value. Both NULLIFY and EVALUATE have the property that they only consider an

```

REPLACE( $T, r, U'$ ):
  let
     $T, U'$  = consistent, fully attributed semantic trees
     $r$  = a node of  $T$ 
     $U$  = the subtree of  $T$  rooted at  $r$ 
  in
    prune  $U$  from  $T$ 
    graft  $U'$  onto  $T$  at  $r$ 
    NULLIFY( $T$ , {attributes of  $r$ })
    EVALUATE( $T$ , {attributes of  $r$  that are ready for evaluation})
NULLIFY( $T, S$ ):
  let
     $T$  = a consistent, fully attributed semantic tree
     $S$  = a set of attribute instances
     $b, c$  = attribute instances
  in
    while  $S \neq \emptyset$  do
      Select and remove an attribute instance  $b$  from  $S$ 
      nullify  $b$ 
      for each  $c$  that is a successor of  $b$  do
        if  $c$  is available then Insert  $c$  into  $S$ 
      od
    od
EVALUATE( $T, S$ ):
  let
     $T$  = a partially evaluated semantic tree in which all successors of unavailable
      attributes are unavailable
     $S$  = the set of attribute instances of  $T$  that are ready for evaluation
     $b, c$  = attribute instances
  in
    while  $S \neq \emptyset$  do
      Select and remove a vertex  $b$  from  $S$ 
      evaluate  $b$ 
      for each  $c$  that is a successor of  $b$  do
        if  $c$  is ready for evaluation then Insert  $c$  into  $S$ 
      od
    od

```

Algorithm 2. Subtree replacement using nullification/reevaluation.

attribute once, so the total work done by Algorithm 2 is linear in the number of attributes considered.

4.3 Suboptimal Behavior

A derivation-tree modification directly affects the values of the attributes at the point of modification. In our simple model of editing, attribute values must be updated in response to each modification to leave the semantic tree consistent and fully attributed. Out of the entire collection of attributes in the tree, only certain ones require new values. To be more precise, let T' denote the inconsistent tree resulting from a subtree replacement, and let T'' denote T' after it has been updated. We define *AFFECTED* to be the set of attribute instances that have different values in the two trees. Because $O(|\text{AFFECTED}|)$ is the minimal amount of work required to update T' after subtree replacement, we say that an

incremental evaluator is *optimal-time* if it runs in $O(|\text{AFFECTED}|)$ steps, where semantic function evaluations are counted as unit steps. It is important to bear in mind that **AFFECTED** is not known a priori and can only be inferred from the updating process itself.

The nullification/reevaluation evaluator given as Algorithm 2 is not optimal because the size of the set of *all* attributes that depend on attributes of the modification point is not related to the size of **AFFECTED** in any fixed way. Consequently, the evaluator may do extensive propagations even when **AFFECTED** is a very small set.

The naive change propagation evaluator given as Algorithm 1 is very sensitive to the order in which attributes are selected as candidates for new values. With the right selections, a tree is updated optimally; however, there is no guarantee that the right selections will be made; and, when attributes are selected in the wrong order, not only is the algorithm suboptimal, but the time it uses can be nonlinear in the number of attributes considered.

As a heuristic in developing an optimal-time change propagation algorithm, we note the following. If, in the course of propagating new values, an attribute is ever (temporarily) reassigned a value other than its correct final value, spurious changes are apt to propagate arbitrarily far beyond the boundaries of **AFFECTED**, leading to suboptimal running time. To avoid this possibility, a change propagator should schedule attribute reevaluations such that any new value computed is necessarily the correct final value. That is, an attribute should not be reevaluated until all of its arguments are known to have their correct final values. This suggests that what is needed is an enumeration of **AFFECTED** in topological order with respect to the dependency graph. In the case of nonincremental evaluation, where all attributes of the tree T must be computed, Knuth's topological sorting algorithm [20] has been applied to the dependency graph $D(T)$ to produce evaluators that run in time $O(|D(T)|)$ [17, 22]. In our case of incremental evaluation, what is needed is an algorithm that will *generate* **AFFECTED** in topological order in time $O(|\text{AFFECTED}|)$.

5. OPTIMAL-TIME CHANGE PROPAGATION

This section describes a second version of the change propagation algorithm **PROPAGATE** that identifies attributes in **AFFECTED** and computes their new, final values in topological order with respect to the dependency graph $D(T)$ [35]. Both the total number of semantic function applications and the total cost of bookkeeping operations are proportional to the size of **AFFECTED**; consequently, this **PROPAGATE** is asymptotically optimal in time.

The optimal-time version of **PROPAGATE** can be understood as a generalization of Knuth's topological sorting algorithm. **PROPAGATE** (like topological sorting) keeps a work-list of attributes that are ready for reevaluation (enumeration); an attribute is placed on the work-list when its in-degree is reduced to zero in a scheduling graph whose edges reflect dependencies among attributes that have not yet been reevaluated (enumerated). Whereas in topological sorting the vertices of the scheduling graph are known a priori, in **PROPAGATE** the set of vertices of the scheduling graph is generated dynamically at the same time as it is being enumerated. What makes **PROPAGATE** asymptotically optimal is

that the scheduling graph never grows larger than $O(|\text{AFFECTED}|)$; on each step, the size of the scheduling graph is proportional to the number of attributes that have received new values since updating began. Vertices of the initial scheduling graph represent just the attributes of the point of subtree replacement. Thereafter, the scheduling graph expands only when changes propagate to attributes that are arguments of attributes outside the current graph.

5.1 Getting Started

Let us suppose a subtree replacement takes place at node r . By virtue of the way attribute values are exchanged during subtree replacement, all inconsistent attributes are confined to r when change propagation is initiated. PROPAGATE would not make any progress reevaluating attributes of other nodes, so it should start off by reevaluating an attribute of r . Furthermore, we wish to choose an attribute of r whose arguments are guaranteed not to change, thereby assuring that the new value computed is the correct final value.

To make the right selection, it is necessary to know about transitive dependencies among the attributes of r . If b and c are attributes of r , and c transitively depends on b , then b must be reevaluated before c . These relationships can be represented by a directed graph whose vertices are the attributes of r and whose edges represent transitive dependencies among the attributes.

To discuss this idea more precisely, we make the following definitions:

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

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

- (2) The *deletion* of B from A is

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

Note that deletion deletes only edges.

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

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

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

Transitive dependencies among attributes of a given nonterminal instance s are represented locally by *subordinate* and *superior characteristic graphs*. We let each node s in a semantic tree be labeled with its subordinate characteristic graph, denoted $s.C$, its superior characteristic graph, denoted $s.\bar{C}$, or both. The subordinate characteristic graph at node s is the projection of the dependencies of the subtree rooted at s onto the attributes of s . To form the superior characteristic graph at node s , we imagine that the subtree rooted at s has been pruned from the semantic tree and project the dependency graph of the remaining tree onto the attributes of s . Note that the vertices of the characteristic graphs at s correspond to the attributes of s .

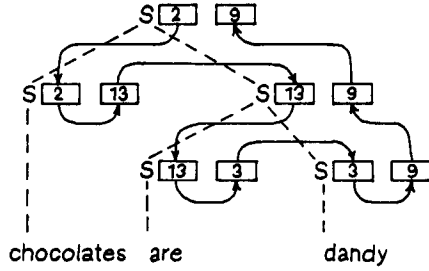


Fig. 2. A freestanding tree.

Formally, let s be a node in semantic tree T , let the subtree rooted at s be denoted T_s , and let V_s denote the vertices of $D(T)$ that correspond to the attributes of s . The subordinate and superior characteristic graphs at s are defined by

$$s.C \equiv D(T_s)/V_s;$$

$$s.\bar{C} \equiv (D(T) - D(T_s))/V_s.$$

Knowing the subordinate and superior characteristic graphs at the point of subtree replacement r allows us to construct the graph $r.C \cup r.\bar{C}$. An edge of this graph represents a transitive dependence between two attributes of r . An attribute in this graph that has in-edges depends on one of the other attributes of r ; consequently, it is not a suitable first choice for reevaluation. An attribute with in-degree zero does not depend on any of the other attributes of r and therefore is a suitable first choice. There is at least one such attribute because we are working with noncircular attribute grammars.

Example. Continuing our example, suppose the subtree “candy is dandy” of Figure 1 is replaced by the freestanding tree shown in Figure 2, which apparently originated in a context in which the previous word ended in column 2. Then, after grafting but before change propagation, the attribute values at the point of subtree replacement r are $r.previous = 2$ (from the inherited attribute of the root of the replaced subtree) and $r.last = 5$ (from the synthesized attribute of the root of the replacing subtree). At r , the subordinate characteristic graph $r.C$ is



reflecting the transitive dependence of $r.last$ on $r.previous$. The superior characteristic graph $r.\bar{C}$ is



the absence of edges reflecting the fact that $r.previous$ does not at all depend on $r.last$. The initial scheduling graph $r.C \cup r.\bar{C}$ is thus



consequently, change propagation starts by reevaluation of the inconsistent attribute value $r.previous = 2$.

5.2 The Updating Process

The previous section argues that the graph $r.C \cup r.\bar{C}$ must be constructed in order to choose the first attribute for reevaluation. In general, this graph represents a partial order that PROPAGATE must respect *throughout* the updating process. As updating progresses it is necessary to know more than just the dependency relationships among the attributes of r . When the value of an attribute instance is changed, all attributes that use it as an argument may become inconsistent; it is necessary to take into account the dependencies that involve these attributes.

To schedule reevaluations, PROPAGATE employs a graph M , called the *model*, and a set S , used as a work-list. M is a generalization of the graph discussed in the previous section, which represents dependencies among the attributes of a connected region of the tree, rather than just dependencies among the attributes of a single node. A vertex of M corresponds to an attribute; an edge of M represents a functional dependence, which may be either a direct dependence or a transitive dependence. In particular, M contains

- (1) edges representing direct dependencies in the modeled region of the tree,
- (2) edges of the superior characteristic graph of the apex of the region, and
- (3) edges of the subordinate characteristic graphs of the frontier of the region.

Characteristic-graph edges represent transitive dependencies transmitted entirely outside the modeled region of the tree.

M is initially $r.C \cup r.\bar{C}$, and S is initially the set of vertices of M with in-degree zero. As long as M covers the affected region of the tree, PROPAGATE does a topological enumeration of M . As each attribute b is enumerated, it is reevaluated, and the old value and the new value are compared; if they differ, and if b is an argument of an attribute instance that is outside the current model M , then M is expanded by one production instance so that it includes the successors of b .

To describe an expansion precisely, we define the functions ExpandedSubordinate and ExpandedSuperior, which produce graphs that are refinements of a node's characteristic graphs. If node s_0 is the parent node in production instance $p: \langle s_0, s_1, \dots, s_k \rangle$, we define

$$\text{ExpandedSubordinate}(s_0) \equiv D(p) \cup s_1.C \cup \dots \cup s_k.C.$$

For any other node s_j in the production instance, we define

$$\begin{aligned} \text{ExpandedSuperior}(s_j) \\ \equiv D(p) \cup s_0.\bar{C} \cup s_1.C \cup \dots \cup s_{j-1}.C \cup s_{j+1}.C \cup \dots \cup s_k.C. \end{aligned}$$

A model is expanded by the procedure EXPAND, given in Figure 3. In addition to deleting a characteristic graph from the model and augmenting the model with the corresponding expanded characteristic graph, an expansion also involves making insertions into the work-list S . At the time an attribute is brought into the model, if its in-degree in the model is zero, it is inserted into the work-list because it is ready to be reevaluated. Because an expansion is limited to a single production, it has a bounded cost for a given grammar.

PROPAGATE, stated below as Algorithm 3, interleaves topological enumeration and attribute reevaluation with calls to EXPAND. When PROPAGATE

```

EXPAND( $M, b, S$ ):
  let
     $M$  = a directed graph
     $b, c$  = attribute instances
     $S$  = a set of attribute instances
  in
    if there exists  $c$ , a successor of  $b$  in  $D(T)$  that is not in  $M$ 
      and  $\text{TreeNode}(c)$  is a child of  $\text{TreeNode}(b)$  then
         $M := (M - \text{TreeNode}(b).C) \cup \text{ExpandedSubordinate}(\text{TreeNode}(b))$ 
        Insert into  $S$  all vertices of  $\text{ExpandedSubordinate}(\text{TreeNode}(b))$  whose in-
        degree in  $M$  is 0
    if there exists  $c$ , a successor of  $b$  in  $D(T)$  that is not in  $M$ 
      and  $\text{TreeNode}(c)$  is the parent of  $\text{TreeNode}(b)$  then
         $M := (M - \text{TreeNode}(b).\bar{C}) \cup \text{ExpandedSuperior}(\text{TreeNode}(b))$ 
        Insert into  $S$  all vertices of  $\text{ExpandedSuperior}(\text{TreeNode}(b))$  whose in-
        degree in  $M$  is 0

```

Fig. 3. Expanding a model.

```

PROPAGATE( $T, r$ ):
  let
     $T$  = a fully attributed semantic tree
     $r$  = a nonterminal node of  $T$  containing any inconsistent attributes of  $T$ 
     $S$  = a set of attribute instances
     $M$  = a directed graph
     $b, c$  = attribute instances
    Oldvalue, Newvalue = attribute values
  in
     $M := r.C \cup r.\bar{C}$ 
     $S :=$  the set of vertices of  $M$  with in-degree 0 in  $M$ 
    while  $S \neq \emptyset$  do
      Select and remove a vertex  $b$  from  $S$ 
      Oldvalue := value of  $b$ 
      evaluate  $b$ 
      Newvalue := value of  $b$ 
      if Oldvalue  $\neq$  Newvalue and  $M$  does not contain all the successors of
         $b$  in  $D(T)$  then EXPAND( $M, b, S$ )
      for each  $c$  that is a successor of  $b$  in  $M$  do
        Remove edge  $(b, c)$  from  $M$ 
        if  $\text{in-degree}_M(c) = 0$  then Insert  $c$  into  $S$ 
      od
    od

```

Algorithm 3. Optimal-time change propagation.

terminates, M consists of all attributes of all production instances in which an attribute has changed value. All dependency-graph edges of this region have been inserted into M by the expansion process and have been removed from M by the topological enumeration process.

The number of vertices and edges introduced into M by an expansion is bounded by the size of the largest production in the grammar. M is only enlarged when we find a new member of **AFFECTED**; consequently, its maximum size is

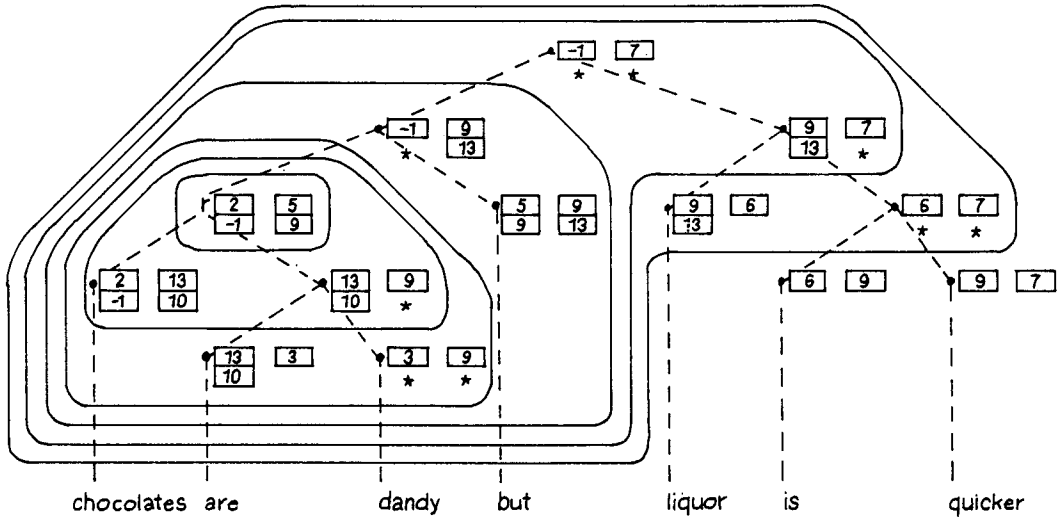
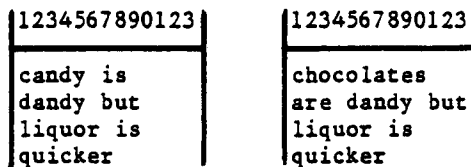


Fig. 4. Progression of models.

$O(|\text{AFFECTED}|)$). The cost of considering a vertex is one semantic-function application and a constant amount of bookkeeping work. The total number of semantic-function applications and the total cost of bookkeeping operations in PROPAGATE are $O(|\text{AFFECTED}|)$; thus, PROPAGATE is asymptotically optimal in time.

Characteristic-graph edges, representing transitive dependencies in $D(T)$, are crucial to the optimal behavior of PROPAGATE. The presence of characteristic-graph edges ensures that an attribute is never updated until all its ancestors are consistent; consequently, an attribute can never be assigned a temporarily incorrect value during updating. Removing a characteristic-graph edge allows PROPAGATE to skip, in unit time, arbitrarily large sections of $D(T)$ in which values do not change.

Example. As a result of replacing “candy is dandy” in Figure 1 with “chocolates are dandy” from Figure 2, PROPAGATE updates attribute values in the derivation tree to reflect the desired change in display layout:



The progression of six models generated in the course of change propagation is indicated in Figure 4. Attributes in AFFECTED appear as double boxes, with original value above final value. Every attribute that is eventually included in the model is reevaluated. Note that the model never expands to include the attributes of either “is” or “quicker” because, although the attribute S.previous of the

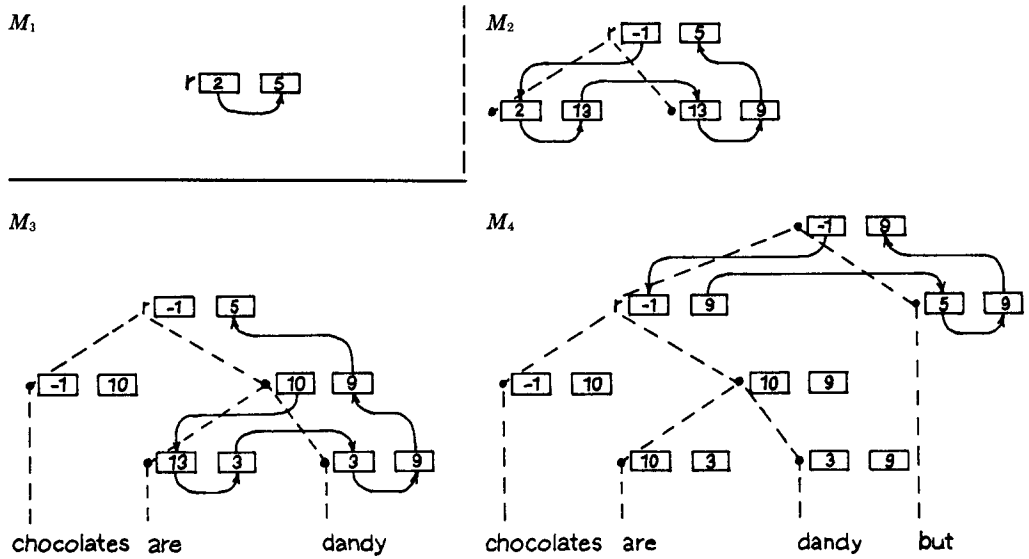


Fig. 5. The first four models.

phrase “is quicker” is recomputed, it does not change value. In Figure 5, detailed snapshots of the first four models just after expansion are shown.⁴

5.3 Correcting a Shortcoming

As presented, PROPAGATE has a shortcoming: an attribute instance that becomes part of M eventually gets evaluated, even if none of its arguments receives a new value. For example, Figure 4 contains nine attribute instances (marked with *’s) that need never have been reevaluated. Furthermore, this can happen to an attribute instance not just once, but up to three times. For example, note in Figure 5 that the appearance in model M_4 of an edge to the attribute $r.previous$ will result in its being reevaluated yet again, even though it clearly cannot get a new value from the second reevaluation. Since, in general, evaluations may be expensive, this is undesirable behavior. Note that this is not a counterexample to the optimal time bound; in general, attributes at the cursor location can be introduced into M (and evaluated) at most three times, while all other attributes can be introduced into M at most twice.

Such needless evaluations can be avoided by using an additional set, named `NeedToBeEvaluated`, as follows:

- (1) `NeedToBeEvaluated` is initialized to contain all the vertices of the initial model;

⁴Our sample attribute grammar serves to illustrate the behavior of the algorithm for arbitrary noncircular attribute grammars, even though it is an instance of the restricted class of L -attributed grammars, for which it is sufficient to reevaluate attributes during a left-to-right traversal starting at node r , descending no further when attributes no longer change value, and halting upon finding an ancestor of r whose attributes do not change value.

```

PROPAGATE( $T, r$ ):
  let
     $T$  = a fully attributed semantic tree prepared for propagation at  $r$ 
     $r$  = a nonterminal node of  $T$  containing any inconsistent attributes of  $T$ 
     $S, \text{NeedToBeEvaluated}$  = sets of attribute instances
     $M$  = a directed graph
     $b, c$  = attribute instances
    changed = Boolean
    Oldvalue, Newvalue = attribute values
  in
     $M := r.C \cup r.\bar{C}$ 
     $S :=$  the set of vertices of  $M$  with in-degree 0 in  $M$ 
     $\text{NeedToBeEvaluated} :=$  the set of vertices of  $M$ 
    while  $S \neq \emptyset$  do
      Select and remove a vertex  $b$  from  $S$ 
      changed := false
      if  $b \in \text{NeedToBeEvaluated}$  then
        Remove  $b$  from  $\text{NeedToBeEvaluated}$ 
        Oldvalue := value of  $b$ 
        evaluate  $b$ 
        Newvalue := value of  $b$ 
        if Oldvalue  $\neq$  Newvalue then
          changed := true
          if  $M$  does not contain all the successors of  $b$  in  $D(T)$  then
            EXPAND( $M, b, S$ )
      for each  $c$  that is a successor of  $b$  in  $M$  do
        Remove edge  $(b, c)$  from  $M$ 
        if in-degree $_M(c) = 0$  then Insert  $c$  into  $S$ 
        if changed = true then Insert  $c$  into  $\text{NeedToBeEvaluated}$ 
    od
  od

```

Algorithm 4. Improved optimal-time change propagation.

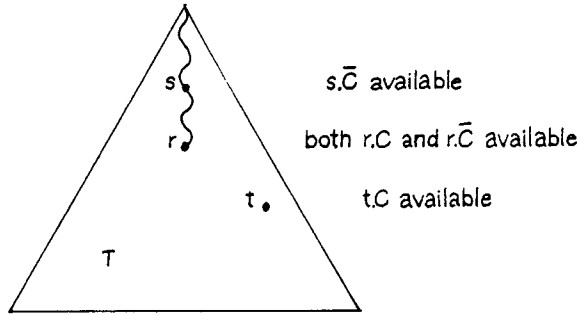
- (2) when the value of an attribute instance b is changed, every successor of b is inserted into NeedToBeEvaluated ;
- (3) when b is removed from S , it is reevaluated only if $b \in \text{NeedToBeEvaluated}$.

These ideas are incorporated into the version of PROPAGATE presented as Algorithm 4.

5.4 Characteristic Graphs, Cursor Motion, and Subtree Replacement

Until now, we have tacitly assumed that both subordinate and superior characteristic graphs were maintained at each node of the tree. However, a subtree replacement can radically alter transitive dependencies among attributes. In fact, because a subtree replacement at node r can alter characteristic graphs arbitrarily far away from r , maintaining every characteristic graph in the tree would make subtree replacements too expensive.

Fortunately, PROPAGATE does not need every characteristic graph. After a subtree replacement at node r , PROPAGATE never needs subordinate characteristic graphs at any of the nodes on the path from r to the root of the tree, and it never needs superior characteristic graphs anywhere else. PROPAGATE needs

Fig. 6. T prepared for propagation at r .

both characteristic graphs only at r . We say that T is *prepared for propagation at r* when, as in Figure 6,

- (1) r is labeled with both its subordinate characteristic graph, $r.C$, and its superior characteristic graph, $r.\bar{C}$;
- (2) each node s on the path from r to $\text{root}(T)$ is labeled with its superior characteristic graph, $s.\bar{C}$; and
- (3) each node t not on the path from r to $\text{root}(T)$ is labeled with its subordinate characteristic graph, $t.C$.

The editor maintains the invariant that the semantic tree is prepared for propagation at the position of the editing cursor. This invariant must be reestablished after each movement of the editing cursor to a new location. Every cursor motion can be defined as a sequence of the operations `AscendToParent` and `DescendToChild(j)`. Given that the editing cursor is positioned at node r of T and that T is prepared for propagation at r , `AscendToParent` has the side effect

$$\text{parent}(r).C := \text{ExpandedSubordinate}(\text{parent}(r)) / \{\text{attributes of parent}(r)\}.$$

`DescendToChild(j)` has the side effect

$$r_j.\bar{C} := \text{ExpandedSuperior}(r_j) / \{\text{attributes of } r_j\}$$

where r_j denotes the j th child of r . For a given grammar, each of these updates has unit cost. A movement of the editing cursor over a path of length m in the semantic tree costs $O(m)$.

The invariant that the tree is prepared for propagation at the position of the editing cursor must also be reestablished after a subtree replacement before `PROPAGATE` is called. By retention of subordinate characteristic graphs when a subtree is pruned, a freestanding tree is prepared for propagation at its root. After a subtree U at node r is replaced by a freestanding tree U' with root s , setting the superior characteristic graph at the cursor to be $r.\bar{C}$ and the subordinate characteristic graph to be $s.C$ reestablishes the invariant.

5.5 Operations Other Than Subtree Replacement

Existing language-based editors are either *structure editors*, such as Emily [12], MENTOR [7], and the Gandalf editor [26], *text editors* that employ an incre-

mental parser, such as the CAPS editor [39] and the PDE1L editor [27], or *hybrid editors* that combine both techniques, such as the editor of the Cornell Program Synthesizer [38].

Nearly all these editors represent programs as derivation trees of the language's abstract syntax, and the core of each system is a set of primitives for manipulating abstract-syntax trees. Exactly which operations are included among the primitives depends to a large extent on the system's user interface. In a structure editor, editing consists of a sequence of *deriving*, *pruning*, and *grafting* operations interleaved with cursor movements that shift the focus of attention in the tree. An editor employing an incremental parser uses additional primitives, such as a *split* operation that breaks up a tree into smaller trees and a *join* operation that assembles smaller trees into a larger tree [13, 29].

So far, the only operations we have discussed are subtree replacement and cursor movement. Insertion, deletion, and derivation have been treated as special cases of subtree replacement; thus, the algorithms presented earlier are suitable for language-based editors with a structural interface.

However, not all editing operations in language-based editors involve a single subtree replacement. In an editor that employs an incremental parser, a single program modification may involve a complex restructuring of the entire tree. In an editor that supports transformations, operations alter nodes in the interior of a tree, rather than a whole subtree. Although all editing operations in language-based editors can be defined as a sequence of subtree replacements and cursor motions, REPLACE is not suitable for such compositions of operations, because REPLACE updates the tree with each subtree replacement. Instead, updating of attribute values should be carried out only after the whole sequence of structural modifications has been completed.

As presented above, PROPAGATE cannot be used to update a tree after an arbitrary modification to a program tree, because until now we have assumed that all inconsistencies are initially confined to the attributes of a single tree-node. This restriction can be relaxed by making a simple change to PROPAGATE. Let R denote the smallest connected region of T that includes all nodes affected by a restructuring; all initially inconsistent attributes of T are attributes of this region. Instead of passing PROPAGATE a single node r , we pass R ; instead of initializing the model M to $r.C \cup r.\bar{C}$, PROPAGATE initializes M to

$$D(R) \cup \text{root}(R).\bar{C}$$

∪ subordinate characteristic graphs of all
nodes on the frontier of R .

The set NeedToBeEvaluated is initialized to contain all attributes of R ; the work-list S is initialized to all vertices of M with in-degree zero. By starting off in this way, with M containing all the inconsistent attributes, PROPAGATE will be able to update the tree correctly.

Similarly, if the normal-form restriction is relaxed, then inconsistencies are not restricted to the attributes of r but may occur in attributes of the parent, siblings, and children of r as well. By passing PROPAGATE the appropriate R , we may drop the normal-form restriction that has been assumed throughout this paper.

6. EFFICIENCY CONSIDERATIONS

The discussion above has ignored questions of efficiency other than the asymptotic behavior. This section discusses three ways by which performance can be improved, two of which are aimed at reducing the overhead of the attribute-updating mechanism, and one that is aimed at reducing the overhead of building and storing attribute values.

6.1 Optimizations for Attributes Defined by Identity Functions

When change propagation is employed to update a semantic tree, the new attribute value computed by each semantic-function application is compared to the old attribute value to see if changes need to be propagated further. Because testing equality of attribute values may be an expensive operation, it will be advantageous if we can avoid performing some of the equality tests.

Such an optimization is possible for attributes defined by *copy rules*, that is, defined by identity functions. This is an important optimization, because in practice a large proportion of semantic functions are identity functions [40].

The basic idea is that, if the first attribute instance in a chain of copy rules changes value, then the rest of the elements in the chain must also change value; thus it is wasteful to test the rest of the elements of the chain to see if they change value. It is important to note, however, that this is only true when the old values in the chain are consistent (i.e., identical); change propagation may terminate without reaching the end of the chain when some of the old values are inconsistent. However, the only possible inconsistent instances of attributes defined by copy rules are attributes of nonterminals in R , the region of the tree that was modified.⁵ Thus, it is necessary to treat each attribute of all nonterminals in R that are defined by copy rules as if it were not part of any chain, and to test its old value and its new value for equality whenever it is recomputed.

A further optimization is also possible for chains of copy rules. During change propagation, every time the model expands, it expands to cover dependencies in only a single additional production instance. For attributes defined by arbitrary semantic functions, this is crucial to the optimal behavior of PROPAGATE, because changes may not propagate beyond that production instance. However, for attributes defined by identity functions, once we detect that a chain has elements outside the model, the model may as well be expanded to include all such elements as well as all of their immediate successors. By expanding the model with this larger increment, we save the overhead involved in doing repeated expansions. Again, it is necessary to treat the attributes of nonterminals in R that are defined by copy rules as if they were not part of any chain.

6.2 Demand Attributes

Until now, we have abided by the requirement that each attribute in a program tree be given a consistent value after every editing operation. The assumption of this requirement is open to challenge. In particular, if the value of an attribute is in no way observable immediately after an editing operation, then there is no reason to insist on that value being correct. Instead, we could delay reevaluation

⁵ Recall that, in the case of subtree replacement, R consists of a single node.

until the value is needed in the computation of some aspect of the environment that *is* observable to the user. One may argue that the cost of such unobserved reevaluations is best prorated across many editing transactions in the interest of instantaneous response when the value finally is needed. On the other hand, repeated useless reevaluations of the attributes that are only rarely used might well destroy the ability of an editor to respond quickly to frequent, commonplace transactions.

We address this argument by extending our simple model of editing with a class of *demand attributes* that are given values only when necessary, that is, when a demand is placed on them for their value. A demand would arise either directly from a user query, from a need to display an attribute on the screen, or from a neighboring attribute needing to use the value as an argument. In some situations, it will be advantageous to intermix demand attributes and regular attributes, so we allow demand attributes to be arguments of regular attributes and vice versa.

Before discussing incremental evaluation of demand attributes, it is worthwhile to consider the demand concept as a paradigm for (nonincremental) evaluation. Earlier, in Section 4.3, we discussed how topological sorting can be turned into an algorithm for attribute evaluation by evaluating a vertex's semantic function when the vertex would normally be enumerated in the topological order [17]. However, there is another well-known algorithm for producing a linear ordering of a directed acyclic graph: start from the vertices with no successors, treat the graph as if all edges had been reversed in direction, and do a depth-first search, listing the vertices in endorder. This algorithm can also be turned into an attribute evaluation algorithm by evaluating a vertex's semantic function when the vertex is ready to be enumerated in the linear order. This algorithm can be thought of as a *demand evaluator* that fulfills demands for the values of the attributes with no successors.

In the version of PROPAGATE stated below as Algorithm 5, the demand attributes are treated just like the regular attributes when none of their arguments changes value; that is, a demand attribute keeps its old value if it is not a member of *NeedToBeEvaluated* when it is removed from the work-list. If one of its arguments has changed value, a demand attribute is given the value **null**, and receives a value later only if the value is needed for evaluating a regular attribute.

Note that the subordinate and superior characteristic graphs used by PROPAGATE may be thought of as demand attributes that are linked to the cursor. The computations referred to earlier as side effects of *AscendToParent()* and *DescendToChild()*, namely,

$$r.C := \text{ExpandedSubordinate}(r) / \{\text{attributes of } r\}$$

and

$$r.\bar{C} := \text{ExpandedSuperior}(r) / \{\text{attributes of } r\},$$

are just the semantic equations that define two graph-valued attributes. When the cursor is moved to node *r* or when a subtree replacement takes place at node *r*, a demand is placed on *r.C* and *r.C̄*.

It has not escaped our attention that allowing user-defined, cursor-linked, demand attributes could be a valuable mechanism for certain facilities of lan-

```

PROPAGATE( $T, r$ ):
  let
     $T$  = a fully attributed semantic tree prepared for propagation at  $r$ 
     $r$  = a nonterminal node of  $T$  containing any inconsistent attributes of  $T$ 
     $S$ , NeedToBeEvaluated = sets of attribute instances
     $M$  = a directed graph
     $b, c$  = attribute instances
    changed = Boolean
    Oldvalue, Newvalue = attribute values
  in
     $M := r.C \cup r.\bar{C}$ 
     $S :=$  the set of vertices of  $M$  with in-degree 0 in  $M$ 
    NeedToBeEvaluated := the set of vertices of  $M$ 
    while  $S \neq \emptyset$  do
      Select and remove a vertex  $b$  from  $S$ 
      changed := false
      if  $b \in$  NeedToBeEvaluated then
        Remove  $b$  from NeedToBeEvaluated
        Oldvalue := value of  $b$ 
        if  $b$  is a demand attribute then set  $b$  to null
        else for each argument  $c$  of  $b$  that is a demand attribute do
          DEMANDVALUE( $c$ )
        od
        evaluate  $b$ 
        Newvalue := value of  $b$ 
        if Oldvalue  $\neq$  Newvalue then
          changed := true
          if  $M$  does not contain all the successors of  $b$  in  $D(T)$  then
            EXPAND( $M, b, S$ )
          for each  $c$  that is a successor of  $b$  in  $M$  do
            Remove edge  $(b, c)$  from  $M$ 
            if in-degree $_M(c) = 0$  then Insert  $c$  into  $S$ 
            if changed = true then Insert  $c$  into NeedToBeEvaluated
          od
        od
    DEMANDVALUE( $b$ )
  let
     $b, c$  = attribute instances
  in
    while there exists  $c$ , an unavailable argument of  $b$ , do
      DEMANDVALUE( $c$ )
    od
    evaluate  $b$ 

```

Algorithm 5. Change propagation in the presence of demand attributes.

guage-based editors. For example, if one were to build a program-transformation editor using the scheme described in [11], the appropriate way to treat the “left-forward” and “right-backward” attributes, which are used to determine if a transformation preserves correctness in a given context, would be to make them demand attributes of the cursor.

6.3 Efficient Representations of Large Attributes

Attribute grammars often employ attributes belonging to data types that require a large data structure to represent a single value. For instance, attribute grammars

for programming languages commonly use attributes whose values are symbol tables. Both the space needed for storing large value-structures and the time needed for creating them may impose a great deal of overhead on an attribute-grammar-based system.

It is common practice for large attributes to be accessed through one level of indirection so that a single structure represents the value of several attribute instances that have identical values [32, 37]. A second benefit of this strategy is that semantic functions that are identity functions can copy pointers rather than having to copy entire structures.

This strategy can be taken a step further by arranging for the structures that represent *nearly* identical values to share *most* of the same substructure in common. When this is done, storing two nearly identical values requires only marginally more space than storing just one of the values; a semantic function that returns a value nearly identical to one of its arguments requires little time to construct the return value.

For example, a symbol table can be represented by a linked list that is accessed through a pointer to the head of the list. A linked-list representation of a symbol table is a sharable data structure because, when the names contained in one table are a subset of the names contained in the other table, both tables can be represented using a single list that has the common names at the tail of the list. A new name can be added to the symbol table by concatenating it to the head of the list. The full set of names is accessed through a pointer to the head of the list; the smaller set of names is accessed through a pointer to the common tail. The idea of implementing symbol-table attributes with linked lists so that storage can be shared in common is not a new one, but, to the authors' knowledge, linked lists have been the only sharable structures previously used in attribute-grammar-based systems.

However, a 2-3 tree is also a sharable structure if the tree is manipulated in a slightly nonstandard way. Normally, the elements of an ordered set are associated with the leaves of a tree in increasing order from left to right, and each interior node of the tree is labeled with five items: the addresses of its two or three children, the value of the largest element stored in the subtree rooted at the leftmost child, and the value of the largest element stored in the subtree rooted at the middle child [1]. During the course of such operations as INSERT, DELETE, MIN, SPLIT, and CONCATENATE, the tree may be restructured, in which case these fields take on new values.

We define a *sharable 2-3 tree* to be a standard 2-3 tree on which all operations are carried out in the usual fashion, except that, whenever one of the fields of an interior node M would normally be changed, we create a new node N that duplicates M , and we change the field in N . Of course, to be able to treat N as the child of $\text{parent}(M)$, it is necessary to change the appropriate child-field in $\text{parent}(M)$, so we create a new node that duplicates $\text{parent}(M)$, and so on, all the way to the root of the tree. Thus, new nodes are introduced for each of the original nodes along the path from M to the root of the tree. Because an operation that restructures a standard 2-3 tree may modify all of the nodes on the path to the root anyway, and because a single operation on a standard 2-3 tree that has n nodes takes at most $O(\log n)$ steps, the same operation on a sharable 2-3 tree introduces at most $O(\log n)$ additional nodes and also takes at most $O(\log n)$

steps. The new tree resulting from the operation shares the entire structure of the original tree except for the nodes on a path from N to the root, plus at most $O(\log n)$ other nodes that may be introduced in order to maintain the 2-3 property.

Sharable 2-3 trees are very versatile structures that can be used to implement efficient INSERT, DELETE, MEMBER, MIN, SPLIT, and CONCATENATE operations on ordered sets. For symbol tables, a sharable 2-3 tree representation is better than a linked-list representation because MEMBERSHIP tests and INSERT operations on an n -entry symbol table require at most $O(\log n)$ steps with a sharable 2-3 tree, but $O(n)$ steps (expected time) with a linked-list representation.

For further discussion of efficient representations of large attributes, the reader is referred to [34].

7. COMPARISON WITH ALTERNATIVE METHODS

The techniques presented above can be used in the design of systems capable of generating editors with context-dependent facilities. What sets this work apart from other work on language-based editors is the use of a formal framework for implementing such facilities. One approach that has been taken by previous systems for generating language-based editors has been to provide built-in mechanisms for a few special problems, such as block-structured scoping of variable names. This approach, which is used in the generator for Emily [12] and the generator for PCM [41], has the drawback that new applications will inevitably run up against the system's limitations.

A second approach, which is called the *semantic-action* approach, is to support designer-specified, semantic-action routines for making updates to the internal program representation; during editing, each operation that affects a node of type X invokes an action associated with the category X . An action is an imperative routine that can walk the program tree making updates to nodes of the tree as well as to global data structures. This approach is used in the generator for AVID [21] and the generator for Gandalf [24].

The major drawback of the semantic-action approach is the imperative nature of the editor specification. In addition to specifying semantic *actions* to occur when elements are inserted into a program, it is also necessary to specify semantic *retractions* to occur when elements are deleted from a program; the latter routines must provide a method for *undoing* the effects of the semantic actions. Though the procedures are associated with individual nonterminals of the grammar, such a specification is not as modular as it might first appear. Each routine controls a tree-walk and may depend on the structure of the entire tree; consequently, the structure of much of the grammar may have to be hard-coded into each routine. Because the order in which programs are developed is not fixed in advance, in writing the semantic routines the editor-designer is faced with the task of devising an updating strategy that maintains the data structure consistently, regardless of the order in which the program is developed. In practice, there may also be routines for other operations besides insertion and deletion, for example, operations associated with moving the editing cursor [26]. In general, the task of getting all of these procedures and their side effects to cooperate as desired may place a considerable burden on the editor-designer.

In contrast, the approach described in this paper, in which the syntax and semantics of the target language are specified with an attribute grammar, avoids the complexity inherent in the semantic-action approach because of the declarative nature of attribute grammars. An attribute grammar describes relationships between attributes, and the propagation of semantic information throughout a program tree is implicit in the formalism. As we have shown in this paper, when a program is modified, consistent relationships among the attributes can be reestablished automatically; consequently, there is no need for an explicit notion of undoing a semantic action or reversing the effect of a semantic analysis. When an editor is specified with an attribute grammar, the method for achieving a consistent state after an editing modification is not part of the specification.

The advantages of attribute grammars are offset by certain efficiency problems. Because attributes are defined by semantic equations with strictly local dependencies, when a program is modified (and attribute values recomputed), new values can flow from attribute to attribute only along edges of the derivation tree. With the semantic-action approach, however, the editor-designer is allowed to write action routines that update the tree by other means. In particular, by using data structures that record nonlocal dependencies in the tree, the updating routines can skip over arbitrarily large sections of the tree that an incremental attribute evaluator must visit node by node.

For example, suppose we want to enforce the constraint that the declarations and uses of identifiers in a program be consistent. With the semantic-action approach, this constraint can be implemented using a symbol table for each block in which the entry for an identifier i points to a chain of all uses of i in that block [15, 26]. When a declaration is deleted, the use chains are employed to immediately access uses of variables that were formerly declared. Of course, the action routines must also handle the additional task of maintaining the consistency of the nonlocal-dependency structures. For example, when a group of statements is deleted, the entire group must be traversed to find all uses that must be deleted from their use chains.

With the attribute-grammar approach, the constraint can be expressed with a grammar that threads a symbol-table attribute through the tree, as described in [4]. When declarations are inserted or deleted, the recomputed symbol-table attribute value propagates through the entire scope of the declarations; when a group of statements are inserted, the incremental evaluator essentially traverses the entire group, reevaluating semantic functions that check for undeclared-variable errors. On the other hand, a group of statements can be deleted by a simple subtree-pruning operation without traversing the group.

Johnson and Fischer have considered a combination of the attribute-grammar and the semantic-action approaches in which the standard definition of an attribute grammar is extended with the concept of nonsyntactic attribute flow [15]. Their objective is to have attributes flow "directly to where they are needed, rather than being restricted in their flow to paths in the parse tree of a program." They extend standard attribute-grammar notation with assertions on grammar symbols, where the assertions may contain quantifiers and some special relational operators for describing intersymbol relationships. The virtue of such an extension is that it allows a more perspicuous specification of context dependencies than the normal rule-by-rule specification, yet it remains formal and declarative.

While their goal is laudable, in [15] Johnson and Fischer, unfortunately, do not show how editors can be generated from their extended specification language. The essential difference between their approach and the semantic-action approach is that they make a distinction between the process of updating *links* between nonterminals related by an assertion and the process of updating attribute *values*; they use an incremental attribute evaluator for updating attribute values, but use action routines for updating links. For the example of type checking in an ALGOL-like language, reasonably efficient hand-coded “pseudo-evaluation” and “roll-back” actions for maintaining links are shown; what is not shown is how such an implementation can be derived automatically from the assertions of their extended specification language.

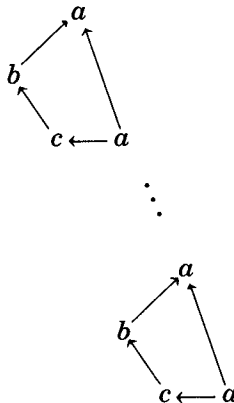
A second problem is that the incremental attribute-evaluation algorithm presented in [15] is just naive change propagation; consequently, unless the dependencies are restricted in such a way that there is never more than one path between any two attributes, the number of attributes considered may be arbitrarily greater than the number of attributes given new values, and the number of steps required to update the attributes may be nonlinear in the number of attributes considered. Furthermore, due to the presence of nonlocal dependencies, this problem can not be fixed by applying the methods we have presented in Section 5 of this paper.

APPENDIX

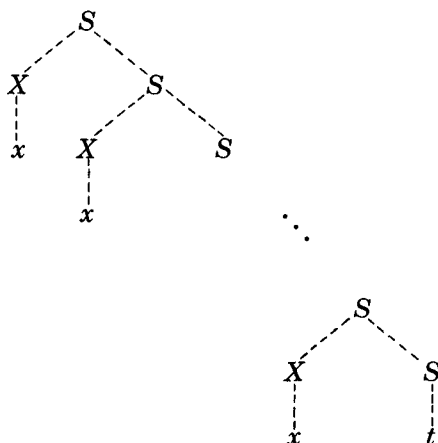
The following grammar causes Algorithm 1, naive change propagation, to require time nonlinear in the size of **AFFECTED**.

$$\begin{aligned}
 S_1 &\rightarrow X S_2 \\
 &\quad X.b = S_2.a \\
 &\quad S_1.a = X.b + S_2.a \\
 S &\rightarrow s \\
 &\quad S.a = 1 \\
 S &\rightarrow t \\
 &\quad S.a = 2 \\
 X &\rightarrow x \\
 &\quad X.b = X.c
 \end{aligned}$$

Dependency graphs of the derivation trees of this grammar have the form



If manipulations of the work-list S in Algorithm 1 obey LIFO order, updating proceeds depth-first through the dependency graph and may take exponential time. For example, suppose we replace the rightmost production instance, $S \rightarrow t$, with $S \rightarrow s$ in



If, after updating an attribute labeled a , PROPAGATE always inserts the successor labeled c into the work-list before inserting the successor labeled a , the time required to update the tree satisfies the recurrence relation

$$T(h) = 2T(h - 1) + \text{const}$$

where h is the height of the derivation tree. Thus, $T(h)$ is exponential in the worst case.

Alternatively, if manipulations of the work-list S obey FIFO order, updating proceeds breadth-first through the dependency graph and may exhibit quadratic behavior. The proof of this observation, omitted because of its length, appears in [34].

ACKNOWLEDGMENTS

We are grateful for the comments and suggestions of Bowen Alpern, Joe Bates, David Gries, Mark Horton, Susan Horwitz, Mike O'Donnell, Barry Rosen, and the three referees. We would also like to thank Bernard Martin for letting us use the facilities of the Conservatoire National des Arts et Metiers to prepare the manuscript of this paper.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
2. ARTHUR, J., AND RAMANATHAN, J. Design of analyzers for selective program analysis. *IEEE Trans. Softw. Eng. SE-7*, 1 (Jan. 1981), 39-51.
3. BABICH, W.A., AND JAZAYERI, M. The method of attributes for data flow analysis. Part I: Exhaustive analysis; Part II: Demand analysis. *Acta Inf.* 10, 3 (Oct. 1978), 245-272.
4. BOCHMANN, G.V. Semantic evaluation from left to right. *Commun. ACM* 19, 2 (Feb. 1976), 55-62.
5. DEMERS, A., REPS, T., AND TEITELBAUM, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In Conference Record of the 8th Annual ACM

- Symposium on Principles of Programming Languages, Williamsburg, Va., Jan. 26–28, 1981, pp. 105–116.
6. DONZEAU-GOUGE, V., HUET, G., KAHN, G., AND LANG, B. Programming environments based on structured editors: The MENTOR experience. Tech. Rep., INRIA, Le Chesnay, France, May 1980.
 7. DONZEAU-GOUGE, V., HUET, G., KAHN, G., LANG, B., AND LEVY, J.J. A structure-oriented program editor. Tech. Rep., IRIA-LABORIA, Le Chesnay, France, 1975.
 8. FANG, I. FOLDS, a declarative formal language definition system. Tech. Rep. STAN-CS-72-329, Computer Science Dept., Stanford Univ., Stanford, Calif., Dec. 1972.
 9. FARROW, R.W. Attributed Grammar Models for Data Flow Analysis. Ph.D. dissertation, Dept. of Mathematical Sciences, Rice Univ., Houston, Tex., May 1977.
 10. GANZINGER, H., RIPKEN, K., AND WILHELM, R. Automatic generation of optimizing multipass compilers. In *Information Processing 77; Proceedings of the IFIP Congress 77*, B. Gilchrist (Ed.). Elsevier North-Holland, New York, 1977, pp. 535–540.
 11. GERHART, S.L. Correctness-preserving program transformations. In Conference Record of the 2d ACM Symposium on Principles of Programming Languages, Palo Alto, Calif., Jan. 20–22, 1975, pp. 54–66.
 12. HANSEN, W. Creation of Hierarchic Text with a Computer Display. Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif., June 1971.
 13. JALILI, F., AND GALLIER, J.H. Building friendly parsers. In Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., Jan. 25–27, 1982, pp. 196–206.
 14. JAZAYERI, M., OGDEN, W.F., AND ROUNDS, W.C. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Commun. ACM* 18, 12 (Dec. 1975), 697–706.
 15. JOHNSON, G.F., AND FISCHER, C.N. Non-syntactic attribute flow in language based editors. In Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., Jan. 25–27, 1982, pp. 185–195.
 16. KASTENS, U., HUTT, B., AND ZIMMERMANN, E. *Lecture Notes in Computer Science*, vol. 141: *GAG, a Practical Compiler Generator*. Springer-Verlag, New York, 1982.
 17. KENNEDY, K., AND RAMANATHAN, J. A deterministic attribute grammar evaluator based on dynamic sequencing. *ACM Trans. Program. Lang. Syst.* 1, 1 (July 1979), 142–160.
 18. KNUTH, D.E. Semantics of context-free languages: Correction. *Math. Syst. Theory* 5, 1 (Mar. 1971), 95–96.
 19. KNUTH, D.E. Semantics of context-free languages. *Math. Syst. Theory* 2, 2 (June 1968), 127–145.
 20. KNUTH, D.E. *The Art of Computer Programming*, vol. 1: *Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968, pp. 258–268.
 21. KRAFFT, D. AVID: A System for the Interactive Development of Verifiably Correct Programs. Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1981.
 22. LEWIS, P.M., ROSENKRANTZ, D.J., AND STEARNS, R.E. Attributed translations. *J. Comput. Syst. Sci.* 9, 3 (Dec. 1974), 279–307.
 23. LORHO, B. Semantic attributes processing in the system DELTA. In *Lecture Notes in Computer Science*, vol. 47: *Methods of Algorithmic Language Implementation*, A. Ershov and C.H.A. Koster (Eds.). Springer-Verlag, New York, pp. 21–40.
 24. MEDINA-MORA, R. Syntax-Directed Editing: Towards Integrated Programming Environments. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Mar. 1982.
 25. MEDINA-MORA, R., AND FEILER, P. An incremental programming environment. *IEEE Trans. Softw. Eng. SE-7*, 5 (Sept. 1981), 472–482.
 26. MEDINA-MORA, R., AND NOTKIN, D.S. ALOE users' and implementors' guide. Tech. Rep. CMU-CS-81-145, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Nov. 1981.
 27. MIKELSONS, M., AND WEGMAN, M.N. PDEIL: The PLIL program development environment; principles of operation. Res. Rep. RC8513, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Nov. 1980.
 28. MILTON, D.R., KIRCHHOFF, L.W., AND ROWLAND, B.R. An ALL(1) compiler generator. In Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colo., Aug. 6–10, 1979. *SIGPLAN Notices (ACM)* 14, 8 (Aug. 1979), 152–157.
 29. MORRIS, J.M., AND SCHWARTZ, M.D. The design of a language-directed editor for block-structured languages. *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983.

- ured languages. In Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland, Ore., June 8-10, 1981. *SIGPLAN Notices (ACM)* 16, 6 (June 1981), 28-33.
30. NEEL, D., AND AMIRCHAHY, M. Semantic attributes and improvement of generated code. In Proceedings of the ACM Annual Conference, San Diego, Calif., Nov. 1974, vol. 1, pp. 1-10.
 31. RÄIHÄ, K.-J. Bibliography on attribute grammars. *SIGPLAN Notices (ACM)* 15, 3 (Mar. 1980), 35-44.
 32. RÄIHÄ, K.-J. Dynamic allocation of space for attribute instances in multi-pass evaluators of attribute grammars. In Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colo., Aug. 6-10, 1979. *SIGPLAN Notices (ACM)* 14, 8 (Aug. 1979), 26-38.
 33. RÄIHÄ, K.-J., SAARINEN, M., SOISALON-SOININEN, E., AND TIENARI, M. The compiler writing system HLP (Helsinki Language Processor). Rep. A-1978-2, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland, Mar. 1978.
 34. REPS, T. Generating language-based environments. Tech. Rep. 82-514 and Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1982.
 35. REPS, T. Optimal-time incremental semantic analysis for syntax-directed editors. In Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, N.M., Jan. 25-27, 1982, pp. 169-176.
 36. REPS, T. The Synthesizer Editor Generator: Reference manual. Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Sept. 1981.
 37. SCHULZ, W.A. Semantic Analysis and Target Language Synthesis in a Translator. Ph.D. dissertation, Univ. of Colorado, Boulder, Colo., 1976.
 38. TEITELBAUM, T., AND REPS, T. The Cornell Program Synthesizer: A syntax-directed programming environment. *Commun. ACM* 24, 9 (Sept. 1981), 563-573.
 39. WILCOX, T.R., DAVIS, A.M., AND TINDALL, M.H. The design and implementation of a table driven, interactive diagnostic programming system. *Commun. ACM* 19, 11 (Nov. 1976), 609-616.
 40. WILNER, W.T. Declarative Semantic Definition As Illustrated by a Definition of Simula 67. Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif., June 1971.
 41. YONKE, M.D. A knowledgeable, language-independent system for program construction and modification. Res. Rep. ISI/RR-75-42, Information Sciences Institute, Univ. of Southern California, Los Angeles, Calif., Oct. 1975.