Optimal-time Incremental Semantic Analysis for Syntax-directed Editors

Thomas Reps

TR 81-453

March 1981
(Revised November 1981)

To be presented at the
Ninth Annual ACM SIGACT-SIGPLAN Symposium on
Principles of Programming Languages
Albuquerque, NM, January 25-27, 1982

This work was supported in part by the National Science Foundation
under Grant MCS80-04218.

Optimal-time Incremental Semantic Analysis for Syntax-directed Editors*

Thomas Reps
Cornell University

Abstract

Attribute grammars permit the specification of static semantics in an applicative and modular fashion, and thus are a good basis for syntax-directed editors. Such editors represent programs as attributed trees, which are modified by operations such as subtree pruning and grafting. After each modification, a subset of attributes, AFFECTED, requires new values. Membership in AFFECTED is not known a priori; this paper presents an algorithm that identifies attributes in AFFECTED and recomputes their values. The algorithm is time-optimal, its cost is proportional to the size of AFFECTED.

1. Introduction

A syntax-directed editor provides powerful, yet disciplined, editing functions in terms of the grammar of a programming language. The success of syntax-directed editors, such as Emily [Han] and MENTOR [DHK], and syntax-directed programming environments, such as PDEIL [MiW], Gandalf [MeF], and the Cornell Program Synthesizer [TeR], has encouraged us to develop a tool for generating such systems from language descriptions. Because these editors are basically editors for the derivation trees of a context-free grammar, a new language can be handled by changing the tables that encode the grammar. A generator builds the proper tables from a grammar for the language.

In fact, this is an oversimplification because syntax-directed editors may have non-context-free facilities. For example, the Synthesizer enforces the constraint that all variables must be declared at the head of the procedure. Such features cannot be described by context-free grammars; to build editors that accommodate such features a more powerful formalism is needed that describes static semantics in addition to syntax. For maximum feed-back to the programmer, the editor should perform semantic analysis after each editing operation. For efficiency, analysis should be done incrementally, re-using as much old information as possible [Ros].

The static semantics of a language may be described in an applicative and modular fashion by an attribute grammar. An editor based on

---

* This work was supported in part by the National Science Foundation under Grant MCS80-04218.

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

this formalism represents a program as an attributed tree, and programs are modified by derivation tree operations such as pruning, grafting, and deriving [DRT]. A derivation tree modification directly affects the values of the attributes of the modification point; to perform incremental semantic analysis, attribute values must be updated in response to each modification. Out of the entire collection of attributes in the tree, only certain ones require new values; these we denote by AFFECTED, but it should be understood that AFFECTED is not known a priori. Elements of AFFECTED depend in some way on the attributes of the modification point, but not every attribute that depends on these attributes requires a new value. An optimal algorithm has a cost proportional to the size of AFFECTED.

A two-step incremental evaluation algorithm, consisting of nullification followed by evaluation, is presented in [DRT]. The first step propagates the special value **null** to all attributes that depend on the attributes of the modification point; new values are then introduced, and propagated through the tree by the second step. The two-step algorithm 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 two-step algorithm does extensive propagations even for operations that are potentially free.

The algorithm reported in this paper is a substantial improvement over that of [DRT]; both the total number of semantic function applications and the total cost of bookkeeping operations are bounded by $O(|AFFECTED|)$. The author has implemented the algorithm at Cornell as part of the Synthesizer Generator [Rep] -- a system for automatically generating program development systems. This system has been used to build experimental editors in which attributes control pretty-printing, language constraints, translation, global data-flow analysis, and verification.

2. Attribute grammars and attribute evaluation

Attribute grammars [Knu 68b] allow the semantics of a language to be specified along with its syntax. 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 are semantic functions that specify how attributes receive values in terms of the values of other attributes in the production. Attributes are divided into two classes: synthesized attributes and inherited attributes. Each semantic function defines a value for a synthesized attribute of the left-side nonterminal or an inherited attribute of a right-side symbol. Thus, synthesized attributes pass information up the derivation tree, and inherited attributes pass information down the derivation tree. For brevity, the arguments of the semantic function defining the value of attribute b are referred to as the arguments of b.

Functional dependencies among attributes in a production p or a derivation tree T can be represented by a dependency graph, denoted D(p) or D(T) respectively, defined as follows:

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

b) If attribute b is an argument of attribute c, the graph contains a directed edge (b', c').

Although closely related, an attribute instance b in derivation tree 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'.

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. To determine the "meaning" of a string, first construct its semantic tree with an assignment of **null** to each attribute instance, and then evaluate the semantic functions of as many attribute instances as possible. The latter process is termed attribute evaluation. In an attribute grammar, the order in which attributes are evaluated is arbitrary, subject to the constraint that each semantic function be evaluated only when all of its argument attributes are available, i.e. non-**null**. When all the arguments of an unavailable attribute instance are available, we say it is ready for evaluation.

A semantic tree is fully attributed if a value is associated with each of its attribute instances; it is partially attributed if the value of at least one attribute instance is unavailable. To further characterize semantic trees we introduce the notion of consistency. We say an attribute instance b is inconsistent if:

a) b is available, and

b) the arguments of b are available, and

c) the value of b is not equal to its semantic function applied to the arguments.

In all other cases, we say b is consistent. 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.

This paper deals only with attribute grammars that meet the conditions of well-formedness and noncircularity. An attribute grammar is noncircular when the dependency graph of every possible derivation tree is acyclic. 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 function for all the synthesized attributes of the left-side nonterminal and all the inherited attributes of the right-side symbols.

Using these definitions, Algorithm 1 states the method of [Knu 68b] for evaluating a semantic tree of a well-formed, noncircular attribute grammar. Each attribute instance of T causes exactly one semantic function evaluation. Furthermore, EVALUATE implicitly requires that the attributes of T be evaluated in an order that respects the partial order given by D(T), the attribute dependency graph of T.

Algorithm 1: Attribute evaluation.

```
EVALUATE(T):
   let
      T = an unevaluated semantic tree
      b = an attribute instance
   in
      do there exists b ready for evaluation →
            evaluate b
      od
```

In practice, EVALUATE must have some method of selecting the next attribute instance to evaluate; we clearly do not want the total cost of this bookkeeping to be more than $O(|D(T)|)$. One satisfactory method is to perform a topological sort [Knu 68a]. One may perform the topological sort first and then evaluate the attribute instances in the resulting order [LRS, KeR], or one may interleave sorting with attribute evaluation, a process we call topological evaluation.

The topological evaluation algorithm stated below as Algorithm 2 is essentially topological sort, except that in line (*) attributes are evaluated instead of being assigned a topological number. The algorithm uses a graph G, which is initially the dependency graph D(T), and a set S, which is initially the set of vertices of D(T) with indegree 0. S is a worklist containing the set of attributes that are ready for evaluation. G changes as attributes are evaluated to reflect the dependencies of unevaluated attributes only. The indegree of each vertex reflects the number of unevaluated attributes that the vertex depends upon. Thus, an edge (b, c) in G indicates that attributes b and c are unevaluated and c depends on b. This means that upon evaluation of an attribute b, all its outgoing edges (b, c) are deleted from G, and if this action causes c to have indegree 0, c is inserted in the worklist S because it is ready for evaluation.

The algorithm relies substantially on the fact that D(T), and hence G, is acyclic. If G has an edge, there must be at least one unevaluated attribute of G with indegree 0; thus, S is not empty. Upon termination S is empty, so G contains no edges; therefore, all attributes have been evaluated.

Later, in Section 4, we describe PROPAGATE, a procedure to update attribute values after modifications to a semantic tree. Like topological evaluation, PROPAGATE uses a worklist to keep track of attributes that are ready for reevaluation, and a graph to represent functional dependencies between attributes that have not been reevaluated.

3. Incremental attribute evaluation

The creation of a program using a syntax-directed editor entails growing a derivation tree. During program development, the derivation

Algorithm 2: Topological evaluation.

```
EVALUATE(T):
   let
      T = an unevaluated semantic tree
      G = a directed graph
      S = a set of attribute instances
      b, c = attribute instances
   in
      G := D(T)
      S := the set of vertices of G with indegree 0
      do S ≠ ∅ →
         Select and remove a vertex b from S
(*)      evaluate b
         do there exists c, a successor of b in G →
            Remove (b, c) from G
            if indegree_G(c) > 0 → skip
             ▯ indegree_G(c) = 0 → Insert c into S
            fi
         od
      od
```

tree is partial, but we wish to perform as much semantic analysis as possible incrementally. To facilitate this, we require a completing production $X \rightarrow \perp$ for each nonterminal symbol X. The symbol $\perp$ denotes "unexpanded" and defines the semantics of a missing subtree. 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 incrementally, its tree may be fully attributed.

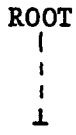
The modification of a program entails restructuring a derivation tree by tree operations defined as follows: Let T be a consistent, fully attributed, semantic tree and U be a subtree of T with root node r labeled X. U is pruned from T by removing the subtree rooted at r. Let U' be a consistent, fully attributed 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. In general, the resulting tree T' will not be consistent, but the inconsistencies are confined to the attributes of r'.[1] We define subtree replacement of U by U' as the pruning of U fol-

[1]The definition given above for grafting differs from that in [DRT], where the inherited attribute values of r are assigned to the inherited attribute instances of r'. When inherited attributes of r are retained, T' may have inconsistent attributes in the production instance that

lowed by the grafting of U' in its place.

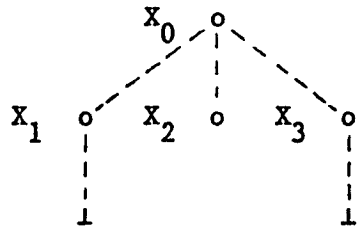
Because modifications may be made at any location in the program, the system must deal with free-standing trees derived from any of the nonterminals of the grammar, not just ones derived from the root symbol. For example, a subtree removed at X becomes a free-standing tree with root X. Such trees are retained so that they can be inserted into the program elsewhere. Free-standing trees are also made consistent and fully attributed. In a free-standing tree T, the arguments of the inherited attributes of its root are not in T; by convention we allow inherited attributes of the root of a free-standing tree to have arbitrary values.

The subtree replacement operation can be used to model most syntax-directed editors, regardless of the user interface. 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 deletion is viewed as the replacement of a subtree U (with root X) by an instance of the completing production of X. Each insertion at an unexpanded nonterminal labeled X is viewed as the replacement of an instance of the completing production of X by a free-standing tree U' with root X.

Different editors merely have different methods for creating U'. In an editor where each insertion is a single derivation, U' is a tree consisting of a single production instance with instances of completing productions derived from each nonterminal of the right part. For example, when the production $X_0 \rightarrow X_1\ X_2\ X_3$ is derived, where X_1 and X_3 are nonterminals, U' is:



In an editor where one makes insertions by typing program text directly, the text is sent to a parser, which constructs U'. In either case, EVALUATE is applied to U' to make it consistent and fully attributed before it is grafted onto the program tree.

The task of an incremental attribute evaluator is to produce a consistent, fully attributed tree after each subtree replacement. Of course, EVALUATE could be applied to completely reevaluate the tree, but

---

derived r', as well as in the production instance derived from r'.

the goal is to minimize work by confining the scope of reevaluation required after each subtree replacement. Let T' denote an inconsistent tree resulting from a subtree replacement, and T" denote T' after it has been made consistent and fully attributed. The dependency graphs D(T') and D(T") have the same vertices and edges, but some of the vertices may have different values. We define AFFECTED to be the set of vertices that have different values in the two graphs; thus, O(|AFFECTED|) is the minimal amount of work required after subtree replacement. The remainder of this paper describes PROPAGATE(T, r), an incremental attribute evaluator that makes tree T consistent after a subtree replacement at node r, and whose total cost is O(|AFFECTED|).

4. Change propagation

To update a tree, an incremental attribute evaluator must find and reevaluate inconsistent attribute instances. When the value of attribute instance b is changed to make b consistent, the successors of b may become inconsistent. The evaluator propagates such changes in attribute values by following attribute dependencies. This approach to incremental evaluation for attribute grammars is called change propagation [DRT].

Change propagation is appealing because if reevaluating an attribute yields a value equal to its old value, changes need not be propagated further. However, this strategy by itself does not guarantee optimal behavior; if attribute dependencies are followed blindly, change propagation will not behave optimally.

Non-optimal behavior is a consequence of assigning temporary values to attributes. To illustrate this, let us suppose b is a member of AFFECTED with value x that will eventually be updated to y. If b is temporarily assigned the value z, then other temporary values are apt to be assigned to attribute instances arbitrarily far beyond the boundaries of AFFECTED that transitively depend on b.

The main result of this paper is the procedure PROPAGATE, a change propagation algorithm whose total cost is proportional to the size of AFFECTED. As in topological evaluation, PROPAGATE keeps a worklist of attributes that are ready for reevaluation, and it schedules reevaluations according to an attribute's position in a graph that reflects dependencies among attributes that have not been reevaluated.

The secret to optimal updating is to keep the scheduling graph from growing larger than O(|AFFECTED|). On each step of PROPAGATE, 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.

4.1. 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 attributes of r when change propa-

gation is initiated. PROPAGATE would not make any progress reevaluating attributes of other nodes, so it should start off by reevaluating an attribute of r. However, if the wrong attribute of r is chosen, it could be given a temporary value, and PROPAGATE would not behave optimally.

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 use the following terminology:

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:

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

The deletion of B from A is:

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

Note that deletion deletes only edges.

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

Transitive dependencies are represented locally by subordinate and superior characteristic graphs. We let each node r in a semantic tree be labeled with its subordinate characteristic graph, denoted r.C, its superior characteristic graph, denoted r.$\bar{C}$, or both. The subordinate characteristic graph at node r is the projection of the dependencies of the subtree rooted at r onto the attributes of r.[2] To form the superior characteristic graph at node r, we imagine that the subtree rooted at r has been pruned from the semantic tree, and project the dependency graph of the remaining tree onto the attributes of r. Note that the vertices of the characteristic graphs at r are only attributes of r.

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

---

[2]Subordinate characteristic graphs have been used in [Knu 71] as part of the test for attribute grammar circularity and in [CoH] in connection with a (non-incremental) attribute evaluator.

$$r.C \equiv D(T_r)/V_r$$

$$r.\overline{C} \equiv (D(T) - D(T_r))/V_r$$

Knowing the subordinate and superior characteristic graphs at r allows us to construct the graph $r.C \cup r.\overline{C}$. 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 indegree 0 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.

4.2. The updating process

The previous section argues that the graph $r.C \cup r.\overline{C}$ 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. However, 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 uses a graph M, called the model, which is initially $r.C \cup r.\overline{C}$, and a set S, which is initially the set of vertices of M with indegree 0. M is a generalization of the graph discussed in the previous section; it represents dependencies among the attributes of a connected region of the tree, not just the 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, the model contains:

a) edges representing direct dependencies in the modeled region of the tree,

b) the edges of the superior characteristic graph of the apex of the region, and

c) 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.

As long as M covers the affected region of the tree, PROPAGATE does a topological evaluation of M. Every time an attribute b is reevaluated, the old value and the new value are compared; if they differ, and if b is an argument of an attribute that is outside the model, then the model is expanded by one production instance so that it cover the successors of b.

To describe an expansion formally, 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: $\langle s_0, s_1, \ldots, s_k \rangle$, we define:

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

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

$$\text{ExpandedSuperior}(s_j) \equiv D(p) \cup s_0.\overline{C} \cup s_1.C \ldots s_{j-1}.C \cup s_{j+1}.C \ldots s_k.C$$

A model is expanded by the procedure EXPAND, given below. In addition to deleting a characteristic graph from the model, and unioning the model with the corresponding expanded characteristic graph, an expansion also involves making insertions into the worklist S. At the time an attribute is brought into the model, if its indegree in the model is 0, it is ready to be reevaluated, so it is inserted into the worklist. Because an expansion is limited to a single production, it has unit cost for a given grammar.

4.3. PROPAGATE: An optimal-time incremental attribute evaluator

PROPAGATE, an incremental attribute evaluator stated below as Algorithm 3, interleaves topological evaluation with calls to EXPAND. When PROPAGATE terminates, M consists of the attributes of the portion of the tree affected by the updating process. 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 evaluation process.

```
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 TreeNode(c) is a child of TreeNode(b) then
         M := (M - TreeNode(c).C) ∪ ExpandedSubordinate(TreeNode(b))
         Insert into S all vertices of ExpandedSubordinate(TreeNode(b))
               whose indegree in M is 0
      if there exists c, a successor of b in D(T) that is not in M
            and TreeNode(c) is the parent of TreeNode(b) then
         M := (M - TreeNode(b).C̄) ∪ ExpandedSuperior(TreeNode(b))
         Insert into S all vertices of ExpandedSuperior(TreeNode(b))
               whose indegree in M is 0
```

Algorithm 3: Change propagation.

```
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 ∪ r.C̄
      S := the set of vertices of M with indegree 0 in M
      do S ≠ ∅ →
         Select and remove a vertex b from S
         Oldvalue := value of b
         evaluate b
         Newvalue := value of b
         if Oldvalue = Newvalue or M contains all the successors of b in D(T) →
               skip
          ▯ Oldvalue ≠ Newvalue and M does not contain all the successors
                 of b in D(T) →
               EXPAND(M, b, S)
         fi
         do there exists c, a successor of b in M →
            Remove edge (b, c) from M
            if indegree_M(c) > 0 → skip
             ▯ indegree_M(c) = 0 → Insert c into S
            fi
         od
      od
```

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 member of AFFECTED; consequently, its maximum size is O(|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(|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 temporary 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.

4.4. 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, 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 everywhere else. PROPAGATE needs both characteristic graphs only at r. We say that T is prepared for propagation at r when, as in Figure 1 below,

a) r is labeled with both its subordinate characteristic graph, r.C, and its superior characteristic graph, r.$\overline{C}$,

b) each node s on the path from r to root(T) is labeled with its superior characteristic graph, s.$\overline{C}$, and

c) each node t not on the path from r to 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:

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

DescendToChild(j) has the side effect:

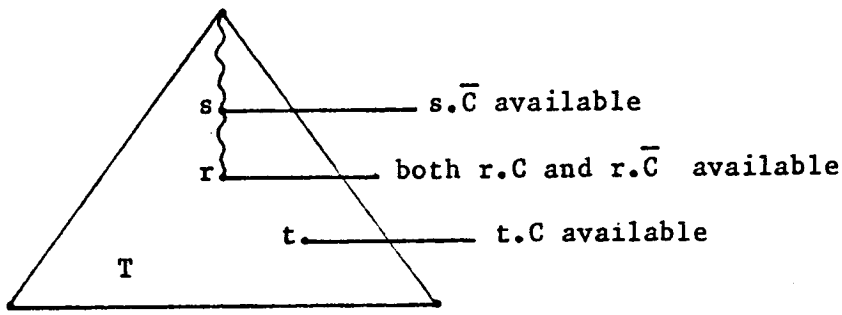


Figure 1. T prepared for propagation at r.

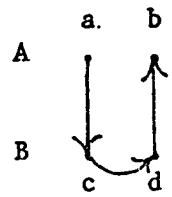
$$r_j.\overline{C} := \text{ExpandedSuperior}(r_j)/\{\text{attributes of } r_j\}$$

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 retaining subordinate characteristic graphs when a subtree is pruned, a free-standing tree is prepared for propagation at its root. After a subtree U at node r is replaced by a free-standing tree U' with root s, setting the superior characteristic graph at the cursor to be $r.\overline{C}$ and the subordinate characteristic graph to be s.C reestablishes the invariant.

4.5. 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. Furthermore, this can happen to an attribute instance not once, but up to three times. For example, when the cursor is positioned at B in:



The initial configuration of M is:

In the following action sequence c is evaluated twice:

```
evaluate c
evaluate d
expand M, introducing a and b
evaluate b
evaluate a
evaluate c
```

It is clear that c cannot get a new value from the second evaluation, and since evaluations may be expensive this is undesirable behavior. Note that this is not a counter-example 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.

We can avoid such needless evaluations by using an additional set, named NEEDEVAL, as follows:

a) NEEDEVAL is initialized to contain all the vertices of the initial model

b) when the value of an attribute instance b is changed, every successor of b is inserted into NEEDEVAL

c) when b is removed from S, it is reevaluated only if b∈NEEDEVAL.

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

Algorithm 4: Change propagation.

```
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, NEEDEVAL = sets of attribute instances
      M = a directed graph
      b, c = attribute instances
      changed = Boolean
      Oldvalue, Newvalue = attribute values
   in
      M := r.C ∪ r.C̄
      S := the set of vertices of M with indegree 0 in M
      NEEDEVAL := the set of vertices of M
      do S ≠ ∅ →
         Select and remove a vertex b from S
         if b ∉ NEEDEVAL → changed := false
          ▯ b ∈ NEEDEVAL →
            Remove b from NEEDEVAL
            Oldvalue := value of b
            evaluate b
            Newvalue := value of b
            if Oldvalue = Newvalue → changed := false
             ▯ Oldvalue ≠ Newvalue and M contains all the successors of b in D(T) →
                  changed := true
             ▯ Oldvalue ≠ Newvalue and M does not contain all the
                     successors of b in D(T) →
                  changed := true
                  EXPAND(M, b, S)
            fi
         fi
         do there exists c, a successor of b in M →
            Remove edge (b, c) from M
            if indegree_M(c) > 0 → skip
             ▯ indegree_M(c) = 0 → Insert c into S
            fi
            if ¬changed → skip
             ▯ changed → Insert c into NEEDEVAL
            fi
         od
      od
```

5. Summary

This paper has discussed a necessary algorithm if attribute grammars are to be used for building syntax-directed programming environments. In an language-specific environment based on this formalism, the static semantics of the language are specified by an attribute grammar. The objects handled by the system are represented internally as consistent, fully attributed semantic trees. When an object is altered, consistent values are reestablished throughout its tree incrementally in response to each modification.

Each tree has a distinguished location, called the cursor position, that may be moved around in the tree. The system maintains the invariant that a tree is prepared for propagation at the position of its cursor. For a given grammar, moving the cursor to a parent or to a child and reestablishing the invariant has unit cost; a movement of the cursor over a path of length m in the tree costs O(m).

The basic editing operation is REPLACE(T, r, U'), given below, which replaces the subtree of T that is rooted at r by U' and reestablishes consistent values throughout T.

```
REPLACE(T, r, U'):
   let
      T = a consistent, fully attributed, semantic tree that is prepared
            for propagation at r
      r = a node of T
      U' = a consistent, fully attributed, semantic tree that is prepared
            for propagation at its root, and the syntactic label of its root
            matches the syntactic label of r
   in
      prune the subtree at r from T
      graft U' onto T at r
      PROPAGATE(T, r)
```

In REPLACE, both the total number of semantic function applications and the total cost of other bookkeeping are bounded by O(|AFFECTED|).

We have built a prototype environment construction tool, called the Synthesizer Generator [Rep], that handles static semantics using the methods described in this paper.

6. Acknowledgments

I am deeply indebted to Tim Teitelbaum and Alan Demers, first, for the pleasurable collaboration that resulted in [DRT], second, for encouraging me to look for still better results, and finally, for suggestions while this paper was being prepared. Bowen Alpern, Joseph Bates, David Gries, Mark Horton, Susan Horwitz, and Barry Rosen all provided helpful comments.

References

[CoH]
Cohen, R. and Harry, E. Automatic generation of near-optimal linear-time translators for non-circular attribute grammars. Conference Record of the Sixth ACM Symposium on Principles of Programming Languages, January 1979, 121-134.

[DRT]
Demers, A., Reps, T., and Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. Conference Record of the Eighth ACM Symposium on Principles of Programming Languages, January 1981, 105-116.

[DHK]
Donzeau-Gouge, V., Huet, G., Kahn, G., Lang B., and Levy, J.J. A structure-oriented program editor. Technical Report, IRIA-LABORIA, France 1975.

[Han]
Hansen, W. Creation of hierarchic text with a computer display. Ph.D Thesis, Computer Science Dept., Stanford University, June 1971.

[KeR]
Kennedy, K. and Ramanathan, J. A deterministic attribute grammar evaluator based on dynamic sequencing. ACM Trans. on Prog. Lang. and Sys. 1, 1 (July 1979) 142-160.

[Knu 68a]
Knuth, D.E. The Art of Computer Programming, Vol. I: Fundamental Algorithms. Addison-Wesley, Reading, Mass., 1968, 258-268.

[Knu 68b]
Knuth, D.E. Semantics of context-free languages. Mathematical Systems Theory 2, 2 (1968), 127-145.

[Knu 71]
Knuth, D.E. Semantics of context-free languages: correction. Mathematical Systems Theory 5, 1 (1971), 95-96.

[LRS]
Lewis, P.M., Rosenkrantz, D.J., and Stearns, R.E. Attributed translations. Journal of Computer and Systems Sciences 9, 3 (December 1974), 279-307.

[MeF]
Medina-Mora, R. and Feiler, P. An incremental programming environment. IEEE Transactions on Software Engineering SE-7, 5 (September 1981) 472-482.

[MiW]
Mikelsons, M. and Wegman, M.N. PDE1L: The PL1L program development environment principles of operation. Research report RC8513, IBM Watson Research Center, Yorktown Heights, November 1980.

[Rep]
Reps, T. The Synthesizer Editor Generator: Reference Manual. September 1981.

[Ros]
Rosen, B.K. Linear cost is sometimes quadratic. Conference Record of the Eighth ACM Symposium on Principles of Programming Languages, January 1981, 117-124.

[TeR]
Teitelbaum, T. and Reps, T. The Cornell Program Synthesizer: a syntax-directed programming environment. Communications of the ACM 24, 9 (September 1981) 563-573.