RICE UNIVERSITY
Houston, Texas

# Hierarchical Attribute Grammars: Dialects, Applications and Evaluation Algorithms

by

**Alan Carle**
carle@cs.rice.edu

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

**Doctor of Philosophy**

May, 1992

Rice University
Department of Computer Science
Technical Report
Rice COMP TR97-270
January, 1997

# Hierarchical Attribute Grammars: Dialects, Applications and Evaluation Algorithms

Alan Carle
(carle@cs.rice.edu)

## Abstract

Although attribute grammars have been applied successfully to the specification of many different phases of analysis and transformation of complex language processing systems, including type checking, data flow analysis, constant propagation and dead code elimination, little success has been achieved in applying attribute grammars to the specification of complete systems such as multiple pass code optimizers or automatic parallelizers. This is the direct result of the failure of typical attribute grammar dialects to provide any means for composing attribute grammar specifications of sub-computations to create a specification of a complete hierarchical computation.

This dissertation introduces the notion of *hierarchical attribute grammars*, a set of attribute grammar dialects that are suitably expressive to permit the natural description of complex computations through the composition of attribute grammar specified sub-computations within the context of the original attribute grammar formalism. The set of hierarchical dialects includes Schulz's attributed transformations, the Attribute Coupled Grammars of Ganzinger and Giegerich, SSL, the specification language of the Synthesizer Generator of Reps and Teitelbaum, the Higher Order Attribute Grammars of Vogt, Swierstra and Kuiper, and a new dialect *Modular Attribute Grammars*. The relationships between these five dialects are examined, and examples of Modular Attribute Grammar specifications are presented.

For hierarchical attribute grammar dialects to be useful, efficient batch and incremental evaluators for hierarchical dialects must be developed. Therefore, the majority of this dissertation is dedicated to the presentation of new batch and incremental evaluation algorithms for hierarchical specifications.

# Acknowledgments

I would like most to thank my thesis advisor, Lori Pollock, and the members of my thesis committee, Robert Bixby, Hans Boehm and Robert Hood for their advice, cooperation, and perseverance through both time and space.

I would also like to thank Federal Express for being the shepherd of my thesis, carefully guiding drafts, corrections, and signatures, back and forth between Texas and Pennsylvania, Texas and Germany, and Texas and California.

I am grateful to IBM and to the National Science Foundation for partial support of this research.

Finally, I would also like to thank my wife, Janice, for her continued love and support throughout my many years of graduate school. I hope she continues to love me when I'm not a student.

# Contents

# Chapter 1

# Introduction and Overview

## 1.1 Introduction

Attribute grammars have been found to be quite effective for defining the individual analysis and transformation components of language translators, but notoriously ineffective for describing large transformational systems such as complete language translators, including compilers, optimizers and parallelizers. Very early in the history of attribute grammars it was discovered that complex systems could be described by informally composing sequences of simple translators, each described by its own attribute grammar specification, to form a single complex translator. Unfortunately this informal approach suffers from two significant deficiencies: it only provides for sequential composition of modules, and, since the resulting system description is not an attribute grammar, the complete system is not amenable to incremental evaluation by the efficient incremental evaluation techniques developed for attribute grammars.

More recently, several new attribute grammar dialects have been developed with the common goal of permitting complex specifications to be built through explicit composition of smaller, more manageable modules, each independently defined by its own attribute grammar specification. We refer to the set of new dialects as the class of *hierarchical attribute grammars*. The class includes Schulz's attributed transformations, the Attribute Coupled Grammars of Ganzinger and Giegerich, the specification language of the Synthesizer Generator of Reps and Teitelbaum called SSL, the Higher Order Attribute Grammars of Vogt, Swierstra and Kuiper, and a new dialect introduced in this dissertation, *Modular Attribute Grammars*. We will refer to attribute grammar dialects that do not support the modular approach to specification of complete systems as *non-hierarchical attribute grammars*.

The hierarchical attribute grammar dialects support a "divide and conquer" approach to the problem of specifying a complex system in which the various transformation and analysis modules comprising the complete system are separated into independently specified modules much like the modules of a multipass optimizing compiler. Each module in the system is defined as a mapping from annotated tree-structured (or dag-structured) values to annotated tree-structured (or dag-structured) values, and is described by its own attribute grammar. Composition of modules is supported directly within the attribute grammar framework by permitting modules defined by attribute grammars to be recursively invoked as semantic functions.

For the hierarchical attribute grammar dialects to be accepted as a tool for describing complex language processing tools, efficient algorithms for batch and incremental evaluation of hierarchical attribute grammar specifications must be developed. If a set of functions can be composed together, then the batch evaluators for each of the attribute grammar specifications of the functions can also be composed together, since the batch evaluator for a module maps a module's input to its output. Therefore, a batch evaluator for hierarchical attribute grammars simply requires the recursive application of a batch evaluator for non-hierarchical attribute grammars to each of the modules in the hierarchical specification.

Traditional incremental evaluators for attribute grammars map a database of previously computed facts for a module and a description of a change to the module's input into a new database of computed facts for that module and the module's new output. If a set of functions can be composed together, then the incremental evaluators for each of the attribute grammar specifications of the functions cannot be directly composed together, since the incremental evaluator for each module does not construct a description of a change to the module's previous output value. Therefore, an incremental evaluator for hierarchical attribute grammars cannot be constructed by simply applying an incremental evaluator to each module of the hierarchical specification. The majority of this dissertation is dedicated to the presentation of new incremental evaluation algorithms for hierarchical attribute grammars.

## 1.2 Research Overview

The dissertation is organized as follows:

- Chapter 2 reviews the attribute grammar formalism and presents tree-walking and applicative algorithms for both batch and incremental evaluation of attribute grammars.

- Chapter 3 presents two of Knuth's early attribute grammar examples which we consider to be hierarchical specifications, and then surveys the existing hierarchical dialects. Based on deficiencies in the notion of annotated abstract syntax underlying each of the survey dialects, an improved notion of annotated abstract syntax and a new hierarchical attribute grammar dialect, Modular Attribute Grammars, which is based on the improved notion of annotated abstract syntax, are introduced. We believe Modular Attribute Grammars to be better suited than any of the surveyed dialects for the modular specification of complex hierarchical systems.

- Chapter 4 demonstrates the application of Modular Attribute Grammars to the description of a pair of source-to-source code optimizers for a very simple programming language. Both optimizers contain a module that is responsible for performing data flow analysis over a program. Data flow analysis is typically first described as a circular attribute grammar, and then, since most attribute grammar evaluators are incapable of evaluating circular specifications, the circular attribute grammar is translated into a significantly more complex non-circular attribute grammar. It is demonstrated that, instead of using circularity to define data flow analysis, it is possible to specify data flow analysis using a hierarchical attribute grammar that uses recursion between non-circular attribute grammars instead of circularity within a single attribute grammar.

- Chapter 5 presents both tree-walking and applicative algorithms for batch evaluation of hierarchical attribute grammars.

- Chapter 6 examines tree-walking algorithms for incremental evaluation of hierarchical attribute grammars and introduces the notion of *matching* which is responsible for explicitly comparing a module's old and new input terms in order to determine how the new input term has been derived from the old input term. As an example of a matching algorithm and in order to introduce notation, Teitelbaum and Chapman's published algorithm for incremental evaluation of Higher Order Attribute Grammars is presented. A high-level description of a matching-based incremental evaluator for modules of hierarchical attribute grammars is then presented along with a lower bound on the cost of evaluating modules using this matching-based approach. Finally, it is demonstrated that this matching-based approach is unable to evaluate modules of a hierarchical specification optimally since it does not take into consideration the values of attributes during matching.

- Chapter 7 presents four matching algorithms for use in the evaluator for modules of a hierarchical attribute grammar described in Chapter 6. The algorithms are designed to satisfy a pair of heuristics that are intended to maximize the amount of information that is retained for use during later invocations of each module instance.

- Chapter 8 presents an incremental evaluation algorithm for hierarchical attribute grammars that is designed to take attribute context into account in its selection of old attribute subtrees in an attempt to minimize the cost of evaluating module instances in a hierarchical system. The evaluator evaluates entire hierarchical specifications optimally with respect to any evaluation algorithm that explicitly constructs a consistent attribute tree for each module instance and requires that the attribute tree be traversed from its root by following parent and child links between adjacent attribute tree nodes. The chapter concludes by describing the applicative approach to evaluation of hierarchical attribute grammars and by comparing the applicative approach with the optimal incremental evaluation algorithm.

- Chapter 9 summarizes the major contributions of this thesis.

# Chapter 2

# An Attribute Grammar Review

We now present a short review of attribute grammars and batch and incremental evaluation algorithms for attribute grammars. For a more formal presentation of attribute grammars, the reader should consult the survey by Deransart, Jourdan and Lorho [DJL88] and the two papers [Cou84] and [Eng84].

## 2.1 The Language

Knuth introduced the attribute grammar formalism as a tool for mathematically defining the semantics of context free languages [Knu68, Knu71b, Knu71a]. However, as a tool for defining semantics, attribute grammars have fallen out of favor in preference to other techniques, mainly denotational semantics. Currently, attribute grammars are used to specify programming language translators and incremental language processing tools.

An attribute grammar consists of a context free grammar $G$ and an attribute system $AS$. $AS$ associates sets of storage cells known as *attributes* with each symbol in the grammar $G$, and sets of equations known as *semantic rules* with each production in $G$. Each semantic rule contains a *semantic function* that defines the value of an attribute at a derivation tree node in terms of the values of a subset of the attributes at adjacent nodes in the derivation tree. Each semantic function is actually a composition of functions which are themselves confusingly referred to as semantic functions. The entire composition of functions in a semantic rule is sometimes referred to as the *body* of the semantic rule.

Attributes are divided into two sets: *inherited* attributes which communicate information down towards the leaves of a derivation tree, and *synthesized* attributes which communicate information up towards the root of the derivation tree. Inherited attributes of a derivation tree node $N$ are defined by semantic rules associated with the production applied at $N$'s parent, whereas synthesized attributes at $N$ are defined by rules associated with the production applied at $N$, itself. Attributes can additionally be classified as IN or OUT attributes of a production depending on whether they carry information into or out of a production applied at some node in a derivation tree. IN attributes, consisting of the inherited attributes at the root of a production, say $R$, and the synthesized attributes at each of the children of $R$, communicate information into the production for use as arguments of semantic rules associated with production $R$. OUT attributes, consisting of the synthesized attributes at the root of a production $R$ and the inherited attributes at each of the children of $R$, communicate information out of the production for use as arguments of semantic rules associated with the productions applied just above and below the production $R$. An attribute grammar is in *Bochmann Normal Form* if and only if every OUT attribute of a production is defined in terms of zero or more IN attributes of that production [Boc76]. Throughout this dissertation, we assume that attribute grammars are normalized in this fashion.

An attribute system describes a mapping from a derivation tree $D_S$ of a sentence $S$ in a context free language $L(G)$ into a computation graph $C_{D_S}$. Each vertex in $C_{D_S}$ corresponds precisely to an attribute associated with a node in $D_S$. Each vertex is labeled with the semantic function in its defining semantic rule. Edges in $C_{D_S}$ represent the functional dependencies between attributes associated with nodes in $D_S$. A set of distinguished vertices in $C_{D_S}$ corresponds to a set of attributes associated with the root of the derivation tree $D_S$ and is considered to be the set of *result vertices*. The size of $C_{D_S}$ is linear in the size of $D_S$, since an attribute grammar associates a (usually small) constant number of attributes with each of the derivation tree nodes.

The "meaning" of a derivation tree $D_S$ defined by an attribute grammar is the value that results from explicitly constructing the computation graph $C_{D_S}$ for the derivation tree according to the attribute grammar specification, and then "evaluating" the computation graph. More explicitly, let $p$ be a production in the context free grammar $G = (N, T, P, Z)$ underlying the attribute grammar specification:

$$p : X_0 \longrightarrow X_1 X_2 \ldots X_{n_p}$$

where $X_i \in N$. If $out(p, i)$ and $in(p, i)$, $p \in P, 0 \leq i \leq n_p$, are the set of output and input attributes of $X_i$, then the local dependency graph $LDG(p)$ represents the relation:

$$in(i) \;\; LDG(p) \;\; out(j) \Longleftrightarrow in(i) \;\text{ is used to compute }\; out(j).$$

The computation graph $C_{D_S}$ is constructed by *pasting* together a copy of the local dependency graph for the production applied at each node of the derivation tree. The computation graph can be constructed during a depth-first traversal of $D_S$. Whenever a node is reached during the traversal, the appropriate local dependency graph is created and attached to $C_{D_S}$. Evaluation of $C_{D_S}$ can then be performed in time linear in the size of $D_S$ by topologically sorting the graph, evaluating vertex functions in topological order, and finally returning the values of the result vertices.

Unfortunately, even though this technique is extremely simple and will run in time linear in the size of the input derivation tree, there are several problems with this approach. First, all attributes are evaluated, including those that do not contribute to the value of the result attributes. For many purposes, we can ignore this problem by assuming that all attributes in an attribute grammar are *useful*, i.e., all attributes contribute to the values of the result attributes. Therefore, at least a linear amount of work must be performed during attribute evaluation, assuming that semantic function evaluations require constant time. Second, although the graph-evaluation approach can be applied to any well-formed attribute grammar, the generality of the approach requires applying a *dynamic* evaluation strategy that makes it impossible to take advantage of locality and predictability to streamline the evaluation process. A *static* evaluation strategy is described below. Third, the graph-evaluation approach is only guaranteed to terminate with the successful computation of the result attribute when it is applied to evaluate a *well-formed* attribute grammar – an attribute grammar that generates only acyclic dependence graphs for any input tree. Unfortunately, the membership test for well-formed attribute grammars is the circularity test first described correctly in [Knu71b] which has been proven to be intrinsically exponential [JOR75, Jaz81].

Due to the space requirements of the computation graph and the generality of the graph traversal, the computation graph-based evaluation approach has typically been considered to be the "defining" technique for attribute grammar evaluation – it has not been considered to be a practical evaluation algorithm. More efficient evaluation techniques for attribute grammars have been devised for subclasses of well-formed attribute grammars. Two of these evaluation approaches are presented in the next section.

Modern usage of attribute grammars drops the suggestion that the input to an attribute grammar evaluator is the text string $S$ that must be parsed to create the derivation tree $D_S$. We refer to this original approach as the *string-based approach*. Instead, the input to the evaluator will be taken to be an abstract syntax tree built by applying node constructors representing productions in some underlying context free grammar. This approach will be referred to as the *structure-based approach*. This dissertation adopts the notion that an attribute grammar defines a function directly over abstract syntax trees. An operator *op* will be associated with each production in a context free grammar $G = (N, T, P, Z)$, and each syntax tree node will be labeled with the operator *op* and a nonterminal $n \in N$. We will say that a production $p \in P$ is "applied" at a syntax tree node, but the node is labeled with the operator *op* that corresponds to $p$.

Throughout this dissertation, the evaluation process will be referred to in the following ways:

- "Evaluating an attribute grammar" – the abstract syntax tree $D_S$ is implied,
- "Evaluating a tree" – the existence of an attribute grammar $AG$ is implied.
- "Attributing a tree $D_S$" – the process of evaluation computes the values of "attributes" associated with the nodes of $D_S$. The existence of an attribute grammar $AG$ is implied.

Knuth proved, using a now well-known construction, that the attribute grammar formalism is powerful enough to define any computable function over sentences in context free languages if the semantic function sublanguage is capable of defining any computable function [Knu68]. However, expressing general computations as attribute grammars by applying Knuth's construction is not a particularly useful method for specifying the solutions to complex problems since the majority of the computation will be encapsulated within a very small number of very complex semantic functions. These attribute grammars are generally considered to be "not in the spirit of the attribute grammar formalism." However, there is no good definition of what constitutes an attribute grammar that is "in the spirit." Traditional attribute grammar evaluation schemes are based upon the assumption that the computation described by an attribute grammar is evenly distributed throughout the semantic functions.

## 2.2 Approaches to Batch Evaluation

We now present two approaches to the evaluation of attribute grammars:

- The *Tree-Walking Approach* traverses a syntax tree computing attribute values that are then stored in cells associated with each of the syntax tree nodes. "Associated" does not necessarily mean "attached."

- The *Applicative Approach* first translates an attribute grammar description into a program consisting of a collection of recursive applicative functions, and then performs evaluation by applying the function defining the result attribute of the attribute grammar to the syntax tree to be attributed.

### 2.2.1 The Tree-Walking Approach

As an example of a static approach to evaluation, we now present the tree-walking evaluation algorithm for ordered attribute grammars, a subclass of well-formed attribute grammars. The tree-walking approach to ordered attribute grammar evaluation has been described as a "distributed-control" evaluator in the sense that a very simple "machine" is associated with each node in the tree to be evaluated [RT89]. Initially, the machine at the root of the tree is active. The ordered attribute grammar evaluator, presented in Figure 2.1 and Figure 2.2, spends no time making runtime decisions about which attributes to evaluate next — all of that information is encapsulated in a table of "visit-sequences" or "plans" generated prior to runtime. The visit-sequences are transition tables for the machines and govern the evaluation of attributes and the transfer of control from node to node in the tree. Each visit-sequence consists of a series of the following instructions:

- **EVAL(i,a)** - evaluate attribute $a$ of the $i$th child of the current node
- **VISIT(i,r)** - perform the $r$th visit to the $i$th child of the current node
- **VISIT-PARENT(r)** - perform the $r$th visit to the parent of the current node

The VISIT and VISIT-PARENT instructions transfer control to the $i$th child or the parent of the currently active machine and make the machine for the child or parent become the active machine. Once a VISIT instruction to the $i$th child of a node $N$ has been executed, control remains within the subtree rooted by the $i$th child of $N$ until a VISIT-PARENT instruction is executed at the child to return control to $N$. A node labeled with operator $op$ corresponding to the production $p : X_0 \rightarrow X_1 X_2 \ldots X_{n_p}$ will be visited *NumVisits*$(X_0)$ times by its parent, and will visit its $i$th child, with $1 \leq i \leq n_p$, *NumVisits*$(X_i)$ times.

In addition to passing control from node to node, VISIT and VISIT-PARENT instructions can be logically viewed as communicating sets of newly computed attribute values to the visited node. With respect to node $N$, a VISIT from a node $N$ to its $i$th child $N_i$ passes the values of a set of inherited attributes associated with $N_i$ to the machine associated with $N_i$ so that node $N_i$ can compute values for a subset of its synthesized attributes and then pass those values back to $N$ with the execution of a VISIT-PARENT instruction.

There is actually no need to pass the values of the sets of attributes when VISIT and VISIT-PARENT instructions are executed since those attributes can be accessed directly from the visited node. In addition, the members of the subsets of available inherited and synthesized attributes that become available with each visit are implicit in the construction of visit-sequences, i.e., the $k$th visit to a node labeled with nonterminal $n$ from its $i$th child always provides the same set of synthesized attribute values, and the $k$th visit to the $i$th child of a node labeled with nonterminal $n$ always provides the same set of inherited attribute values. The sets of attribute values that are provided by visits to nodes can be represented by a pair of tables, *AvailInhAttrs* and *AvailSynAttrs*. The table *AvailInhAttrs* maps each nonterminal $n$ and visit number *visitNum*, for $1 \leq visitNum \leq NumVisits(n)$, to the ordered sequence of inherited attributes whose values

```
Function Evaluate(TreeRoot)
  TreeNode = TreeRoot
  index = 1
  forever do
    if plan[TreeNode.op][index] is EVAL(i,a)
      args = ArgList(TreeNode,i,a)
      TreeNode[i].a = apply semfunc[TreeNode.op,i,a] to args
      index = index + 1
    elseif plan[TreeNode.op][index] is VISIT(i,r)
      index = MapVisitToIndex(TreeNode[i].op,0,r)
      TreeNode = TreeNode[i]
    elseif plan[TreeNode.op][index] is VISIT-PARENT(r)
      if TreeNode == TreeRoot
        return ResultAttr(TreeRoot)
      index = MapVisitToIndex(TreeNode.parent.op,TreeNode.son,r)
      TreeNode = TreeNode.parent
    fi
  od
end
```

**Figure 2.1** Evaluate: for Batch Evaluation of an Ordered Attribute Grammar (Kasten's Algorithm)

```
Function ArgList(node,index,attr)
  – – create a list of values to pass as arguments to semfunc[node.op,index,attr]
  argsdesc = ArgsDescFor(node.op,index,attr)
  – – argsdesc is a sequence of attribute references of the form ATTR-REF(i,a)
  args = NULL
  for each arg in argsdesc
    – – arg is ATTR-REF(i,a)
    add attribute node[i].a to args
  return args
end
```

**Figure 2.2** ArgList: for Batch Evaluation of an Ordered Attribute Grammar are available immediately before the *visitNum*th visit to a node labeled with a nonterminal $n$. The table *AvailSynAttrs* maps each nonterminal $n$ and visit number *visitNum* to the ordered sequence of synthesized attributes whose values are available immediately after the *visitNum*th visit from a node labeled with a nonterminal $n$ to its parent.

It is unnecessary to explicitly record the index of the VISIT or VISIT-PARENT instruction for later resumption of the machine at that node, since the appropriate index can be computed in constant time from information available at runtime and from a static table *MapVisitToIndex* that maps an operator, the index of a child, and a visit number $r$ to the index of the instruction in the plan for the node being visited.

In addition to the visit-sequences and the *MapVisitToIndex* table, the ordered evaluator requires descriptions of the semantic functions to invoke to compute the values of each attribute in the specification along with descriptions of the arguments to pass to those functions. It is assumed that descriptors for lists of arguments consist of a series of instances of elements of the form *ATTR-REF(i,a)*, a reference to attribute $a$ of the $i$th child of the current node, where the 0th child of a node is just the node itself.

Figure 2.3 depicts three productions from a simple attribute grammar. Figure 2.4 depicts the visit-sequences for the three operators $OP1$, $OP2$ and $OP3$.

### 2.2.2 The Applicative Approach

The applicative approach to attribute grammar evaluation is based on the translation of an attribute grammar into a program consisting of a collection of recursive functions in some functional programming language [Kat84, FZ82]. Here it is assumed that the attribute grammar has only a single result attribute associated with its root. Evaluation over an input tree $T$ is performed by applying the function defining the value of the result attribute to the input tree $T$. The set of recursive functions acts as an "output oriented" evaluator [Saa78], since attribute evaluations are performed on demand leading from the original request to compute the result attribute.

```
N ::= OP1(M)
  M.d = f1(N.a)
  M.e = f2(N.a)
  M.f = f3(N.b,M.g)
  N.c = f4(M.h)

N ::= OP2(M)
  M.d = f5(N.a)
  M.e = f6(N.a)
  M.f = f7(N.b)
  N.c = f8(M.g,M.h)

M ::= OP3()
  M.g = f9(M.d,M.e)
  M.h = f10(M.f)
```

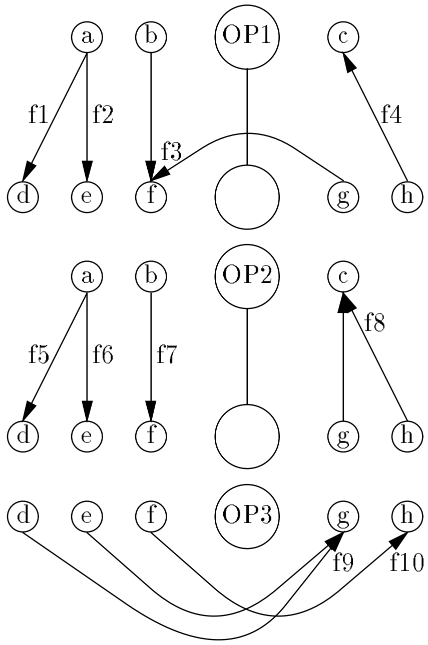


**Figure 2.3** An Attribute Grammar Fragment

```
Visit-Sequence(OP1) = Visit-Sequence(OP2) =
[
                        -- values of N.a and N.b are available
    EVAL(1,d)           -- evaluate M.d
    EVAL(1,e)           -- evaluate M.e
    VISIT(1,1)          -- first visit computes M.g
    EVAL(1,f)           -- evaluate M.f
    VISIT(1,2)          -- second visit computes M.h
    EVAL(0,c)           -- evaluate N.c
    VISIT-PARENT(1)     -- final visit to parent with N.c computed
]

Visit-Sequence(OP3) =
[
                        -- values of M.d and M.e are available
    EVAL(0,g)           -- evaluate M.g
    VISIT-PARENT(1)     -- first visit to parent with M.g computed
                        -- values of M.d, M.e and M.f are available
    EVAL(0,h)           -- evaluate M.h
    VISIT-PARENT(2)     -- final visit to parent with M.h computed
]
```

**Figure 2.4** Visit-Sequences for the Attribute Grammar Fragment

The applicative evaluation scheme is general; techniques have been demonstrated for translation of well-formed attribute grammars into recursive functions [CFZ82, Kat84]. Simplified techniques for subclasses of well-formed attribute grammars are easy to construct.

The translation of ordered attribute grammars, a subclass of well-formed attribute grammars introduced by Kastens [Kas80], is based on the notion that each nonterminal symbol in an attribute grammar represents a set of functions, one function per synthesized attribute associated with the nonterminal symbol, and each mapping a syntactic subtree and a subset of inherited attributes to a synthesized attribute value [CFZ82]. Critical to this translation is the notion of an *argument selector* [Eng84]. For each synthesized attribute of a nonterminal symbol, the argument selector is the set of inherited attributes upon which the synthesized attribute is functionally dependent, i.e., the set of inherited attributes whose values may possibly be needed to compute the value of the synthesized attribute.

We now outline the translation of an ordered attribute grammar into a set of recursive functions. A function is constructed for each pair $(N, s)$ where $s$ is a synthesized attribute of nonterminal $N$. The function, referred to as $N$–$s$, computes the value of $s$ at any node labeled $N$ when invoked with an appropriate set of arguments. The parameter list for $N$–$s$ contains a syntax tree-valued formal parameter $T$ and a formal parameter for each attribute in the argument selector for synthesized attribute $s$ of nonterminal $N$. The function $N$–$s$ selects a nested expression to execute based on the operator $op$ applied at the root of the syntax tree $T$. The nested expression is a composition of semantic functions associated with operator $op$ and invocations of generated functions computing the synthesized attributes at the children of $T$. The function invocations compute the synthesized attribute result of the function as well as the inherited attributes of the children upon which the synthesized attributes of the children depend. The arguments passed to $N$–$s$ may be used as arguments to the function computing the result of $N$–$s$ or as arguments to the functions that compute inherited attributes at the children. Whenever a synthesized attribute of a child is needed, the function responsible for computing that synthesized attribute is invoked with the appropriate set of inherited attributes and the appropriate syntactic subtree. To evaluate an input tree $T$, invoke the function $Z$–$s$ with the single parameter $T$, where nonterminal $Z$ is the start symbol of the grammar and $s$ is the result attribute associated with $Z$.

Figure 2.5 depicts the three functions $N$–$c$, $M$–$g$ and $M$–$h$ generated by translating the attribute grammar fragment from Figure 2.3.

Although execution of the program constructed by translation of an attribute grammar will correctly compute the result attribute given a tree $T$ to be attributed, execution of the functional program may take time exponential in the size of $T$ since computed attribute values are not stored by the recursive program

```
N-c(T,a,b) =
  case production(T) of
    [N ::= OP1(M)] return f4(M-h(T[1],f3(b,M-g(T[1],f1(a),f2(a)))))
    [N ::= OP2(M)] return f8(M-g(T[1],f5(a),f6(a)),M-h(T[1],f7(b)))
  esac

M-g(T,d,e) =
  case production(T)
    [M ::= OP3()] return f9(d,e)
  esac

M-h(T,f) =
  case production(T)
    [M ::= OP3()] return f10(f)
  esac
```

**Figure 2.5** Recursive Functions for the Attribute Grammar Fragment

and attributes may be reevaluated multiple times [Eng84]. This exponential behavior can be avoided by explicitly caching computed attribute values into the nodes of the tree $T$ and then reusing those values anytime a request is made to recompute one of them. This caching of computed values can also be achieved by using function caching [Mic68] to "short-circuit" attempts to invoke the same recursive procedure to recompute attribute values. Applying an efficient caching technique gives a linear time attribute grammar evaluator.

## 2.3 Approaches to Incremental Evaluation

The goal of any incremental algorithm is to make use of intermediate results computed while solving a prior instance of a problem to reduce the amount of work required to solve a new, "nearby" problem. An incremental algorithm must, therefore, *retain* a database of intermediate results computed during previous solutions and then identify the intermediate results to *reuse* for a later solution.

Incremental evaluation algorithms for attribute grammars have seen extensive use in the context of language-based editors, as pioneered by the work of Reps and Teitelbaum in the Synthesizer Generator [RT89]. In the language-based editing framework, a user explicitly applies editing operations to a syntax tree and the incremental evaluator is responsible for quickly computing values of attributes indicating various kinds of derived information which are then displayed directly to the user. In a context truer to the definition of attribute grammars as computing values for a set of result attributes, the goal of incremental evaluation is to update the values of the result attributes in response to changes to the tree being evaluated.

The tree-walking and applicative approaches to batch evaluation have both been extended to support incremental evaluation of attribute grammars. We now describe the resulting evaluators. Both approaches must, of course, provide mechanisms to retain and reuse intermediate results from prior evaluations.

### 2.3.1 The Tree-Walking Approach

The majority of the research into incremental evaluation of attribute grammars, beginning with the seminal work by Demers, Reps and Teitelbaum, has concentrated on tree-walking evaluation algorithms [RTD83, Rep84, Yeh83, YK88, JQ85, WJ88, JS86, Jon90, Hoo87].

An attribute is said to be *consistent* if its value is equal to the value computed by applying its defining semantic function to its arguments, otherwise it is *inconsistent*. Similarly, a region of an attributed tree or a computation graph is consistent if all of the attributes in the region are consistent. An attributed tree is consistent if all of its attributes are consistent. An attributed tree, all of whose attributes are consistent, will be referred to as being *consistently* attributed.

We assume a tree editing scheme which maintains a single *primary* syntax tree and a set of *secondary* syntax subtrees. A *pruning* operation detaches a complete subtree from the primary tree and adds it to the set of secondary subtrees. A *grafting* operation either attaches a new complete subtree to the primary tree or attaches one of the secondary subtrees to the primary tree, in the process removing the subtree from the set of secondary subtrees. The incremental evaluator will be invoked to update attributes associated with the primary syntax tree after a sequence of editing operations transforms a complete primary tree into a new complete tree. We will refer to the primary syntax trees before and after the application of a sequence of editing operations as $T$ and $T'$, respectively.

Prior to the creation of $T'$ from $T$, the primary syntax tree $T$ will be consistently attributed and all of the secondary syntax trees will be consistently attributed, except at their roots. The primary and secondary trees act as the retained database of previously computed values. Reuse occurs implicitly as the sequence of syntax tree edits are applied to construct $T'$.

We choose to separate syntax tree editing from attribute value reuse by introducing a structure $A_T$, referred to as an *attribute tree*, whose nodes will contain the attribute storage cells that are typically considered to be attached to nodes in a syntax tree $T$. The attribute tree $A_T$ has nodes corresponding to each of the nodes of the syntax tree $T$ and fields associated with each of the nodes which are the actual attribute storage cells. We assume that each node in $T$ has a direct link to the node in $A_T$ that contains its attributes, and that each node in $A_T$ contains a backlink initialized to identify the corresponding node in $T$. An attribute that is "associated" with a node in $T$ will actually be stored in the node in $A_T$ that represents the node in $T$. Instead of using the phrase "consistently attributed attribute tree," we will simply say "consistent attribute tree."

The attribute tree will act as a structured database of previously computed intermediate values. Reuse of previous values will now occur as the result of an operation, referred to as *initialization*, that constructs the new (probably) inconsistent attribute tree $A_{T'}$ from $A_T$ given the sequence of editing operations that created $T'$ from $T$. The new inconsistent attribute tree will also be referred to as the initial inconsistent tree. The new attribute tree $A_{T'}$ is formed by applying each of the edits to $T$ to the attribute tree $A_T$ in an obvious fashion, with the result being a new attribute tree containing new attribute tree nodes and a set of old attribute tree *components*, where each old component is a maximal contiguous set of attribute tree nodes from $A_T$.

Initialization implicitly classifies the attributes of $A_{T'}$ as follows (where ATTR is the set of all attributes in $A_{T'}$):

- RETAINED. The subset of attributes in ATTR with initial values taken from $A_T$.
- NEWBORN. The subset of attributes in ATTR that never previously existed and are initially assigned the value NULL. Note that ATTR = RETAINED $\cup$ NEWBORN.
- EDITDEP. The subset of attributes in ATTR whose values may have become inconsistent as a direct result of the application of an editing operation. We consider NEWBORN attributes to be in EDITDEP since NEWBORN attributes come into existence when new syntax nodes are inserted into the syntax tree by an editing operation. A RETAINED attribute is in EDITDEP if it appears in an old component but depends upon attributes in a different old component.

After initialization has been performed to build the attribute tree $A_{T'}$, a process known as *propagation* or *change propagation*, is invoked to reevaluate attributes in the initial inconsistent tree in order to restore it to a consistent state. Of course, as inconsistent attributes are identified and recomputed, attributes depending on those recomputed attributes may become inconsistent and need to be recomputed. As for batch evaluation algorithms, efficient techniques take into account the attribute grammar subclass of the attribute grammar defining the underlying computation in order to determine an effective ordering for recomputing inconsistent attributes. Propagation may take advantage of navigation information left by initialization.

The change propagation process implicitly classifies the attributes of $A_{T'}$ as follows:

- NOTEQUAL. The subset of attributes in RETAINED whose initial values were found to be inconsistent, i.e., those attributes which were assigned values during propagation that were not equal to their values in the initial inconsistent attribute tree.
- EQUAL. The subset of attributes in RETAINED whose initial values were found to be consistent, i.e., those attributes whose values in the new attribute tree are equal to their values in the inconsistent attribute tree.

- AFFECTED. The subset of attributes in ATTR whose values must be computed to construct a consistent attribute tree $A_{T'}$. AFFECTED = NOTEQUAL $\cup$ NEWBORN.

The goal of an "optimal" incremental evaluator is to limit the total amount of work required to update $A_{T'}$ to be O(|AFFECTED|), for a fixed attribute grammar, given the assumption that all semantic functions in an attribute grammar have roughly the same cost. Limiting the number of semantic function evaluations performed during incremental evaluation to be within the optimal O(|AFFECTED|) bound is easy – topologically sort the complete dependence graph and then recompute attributes which are NEWBORN, EDITDEP, or have already been found to be in NOTEQUAL in order. Of course, the topological sorting of the graph exceeds the O(|AFFECTED|) bound. Time-optimal algorithms must, therefore, in constant time locate the "first" inconsistent attribute, and, at any time during propagation, determine the "next" inconsistent attribute to evaluate in accordance with the topological ordering of all of the attributes in the syntax tree.

Reps and Teitelbaum describe an optimal incremental evaluator for ordered attribute grammars that is based on performing incremental evaluation after a single subtree replacement operation in which a subtree of a previously consistent attribute tree is replaced by a subtree that was attributed in a different context [Rep82, Rep84]. This editing model is referred to as the *single subtree replacement model.* The version of the ordered incremental evaluator shown in Figure 2.6 takes an old attribute subtree and a new attribute subtree as

```
Function Evaluate(Oldnode,Newnode)

  – – Apply the edit to the syntax tree and fix up the attribute tree
  SwapTrees(Oldnode,Newnode)
  SwapAttrTrees(Oldnode.Attrnode, Newnode.Attrnode)

  – – Add the root of the new subtree and its parent into Reactivated
  InitReactivated(Newnode.Attrnode)

  – – Start evaluation at the parent of the new subtree
  AttrNode = Newnode.Attrnode.parent
  index = MapVisitToIndex(AttrNode,0,1)

  forever do
    if plan[AttrNode.op][index] is EVAL(i,a)
      oldvalue = AttrNode[i].a
      args = ArgList(AttrNode,i,a)
      AttrNode[i].a = apply semfunc[AttrNode.op,i,a] to args
      if oldvalue != AttrNode[i].a
        ExtendReactivated(AttrNode,i,a)
      index = index + 1
    elseif plan[AttrNode.op][index] is VISIT(i,r)
      if IsReactivated(AttrNode[i])
        index = MapVisitToIndex(AttrNode[i].op,0,r)
        AttrNode = AttrNode[i]
      else
        index = index + 1
      fi
    elseif plan[AttrNode.op][index] is VISIT-PARENT(r)
      if AttrNode == AttrRoot
        return ResultAttr(AttrRoot)
      elseif IsReactivated(AttrNode.parent)
        index = MapVisitToIndex(AttrNode.parent.op,AttrNode.son,r)
        AttrNode = AttrNode.parent
      else
        index = index + 1
      fi
    fi
  od
end
```

**Figure 2.6** Evaluate: for Incremental Evaluation of an Ordered Attribute Grammar (Reps and Teitelbaum's Algorithm) arguments, and explicitly invokes the functions *SwapTrees* and *SwapAttrTrees* to apply the user's edit to the primary syntax tree and to update the attribute tree to represent the new syntax tree.

The ordered evaluator algorithm takes advantage of the fact that the plans used for batch evaluation of an ordered attribute grammar define a total ordering on the evaluation of attributes that is guaranteed to compute consistent attribute values for every attribute in a tree being evaluated. By skipping VISIT and VISIT-PARENT instructions to attribute tree nodes which are not in a lazily created set *Reactivated*, the algorithm is able to reevaluate all of the inconsistent attribute values in time proportional to the size of *Reactivated*, which never grows to be larger than O(|AFFECTED|). The *Reactivated* set contains only those attribute tree nodes that are known to contain inconsistent attribute values or to depend upon inconsistent attribute values. The *Reactivated* set is managed by the functions *InitReactivated*, *ExtendReactivated*, and *IsReactivated*.

Reps and Teitelbaum's ordered evaluator can easily be extended to update attribute trees in which all of the EDITDEP attributes are associated with a contiguous region of attribute tree nodes. After the new inconsistent attribute tree has been constructed, the evaluator simply initializes the *Reactivated* set to contain all of the nodes in the inconsistent region and then starts interpreting visit-sequences at the root of the region. This extension will prove useful later when we discuss tree-walking incremental evaluation algorithms for hierarchical attribute grammars.

### 2.3.2 The Applicative Approach

Pugh recently investigated applicative schemes for the incremental evaluation of functional programs using function caching [Pug88]. Pugh demonstrated that an incremental evaluator for pure functional programs could be constructed by combining the interpreter for the functional language with efficient function caching algorithms. The function cache consists of a flat database of tuples, each tuple containing a function, a set of arguments to that function, and the result previously computed by applying that function to the arguments. Each time a function is invoked with a set of arguments, the function cache is queried to determine if that function has been previously invoked with those arguments. A new function invocation with identical arguments, in a value-oriented sense, to a previous invocation's arguments can simply extract the stored result from the cache without reinvoking the function. Application of an efficient function caching interpreter provides incremental evaluation of any functional program, although the efficiency of incremental evaluation depends on the computation. If computations are suitably stable, i.e., if in response to changes to the input of a computation, many of the functions invoked during the computation are again invoked with the same arguments, then the function caching technique will prove to be quite effective for the incremental evaluation of those computations. Pugh provides efficient probabilistic algorithms for implementing function caching, including techniques for performing the equality test on term-structured values.

In order to compare incremental evaluation by function caching with incremental evaluation of attribute grammars, Pugh showed that an incremental evaluator for attribute grammars can be created by applying the function caching interpreter to the functional program constructed by translating an attribute grammar as described in Section 2.2.2. Function caching was already shown to be useful for preventing unnecessary

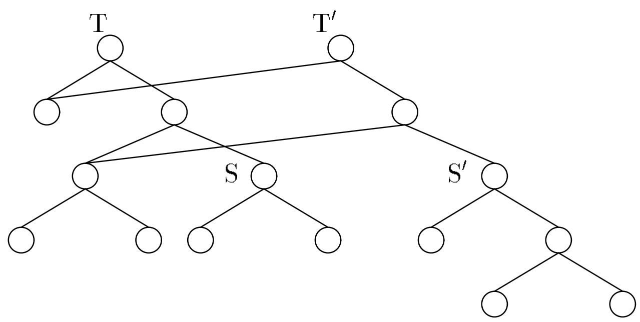


**Figure 2.7** Applicative Construction of $T'$ from $T$

reevaluation of attributes during batch evaluation of the functional program created by translating an attribute grammar. If edits to syntax trees are performed applicatively, then incremental evaluation simply requires using a function cache to retain computed values across evaluations of the attribute grammar.

In particular, batch evaluation of the attribute grammar over a tree $T$ is performed by invoking the function defining the result attribute at the root of $T$. As functions are evaluated to compute the result attribute for $T$, tuples are constructed and entered into the function cache. Incremental evaluation of a tree $T'$, where $T'$ is an applicatively modified version of the tree $T$, is then performed by invoking the function defining the result attribute at the root of $T'$ in the presence of the function cache created during evaluation of the tree $T$. Let the tree $T'$ be created by replacing a single subtree $S$ of the tree $T$ with a new subtree labeled $S'$. Applicative replacement of $S$ by $S'$ in $T$ requires creating new nodes to represent each of the nodes on the path from the root of $T$ to subtree $S$ as shown in Figure 2.7.

Pugh argues that applying the function caching interpreter to perform incremental evaluation requires $O(|\text{AFFECTED}| + |\text{PATH}|)$ function applications, where PATH is the set of nodes on the path from the root of $T'$ to the root of the subtree $S'$, inclusive. It can easily be shown that this result can be generalized for a multiple subtree replacement model in which multiple subtrees $S_1, S_2, \ldots, S_n$ are simultaneously replaced with the subtrees $S'_1, S'_2, \ldots, S'_n$. In this case, the incremental evaluator runs in $O(|\text{AFFECTED}| + |\text{PATHS}|)$ time, where PATHS includes all of the nodes in the new syntax tree $T'$ reaching from the root of $T'$ to the roots of each of the subtrees $S'_1, S'_2, \ldots, S'_n$.

## 2.4 Summary

This chapter has reviewed the attribute grammar formalism and both tree-walking and applicative algorithms for batch and incremental evaluation of attribute grammars in order to provide a foundation for the discussion of hierarchical attribute grammars and their evaluation.

# Chapter 3

# Hierarchical Attribute Grammars

Hierarchical attribute grammar dialects provide the necessary features to permit complex systems to be described recursively using a collection of modules each defined by their own attribute grammar. Each module describes a translation from one "annotated abstract syntax tree" to another "annotated abstract syntax tree" representing the result of analysis and transformation for that module.

## 3.1 Seminal Examples of Hierarchical Attribute Grammars

We now present two simple historical examples of attribute grammars that we consider to be hierarchical. These examples are both taken from the paper "Examples of Formal Semantics" [Knu71a]. The examples clearly demonstrate the advantages of taking a pipelined or hierarchical approach to the specification of systems.

In [Knuth68], Knuth defined the meaning of programs written in *Turingol*, a simple programming language for describing Turing machine computations, by providing a translation of Turingol programs into collections of quadruples. Later, in [Knu71a], Knuth decided that the original translation of programs into an "artificial representation of Turing machines based on quadruples" did not truly provide insight into the "essential nature of computation itself." A new approach was mandated. The new approach involved defining a pair of attribute grammars, one specifying the translation of a Turingol program into a program in an intermediate language, referred to as TL/I, and the other specifying the meaning of TL/I programs by translating them into English language descriptions of the operations performed by a Turing machine executing those TL/I programs. Knuth states: "For more sophisticated languages than Turingol, it becomes increasingly more important and helpful to introduce intermediate levels of semantics." Although the meaning of Turingol programs could easily have been described using a single specification, splitting the specification into two pipelined components permits either specification to be reused with obvious benefits.

Knuth's second example specification defines the meaning of a reducible lambda expression $E$ to be the string of characters that represents the *reduced* lambda expression for $E$. The fragment of Knuth's specification that is relevant for this discussion is shown in Figure 3.1. A lambda expression $E$ is said to be *reduced* if it does not contain a subexpression of the form $\lambda v.e_1 e_2$. A lambda expression is *irreducible* if it cannot be transformed into a *reduced* expression.

Knuth's attribute grammar, when provided with an unreduced lambda expression containing a subexpression of the form $\lambda v.e_1 e_2$, creates a new string representing the lambda expression after replacing each reference to $v$ in $e_1$ with the text $e_2$. If necessary, i.e., if the resulting expression is not completely reduced, the "meaning" function is then recursively invoked to further simplify the expression. Knuth's attribute grammar manipulates string representations of the lambda expression, hence, the recursive invocation of the meaning function implicitly implies that the passed string first be parsed and then the resulting derivation tree be attributed. When applied to an irreducible lambda expression, the process described by Knuth's attribute grammar will fail to terminate. Knuth's recursive specification corresponds directly to the traditional recursive approach to defining the meaning of reducible lambda expressions.

## 3.2 A Survey of Hierarchical Dialects

We now present a survey of attribute grammar dialects that support hierarchical approaches to the specification of complex language processing systems. The surveyed dialects include:

- Schulz's Cascaded Attributed Transformations,

$$
\begin{array}{llll}
\text{P1:} & S ::= E & & \\
& \text{meaning(S)} & = & \text{if reduced}(E) \\
& & & \quad \text{then text}(E) \\
& & & \quad \text{else meaning(text}(E)) \\
& \ldots & & \\
\text{P2.1:} & V ::= x & & \\
& \ldots & & \\
\text{P2.2:} & V_1 ::= V_2' & & \\
& \ldots & & \\
\text{P3.1:} & E ::= V & & \\
& \text{text}(E) & = & \text{if index}(V) \text{ in bound}(E) \text{ then } \ldots \\
& \ldots & & \\
\text{P3.2:} & E_1 ::= \lambda V\, E_2 & & \\
& \text{text}(E_1) & = & \text{if arg}(E_1) \text{ then } \ldots \\
& \ldots & & \\
\text{P3.3:} & E_1 ::= (E_2 E_3) & & \\
& \text{text}(E_1) & = & \text{if function}(E_2) \text{ then } \ldots \\
& \ldots & &
\end{array}
$$

**Figure 3.1** Fragments from Knuth's Hierarchical Example

- Reps and Teitelbaum's Synthesizer Specification Language (SSL),
- Ganzinger and Giegerich's Attribute Coupled Grammars,
- Ganzinger and Giegerich's Extended Attribute Coupled Grammars, and
- Vogt, Swierstra and Kuiper's Higher Order Attribute Grammars.

Each of these dialects takes advantage of the notion of constructing complex systems by composing a set of non-procedurally specified modules. Each dialect is distinguished by the facilities provided for specifying the composition of modules, communicating facts between pairs of modules, and specifying the computation performed by each module.

### 3.2.1 Cascaded Attributed Transformations

In 1976, Schulz examined *cascaded attributed transformations* as a tool for describing programming language translators. A cascaded attributed transformation is built from a composition of a sequence of modules, each defined by an attributed transformation. An attributed transformation consists of an attribute grammar augmented with a template-based mechanism for synthesizing target language terms. Instead of adopting a pure attribute grammar approach to describing modules of a cascaded translator, Schulz decided to adopt a mixed formalism for two reasons. First, Schulz believed that, although target language synthesis should be based on semantic values computed during the attribute evaluation process, the process of synthesizing the target language result should be treated as a distinct notion and specified separately. This is mostly a matter of taste since, as demonstrated by Schulz, an attributed transformation can easily be translated into a traditional attribute grammar by encoding templates into sets of semantic rules. Templates can then be seen simply as a shorthand, and the notion of cascaded attributed transformations can be viewed as a formalization of Knuth's pipelined example. Second, Schulz believed that space considerations in the 1970's prohibited practical language translators from explicitly constructing semantic values as large as the target language result. Instead, the resulting string should be emitted on the fly. This issue is mitigated in modern times by the availability of large computer memories.

Schulz argues strongly in favor of a structure-based approach in which cascaded attributed transformations describe translations between tree-structured representations of target language terms. His motivations for adopting a structure-based approach are essentially the same as those underlying this thesis. First, facts, referred to as *intrinsic* attributes, may be attached directly to syntax tree nodes for use by later modules. Second, explicit retention of the tree structure makes it possible to serially *cascade* a sequence of modules without the need to parse the input to each module. The tree-structured input and output of an attributed transformation are referred to as the input tree and output tree, respectively.

Because each module in a cascaded attributed transformation is split into two components: (1) an attribute system defining computations whose results are used to guide output tree synthesis, and (2) a template-based description actually constructing the output tree, it is unnecessary for the attribute system to use data domains of tree-structured values for attributes. Output trees are constructed entirely by templates.

An attributed transformation associates a set of template names with each of the nonterminals and terminals in a module's input grammar in addition to the standard sets of inherited and synthesized attributes. A set of templates (or template rules) is associated with each production and defines a value for each of the template names associated with the nonterminal labeling the lefthand side of each production. A template associated with production $P$ and defining a template name is essentially a derived operator consisting of a nest of conditionals and constructors with "holes." These "holes" are identified with template names associated with the righthand side nonterminals and terminals in $P$, functions defining the values of intrinsic attributes given the values of the synthesized attributes of $P$, and terminals in the righthand side of $P$. Since templates may only refer to template names of the children of a node, the target tree is constructed during a bottom-up traversal of the source tree. Prior to this synthesis of the output tree, attribute evaluation computes values for each of the attributes defined in the attribute system underlying the attributed transformation.

Schulz's template mechanism includes a restriction that is mirrored in many of the other hierarchical dialects that guarantees that any module's output is a tree-structured, rather than a dag-structured, value. Schulz's stated goal in restricting the output of each module to be a tree is to guarantee that the size of the output of any module is proportional to the size of that module's input tree. An "unshared" representation of the output of a module will be constructed if it is necessary to build a textual representation of the result of an attributed transformation or to build a tree suitable as the input to another attributed transformation.

The restrictions placed on templates to avoid the construction of dags are the following: (1) a template name associated with a production may appear in at most one of the template rules associated with each production, and (2) if a template name appears more than once in the same rule, then only one of the references to the template name may contribute its subtree to the tree constructed by the template. The first restriction guarantees that no subtree will be included in two distinct subtrees that could eventually both be used as arguments to the same constructor, indirectly generating a dag-valued term. The second restriction prohibits any single template rule from directly constructing a dag value by including the same subtree twice. Obviously, these restrictions are pessimistic, since, even if the same subtree is included in two different trees, those two trees may never be combined to create a dag-valued term.

Finally, Schulz describes a notion of *telescoping* pairs of attributed transformations to combine two specifications of attributed transformations into a single specification. The goal of composing two specifications into a single specification is to eliminate the space required to store the intermediate term generated by evaluation of the first specification. Unfortunately, the specification that results from telecoping pairs of transformations is significantly more complex than either of the two specifications being composed.

Schulz's cascaded attributed transformation dialect is the first formalization of the fundamental concept underlying hierarchical attribute grammars – the use of attribute grammars to define modules. Apparently, cascaded attributed transformations were never implemented.

### 3.2.2 The Synthesizer Specification Language (SSL)

In 1984, Reps and Teitelbaum introduced the Synthesizer Generator, a specification-based tool for constructing language-based editors. Editor specifications are written in the Synthesizer Specification Language (SSL), which is essentially an attribute grammar specification language augmented with features for defining facilities of language-based editors. SSL is the most visible of the hierarchical attribute grammar dialects due to the widespread use of the Synthesizer Generator. The attribute grammar component of an SSL specification consists of an abstract syntax-based definition of a context free grammar and a definition of an attribute system. This survey focuses on the attribute grammar aspects of an SSL specification and ignores all other aspects of a specification.

Abstract syntax in an SSL specification is defined as a collection of *phyla* and *operators*, where phyla represent the nonterminals of a context free grammar, and operators give names to the alternative productions that can be derived from a phylum [Donzeau-Gouge et al. 1975, Medina-Mora and Notkin 1981]. SSL provides primitive phyla for representing terminal classes such as "integer", "string", and "boolean." A primitive phylum is defined by a possibly infinite set of nullary operators, one operator per member of the terminal class. For example, "integer" might be defined as the infinite set of terms "0()", "1()", "2()", and so forth. Terminal symbols that carry no semantic information do not appear in the phylum/operator definition of a context free grammar, but instead appear in the definition of the concrete syntax of the language. Phyla that are not primitive are referred to as user-defined phyla. An SSL specification must identify a single user-defined phylum as the *root* phylum. There is no requirement that all phyla in a specification be derivable from the root phylum.

A structured value is called a *term* and is defined by the application of a $k$-ary operator to $k$ terms having the appropriate phylum. Hence, operators are term constructors. Each node in a term constructed using the phylum/operator mechanism is labeled with the operator applied at that node. A term is a representation of a rooted, labeled tree structure, but may be implemented as directed acyclic graphs with sharing of components. A component of a term will be referred to as a *subterm*.

A novel feature of an attribute system in an SSL specification is that attributes take their types from either the set of predefined phyla or the set of user-defined phyla. Therefore, every attribute value in an SSL specification is a term – a degenerate term if the attribute's type is taken from the set of predefined phyla.

An SSL specification associates a set of attributes with each user-defined phylum, and a set of semantic rules with each of the operators in a specification as is expected. Attributes and semantic rules may be associated with phyla and operators that are not derivable from the root phylum of the specification. These attributes and semantic rules come into play with the application of a mechanism, referred to as an *attribution expression* in SSL, that permits the attribute grammar evaluator to be recursively applied to compute functions over the values of term-valued attributes by using the attribution mechanism itself.

An attribution expression is invoked with a set of values and a term to attribute. Since terms are represented as dags internally, the first step of processing an attribution expression is to copy the term to create a tree without sharing. Next, the set of values is assigned to the inherited attributes at the root of the new tree and evaluation commences. Actually, general attribution expressions were never fully implemented and the set of values must be empty; however, the Synthesizer Generator Reference Manual [RT87] describes a technique that achieves the same effect: (1) create a new operator having $k+1$ children to hold the $k$ inherited attribute values and the term to be attributed, (2) extend the module that is to be invoked with new semantic rules that assign values to the inherited attributes associated with the last child of the new operator. It is precisely because of the notion of an attribution expression that SSL is considered to be hierarchical.

The body of each semantic rule in an SSL specification is a composition of identity functions, conditionals, predefined functions over primitive phyla, user-defined functions over user-defined phyla, term constructors, pattern queries for selecting subterms from terms, and attribution expressions. To enable termination of a recursive specification, expressions within conditionals are evaluated only on demand. The arguments passed to a semantic rule associated with a production may include attribute references and *syntactic references*. A syntactic reference to the $i$th component in a semantic rule associated with a node $N$ retrieves the term rooted by node $N$ if $i == 0$, and the term rooted by the $i$th child of node $N$, otherwise. Since terminals in a context free grammar are represented as predefined phyla, syntactic references permit semantic rules to access the lexical values of those terminals and also permit semantic rules to be applied to entire syntactic subterms of any module's input term. This second application of syntactic references is not strictly necessary – instead, semantic rules could be defined to explicitly synthesize copies of the appropriate portions of the input term. In fact, the interaction between syntactic references and destructive editing models, such as the one supported by editors generated by the Synthesizer Generator, complicates incremental evaluation techniques. Reps and Teitelbaum's incremental evaluation algorithm is not optimal in the presence of syntactic references [TC90].

Although attribution expressions provide a means of invoking semantic functions defined by attribute grammar specifications, we believe that SSL has a key deficiency that keeps it from being the ideal hierarchical attribute grammar dialect – a complete specification defines only a single attribute system associating attributes and semantic rules with a single collection of phyla and operators. Thus, there is no language-supported mechanism for defining independent modules which share the same underlying syntax. To a limited extent, a set of independent modules can be simulated by replicating the original phyla and operators and then associating the attributes and rules for each logically independent module with one of the copies of the original phyla and operators. Using this technique enables each of the modules in a hierarchical specification to be defined independently, but unfortunately each module is then defined over terms constructed from different sets of phyla and operators. Therefore, the set of operators that a module uses to construct terms depends, in a non-modular manner, on the context of the module within a complete specification.

### 3.2.3 Attribute Coupled Grammars

Coincident with the work of Reps and Teitelbaum in 1984, Ganzinger and Giegerich introduced Attribute Coupled Grammars as a mechanism for defining complex language translators that they believed could not be described naturally and modularly using the traditional attribute grammar formalism [GG84]. Just as in Knuth's pipelined example, and Schulz's cascaded attributed transformations, an Attribute Coupled Grammar is simply a sequence of attribute grammars, referred to as *attribute couplings*, each describing a translation from a source language term to a target language term.

An Attribute Coupled Grammar consists of specifications for each module in a translator and an explicit description of a sequence in which to compose the modules. Each translator module is defined by an attribute grammar that describes one syntax-directed phase of analysis and transformation.

A sequence of attribute couplings meeting certain restrictions can be composed into a single specification using a technique referred to by Ganzinger and Giegerich as *descriptional composition*, essentially an extended and much more formally studied version of Schulz's notion of telescoping. The attribute coupling that results from descriptional composition may be considerably more complicated than any of the specifications being composed. This complexity can be seen in both the total number of attributes in the resulting specification and in the fact that descriptional composition of a sequence of attribute grammars is not closed with respect to attribute grammar subclasses. In [GG84], Ganzinger and Giegerich prove a sharp upper bound of $| \alpha | \star | \beta |$ on the number of attribute rules in the specification resulting from the descriptional composition of a pair of grammars $\alpha$ and $\beta$, where $\alpha$ has $| \alpha |$ attributes and $\beta$ has $| \beta |$ attributes. In [Gie88], Giegerich proves that descriptional composition of a pair of absolutely noncircular attribute grammars may construct a grammar that is not a member of the absolutely noncircular subclass.

Ganzinger and Giegerich's original formulation of Attribute Coupled Grammars defines an attribute coupling to be an attribute grammar that satisfies a restriction referred to as the syntactic single use requirement, or SSUR, that ensures that descriptional composition is well-defined. This restriction mirrors the restriction in Schulz's attributed transformations that guarantees that all syntactic terms generated during evaluation are tree-structured, not dag-structured. In [GGAI], the SSUR restriction is eliminated from the actual definition of attribute couplings; however, only SSUR attribute couplings are subject to descriptional composition.

### 3.2.4 Extended Attribute Coupled Grammars

In addition to formalizing sequential composition of attribute grammars, Ganzinger and Giegerich's 1984 paper also includes a short description of several extensions to the Attribute Coupled Grammar formalism that transform Attribute Coupled Grammars into a truly hierarchical dialect by permitting an attribute grammar, or a sequence of attribute grammars, to be used to define semantic functions over syntactic attributes. We refer to this concept as *Extended Attribute Coupled Grammars*, since apparently no explicit name for the concept has been provided. MARVIN, described in [GGV86], is an implementation of Extended Attribute Coupled Grammars. The fact that MARVIN is implemented within MARVIN demonstrates the power of the hierarchical concept. It is not clear from the literature what restrictions are placed on attribute couplings for use in Extended Attribute Coupled Grammars, but apparently MARVIN is implemented without the SSUR restriction and dag-valued terms may be created.

### 3.2.5 Higher Order Attribute Grammars

The Higher Order Attribute Grammars of Vogt, Swierstra and Kuiper [VSK89] are distinguished from traditional attribute grammars in that they allow the tree being evaluated to be expanded during evaluation. In essence, certain subtrees of the syntax tree are not known, a priori, and only become known lazily during the evaluation process. An initial syntax tree consists of a collection of syntax tree nodes, some of which contain unexpanded *nonterminal attributes*. During evaluation, the values of tree-valued attributes will be assigned to the nonterminal attributes to extend the syntax tree. As the syntax tree is expanded, the evaluation process continues, computing values of attributes appearing throughout the old and new components of the syntax tree. To simplify the notation, the name of each nonterminal attribute indicates the kind of syntax tree values that may be assigned to it. For example, nonterminal attribute $V$ can be assigned any value derived from nonterminal $V$ in the underlying context free grammar.

Unfortunately, the notion of Higher Order Attribute Grammars, as defined by Vogt, Swierstra and Kuiper, does not take into consideration two effects of building structured data by using constructors. First, values built by constructors will, in general, be dag-valued, not tree-valued, since they are constructed by invoking term constructors on arguments that may not represent unique structures. Second, it is possible for a single tree-structured value to be assigned to two different nonterminal attributes. To remedy this problem, we redefine the Higher Order Attribute Grammar notion so that whenever a term is installed into a nonterminal attribute, a new tree representing that term will be constructed and installed instead. Attribute evaluation will, therefore, always be performed over a tree.

Assigning a term value to a nonterminal attribute $V$ implicitly defines a set of functions, one function per synthesized attribute associated with the nonterminal $V$. This is precisely the same way that any subtree of the syntax tree defines a set of functions in the traditional attribute grammar framework. Each function is realized implicitly by the attribute evaluation process as attributes throughout the term are evaluated in accordance with some topological ordering of the dependence graph. Higher Order Attribute Grammars, therefore, are not as different from traditional (first order) attribute grammars as they may initially seem. In fact, it is relatively straightforward to translate a Higher Order Attribute Grammar specification into a hierarchical dialect, by replacing the installation of a term into a nonterminal attribute into the explicit evaluation of a set of attribute grammar defined semantic functions. This translation is described in Appendix A.

In a Higher Order Attribute Grammar, the attribute evaluator cannot be directly invoked as a semantic function; however, the same effect can be achieved by assigning the term to be attributed to a nonterminal attribute and then referring to a synthesized attribute of that term. Unfortunately, as in SSL, only a single set of semantic rules may be associated with any production in the context free grammar underlying the higher order specification. Hence, even though we can simulate the ability to attribute a term, the function computed by attribution is not defined independently from the rest of the specification. As in SSL, tricks can be applied to split the context free grammar so that different functions can be defined over terms from essentially the same base syntax.

Vogt, Swierstra and Kuiper describe a notion equivalent to the SSL notion of syntactic reference that provides a means of capturing subtrees of the underlying syntax tree as an attribute value. They suggest that syntactic references are equivalent to a collection of semantic functions explicitly synthesizing an unattributed copy of the syntax tree during evaluation. Recently, Teitelbaum and Chapman have suggested adopting Higher Order Attribute Grammars within the Synthesizer Generator as a means of providing macro facilities within generated editors [TC90]. Teitelbaum and Chapman adopt a notion of Higher Order Attribute Grammars that differs from that presented by Vogt, Swierstra and Kuiper in the interpretation of syntactic references in which syntactic references include the values of all of the attributes associated with the syntax nodes in addition to the syntax nodes themselves, and in which syntactic references are implemented by simply returning a pointer to the appropriate attributed syntax tree.

Unfortunately, to implement Teitelbaum and Chapman's notion of syntactic references, access to a syntactic subtree that still contains unexpanded nonterminal attributes or is not fully attributed must be prohibited. Otherwise, the value returned by a syntactic reference will be incomplete and may be modified later during evaluation either by extending the syntactic term by assigning a term to a nonterminal attribute or by evaluating attributes associated with the syntactic term and assigning those attributes to the nodes of the syntactic term. Safeguarding access to incomplete subtrees requires augmenting the attribute grammar with new attributes and new semantic rules to guarantee that syntactic references are not evaluated before the entire subtree being referenced has been expanded, and, if necessary, attributed. These additional dependences must then be considered in the scheduling of attribute evaluations.

### 3.3 An Improved Mechanism for Defining Abstract Syntax

The hierarchical dialects that have just been surveyed are each based on notions of abstract syntax that hinder the use of those dialects in specifying a complex system as a collection of independently specified modules. In this section, we describe an extension to SSL's phylum/operator grammar definition mechanism that we believe to be more suitable for the specification of modular systems. In the next section, we describe a new hierarchical attribute grammar dialect based upon this new notion of abstract syntax.

We adopt SSL's view that syntactic terms are defined by the application of operators to an appropriate number of properly typed subterms and logically represent rooted, labeled tree structures, which may happen to be implemented as directed acyclic graphs with sharing of subcomponents. Irrespective of the representation of syntactic terms, the result of evaluating a term according to a module specification is defined to be the result of evaluating the tree represented by the term.

In typical formulations of abstract syntax and attribute systems defined over abstract syntax, syntactic terms consist of sets of "syntactic" nodes and "semantic" leaves, where semantic leaves correspond to atomic values such as strings or integers. An attribute system then associates attributes with the syntactic nodes and the attribute evaluator traverses only the syntactic nodes in the term to perform evaluation. In a hierarchical setting, it is extremely useful to be able to associate sets of facts with various nodes in a syntactic term to be passed to a module for evaluation. These facts, referred to annotations, might be the result of a computation by a module that performs analysis over a program fragment, for example, the sets of data flow facts gathered by performing dataflow analysis over a program. Annotations should be named and typed, and should be permitted to contain any kind of data that can be computed or manipulated by modules. With respect to the attribute evaluation process, however, annotations are intended to act just like semantic leaves – they should not be attributed.

To some extent, annotations can be simulated in a phylum/operator grammar definition by converting each operator that is to be annotated with a set of typed values into a new operator whose righthand side contains all of the phyla appearing on the righthand side of the original operator plus new phyla representing each of the annotations. Unfortunately, there are two problems with this approach. First, requiring that operators be explicitly replicated and extended leads to a nightmare of additional complexity in the specification of already complex systems, and will make it difficult to decouple specifications of modules in a hierarchical system in any reasonable manner. Second, if annotations are added as new children of syntax nodes, then there is no way to specify that these new children should be considered to be semantic, and, therefore, should not be attributed. Evaluation of attributes associated with the nodes of an annotation may be very expensive, and may result in non-termination of the evaluation process due to recursion since it may be necessary to invoke a module to compute the value of the attribute.

Both problems are addressed by the notion of *term typing* which permits named and typed annotations to be associated with every node in a syntactic term. We assume a base formulation of abstract syntax and attribute data domains just like that used in SSL in which predefined and user-defined phyla are used as the domain of each of the attributes in an attribute grammar specification and the domain and range of the attribute evaluation process itself. The term typing mechanism permits different modules to be defined over a common syntactic structure annotated in different ways according to different term types. This will eliminate the need to proliferate many different versions of the same base syntax. In addition, there will no longer be any confusion about which nodes in any syntactic term are to be attributed by an evaluator. Annotations will be used to store all of the facts previously stored as semantic leaves in terms. Semantic rules appearing in an attribute grammar access the values of annotations by using an "annotation reference," which is written simply as the name of the annotation.

A *term type* is a map from phyla to a (possibly empty) set of named and typed fields, or annotations. Given a collection of phyla and operators defining "unannotated" terms, a term type extends each of the "unannotated" operators with a set of annotations for the phylum containing the operator. All syntax-valued data will have a type defined by a phylum and a term type. For example, an attribute of type $\langle phy,\ tt\rangle$ is a value derivable from phylum $phy$ and annotated according to the term type $tt$. The notation $phy$ is intended to be a shorthand for the type $\langle phy, default\rangle$, where the term type *default* maps each phylum in a phylum/operator definition to a default set of annotations which will always be associated with those phyla. For example, the binding of variable names to syntax nodes representing identifiers might be included in the default term type. All other term types extend the *default* term type.

A structured value of type $\langle phy, tt\rangle$ is created by the application of a tuple $\langle OP, tt\rangle$, with $OP$ a $k$-ary operator from phylum $phylum(OP)$, to $k + |tt(phylum(OP))|$ terms of the appropriate type, where the first $k$ terms have type $\langle phy_i, tt\rangle$, where $phy_i$ is the phylum of the $i$th syntactic subterm of $OP$ and the remaining $|tt(phylum(OP))|$ terms have types $tt(phylum(OP))[j]$, where $tt(phylum(OP))[j]$ is the $j$th type in the sequence of types associated with phylum $phylum(OP)$ by term type $tt$, for $1 \leq j \leq |tt(phylum(OP))|$. An application of the operator $OP$ as a node constructor is taken as shorthand for the application of the tuple $\langle OP,\ default\rangle$.

```
GRAMMAR
  root       ::= Root(program).
  program    ::= Program(identifier stmts).
  stmts      ::= Stmtsnull().
  stmts      ::= Stmtpair(stmt stmts1).
  stmt       ::= Assign(identifier expr).
  stmt       ::= IfThenElse(expr stmts1 stmts2).
  stmt       ::= While(expr stmts).
  expr       ::= Ident(identifier) | Sum(expr1 expr2) | ....
  identifier ::= Identifier().
```

**Figure 3.2** A Simple Phylum/Operator Grammar Definition

```
TERM-TYPE default
  annotation identifier id s.

TERM-TYPE tt1
  annotation stmts idlist in, idlist out.

TERM-TYPE tt2
  annotation stmts ⟨stmts, tt1⟩ stmts1, stmts stmts2.
```

**Figure 3.3** Term Type Definitions

Figure 3.2 presents a simple phylum/operator grammar definition. A set of term type definitions are shown in Figure 3.3. Types *id* and *idlist* are builtin types. As suggested above, the term type *default* annotates each node derivable from phylum *identifier* with an annotation *s* of type *id*, i.e., each node with operator *Identifier* has an annotation *s*. The *tt1* term type associates annotations *in* and *out* both of type *idlist* with the phylum *stmts*. The *tt2* term type associates an annotation *stmts1* of type $\langle stmts, tt1\rangle$ and an annotation *stmts2* of type *stmts* with phylum *stmts*.

## 3.4 Modular Attribute Grammars

We now sketch a new hierarchical attribute grammar dialect based upon the notion of annotated abstract syntax described in the previous section. Our primary goal in presenting this new dialect is to define the set of features that we believe should be provided in a hierarchical dialect and supported by the batch and incremental evaluation algorithms presented later in this dissertation.

A *Modular Attribute Grammar* definition contains the following:

- A phylum/operator grammar definition consisting of a set of phylum definitions and operator definitions,
- A set of term type definitions associating typed annotations with the phyla in the phylum/operator grammar definition,

- A set of named module definitions defined by attribute grammars,
- A designation of the *root* module.

Each attribute grammar definition of a module includes the following:

- A designated root phylum,
- A designated root term type,
- A set of named and typed attributes each having a designated phylum and term type for each phylum reachable from the root phylum,
- A set of named and typed local attributes associated with each operator,
- A designated result attribute of the root phylum,
- Semantic rules defining each attribute associated with each phylum reachable from the root phylum.

The body of each semantic rule is a composition of the following functions:

- Identity functions,
- Lazily evaluated conditionals,
- Predefined functions defined over predefined phyla,
- User-defined auxiliary functions defined over user-defined and predefined phyla,
- Term constructors,
- A destructuring facility for accessing subterms of terms,
- Module invocations.

The arguments passed to semantic rules are chosen from the following sources:

- Attribute references,
- Annotation references,
- Syntactic references,
- Constants.

The input to each module must be rooted by a node whose operator is derivable from the root phylum of the module, and whose term type is the input term type of the module. Additional arguments may be passed to a module by creating a new root node having those arguments as children or annotations as described in Section 3.2.2. Recursive module invocations are permitted.

During evaluation of a Modular Attribute Grammar over an input term, a set of *module instances* will be invoked. A module instance is a runtime instantiation of a module. The input term will sometimes be referred to as the syntactic input to the root module instance. We define MOD-INSTS to be the set of module instances that are invoked during evaluation of an input term. The module instances in MOD-INSTS are naturally organized as a tree since each module instance, other than the root module instance, is invoked during the evaluation of some other module instance in MOD-INSTS. This tree will be referred to as the *module instance tree*. It is important to understand that the set of module instances invoked by a module instance $M$ depends upon the structure of $M$'s syntactic input and upon computed semantic information, since modules are invoked as the result of evaluating semantic functions associated with particular operators in the input term and since modules may be invoked conditionally.

No restrictions have been placed on the use of syntactic values to guarantee that syntactic values are tree-structured. Therefore, dag-structured syntactic values may be created during evaluation of a hierarchical specification and they may be passed to modules to be evaluated. As mentioned earlier, we treat dags as compact representations of trees, i.e., the result computed by applying a module to a dag-structured input term is just the result computed by applying the module to the (possibly much larger) tree-structured input term which represents the dag-structured term. In fact, we advocate adopting a hash-consed implementation of abstract syntax that makes it possible to perform equality tests between pairs of syntactic values in constant time [Got74]. Hash-consing guarantees that whenever a node constructor is invoked to create a new syntax node that will be the root of a subterm that is structurally identical to a previously created subterm, the node constructor will return a reference to the root of the existing subterm instead of a new syntax node. Thus, hash-consing provides a constant-time equality test for syntactic terms by ensuring that any two terms that are structurally identical are actually the same term. Hash-consing cannot be used in the presence of destructive syntax tree modifications. This issue will be resolved in Chapter 6 when we discuss incremental evaluation of hierarchical attribute grammars.

## 3.5 Summary

Hierarchical attribute grammars, and especially Modular Attribute Grammars, are well-suited to the definition of complex hierarchical systems consisting of many independent modules which can be viewed as tree-to-tree transformations. In the next chapter, we present several modules from a source-to-source optimizer for a very simple programming language. The subsequent chapters then examine the extremely significant question of batch and incremental evaluation of hierarchical attribute grammars.

# Chapter 4

# An Application of Modular Attribute Grammars

In this chapter, we describe a set of module specifications chosen from a pair of complete Modular Attribute Grammar specifications describing source-to-source code optimization for a simple source language. The two specifications will be referred to as Optimizer-I and Optimizer-II. Modules from these two specifications are presented as a means of demonstrating pipelined and hierarchical specifications, the modularity of Modular Attribute Grammars, and the use of recursion as a well-defined alternative to circularity. The longest section of this chapter examines the application of Modular Attribute Grammars to the solution of data flow analysis problems, a set of problems that are often specified by circular attribute grammars and then translated into noncircular specifications.

The simple source language to be transformed by both specifications is shown in Figure 4.1. The specifications make use of term type definitions for term type *default* and term type *dflow* as shown in Figure 4.2.

## 4.1 Optimizer I and Optimizer II Specifications

Schematics of the two systems described by the pair of optimizer specifications are presented in Figure 4.3. Optimizer I is essentially a pipelined specification in which the root module ROOT-I passes its input syntax term through a pipeline consisting of the live variable analysis module DFLOW, the dead code elimination module ELIM, and then the constant folding module CONST. The output of module DFLOW, and, therefore, the input to module ELIM, is an annotated abstract syntax tree of term type *dflow*. The dead code elimination module ELIM invokes the module APPEND zero or more times to concatenate pairs of statement lists. The constant folding module CONST invokes the module FOLD zero or more times to perform expression simplification.

Optimizer II is a more hierarchical specification that reuses the entire Optimizer I specification as a key component of an iterate-until-stable optimizer. Module ITER invokes module ROOT-I to translate an input source term, and then it recursively invokes ITER if the term returned by module ROOT-I is not the same as the term passed to ROOT-I to be optimized.

```
GRAMMAR
  root          ::= Root(program).
  program       ::= Program(identifier stmts).
  stmts         ::= Stmtsnull().
  stmts         ::= Stmtpair(stmt stmts1).
  stmt          ::= Assign(identifier expr).
  stmt          ::= IfThenElse(expr stmts1 stmts2).
  stmt          ::= While(expr stmts).
  stmt          ::= Stmtnull().
  expr          ::= Ident(identifier) | Sum(expr1 expr2) | ....
  identifier    ::= Identifier().

-- auxiliary phylum and operators
  dflowholder   ::= DflowHolder(stmts).
  appendholder  ::= AppendHolder(stmts).
```

**Figure 4.1** A Simple Source Language

```
TERM-TYPE default
  annotation identifier id ident.
  annotation dflowholder idlist out.
  annotation appendholder stmts tailstmts.

TERM-TYPE dflow
  annotation program idlist in, idlist out.
  annotation stmts idlist in, idlist out.
  annotation stmt idlist in, idlist out.
```

**Figure 4.2** Term Types for the Optimizer Specifications

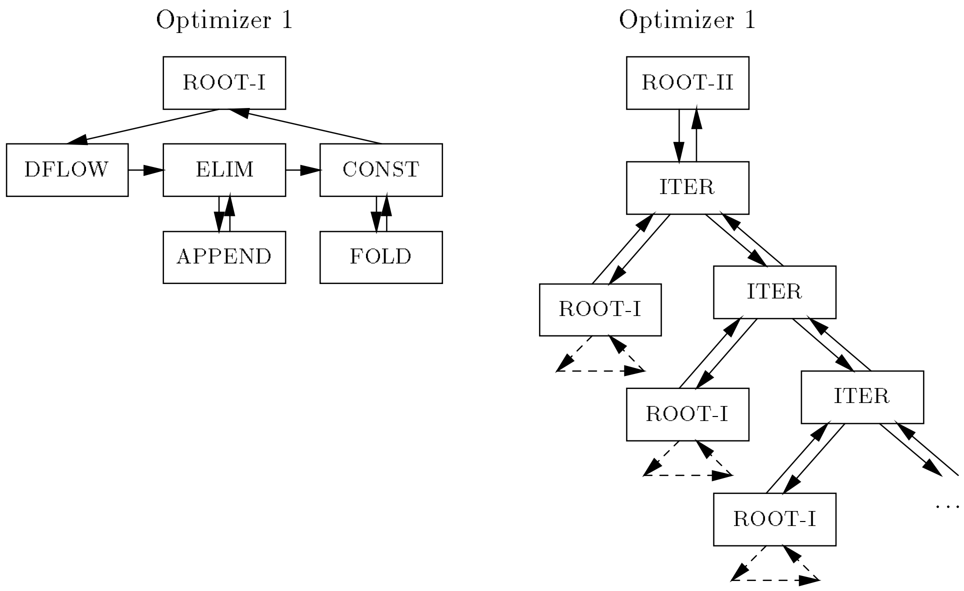


**Figure 4.3** Schematic of Optimizer I and Optimizer II

```
MODULE ROOT-I
default→default
ROOT root – – the root phylum
RESULT tree – – the result attribute of the root phylum

attributes root
  syn root tree
attributes program
  syn program tree

production root ::= Root(program)
  root.tree = Root(program.tree)

production program ::= Program(identifier stmts).
  local ⟨stmts,dflow⟩ dflowres – – phylum stmts, term type dflow
  local stmts elimres
  local stmts constres

  dflowres = DFLOW(stmts) – – stmts is a syntactic reference
  elimres = ELIM(dflowres)
  constres = CONST(elimres)
  program.tree = Program(identifier,constres)
```

**Figure 4.4** Module ROOT-I

Figure 4.4 presents the module specification for module ROOT-I, the root module of Optimizer-I. The semantic rules associated with operator *Program* invoke modules DFLOW, ELIM and CONST in a pipeline to perform optimization over the contents of the statement list in the input program. DFLOW is applied to a syntactic reference to the statement list in the input program and returns an annotated statement list of term type *dflow*, as indicated by the type ⟨stmts,dflow⟩ for local attribute *dflowres*. The annotated statement list is then passed to module ELIM which returns a statement list of term type *default*, as indicated by the type *stmts* for local attribute *elimres*. The statement list created by module ELIM is then passed to module CONST which again returns a statement list of term type *default*, as indicated by the type *stmts* for local attribute *constres*. The complete program resulting from source-to-source optimization is then synthesized into the attribute *tree* associated with the phyla *root* and *program*.

Figure 4.5 presents the specifications for module ROOT-II and module ITER of Optimizer II. ROOT-II passes the entire input term to module ITER which is responsible for iteratively invoking ROOT-I until a fixed point is reached. The function *equal?* compares two terms to determine if they are structurally identical. If terms are implemented using hash-consing as suggested in Section 3.4, then *equal?* can be performed in constant time.

## 4.2 Module APPEND

Figure 4.6 presents a Modular Attribute Grammar specification describing the *append* operation that takes two statement lists and creates a new list consisting of all of the statements in the first list followed by all of the statements in the second list. To append statement lists *headstmts* and *tailstmts*, an *AppendHolder* node is created containing *headstmts* as its only child and *tailstmts* as an annotation. The specification constructs the new list given the two statement lists *headstmts* and *tailstmts* by using the inherited attribute *downstmts* and synthesized attribute *upstmts* to pass the list *tailstmts* down to the *Stmtsnull* node at the end of *headstmts* and to then create the new list by creating new *Stmtpair* nodes containing each statement in *headstmts*.

Evaluation of the APPEND module, as just described, will be very inefficient unless the evaluator realizes that there is no need to actually evaluate attributes associated with syntax nodes contained within the two statement lists *stmts1* and *stmts2* of an *IfThenElse* node and the single statement list *stmts* of a *While* node, since the value of the synthesized attribute *upstmts* of each of these three statement lists is not actually used. Essentially, any syntactic component whose syntactic attributes are not used should be considered to be an annotation rather than a syntactic subterm. To make this even clearer to the evaluator, we suggest using a special notation to indicate that the subterms derived from certain phyla associated with each operator should be considered to be annotations. The notation in Figure 4.7 is intended to indicate that statement lists *stmts1*, *stmts2*, and *stmts* should be treated as annotations. If the evaluation of useless attributes associated with the syntax nodes contained in each of these statement lists is avoided, then the cost of using

```
MODULE ROOT-II
root→root
ROOT root
RESULT tree

attributes root
  syn root tree.

production root ::= Root(program)
  root.tree = ITER(root)
```

```
MODULE ITER
root→root
ROOT root
RESULT tree

attributes root
  syn root tree.

production root ::= Root(program)
  local root result
  result = ROOT-I(root)
  root.tree =
          if (equal?(result,root))
            then result
            else ITER(result)
```

**Figure 4.5** Module ROOT-II and Module ITER

```
MODULE APPEND
default→default
ROOT appendholder
RESULT upstmts

attributes appendholder
  syn stmts upstmts.
attributes stmts
  inh stmts downstmts,
  syn stmts upstmts.

production appendholder ::= AppendHolder(stmts)
  stmts.downstmts = appendholder.tailstmts, -- tailstmts is an annotation
  appendholder.upstmts = stmts.upstmts.
production stmts ::= Stmtsnull()
  stmts.upstmts = stmts.downstmts.
production stmts ::= Stmtpair(stmt stmts1)
  stmts1.downstmts = stmts.downstmts,
  stmts.upstmts = Stmtpair(stmt,stmts1.upstmts).
production stmt ::= IfThenElse(expr stmts1 stmts2)
  stmts1.downstmts = Stmtsnull()
  stmts2.downstmts = Stmtsnull()
production stmt ::= While(expr stmts)
  stmts.downstmts = Stmtsnull()
```

**Figure 4.6** APPEND Module

```
production stmt ::= IfThenElse(expr <stmts1> <stmts2>)
  -- treat <stmts1> and <stmts2> as annotations

production stmt ::= While(expr <stmts>)
  -- treat <stmts> as an annotation
```

**Figure 4.7** A Fragment from Module APPEND

APPEND to concatenate a pair of statement lists will be proportional to the number of statements contained directly in the statement list of the *AppendHolder* node.

## 4.3 Module DFLOW

The expressiveness of the Modular Attribute Grammar dialect can be demonstrated with the specification of several solutions to the *live data flow analysis* problem, a key component of any specification of an optimizing compiler. Data flow analysis has traditionally been described by circular attribute grammar specifications or by noncircular attribute grammars resulting from the translations of these circular attribute grammars. We first present the traditional circular and noncircular attribute grammar formulations of live data flow analysis. Then, we present several different specifications of live data flow analysis, including one specification that uses explicit module-based recursion in place of circularity.

Typically, the computation of data flow facts has been formulated as the least fixed point of a set of recursive equations defined on the nodes of the control-flow graph of a program. Such a formulation can be translated directly into a circular attribute grammar [Far86, SEFR89] where the meaning of the circular attribute grammar is taken to be the least fixed point of the system of equations defined by the grammar [CM79]. As an example of a circular attribute grammar for live data flow analysis, consider Farrow's concise, but circular, attribute grammar shown in Figure 4.8. Farrow's specification computes *live-on-entry* information for each statement in any input program. As defined by Farrow, "a variable $X$ is *live-on-entry* to statement $S$ if there is a control flow path from $S$ to another point $P$ in the program such that $P$ uses the value of $X$ and $X$ is not redefined anywhere along the path from $S$ to $P$." In Figure 4.8, attribute *live* will be assigned the set of identifiers that are live-on-entry to a statement or a list of statements, attribute *out* will be assigned the set of identifiers that are *live-on-exit* from a statement or a list of statements, and attribute *in* will be assigned the set of identifiers actually referenced by each expression. Other examples of circular attribute grammar specifications for data flow analysis problems including: constant propagation, available expressions and reaching definitions, are presented in [ES86, SEFR89].

Since standard batch and incremental attribute grammar evaluation algorithms are not directly applicable to circular attribute grammars, the circular attribute grammars for data flow analysis are usually transformed into noncircular attribute grammars either by ad-hoc or formal means [SEFR89]. Figure 4.9 depicts a version of Farrow's noncircular attribute grammar derived by an ad-hoc translation of the specification shown in Figure 4.8. In order to break the circularity between attributes *stmt.live*, *stmts.live* and *stmts.out* associated with WHILE nodes, Farrow introduced the new attributes *inside* and *thru* for each statement and list of statements. The attributes *live* and *out* associated with statements and lists of statements, and attribute *in* associated with expressions, each have exactly the same meaning and values as in the circular specification. Attribute *inside* will contain the set of identifiers that are referenced by a statement or a list of statements

```
stmt ::= ID = expr.
  stmt.live = (stmt.out - {ID}) ∪ expr.in

stmt ::= IF expr THEN stmts1 ELSE stmts2 END.
  stmt.live = expr.in ∪ stmts1.live ∪ stmt2.live
  stmts1.out = stmt.out
  stmts2.out = stmt.out

stmt ::= WHILE expr DO stmts END.
  stmt.live = stmt.out ∪ (expr.in ∪ stmts.live)
  stmts.out = stmt.out ∪ (expr.in ∪ stmts.live)

stmts ::= stmt ; stmts1.
  stmts.live = stmt.live
  stmt.out = stmts1.live
  stmts1.out = stmts.out

stmts ::= .
  stmts.live = stmts.out
```

**Figure 4.8** Farrow's Circular Attribute Grammar for Live Data Flow Analysis

```
program ::= stmts
  stmts.universe = stmts.ids

stmt ::= ID = expr.
  stmt.ids = {ID} ∪ expr.in.
  stmt.live = (stmt.out - {ID}) ∪ expr.in
  stmt.inside = expr.in
  stmt.thru = stmt.universe - {ID}

stmt ::= IF expr THEN stmts1 ELSE stmts2 END.
  stmt.ids = expr.in ∪ stmts1.ids ∪ stmts2.ids
  stmts1.universe = stmt.universe
  stmts2.universe = stmt.universe
  stmt.live = expr.in ∪ stmts1.live ∪ stmts2.live
  stmts1.out = stmt.out
  stmts2.out = stmt.out
  stmt.inside = expr.in ∪ stmts1.inside ∪ stmts2.inside
  stmt.thru = stmts1.thru ∪ stmts.thru

stmt ::= WHILE expr DO stmts END.
  stmt.ids = expr.in ∪ stmts.ids
  stmts.universe = stmt.universe
  stmt.live = expr.in ∪ stmt.out ∪ stmts.live
  stmts.out = expr.in ∪ stmt.out ∪ stmts.inside
  stmt.inside = expr.in ∪ stmts.inside
  stmt.thru = stmt.universe

stmts ::= stmt ; stmts1.
  stmts.ids = stmt.ids ∪ stmts1.ids
  stmt.universe = stmts.universe
  stmts1.universe = stmts.universe
  stmts.live = stmt.live
  stmt.out = stmts1.live
  stmts1.out = stmts.out
  stmts.thru = stmt.thru ∩ stmts1.thru
  stmts.inside = stmt.inside ∪ (stmts1.inside ∩ stmt.thru)

stmts ::= .
  stmts.ids = {}
  stmts.live = stmts.out
  stmts.inside = {}
  stmts.thru = stmts.universe
```

**Figure 4.9** Noncircular Attribute Grammar for Live Analysis before being redefined, and attribute *thru* will contain the set of identifiers for which there is a definition-free path through the statement or list of statements. Figure 4.9 does not correspond exactly to Farrow's specification since we have included the explicit computation and distribution of the universe of identifiers using attributes *ids* and *universe*. Farrow notes that this noncircular attribute grammar specification is a poor representation of the original circular attribute grammar since it requires a larger number of attributes and an additional operator on sets, namely intersection. In addition, the non-circular specification duplicates a significant portion of the computation described by the circular specification in order to break the circularity.

Farrow's specification, shown in Figure 4.9, is trivially translated into the Modular Attribute Grammar specification shown in Figure 4.10 and Figure 4.11. The specification takes an abstract syntax term representing a program fragment as its input and returns, as its output, a new syntax term that represents the same program fragment annotated with the live data flow analysis facts. This specification will be referred to as the "Noncircular Modular Attribute Grammar Specification of Live Analysis."

The undesirable qualities of the specification resulting from the ad-hoc translation of a circular attribute grammar into a noncircular attribute grammar led Farrow to examine direct evaluation schemes for *circular, but well-defined, attribute grammars* [Far86]. Farrow demonstrated that the members of this subclass of circular attribute grammars meet restrictions that allow them to be translated directly into an executable pro-

```
MODULE DFLOW
default→dflow
ROOT dflowholder – – the root phylum
RESULT tree – – the result attribute

attributes dflowholder
        syn ⟨stmts,dflow⟩ tree.

attributes stmts
        syn idlist ids,
        inh idlist universe,
        syn ⟨stmts,dflow⟩ tree,
        syn idlist thru,
        inh idlist out,
        syn idlist live,
        syn idlist in.

attributes stmt
        syn idlist ids,
        inh idlist universe,
        syn ⟨stmt,dflow⟩ tree,
        syn idlist thru,
        inh idlist out,
        syn idlist live,
        syn idlist in

attributes expr
        syn idlist in

production dflowholder ::= DflowHolder(stmts)
  stmts.universe = stmts.ids,
  dflowholder.tree = stmts.tree,
  stmts.out = dflowholder.out. – – out is an annotation of DflowHolder

production stmts ::= Stmtsnull()
  local idlist live
  stmts.ids = {},
  stmts.live = live = stmts.out,
  stmts.inside = {},
  stmts.thru = stmts.universe,
  – – create a Stmtsnull node of term type dflow and having "live" and "stmts.out" as
  – – its "in" and "out" annotations
  stmts.tree = ⟨Stmtsnull,dflow⟩(live,stmts.out).
```

**Figure 4.10** Noncircular Modular Attribute Grammar for Live Analysis: Part I

```
production stmts ::= Stmtpair(stmt stmts1)
  local idlist live
  stmts.ids = stmt.ids ∪ stmts1.ids,
  stmt.universe = stmts1.universe = stmts.universe,
  stmts.live = live = stmt.live,
  stmt.out = stmts1.live,
  stmts1.out = stmts.out,
  stmts.thru = stmt.thru ∩ stmts1.thru,
  stmts.inside = stmt.inside ∪ (stmts1.in ∩ stmt.thru),
  stmts.tree = ⟨Stmtpair,dflow⟩(stmt.tree,stmts1.tree,live,stmts.out).

production stmt ::= Assign(identifier expr)
  local idlist live
  stmt.ids = {identifier.ident} ∪ expr.in,
  stmt.live = live = (stmt.out - {identifier.ident}) ∪ expr.in,
  stmt.inside = expr.in,
  stmt.thru = stmt.universe - {identifier.ident},
  stmt.tree = ⟨Assign,dflow⟩(ident,expr,live,stmt.out).

production stmt ::= IfThenElse(expr stmts1 stmts2)
  local idlist live
  stmt.ids = expr.in ∪ stmts1.ids ∪ stmts2.ids,
  stmts1.universe = stmts2.universe = stmt.universe,
  stmt.live = live = expr.in ∪ stmts1.live ∪ stmts2.live,
  stmts1.out = stmts2.out = stmt.out,
  stmt.inside = expr.in ∪ stmts1.in ∪ stmts2.in,
  stmt.thru = stmts1.thru ∪ stmts2.thru,
  stmt.tree = ⟨IfThenElse,dflow⟩(expr,stmts1.tree,stmts2.tree,live,stmt.out).

production stmt ::= While(expr stmts)
  local idlist live
  stmt.ids = expr.in ∪ stmts1.ids,
  stmts.universe = stmt.universe,
  stmt.live = live = expr.in ∪ stmt.out ∪ stmts.live,
  stmts.out = expr.in ∪ stmt.out ∪ stmts.inside,
  stmt.inside = expr.in ∪ stmts.inside,
  stmt.thru = stmt.universe,
  stmt.tree = ⟨While,dflow⟩(expr,stmts.tree,live,stmt.out).
```

**Figure 4.11** Noncircular Modular Attribute Grammar for Live Analysis: Part II gram using explicit successive approximation to compute least fixed points for circularly defined attributes. Farrow's technique can be applied whenever every node in a dependency cycle in a circular dependency graph represents an attribute whose domain is a complete partial ordering in which it is possible to test pairs of elements for equality, and where every edge in the dependency cycle is labeled with a monotonic function satisfying the ascending chain condition. These restrictions guarantee that successive approximation will compute the least fixed point as desired.

Farrow's strategy for evaluation extends the applicative approach to batch evaluation described in Section 2.2.2. The resulting evaluator, referred to as the *recursive synth-function evaluator*, consists of a collection of recursive functions, one per synthesized attribute associated with each nonterminal in the context free grammar underlying the attribute grammar specification, exactly as described earlier; however, some of the recursive functions include explicit fixed point computations to compute values for circularly defined attributes by successive approximation.

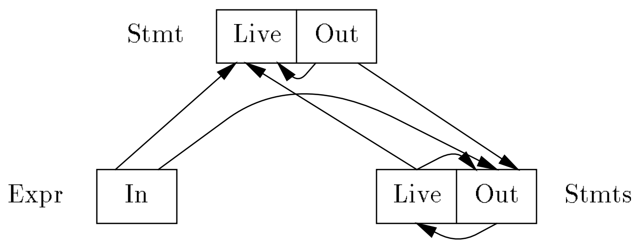


**Figure 4.12** Augmented Dependency Graph for *While*

```
stmt–LIVE(T,OUT) =
  case production(T) of
    [stmt ::= identifier = expr.]
      ...

    [stmt ::= IF expr THEN stmts1 ELSE stmts2 ENDIF.]
      ...

    [stmt ::= WHILE expr DO stmts END.]
      return
        reclet
          exprin = expr–IN(T[1]) – – T[i] is the ith child of T
          evalstmtslive =
            lambda (out live)
              let
                tmpout = out ∪ exprin ∪ live
                tmplive = stmts–LIVE(T[2],tmpout)
              in
                if tmpout ≠ out or tmplive ≠ live
                then evalstmtslive(tmpout,tmplive)
                else tmplive
              endlet
        in
          OUT ∪ exprin ∪ evalstmtslive({},{})
        endlet

    [stmt ::= ]
      ...
  esac
```

**Figure 4.13** Applicative Program Fragment from Recursive Synth-Function Evaluator

Each function created by Farrow's technique computes a synthesized attribute $s$ of a nonterminal $N$ associated with the root of a tree $T$ given a set of values of inherited attributes comprising the argument selector for attribute $s$. If the value of attribute $s$ depends on a circularly-defined attribute in the *augmented* *dependency graph* for that production, then the expression computing the value of $s$ includes explicit code to perform successive approximation of each of the circularly defined attributes. The augmented dependency graph for a production is the union of the dependency graph for the production with the set of all potential dependencies between attributes at the root of the production and the sets of all potential dependencies between attributes at each of the children of the production. The augmented dependency graph associated with the *While* production is shown in Figure 4.12. Notice the cycle containing the pair of attributes *stmts.live* and *stmts.out*. A fragment of the function *stmt–LIVE* created by applying Farrow's translation of the circular attribute grammar from Figure 4.8 is depicted in Figure 4.13.

We now examine a means by which live data flow analysis can be expressed directly as a non-circular Modular Attribute Grammar using recursion in place of circularity. The hierarchical approach to the expression of data flow analysis will mimic Farrow's successive approximation method, but will do so within the Modular Attribute Grammar framework by introducing a recursive module encapsulating the successive approximation step and by defining a module that computes the live-on-entry data flow sets for any statement list given a value for the live-on-exit data flow sets for those statements.

The Modular Attribute Grammar describing live data flow analysis is shown in Figure 4.14. The grammar takes an abstract syntax tree as its input and constructs, as its output, an abstract syntax tree with data flow sets live-on-entry and live-on-exit annotating each of its nodes. The *DflowHolder* node "holds" a list of statements for the purpose of performing live data flow analysis over those statements.

Module DFLOW is invoked with a *DflowHolder* node of term type *default* containing a list of statements over which live data flow analysis is to be performed, and returns a list of statements of term type *dflow* with nodes annotated with the computed data flow facts. Module DFLOW is identical, in spirit, to the original circular specification of live data flow analysis, except for the attribution of the *While* statement and the new semantic equations defining the result attribute *tree*.

The semantic rules for the *While* statement invoke module DFLOWLOOP with a *DflowHolder* node containing the statement list from the *While* statement and the initial value $stmt.out \cup expr.in$ for *stmts.out* to compute the least fixed point for the value *stmts.live* by successive approximation starting with this initial value of *stmts.out*. This value of the least fixed point for *stmts.live* is returned as the *in* annotation on the abstract syntax tree created by module DFLOWLOOP. The value of *stmts.live* computed by successive approximation is then unioned with *expr.in* and *stmt.out* to get the final value for *stmt.live*.

Module DFLOWLOOP takes the value *dflowholder.out* as the current "best approximation" to the value for *stmts.out* and invokes module DFLOW to compute a value for *stmts.live*. As before the new value for *stmts.live* is retrieved from the abstract syntax tree created by DFLOW. A new approximation for *stmts.out* is then formed by unioning together the new value for *stmts.live* with the previous "best approximation" to the value for *stmts.out* stored in *dflowholder.out*. Monotonicity guarantees that the any identifier in the

```
MODULE DFLOWLOOP
default→dflow
ROOT dflowholder
RESULT tree

attributes dflowholder
      syn ⟨stmts,dflow⟩ tree.

production dflowholder ::= DflowHolder(stmts)
  local ⟨stmts,dflow⟩ tree
  local idlist newout

  tree = DFLOW(dflowholder),
  newout = dflowholder.out ∪ tree.in, – – out is an annotation

  dflowholder.tree =
      if dflowholder.out == newout
        then tree
        else DFLOWLOOP(DflowHolder(stmts,newout))
      endif.
```

**Figure 4.14** The Recursive Modular Attribute Grammar Specification of Live Analysis: Part I

```
MODULE DFLOW
default→dflow
ROOT dflowholder
RESULT tree

attributes dflowholder
        syn ⟨stmts,dflow⟩ tree.
attributes stmts
        syn idlist live,inh idlist out,
        syn ⟨stmts,dflow⟩ tree.
attributes stmt
        syn idlist live, inh idlist out,
        syn ⟨stmt,dflow⟩ tree.
attributes expr
        syn idlist ids.

production dflowholder ::= DflowHolder(stmts)
  stmts.out = dflowholder.out, – – out is an annotation
  dflowholder.tree = stmts.tree.

production stmts ::= Stmtsnull()
  local idlist live
  stmts.live = live = stmts.out,
  stmts.tree = ⟨Stmtsnull,dflow⟩(live,stmts.out).

production stmts ::= Stmtpair(stmt stmts1)
  local idlist live
  stmts.live = live = stmt.live,
  stmt.out = stmts1.live,
  stmts1.out = stmts.out,
  stmts.tree= ⟨Stmtpair,dflow⟩(stmt.tree,stmts1.tree,live,stmts.out).

production stmt ::= Assign(identifier expr)
  local idlist live
  stmt.live = live = (stmt.out - {identifier.ident}) ∪ expr.in,
  stmt.tree = ⟨Assign,dflow⟩(identifier,expr,live,stmt.out).

production stmt ::= IfThenElse(expr stmts1 stmts2)
  local idlist live
  stmt.live = live = expr.in ∪ stmts1.live ∪ stmts2.live,
  stmts1.out = stmt.out,
  stmts2.out = stmt.out,
  stmt.tree = ⟨IfThenElse,dflow⟩(expr,stmts1.tree,stmts2.tree,live,stmt.out).

production stmt ::= While(expr stmts)
  local idlist live
  local ⟨stmts,dflow⟩ tree
  tree = DFLOWLOOP(DflowHolder(stmt.out ∪ expr.in,stmts))
  stmt.live = live = stmt.out ∪ expr.in ∪ tree.in,
  stmts.out = {},
  stmt.tree = ⟨While,dflow⟩(expr,tree,live,stmt.out).
```

**Figure 4.15** The Recursive Modular Attribute Grammar Specification of Live Analysis: Part II previous "best approximation" will also be in the new approximation; hence, it is not necessary to construct the new approximation by unioning together the new value for *stmts.live* with the original values of *expr.in* and *stmt.out*. If the new approximation for *stmts.out* based on this new value for *stmts.live* is not identical to the prior approximation for *stmts.out* then it is necessary to invoke DFLOWLOOP recursively with a *DflowHolder* node containing the new approximation for *stmts.out* in order to find a better approximation. Iteration in the cyclic attribute grammar is replaced by recursion in the Modular Attribute Grammar, and the recursion terminates when the value of *stmts.out* reaches the least fixed point.

```
production stmt ::= While(expr < stmts >)
  – – treat < stmts > as an annotation
```

**Figure 4.16** A Fragment from Module DFLOW

This final specification is like module APPEND in that it is wasteful to perform evaluation over the statement list included within *While* nodes since the value of attribute *stmts.live* is not used. In this case, however, both of the statement lists in the *IfThenElse* node should be attributed, since *stmts1.live* and *stmts2.live* are used. The fragment shown in Figure 4.16 can be used to indicate that statement list *stmts* should be treated as an annotation in module DFLOW.

## 4.4 Summary

This chapter has presented a subset of the modules from a pair of source-to-source code optimizers for a simple source language in order to give an impression of the use of hierarchical attribute grammars in specifying potentially complex systems as a set of independent modules. The utility of explicit recursion between modules of a hierarchical attribute grammar was demonstrated in the specification of the root module of an iterate-until-stable optimizer specification and in the specification of live data flow analysis.

# Chapter 5

# Batch Evaluation: Tree-Walking and Applicative Approaches

Although we believe that hierarchical dialects, as exemplified by Modular Attribute Grammars, are an effective tool for specifying complex modular systems, their use for real problems depends upon the availability of efficient algorithms for batch and incremental evaluation. Since hierarchical attribute grammars are, in essence, traditional attribute grammars with explicit support for recursion, it is reasonable to expect that the efficient batch and incremental evaluation algorithms for traditional non-hierarchical attribute grammars can be extended to support evaluation of hierarchical attribute grammars. In this short chapter, we describe a pair of batch evaluation algorithms for hierarchical specifications. The following chapters will examine incremental evaluation algorithms for hierarchical attribute grammars. The algorithms presented in this and the following chapters are intended to be applicable to any hierarchical dialect that provides a subset of the features of Modular Attribute Grammars.

## 5.1 The Tree-Walking Approach

A batch evaluator for hierarchical attribute grammars in which each module is an ordered attribute grammar can be built by applying a slightly extended version of the ordered attribute grammar evaluator described in Section 2.2.1 to each of the modules in a hierarchical system. Since the input to each module in a hierarchical attribute is likely to be a dag-structured term, the ordered evaluator for modules of a hierarchical specification will explicitly construct an attribute tree, as was described in Section 2.3.1 in the context of incremental evaluation of non-hierarchical attribute grammars, to represent the syntactic input to each module.

All attributes computed by a module will be stored in the nodes of that module instance's attribute tree, and the inherited and synthesized attributes associated with each node will be stored in that node. Each attribute tree node will contain a pointer *parent* identifying the attribute tree node which is the parent of that node, and *dagnode* identifying the syntax node the attribute tree node represents. If an attribute tree represents a dag-structured input term, then multiple attribute tree nodes will represent the same syntax node. Annotation references and syntactic references will follow the pointer to a syntax node to retrieve the appropriate annotation or subterm from the term being evaluated. Recursive evaluation of modules occurs naturally whenever a module is invoked.

Function *Evaluate*, shown in Figure 5.1, evaluates each module in a hierarchical attribute grammar specification by invoking *BatchBuildAttrTree*, shown in Figure 5.2, to construct the attribute tree for the syntactic input term, and then proceeds to interpret plans guiding the traversal of the attribute tree and the evaluation of attributes. *Evaluate* invokes *ArgList*, shown in Figure 5.3, to construct lists of values to be passed to each semantic function that is invoked. This version of *ArgList* extends the version presented in Section 2.2.1 to handle annotation references and syntactic references in addition to attribute references. During evaluation of any EVAL instruction, a recursive call to *Evaluate* may be made to evaluate a module.

As an alternative to invoking *BatchBuildAttrTree* to construct the complete attribute tree for a module at the start of evaluation of that module, the attribute tree can be created lazily during the evaluator's traversal. At the start of evaluation, the root node of the attribute tree must be created. Then, new attribute tree nodes will be created and attached to the ever-growing attribute tree just prior to the evaluator's initial visit to each node in the attribute tree. If necessary, each node $N$ in the attribute tree can be deallocated immediately after execution of the last VISIT-PARENT in the plan for $N.parent$.

The cost of batch evaluation of a hierarchical attribute grammar using the tree-walking approach is the sum of the costs of evaluating each of the module instances in MOD-INSTS, the set of module instances invoked

```
Function Evaluate(D)
  – – Copy the dag D to create the attribute tree
  AttrRoot = BatchBuildAttrTree(D,0)
  AttrNode = AttrRoot
  index = 1
  forever do
    if plan[AttrNode.op][index] is EVAL(i,a)
      args = ArgList(AttrNode,i,a)
      AttrNode[i].a = apply semfunc[i,a] to args
      index = index + 1
    elseif plan[AttrNode.op][index] is VISIT(i,r)
      index = MapVisitToIndex(AttrNode[i].op,0,r)
      AttrNode = AttrNode[i]
    elseif plan[AttrNode.op][index] is VISIT-PARENT(r)
      if AttrNode == AttrRoot
        result = ResultAttr(AttrNode)
        – – Deallocate the entire attribute tree
        AttrTreeFree(AttrRoot)
        return result
      fi
      index = MapVisitToIndex(AttrNode.parent.op,AttrNode.son,r)
      AttrNode = AttrNode.parent
    fi
  od
end
```

**Figure 5.1** Evaluate: Batch Evaluation of a Hierarchical Attribute Grammar

```
Function BatchBuildAttrTree(D,son)
  T = NewAttrNode(D.op)
  T.parent = NULL
  T.son = son
  T.dagnode = D
  for each i in [1..NumChildren(D)]
    T[i] = BatchBuildAttrTree(D[i],i)
    T[i].parent = T
  return T
end
```

**Figure 5.2** BatchBuildAttrTree: Batch Evaluation of a Hierarchical Attribute Grammar

```
Function ArgList(anode,index,attr)
  – – create a list of values to pass as arguments to semfunc[index,attr]
  argsdesc = ArgsDescFor(anode.op,index,attr)
  – – argsdesc is a sequence of annotation references and attribute references
  args = NULL
  for each arg in argsdesc
    if arg is ATTR-REF(i,a)
      add attribute anode[i].a to args
    elseif arg is ANNOT-REF(a)
      syn-node = anode.dagnode
      add annotation syn-node.a to args
    elseif arg is SYN-REF(i)
      syn-node = anode.dagnode
      add syn-node[i] to args
    fi
  return args
end
```

**Figure 5.3** ArgList: Batch Evaluation of a Hierarchical Attribute Grammar during batch evaluation. If $S_{inst}$ is the syntactic input term to the module instance *inst* in MOD-INSTS, then, ignoring the cost of module instances invoked by *inst*, the cost of evaluating *inst* is $O(|TREE(S_{inst})|)$, where $TREE(S_{inst})$ is the attribute tree that represents the term $S_{inst}$. The cost of evaluating a complete hierarchical specification is, therefore, $O(\sum_{inst \in \text{MOD-INSTS}} |TREE(S_{inst})|)$.

## 5.2 The Applicative Approach

The applicative approach to batch evaluation of non-hierarchical attribute grammars can be applied directly to batch evaluation of hierarchical attribute grammars. As described in Section 2.2.2, the applicative approach is based upon translating an attribute grammar specification into a functional program that is then executed with the term to be evaluated as its input. To apply the applicative approach in the hierarchical context it is necessary to translate each module in a hierarchical specification into a set of recursive functions named to identify the module for which that function was created, and to translate each module invocation into an invocation of the function that computes the synthesized result attribute for that module. The applicative approach is free from complications arising from the fact that the input term to modules may be dag-structured values since it does not store any information into the nodes of the syntactic term being evaluated and since it does not use parent pointers to traverse the syntactic input term.

Each module $M$, in a complete hierarchical specification, will be translated into a collection of recursive functions named $M$–$N$–$s$, where $N$ is a phylum derivable from the root phylum of module $M$, and $s$ is a synthesized attribute associated with phylum $N$. Each invocation of a module $M$ to evaluate an input term $T$ will be replaced with an invocation of function $M$–$N$–$s$, where phylum $N$ is the root phylum of module $M$, and synthesized attribute $s$ is the result attribute of module $M$. As was the case for batch evaluation of non-hierarchical attribute grammars, efficient batch evaluation requires that attribute values computed during evaluation of the functional program be cached to prevent unnecessary reevaluations of those attributes. Computed attribute values can either be explicitly stored into attributes associated with nodes of an attribute tree representing the syntax term being evaluated, or a function cache can be applied to avoid multiple evaluations of the same attribute. The cost of evaluating a complete hierarchical specification using the resulting evaluator is $O(\sum_{inst \in \text{MOD-INSTS}} |TREE(S_{inst})|)$, where $TREE(S_{inst})$ is a tree that represents the term $S_{inst}$. The tree $TREE(S_{inst})$ will only be explicitly constructed if it will be used to store computed attribute values.

## 5.3 Summary

This short chapter has described a pair of batch evaluators for hierarchical attribute grammars. Essentially, batch evaluation of a hierarchical specification is performed by applying a batch evaluator for non-hierarchical attribute grammars to each module in the hierarchical specification. Although the techniques for batch evaluation of hierarchical attribute grammars follow directly from techniques for batch evaluation of non-hierarchical attribute grammars, as we will see in the next chapter, incremental evaluators for hierarchical attribute grammars and non-hierarchical attribute grammars will differ significantly.

# Chapter 6

# Incremental Evaluation: Tree-Walking Approaches

This chapter, Chapter 7, and Chapter 8 present tree-walking algorithms for incremental evaluation of hierarchical attribute grammars. The applicative approach to incremental evaluation of hierarchical attribute grammars is presented in Chapter 8, as well, as a natural extension to the tree-walking algorithms.

## 6.1 The Tree-Walking Approach

In general, incremental evaluation takes advantage of the similarities between the intermediate results computed during the solution of similar problems to reduce the total work required to compute the solution to a new problem that is *nearby* to some previously solved problem. Incremental evaluation, therefore, requires maintaining a database of intermediate results computed during the solution of a problem and then reusing those intermediate results to compute a solution to a nearby problem. In the context of hierarchical attribute grammars, a "problem" corresponds to the complete evaluation of a hierarchical specification, a "subproblem" corresponds to an invocation of a module instance during evaluation of a hierarchical specification, and an "intermediate result" corresponds to an attribute value.

Batch evaluation of a hierarchical attribute grammar to compute the solution to a problem $P$ leads to the invocation of a set of module instances, one per subproblem in $P$, related by a module instance tree $MI_P$. Invoking a batch evaluator to compute the solution to a problem $P'$ that is nearby to problem $P$ then leads to the invocation of a set of module instances, one per subproblem in $P'$, related by a module instance tree $MI_{P'}$. Presuming that $P$ and $P'$ are sufficiently similar, then many of the subproblems in $P$ and $P'$ should also be similar. Due to the general manner in which modules are invoked during evaluation of a hierarchical specification, there may be subproblems in $P'$ that do not correspond to subproblems in $P$, and subproblems in $P$ that do not correspond to subproblems in $P'$. An efficient incremental evaluation algorithm should take advantage of the intermediate results computed during the solution of the subproblems in $P$ to compute the solutions to the nearby subproblems in $P'$.

Tree-walking incremental evaluation algorithms for hierarchical attribute grammars will maintain an attribute tree as the database of retained facts for each module instance in a hierarchical system and will apply an incremental evaluator to update each module instance, as necessary. Reusing previously computed attribute values amounts to constructing the new inconsistent attribute tree for each module instance from components of the old consistent attribute tree. Unfortunately, as a result of the user's application of an editing operation to the syntax term to be evaluated by the root module of a hierarchical specification, only the root module instance will be invoked with an attribute tree and an explicit edit description indicating how to construct the new inconsistent attribute tree from the previous consistent attribute tree. The other module instances will be invoked with an attribute tree for that module instance and a complete syntax term to be evaluated.

To resolve this problem, we have chosen to take an approach that requires that the new input term for a module instance be explicitly compared with that module instance's previous input term in order to determine which subtrees from the old, consistent attribute tree should be used in constructing the new, inconsistent attribute tree for that module instance. We refer to the process by which the new attribute tree is constructed as *matching* since it is responsible for identifying subtrees of the old attribute tree that can be used to represent regions of the new attribute tree. Once the new attribute tree has been constructed, a traditional change propagation algorithm can be applied to update the tree. Unfortunately, an incremental evaluator based on this matching approach will not be optimal, since, as will be shown in Section 6.5, selecting old attribute subtrees for the new attribute tree in a manner that minimizes the total time required to evaluate a module instance, requires that the attribute value context be taken into consideration during matching. In the next chapter, a pair of heuristics will be introduced in order to guide the construction of the new attribute tree.

Matching is only truly necessary for incremental evaluation of module instances other than the root module instance of a hierarchical system, since the user's explicit editing operations are available for use by an incremental evaluator for the root module instance. To simplify the evaluation strategy for hierarchical specifications, and to permit abstract syntax to be implemented by hash-consing, we have chosen to adopt an editing model in which the user's destructive editing operations are explicitly mapped into applicative updates to the input term for the root module instance, i.e., a subtree replacement operation will be implemented by creating a chain of new syntax nodes to represent the ancestors of the replacement site in the old term. Therefore, incremental evaluation of the root module instance will be performed in exactly the same fashion as all other module instances.

## 6.2 Teitelbaum and Chapman's Algorithm for Incremental Evaluation of Higher Order Attribute Grammars

In this section, we present and critique the only previous attempt at efficient incremental evaluation of hierarchical attribute grammars, Teitelbaum and Chapman's algorithm for incremental evaluation of Higher Order Attribute Grammars. The algorithm consists of a matching algorithm combined with a traditional change propagation algorithm. We will refer to Teitelbaum and Chapman's matching algorithm as the "depth-first algorithm," since it works in a depth-first fashion over the union of the dags representing the old and new syntax terms.

In [TC90], Teitelbaum and Chapman examined the application of Higher Order Attribute Grammars to the description of editing environments, and proposed extending the Synthesizer Generator to support Higher Order Attribute Grammars. Recall from section 3.2.5, Higher Order Attribute Grammars permit a "base syntax tree" to be extended by assigning computed syntax tree components to *nonterminal attributes* associated with syntax tree nodes.

Teitelbaum and Chapman considered several interesting problems that arise with the introduction of the higher order concept to attribute grammars for use in interactive editor-based systems, including: (1) what happens if the user of the editor tries to explicitly modify a "computed" portion of the syntax tree, and (2) what happens if the user makes a change to any portion of the syntax tree and the change subsequently causes nonterminal attributes to become inconsistent, i.e., the new value of the nonterminal attribute computed in response to the explicit syntax modification is different than the value previously held by that nonterminal attribute. Since evaluation was performed over the term previously assigned to the nonterminal attribute, some effort should be made to reuse portions of that previous computation in performing a computation over the new term assigned to the nonterminal attribute. The first problem is not relevant to this thesis, hence, we assume that the user will only be permitted to modify the base syntax tree, not computed portions of the tree. The second problem is addressed precisely by *matching*.

Teitelbaum and Chapman never describe all of the aspects of the evaluator for a complete higher order specification, but instead concentrate on presenting an algorithm for addressing the second problem listed above — how to handle syntax changes propagating to already expanded nonterminal attributes. Therefore, from the context of the work within the Synthesizer Generator, we infer that incremental evaluation will be triggered by the user's application of a single destructive subtree replacement operation to a subtree of the base syntax tree and will be performed by applying a version of an optimal incremental evaluator such as Reps's incremental ordered evaluator that has been extended to process assignments of values to NEWBORN and RETAINED nonterminal attributes.

Teitelbaum and Chapman argue that, in general, term values are conveniently, and most efficiently, represented as dags instead of trees, hence terms assigned to nonterminal attributes should be assumed to be dag-structured values. Since attribute evaluation is defined as a process over trees, assignment of a term $d$ to a nonterminal attribute $D$ first requires that the term be "copied" to break sharing within the term and to create a new tree that is suitable for evaluation, essentially an attribute tree as suggested previously. Adopting Teitelbaum and Chapman's notation here and throughout the presentation of our matching algorithms in the following chapters, if $d$ is a dag term, then $d_{copy}$ is the attribute tree that results from copying $d$ to break sharing. Hence, instead of extending the syntax term by installing terms into nonterminal attributes, the attribute tree representing the syntax term will be extended with the attribute trees representing each of the terms that is installed into a nonterminal attribute.

Batch evaluation of a base syntax term $B$ begins by constructing an initial incomplete attribute tree $B_{copy}$. Each attribute tree node in $B_{copy}$ has a NULL-valued child pointer for each nonterminal attribute associated with the attribute tree node. Whenever a term value $d$ is assigned to a NEWBORN nonterminal attribute $D$, which represents the $i$th child of an attribute tree node $X$, $d$ is assigned to $D$, and a possibly incomplete attribute tree $d_{copy}$ is constructed and assigned to the $i$th child pointer of $X$ and $X$ is recorded as the parent of the root of $d_{copy}$. During evaluation, nonterminal attributes associated with nodes in $d_{copy}$ will be expanded. On completion of batch evaluation over $B$, $B_{copy}$ will be a complete attribute tree.

Incremental evaluation is performed in response to the application of a destructive editing operation that maps the base syntax term $B$ into the new base syntax term $B'$, and maps the attribute tree $B_{copy}$ into the new attribute tree $B'_{copy}$. Given the initial attribute tree $B'_{copy}$, incremental evaluation begins by applying the modified version of Reps's optimal incremental evaluator to the new attribute tree $B'_{copy}$ with the attribute tree node that is the site of application of the subtree replacement operation as the initial element in *Reactivated*. During evaluation, $B'_{copy}$ will be updated whenever a term is installed into a nonterminal attribute. As in batch evaluation, if a NEWBORN nonterminal attribute $D$ is assigned a term $d$, then a (possibly) incomplete attribute tree $d_{copy}$ representing the term $d$ is created and installed into $B'_{copy}$. If a RETAINED nonterminal attribute $D$ with a value of $d$ is assigned a term, say $d'$, then, instead of removing the attribute subtree $d_{copy}$ representing $d$ from the attribute tree $B'_{copy}$ and installing in its place a completely new subtree $d'_{copy}$ representing $d'$, Teitelbaum and Chapman attempt to determine the relationship between $d$ and $d'$, and $d_{copy}$ and $d'_{copy}$ so that components from $d_{copy}$ can be used in constructing $d'_{copy}$. The new attribute tree $d'_{copy}$ will consist of a set of subtrees from $d_{copy}$ and a set of new attribute tree nodes. Once $d'_{copy}$ has been constructed, the subtree $d_{copy}$ of $B'_{copy}$ is replaced with $d'_{copy}$, each of the unattributed nodes and each of the children of the unattributed nodes in $d'_{copy}$ is inserted into the *Reactivated* set, and evaluation of $B'_{copy}$ continues.

As mentioned earlier, we refer to the process of constructing the new attribute tree representing a new term $d'$ from the old attribute tree representing a term $d$ as *matching*. In essence, a "match" is a list of old subtrees from $d_{copy}$ for use in constructing $d'_{copy}$ along with a location in $d'_{copy}$ where each subtree is to be placed. A "correct match" is one that constructs an attribute tree that properly represents the new term to be attributed.

The dags $d$ and $d'$ and the tree $d_{copy}$, shown in Figure 6.1, will be used as a running example throughout the presentation of the depth-first algorithm. Each dag node in the figure is labeled with a unique id. Each tree node in $d_{copy}$ has a label of the form $X/Y$, where $X$ is the unique id representing the tree node itself, and $Y$ is the unique id associated with a dag node in $d$ represented by that tree node.

Let $d' \cap d$ be the forest of dag subterms that $d$ and $d'$ share. Let $(d' - d)[]$ be an incomplete dag formed by removing subterms that are shared by $d$ and $d'$ from the dag $d'$, and let $(d - d')[]$ be the incomplete dag formed by removing subterms that are shared by $d$ and $d'$ from the dag $d$. The notation $d' - d$ will be used to refer to the set of nodes in $d'$ that do not occur in $d$, and the notation $d - d'$ will be used to refer to the

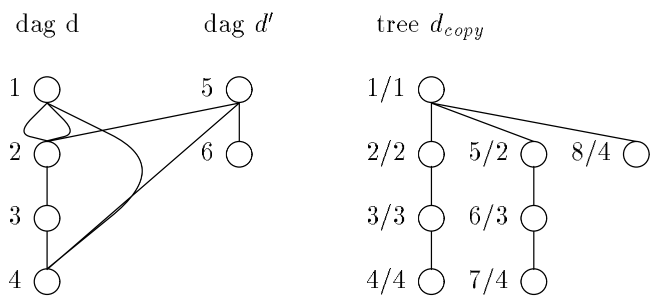


**Figure 6.1** The dags $d$ and $d'$, and the tree $d_{copy}$

set of nodes in $d$ that do not occur in $d'$. Similarly, the notation $d'_{copy} - d_{copy}$ refers to the set of nodes in $d'_{copy}$ that do not occur in $d_{copy}$, and the notation $d_{copy} - d'_{copy}$ refers to the set of nodes in $d_{copy}$ that do not occur in $d'_{copy}$. Figure 6.2 depicts the dags $d' \cap d$, $(d' - d)[]$ and $(d - d')[]$ given $d$ and $d'$ as above.

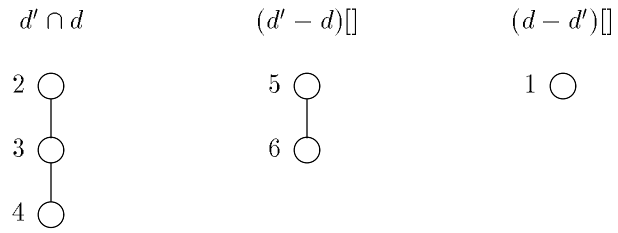


**Figure 6.2** $d' \cap d$, $(d' - d)[]$ and $(d - d')[]$

Teitelbaum and Chapman claim that their algorithm properly constructs the new attribute tree $d'_{copy}$ from components of $d_{copy}$ and does so in time proportional to the number of storage cell deallocations needed to eliminate nodes in $d_{copy}$ that were not used in constructing $d'_{copy}$ and the number of storage cell allocations needed to build the nodes in $d'_{copy}$ which were not available in $d_{copy}$. Thus, the matching algorithm has a running time of $O(|d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}|)$, which Teitelbaum and Chapman claim to be optimal, since this number of allocations and deallocations must be performed to construct $d'_{copy}$. It is true that this number of allocations and deallocations must be performed to construct $d'_{copy}$ from $d_{copy}$ using a particular match, however, there may exist another match for $d_{copy}$ and $d'_{copy}$ having a significantly smaller $|d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}|$. For example, without requiring the matching algorithm to minimize the size of $|d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}|$, Teitelbaum and Chapman's optimal time bound can be achieved by trivially deallocating the entire attribute tree $d_{copy}$ and then allocating the entire new attribute tree $d'_{copy}$.

In order to minimize $|d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}|$, the matching algorithm must construct the new attribute tree $d'_{copy}$ representing $d'$ by reusing maximal subtrees from $d_{copy}$, since

$$|d_{copy} - d'_{copy}| = |d_{copy}| - |d_{copy} \cap d'_{copy}|$$

and

$$|d'_{copy} - d_{copy}| = |d'_{copy}| - |d_{copy} \cap d'_{copy}|,$$

and, therefore,

$$\begin{aligned} |d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}| &= |d_{copy}| - |d_{copy} \cap d'_{copy}| + |d'_{copy}| - |d_{copy} \cap d'_{copy}| \\ &= |d_{copy}| + |d'_{copy}| - 2 * |d_{copy} \cap d'_{copy}|, \end{aligned}$$

which is minimized by maximizing $|d_{copy} \cap d'_{copy}|$ by reusing maximal subtrees from $d_{copy}$ since $|d_{copy}|$ and $|d'_{copy}|$ are constant for a particular pair of dags $d$ and $d'$.

We will use the term *optimal match* to refer to a correct match that reuses maximal subtrees from $d_{copy}$. Unfortunately, the depth-first algorithm, in general, does not construct an optimal match and may exceed the running time of $O(|d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}|)$.

Let $F$ be the set of subtrees from $d_{copy}$ corresponding to the terms in $d' \cap d$. Teitelbaum and Chapman observed that $F$ contains all of the subtrees from $d_{copy}$ that are candidates for use in $d'_{copy}$. The subtrees in $F$ will, therefore, be referred to as *candidates*. The set $F$ of candidates for the example is depicted in Figure 6.3.

The subtrees in $F$ are identified by a multipass marking procedure applied to $d' \cup d$ with each node initially labeled either OLD or NEW. Every syntax node that has appeared within a previously attributed syntactic term will be labeled OLD, including nodes in either the initial syntax tree constructed directly by the front-end editor or nodes in any term that has been installed previously into a nonterminal attribute during evaluation. Newly created syntax nodes will be labeled NEW, including nodes created by the editor itself and nodes created by node-constructing semantic functions.

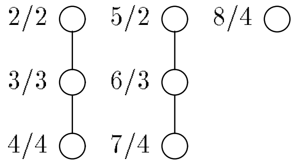


**Figure 6.3** F: Subtrees from $d_{copy}$ that are candidates for $d'_{copy}$.

The term $d'$ that is passed to the matching algorithm consists of a crown of nodes labeled NEW above subterms consisting of nodes labeled OLD, where the term "crown" is used to refer to any contiguous set of nodes rooted by a specified root node of a dag or a tree. The NEW nodes correspond to nodes in $(d' - d)[]$, and the subterms of nodes labeled OLD correspond to nodes in $d' \cap d$. Actually, in Section 6.3.2, it will be shown that nodes labeled OLD are not guaranteed to be in $d' \cap d$. The final step of the matching process labels each of these NEW nodes as being OLD.

During the marking procedure, a node will be labeled as a BOUNDARY node if it is determined to be a direct descendent of a node in $(d' - d)[]$ that is in $d$. A node will be labeled as a ROOT node if, first, it is determined to be a BOUNDARY node, and, later, it is determined to be reachable during a depth-first traversal of $d$ without passing through any other ROOT nodes.

The depth-first algorithm "commits" each subtree in $d_{copy}$ whose root corresponds to a ROOT node in $d' \cap d$ to be a representative of that ROOT node for use in $d'_{copy}$. Given these candidate subtrees, the depth-first algorithm builds $d'_{copy}$ by creating a crown of new, unattributed nodes to connect old subtrees or copies of old subtrees, chosen from the set of candidates. The algorithm deallocates all attribute tree nodes in $d_{copy}$ that are not used in $d'_{copy}$.

The depth-first algorithm constructs $d'_{copy}$ by performing the following five steps:

1. Traverse dag $d'$ in depth-first order from its root to any node with an OLD or BOUNDARY label that is encountered along each path from the root. Label each OLD node that is encountered as a BOUNDARY node. A BOUNDARY node may have an OLD or BOUNDARY node as an ancestor since there may exist multiple paths from the root of $d'$ to any node within $d'$.

   Figure 6.4 shows the labeling of $d$ and $d'$ after this step, assuming that $d$ and $d'$ are defined as above. Nodes 2 and 4 are labeled as BOUNDARY nodes since they were previously OLD nodes and are both reachable from nodes in $(d' - d)[]$.

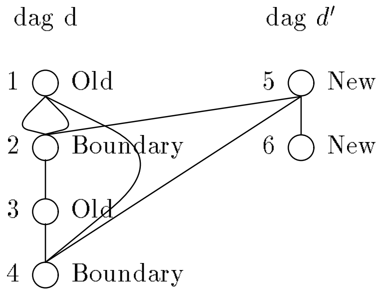


**Figure 6.4** After Step 1: The dags $d$ and $d'$

2. Traverse dag $d$ in depth-first order from its root to any node with a BOUNDARY or ROOT label that is encountered along each path from the root. Label each BOUNDARY node that is encountered as a

ROOT node. Again, a ROOT node in the dag $d$ may have ancestors that are BOUNDARY or ROOT nodes.

Figure 6.5 shows the labeling of $d$ and $d'$ after this step. Nodes 2 and 4 are both labeled as ROOT nodes since they were previously labeled as BOUNDARY nodes and are both direct descendents of nodes in $(d - d')[]$.

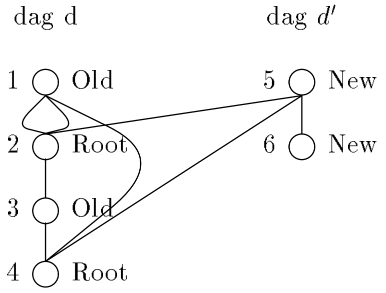


**Figure 6.5** After Step 2: The dags $d$ and $d'$

3. Traverse dag $d$ and tree $d_{copy}$ simultaneously until the first ROOT node is encountered along each path from the root of $d$. When a ROOT node, say $r$, is encountered in $d$, the corresponding node in $d_{copy}$, say $f$, represents the root of a tree in $d_{copy}$ that can be used somewhere in $d'_{copy}$. Update an association map with the new pair $\langle r, f\rangle$ to indicate that the tree rooted by $f$ is available to represent a term rooted by $r$. Since $d$ is a dag, the association map may contain multiple available trees for a single dag term. Later, as $d'_{copy}$ is constructed, the algorithm will reuse as many of the copies of a term as possible. As $d$ is traversed, deallocate any node in $d_{copy}$ that does not correspond to a ROOT node in $d$.

   After this step, the association map contains entries $\langle 2, 2/2\rangle$, $\langle 2, 5/2\rangle$, and $\langle 4, 8/4\rangle$.

4. Traverse dag $d'$ until the first ROOT node is encountered along each path. During the traversal, construct $d'_{copy}$ by creating a new attribute tree node for each node in $d'$ that is labeled NEW, OLD, or BOUNDARY. NEW, OLD, and BOUNDARY nodes each represent nodes for which no copy is available from $d_{copy}$. When a ROOT node $r$ is encountered in $d'$, choose an unused instance of a pair $\langle r, f\rangle$ from the association map, mark the pair as "used", and install subtree $f$ into $d'_{copy}$. If no unused pair $\langle r, f\rangle$ is available, then $d'_{copy}$ needs more copies of the subtree $r$ than $d_{copy}$ required. In this case, choose a "used" pair $\langle r, f\rangle$, create a copy of $f$ and all of its attributes, and install the copy at the appropriate location within $d'_{copy}$.

   Figure 6.6 shows $d'_{copy}$ after this step. Filled circles represent newly created attribute nodes. Traversal of $d'$ creates a new attribute tree node 9/5 for the NEW dag node 5, creates a new attribute tree node 10/6 for the NEW dag node 6, and reuses the subtrees from $d_{copy}$ labeled 2/2 and 8/4.

5. Traverse dag $d'$ until the first OLD, BOUNDARY or ROOT node is encountered along each path and label each traversed node as OLD.

Although not explicitly stated in [TC90], presumably, after the new tree $d'_{copy}$ has been constructed in its entirety, all entries will be removed from the association map.

## 6.3 A Critique of Teitelbaum and Chapman's Depth-First Algorithm

We now present two examples demonstrating deficiencies in the depth-first algorithm. The first example presents a pair of dags $d$ and $d'$ and an attribute tree $d_{copy}$ and demonstrates that due to the complexity of sharing between $d$ and $d'$, the depth-first algorithm non-optimally constructs the new attribute tree $d'_{copy}$. We make no claims that the terms $d$ and $d'$ in this example are typical of terms created by evaluation of

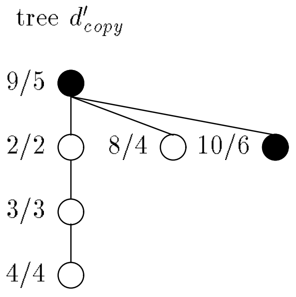


**Figure 6.6** After Step 4: The tree $d'_{copy}$

Higher Order Attribute Grammar specifications, although, it is easy to construct a small specification that constructs exactly this pair of terms in response to a change to the input of the root module instance. The second example demonstrates that the depth-first algorithm is unable to construct the new attribute tree optimally even if terms to be installed into non-terminal attributes are guaranteed to be tree-structured.

### 6.3.1 Example 1: Non-Optimal Matching with Dag-Structured Terms

Figure 6.7 through Figure 6.13 depict an example of the non-optimal construction of $d'_{copy}$ from $d_{copy}$. Figure 6.7 shows two dags $d$ and $d'$ and a tree $d_{copy}$ representing $d$.

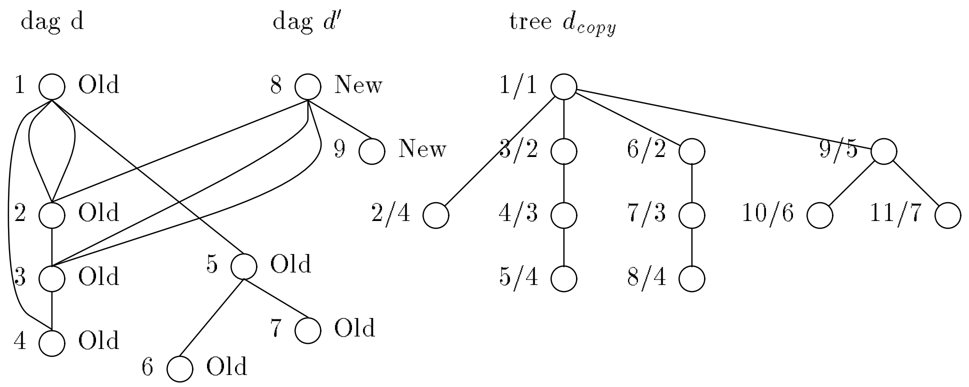


**Figure 6.7** The dags $d$ and $d'$, and the tree $d_{copy}$

Figure 6.8 and Figure 6.9 depict the labelings on dags $d$ and $d'$ after steps 1 and 2 of the algorithm. Figure 6.10 shows the association map entries, while Figure 6.11 depicts the set of subtrees that are identified by step 3 of the algorithm as candidate subtrees from $d_{copy}$ for use in $d'_{copy}$. Figure 6.12 depicts the tree $d'_{copy}$ as constructed by step 4 using the candidate subtrees. Figure 6.13 depicts the optimally constructed tree $d'_{copy}$.

The depth-first algorithm fails to locate and reuse the two subtrees from $d_{copy}$ rooted by nodes 7/3 and 2/4. Essentially, the depth-first algorithm is based upon two mistaken premises:

**Premise 1.** Each candidate subtree in $d_{copy}$ has a root that corresponds to a "root" of a subterm in $d' \cap d$, where a root of a dag term is a node that is in $d' \cap d$ and is a direct descendent of a node in $(d' - d)[]$ and a direct descendent of a node in $(d - d')[]$, i.e., a node labeled ROOT by step 2 of the algorithm.

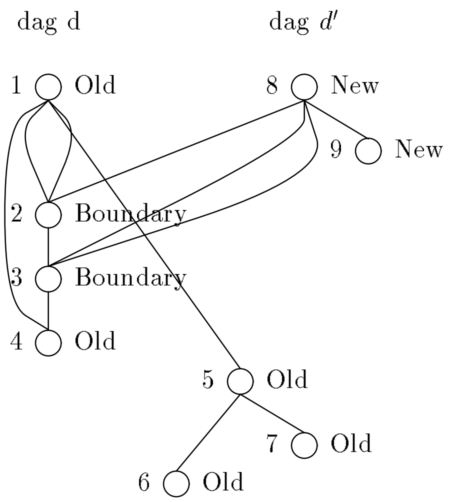


**Figure 6.8** After Step 1: The dags $d$ and $d'$

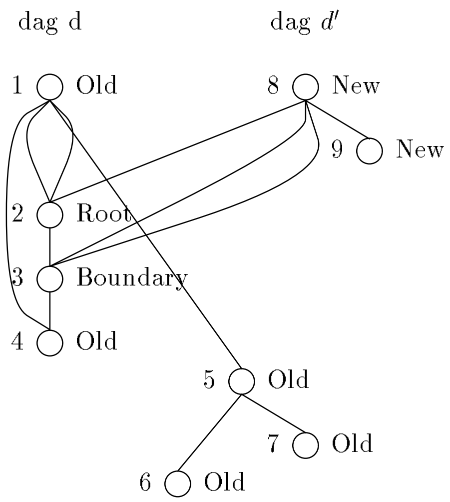


**Figure 6.9** After Step 2: The dags $d$ and $d'$

$\langle 2, 3/2 \rangle$ and $\langle 2, 6/2 \rangle$

**Figure 6.10** After Step 3: Association map entries

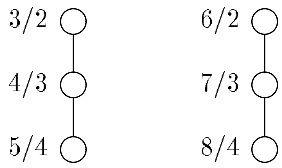


**Figure 6.11** Candidate Subtrees from $d_{copy}$

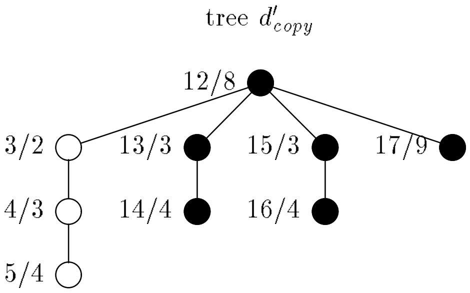


**Figure 6.12** After Step 4: The tree $d'_{copy}$

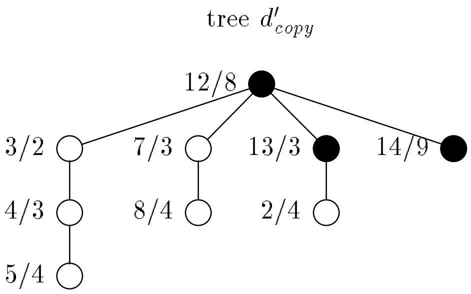


**Figure 6.13** The Optimal Matching: The tree $d'_{copy}$

In the example, although the subtree of $d_{copy}$ rooted by node 2/4 can be used in constructing $d'_{copy}$, since dag node 4 is not a direct descendent of nodes in both $(d' - d)[]$ and $(d - d')[]$, the node is not a root and the subtree rooted by node 2/4 is not even considered to be a candidate for use in $d'_{copy}$.

**Premise 2.** Candidate subtrees should only be used to fill needs for the dag node represented by their roots.

In the example, even though the subtree of $d_{copy}$ rooted by node 6/2 is not needed to represent an instance of dag node 2 in $d'$, the subtree rooted by node 7/3 of the tree rooted by node 6/2 should be used to represent a need for dag node 3 in $d'_{copy}$. The subtree rooted by node 7/3 is not identified as a candidate subtree, since its root is labeled BOUNDARY, not ROOT, because node 3 is not a direct descendent of $(d - d')[]$.

In this example, two small subtrees of $d_{copy}$, which should have been selected and reused in $d'_{copy}$, were not properly identified by the matching algorithm. In practice, these small subtrees could be arbitrarily large and the algorithm could perform an arbitrarily large amount of extra work in constructing $d'_{copy}$. In addition, the number of attributes that must be reevaluated to restore consistency to the sub-optimally constructed attribute tree may be arbitrarily larger than necessary given an optimal match, since attributes associated with attribute tree nodes in the misidentified subtrees may not have needed to be recomputed.

### 6.3.2 Example 2: Interference between Matching Operations

Although the basic mechanism by which the depth-first algorithm selects old attribute subtrees for use in $d'_{copy}$ is flawed in the presence of dag-structured terms, it may appear that in the context of tree-structured terms, the depth-first algorithm will construct the new attribute tree using an optimal match. Unfortunately, even applying the depth-first algorithm to terms that are known to be tree-structured will fail to make optimal use of subtrees from the old attribute tree due to a limitation in the node labeling mechanism used by the algorithm. Recall that labels on syntax nodes in the depth-first algorithm are selected from the set {NEW,OLD,BOUNDARY,ROOT}.

In the depth-first algorithm, as described above, each new node is initially labeled NEW and maintains its NEW label until it is included in a term that is installed in a nonterminal attribute. At that time, the node is relabeled OLD. The matching algorithm uses these labels to locate subterms of $d' \cap d$ by traversing $d'$ from its root to find a node that is labeled OLD, presumably because that node was labeled OLD by the matching algorithm when installing $d$ into nonterminal attribute $D$. Unfortunately, in general, all that can truly be said about a node $N$ in $d'$ that is labeled OLD is that node $N$ appeared in *some* term that had been installed earlier in *some* nonterminal attribute. We now present an example that demonstrates why the single label field on syntax nodes is insufficient.

Consider two overlapping trees $f$ and $g$ which share a subtree $y$. As a result of installing tree $f$, and then tree $g$, into nonterminal attributes, each node in $f$ and $g$ will be labeled OLD. Trees $f$ and $g$ are depicted in Figure 6.14. Next, the subtrees $f'$ and $g'$ are built in such a fashion that they share the subtree $x$

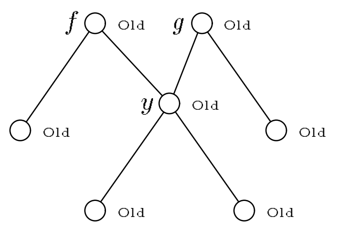


**Figure 6.14** Trees $f$ and $g$ after installation of $f$ and $g$

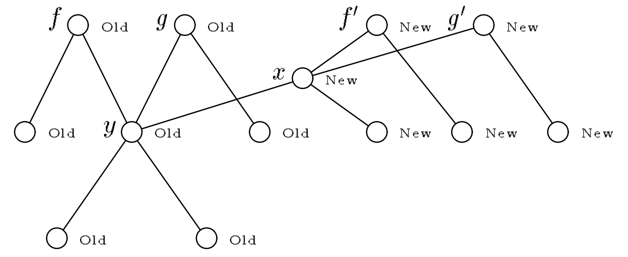


**Figure 6.15** Trees $f$, $g$, $f'$ and $g'$ prior to installation of $f'$ and $g'$

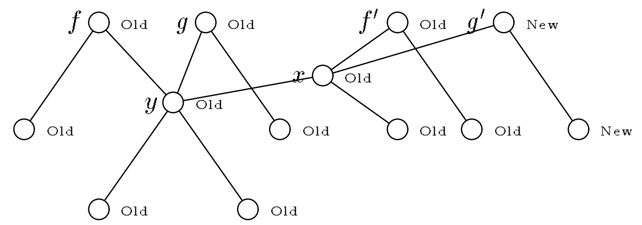


**Figure 6.16** Trees $f$, $g$, $f'$ and $g'$ after installation of $f'$ and prior to installation of $g'$

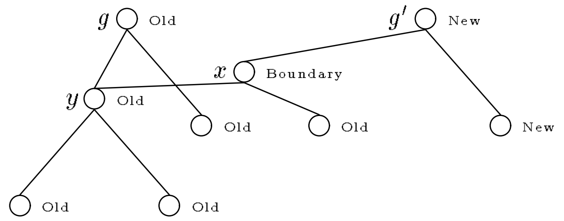


**Figure 6.17** Trees $g$ and $g'$ after Matching step 2

which has the subtree $y$ as a child. The configuration prior to installation of either $f'$ or $g'$ into nonterminal attributes is shown in Figure 6.15. Installing $f'$ into a nonterminal attribute results in labeling each of the nodes in $f'$ as OLD as shown in Figure 6.16. Finally, when $g'$ is installed into a nonterminal attribute, the matching algorithm is invoked with tree $g$ and $g'$. Step 1 of the matching algorithm traverses tree $g'$ to identify potential ROOT nodes. Instead of identifying subtree $y$ as a ROOT, however, it encounters subtree $x$ which is labeled OLD, but which does not correspond to a node in $g$. The root of $x$ is, therefore, labeled as a BOUNDARY node rather than the root of subtree $y$. Step 2 of the algorithm traverses $g$ to convert BOUNDARY nodes to ROOT nodes, but finds no BOUNDARY nodes to convert. The labeling of $g$ and $g'$ after step 2 of the matching algorithm is shown in Figure 6.17. Since no node in $g$ is a ROOT node, no candidate subtrees from $g_{copy}$ are identified for use in $g'_{copy}$, and $g'_{copy}$ must be created entirely from scratch. An optimal match would reuse the subtree from $g_{copy}$ representing subtree $y$ from $g$.

In this example, when the matching operation that is responsible for creating the attribute tree $f'_{copy}$ updates labels on nodes in $f'$, it alters the labels that are needed to guide the construction of $g'_{copy}$.

During their presentation of the algorithm, Teitelbaum and Chapman acknowledge that as the dag $d'$ is traversed, nodes shared with other values may be encountered, but they do not describe the effect this has on the optimality of the construction of $d'_{copy}$.

## 6.4 A Matching-Based Incremental Evaluator

We now transfer the concepts underlying matching in the Higher Order Attribute Grammar setting into the hierarchical attribute grammar setting and describe a matching-based evaluator for modules of hierarchical attribute grammars.

In the context of Higher Order Attribute Grammars, the matching algorithm is invoked each time the syntax tree is extended by the assignment of a term to a nonterminal attribute. Whenever a term is installed into a nonterminal attribute, the matching algorithm is invoked to construct the tree $d'_{copy}$ from the dags $d$ and $d'$ and the tree $d_{copy}$. The attribute subtree $d_{copy}$ is then replaced with $d'_{copy}$, and incremental evaluation of the extended attribute tree continues. In the context of hierarchical attribute grammars, the syntax tree itself is not modified by the evaluation process. Instead, modules are explicitly invoked with syntactic terms to be evaluated according to the attribute grammar specification for that module. Adopting Teitelbaum and Chapman's notation for matching in the hierarchical attribute grammar context, when a module instance that was last invoked to evaluate a term $d$ is invoked to evaluate a term $d'$, the matching algorithm will construct the new attribute tree $d'_{copy}$ from components of the attribute tree $d_{copy}$ representing the dag $d$. The tree $d'_{copy}$ is then updated incrementally.

In the context of Higher Order Attribute Grammars, whenever a new term $d'$ is installed into a nonterminal attribute associated with an attribute tree node $S$, the old term $d$ is available as the retained value of the nonterminal attribute associated with node $S$ and the old attribute tree $d_{copy}$ is available as a child of $S$. Therefore, during batch or incremental evaluation of a Higher Order Attribute Grammar, the attribute tree representing the term most recently installed in any nonterminal attribute is immediately accessible. In the hierarchical setting, however, the batch evaluator does not need to store either the syntax term $d$ or the attribute tree $d_{copy}$, hence, it will be necessary for the incremental evaluator to explicitly save references to these two structures on completion of evaluation of each module instance. Whenever a module instance is invoked, the saved references to $d$ and $d_{copy}$ will be retained as part of the module instance's "saved state," where in general, the saved state for a module instance will contain all of the information from previous evaluations of the module instance that is needed for efficient incremental evaluation of the current module instance.

A complete hierarchical attribute grammar specification will be evaluated by recursively applying a matching-based incremental evaluator to each module in a hierarchical specification. The incremental evaluator is responsible for updating an invoked module instance given the name of a module, a new term $d'$ and the saved state for the invoked module instance. An incremental evaluator based on matching applies a matching algorithm to construct the new attribute tree and then applies a traditional change propagation algorithm to update the resulting tree. During change propagation, whenever a module is invoked as a semantic function, the incremental module evaluator will be invoked recursively with the appropriate module name, syntax term, and saved state.

Let *NumModInvs(op)* be the number of module invocations that occur within the semantic rules associated with operator *op* by a module specification, and let each of the module invocations be numbered with an integer in the range 1 to *NumModInvs(op)*. Each attribute tree node labeled *op* will contain an array *States* of length *NumModInvs(op)*, whose $i$th entry will contain the saved state for module invocation $i$. Each element of the *State* array will initially be assigned the value NULL to indicate that the corresponding module invocation has not yet occurred. Later, whenever the $i$th module invocation actually occurs during evaluation, the value *States[i]* will be passed as the saved state of the invoked module instance. During evaluation, the matching algorithm will update the saved state for each module instance in preparation for subsequent invocations of that module instance.

Function *ModuleEvaluate*, shown in Figure 6.18, first invokes *Match* to construct the new attribute tree $d'_{copy}$, and then invokes the function *Propagate* to update the inconsistent attributes in $d'_{copy}$. *Match* constructs the new attribute tree $d'_{copy}$ representing a dag $d'$ from complete attribute subtrees available in the module instance's saved state. The tree $d'_{copy}$, therefore, consists of a crown of new attribute tree nodes attached to a set of old attribute subtrees, where a *crown* of a tree or a singly-rooted dag consists of a contiguous collection of nodes that contains the root of the tree or dag. We assume that each module in a hierarchical specification is an ordered attribute grammars. Therefore, function *Propagate* can be implemented using the optimal change propagation algorithm described in Section 2.3.1 with the *Reactivated* set initialized to contain the new nodes in the crown of $d'_{copy}$ and the children of those new nodes.

The use of hash-consing to construct syntactic values enables *Propagate* to determine if the old and new values of syntax-valued attributes are equivalent in constant time using a simple pointer equality test, and, thereby, makes it possible for the optimal change propagation algorithm to update the new attribute tree for each module instance *inst* in $O(|\text{AFFECTED}_{inst}|)$ time, where the contents of $\text{AFFECTED}_{inst}$ depends upon which old consistent attribute subtrees were selected by the matching algorithm for use in building the new inconsistent attribute tree. If hash-consing is not used, then the change propagator will be forced to perform an expensive comparison operation on pairs of values, or to use pointer equality as a coarse approximation for determining if two syntactic values are equivalent. In addition, an evaluator that uses pointer equality to approximate equality of syntax-valued attributes may be forced to evaluate a substantially larger number of attributes than an evaluator that uses an exact equality test.

## 6.5 Optimal Evaluation and the Matching-Based Approach

In the non-hierarchical attribute grammar context, an incremental evaluator is invoked in response to a user's editing request which is directly responsible for modifying the old consistent attribute tree to create the new inconsistent attribute tree. Optimal incremental evaluation then requires updating the values of the inconsistent attributes in time proportional to the size of the AFFECTED set, where the contents of the AFFECTED set is determined entirely by the user's edit.

In the hierarchical context, the goal of an optimal evaluator is to minimize the total time required to update a set of module instances in response to a change to the input of the root module instance. Through its selection of old attribute subtrees for reuse in the new attribute tree for module instance $M$, the matching algorithm determines which attributes in the new attribute tree will be in AFFECTED. Therefore, optimal evaluation of a module instance requires minimizing the sum of the cost of matching and the cost of change propagation for that module instance. In addition, the selection of old subtrees for reuse in the new attribute tree for module instance $M$ also determines the cost of updating the attribute tree for each module instance invoked by $M$. This implies that using an optimal incremental evaluator to update each of the module instances in a hierarchical system is insufficient to ensure optimal incremental evaluation of a complete

Function ModuleEvaluate(Module, $d'$, State)
  $d'_{copy}$ = Match($d'$, State)
  return Propagate(Module,$d'_{copy}$)
end

**Figure 6.18** ModuleEvaluate: An Incremental Module Evaluator hierarchical attribute grammar specification. In the rest of this section we only consider the issue of optimal incremental evaluation of module instances.

Let $M$ be a match for a module instance and $\text{REUSED}_M$ be the set of nodes in $d_{copy} \cap d'_{copy}$ for match $M$. Divide $\text{REUSED}_M$ into two disjoint sets: $\text{REUSEDRET}_M$, the set of nodes in $\text{REUSED}_M$ whose attributes are all in RETAINED, and $\text{REUSEDAFF}_M$, the set of nodes in $\text{REUSED}_M$ having attributes in AFFECTED. If $\text{REUSED}_M$ is $d_{copy} \cap d'_{copy}$, then $d_{copy} - \text{REUSED}_M$ is $d_{copy} - d'_{copy}$, and $d'_{copy} - \text{REUSED}_M$ is $d'_{copy} - d_{copy}$. The minimum cost of constructing match $M$ is $\Omega(|d'_{copy} - \text{REUSED}_M| + |d_{copy} - \text{REUSED}_M|)$ since this many attribute nodes must be allocated and deallocated to construct $d_{copy}$ from $d'_{copy}$ using match $M$. The minimum cost of evaluating the attribute tree $d'_{copy}$ is $\Omega(|d'_{copy} - \text{REUSED}_M| + |\text{REUSEDAFF}_M|)$. Summing the cost of matching and evaluation using a match $M$ gives a lower bound of $\Omega(|d'_{copy} - \text{REUSED}_M| + |d_{copy} - \text{REUSED}_M| + |\text{REUSEDAFF}_M|)$. There must exist a match $M = M'$ for $d_{copy}$ and $d'_{copy}$ that minimizes $|d'_{copy} - \text{REUSED}_M| + |d_{copy} - \text{REUSED}_M| + |\text{REUSEDAFF}_M|$. The lower bound for evaluation of a module instance is therefore $\Omega(|d'_{copy} - \text{REUSED}_{M'}| + |d_{copy} - \text{REUSED}_{M'}| + |\text{REUSEDAFF}_{M'}|)$.

This lower bound indicates that a matching algorithm that is only concerned with discovering a match $M$ that maximizes the size of $\text{REUSED}_M$ will be unable to minimize the cost of evaluating a module instance since these algorithms do not attempt to minimize the size of $\text{REUSEDAFF}_M$. An optimal matching algorithm must somehow take attribute context into account during the selection of subtrees from $d_{copy}$ for use in $d'_{copy}$ in order to limit the size of $\text{REUSEDAFF}_M$. Unfortunately, this implies that an optimal evaluator must perform matching and change propagation in a complex interleaved fashion in order to make attribute context available to the matching algorithm. Appendix B presents an example demonstrating the importance of taking attribute context into consideration during matching.

## 6.6 Summary

This chapter motivated the matching-based approach to incremental evaluation of modules of hierarchical attribute grammars and presented and critiqued Teitelbaum and Chapman's incremental evaluator for Higher Order Attribute Grammars. Finally, a high-level description of an incremental evaluator for modules of a hierarchical attribute grammar was presented.

# Chapter 7

# Incremental Evaluation: Matching Algorithms

This chapter presents a series of four matching algorithms for use with the function *ModuleEvaluate* as described in the previous chapter. The new algorithms are based on a pair of heuristics that are intended to improve the utilization of previously computed attribute values.

## 7.1 Matching Heuristics

Since an evaluator for modules of hierarchical specifications that is based on performing matching prior to change propagation will be unable to update modules optimally, we have adopted two heuristics to guide the matching process: the *retention heuristic* and the *maximal reuse heuristic*. The goal of both heuristics is to increase the utilization of previously computed attribute values.

> The *Retention Heuristic* requires that complete subtrees of the old attribute tree $d_{copy}$ that are not used in constructing the new tree $d'_{copy}$ be retained for use by later matching operations. The matching operation must, therefore, be able to construct the new attribute tree $d'_{copy}$ from subtrees taken from a forest of retained attribute trees which will be referred to as $d^*_{copy}$.
>
> The *Maximal Reuse Heuristic* requires reusing maximal subtrees from the old attribute forest $d^*_{copy}$ to construct the new attribute tree $d'_{copy}$.

## 7.2 The Topological Matching Algorithm

This section motivates and presents the *Topological Matching Algorithm (TM)*. The TM algorithm is guaranteed to build the new attribute tree $d'_{copy}$ from maximal components of the old attribute tree $d_{copy}$, and, therefore, satisfies the Maximal Reuse Heuristic. The algorithm, however, makes no attempt to satisfy the Retention Heuristic. The next section presents the Retaining Topological Matching algorithm which extends the TM algorithm to satisfy the Retention Heuristic.

### 7.2.1 The Topological Approach to Matching

Consider a matching algorithm that constructs the new attribute tree $d'_{copy}$ in a top-down manner from complete subtrees of $d_{copy}$ and from new attribute tree nodes, such that during matching, $d'_{copy}$ will consist of a crown of attribute tree nodes connected to a set of "placeholders," where each placeholder is a tuple containing an attribute tree node and an integer $i$, where $1 \leq i \leq k$ and $k$ is the arity of the operator labeling the attribute tree node. A placeholder represents a location into which a new attribute tree node or a complete attribute subtree can be grafted. We will say that a placeholder $\langle n, i \rangle$ is "filled" when a new attribute tree node or an old complete attribute subtree is grafted into the $i$th child slot of attribute tree node $n$ to extend the crown of $d'_{copy}$.

Now consider Figure 7.1 showing the dags $d$ and $d'$ and the attribute tree $d_{copy}$ and Figure 7.2 showing two versions of the attribute tree $d'_{copy}$ that are the result of selecting the subtree $X_{attr}$ to represent $X$ and the result of selecting the subtree $Y_{attr}$ to represent $Y$, respectively. Filled circles in the two versions of $d'_{copy}$ represent new attribute tree nodes. Selecting the subtree $X_{attr}$ to represent $X$ means that the subtree $Y_{attr}$ will be unavailable to represent $Y$ in $d'_{copy}$. Since attribute subtree $Y_{attr}$ includes subtree $X_{attr}$, maximal use

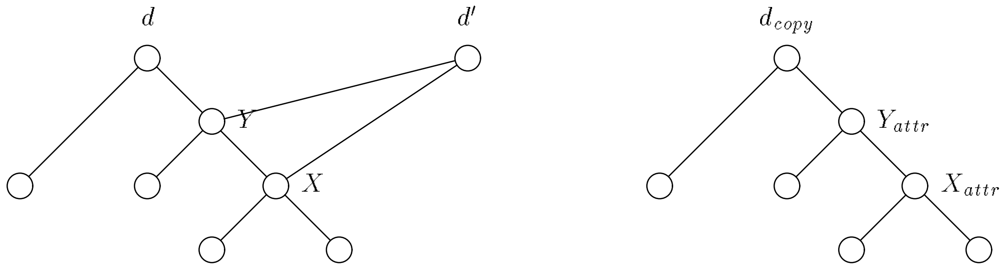


**Figure 7.1** Dags $d$ and $d'$, and an Attribute Tree $d_{copy}$

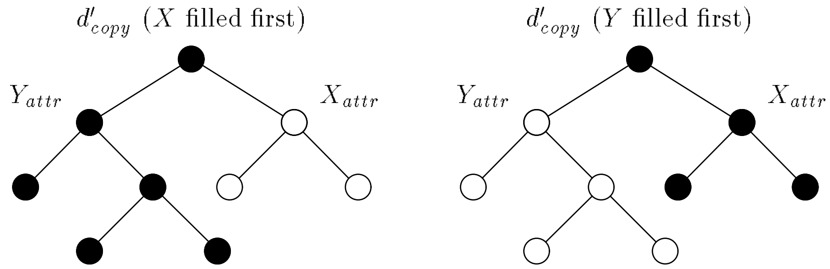


**Figure 7.2** Two New Attribute Trees

of subtrees from $d_{copy}$ requires selecting the subtree $Y_{attr}$ for $Y$ instead. In this example, selecting subtrees $X_{attr}$ and $Y_{attr}$ in the wrong order makes only a small difference in the sizes of the unattributed crowns of the two attribute trees, however, if $Y_{attr}$ had been much larger than $X_{attr}$, then the size difference would have been dramatic.

It can be proven that $d'_{copy}$ is created from maximal subtrees of $d_{copy}$ if subtrees from $d_{copy}$ are selected to fill placeholders at nodes in $d \cup d'$ during a topological traversal of the dag $d \cup d'$. A topological ordering of the dag $d \cup d'$, consisting of a set $V$ of vertices and a set $E$ of edges, is a linear ordering containing all of the vertices in $V$ such that if $e = (u, v)$ is an edge in $E$, then $u$ appears before $v$ in the ordering. In the example of Figure 7.1, the subtree $Y_{attr}$ should be chosen before the subtree $X_{attr}$, since $Y$ "topologically precedes" $X$ in $d \cup d'$, i.e., there exists a valid linear order of $d \cup d'$ in which $Y$ precedes $X$. A pseudo-algorithm based on this topological approach is presented in Figure 7.3.

---

Create the special placeholder $\top$ as the only node in the new crown of $d'_{copy}$.

Repeat the following operation for each dag node $n$ in topological order with respect to $d \cup d'$:

> Repeat for each placeholder $h$ in the crown $d'_{copy}$ that can be filled by an attribute subtree from $d_{copy}$ that represents the subterm rooted by node $n$:
>
> > (*) If there is a complete subtree in $d_{copy}$ that represents the subterm rooted by $n$, then graft that subtree into $h$. Otherwise, graft into $h$ a single new attribute tree node representing the root of $n$ and having placeholders for each of its children.

Remove $\top$ and deallocate all attribute nodes in $d_{copy}$ that were not used in building $d'_{copy}$.

---

**Figure 7.3** The Topological Pseudo-Algorithm for Matching

Lemma 7.1, Lemma 7.2 and Theorem 7.1 prove that matching based on the pseudo-algorithm in Figure 7.3 constructs the new attribute tree $d'_{copy}$ using maximal subtrees from $d_{copy}$. These proofs are based on a pair of functions that map syntax nodes in $d \cup d'$ to integers:

- $AvailD_{copy}$ – maps each syntax node in $d \cup d'$ into the number of attribute tree nodes in $d_{copy}$ representing that node, and
- $AvailD'_{copy}$ – maps each syntax node in $d \cup d'$ into the number of attribute tree nodes in $d'_{copy}$ that will represent that node on completion of matching.

**Lemma 7.1** The topological pseudo-algorithm shown in Figure 7.3 uses $\min(AvailD_{copy}(w), AvailD'_{copy}(w))$ attribute tree nodes from $d_{copy}$ to represent each syntax node $w$ in $d \cap d'$.

**Proof** Choosing attribute subtrees in topological order guarantees that by the time that step (*) of the pseudo-algorithm is executed to select attribute subtrees for syntax node $w$, attribute subtrees representing syntax terms containing $w$ might have already been selected for use in $d'_{copy}$, but no attribute subtrees representing proper subterms of the subterm rooted by $w$ could have been selected. Therefore, at the time that step (*) of the pseudo-algorithm is executed to select attribute subtrees for syntax node $w$, $k$, with $0 \leq k \leq AvailD_{copy}(w)$, of the attribute tree nodes representing syntax node $w$ will have already been moved into $d'_{copy}$, and all of the remaining $k - AvailD_{copy}(w)$ attribute tree nodes will be available for use in filling placeholders at node $w$ in $d'_{copy}$.

If $AvailD_{copy}(w) > AvailD'_{copy}(w)$ then step (*) will use $AvailD'_{copy}(w)$ of the attribute tree nodes in $d_{copy}$ representing node $w$ to fill placeholders representing $w$ in $d'_{copy}$. If $AvailD_{copy}(w) \leq AvailD'_{copy}(w)$ then step (*) will use all $AvailD_{copy}(w)$ attribute tree nodes representing $w$ in $d_{copy}$ to represent $w$ in $d'_{copy}$. Therefore, the pseudo-algorithm reuses $\min(AvailD_{copy}(w), AvailD'_{copy}(w))$ attribute tree nodes to represent $w$ in $d'_{copy}$. □

**Lemma 7.2** Each attribute tree node in $d_{copy}$ that is selected and moved into $d'_{copy}$ is a member of a complete attribute subtree that was moved in its entirety from $d_{copy}$ to $d'_{copy}$.

**Proof** Only complete attribute subtrees are moved from $d_{copy}$ to $d'_{copy}$ by step (*) of the pseudo-algorithm. □

**Theorem 7.1** Selection of subtrees from $d_{copy}$ in topological order with respect to $d \cup d'$, according to the pseudo-algorithm shown in Figure 7.3, uses maximal subtrees from $d_{copy}$ to construct $d'_{copy}$.

**Proof** The maximal use of subtrees from $d_{copy}$ for the construction of $d'_{copy}$ follows directly from the fact that all possible attribute tree nodes in $d_{copy}$ representing each of the nodes in $d \cap d'$ are reused in constructing $d'_{copy}$ according to Lemma 7.1. Lemma 7.2 guarantees that the reused set of attribute tree nodes forms a set of complete attribute subtrees. □

### 7.2.2 The TM Algorithm

TM associates a set PROVIDED and a set NEEDED with each node in $d \cup d'$ and then inserts subtrees and placeholders into the two sets during a topological traversal of $d \cup d'$, in the process, constructing the new attribute tree $d'_{copy}$ from maximal subtrees of $d_{copy}$. The saved state for the TM algorithm consists of the dag $d$ and the attribute tree $d_{copy}$.

The PROVIDED set associated with a node $n$ in $d \cup d'$ will contain the set of all subtrees in $d_{copy}$ representing the subterm rooted by $n$ which were not already selected to fill placeholders for ancestors of node $n$. The NEEDED set associated with a node $n$ will contain the set of placeholders for the subterm rooted by node $n$ in an ever-growing crown of $d'_{copy}$. The subtrees in the PROVIDED set of node $n$ are available to fill placeholders in the NEEDED set of node $n$.

Initially, the PROVIDED and NEEDED sets for every node in $d \cup d'$ will be empty. Elements are inserted into the PROVIDED and NEEDED sets of root($d$) and root($d'$) to represent the initial availability of the entire old attribute tree and to represent the need to construct the entire new attribute tree. The element $d_{copy}$ is inserted into the PROVIDED set for root($d$), and the element $\top$ is inserted into the NEEDED set for root($d'$), where $\top$ represents a placeholder that must be filled by an attribute tree representing all of $d'$. Note that root($d$) and root($d'$) may be the same node in which case $d$ and $d'$ are the same terms and $d_{copy}$ can be used for $d'_{copy}$. It is also possible for $d$ to be a subterm of $d'$, or $d'$ to be a subterm of $d$.

The nodes in $d \cup d'$ are topologically sorted and then processed in order, where processing nodes entails selecting subtrees in PROVIDED to fill placeholders in NEEDED for those nodes[1]. By the time a node $n$ in $d \cup d'$ is selected for processing, all of its ancestors in $d \cup d'$ will have been processed and will have made their contributions to the PROVIDED and NEEDED sets for node $n$. If the PROVIDED set for a node $n$ in $d \cup d'$ is larger than its NEEDED set, then each of the subtrees of the unused subtrees in the PROVIDED set for node $n$ will be inserted into the PROVIDED sets of the children of node $n$. If the NEEDED set for a node $n$ in $d \cup d'$ is smaller than its PROVIDED set, then, for each unfilled placeholder representing node $n$, a placeholder will be inserted into the NEEDED set of each of the children of node $n$. After all of the nodes in $d \cup d'$ have been processed, the node $\top$ will be removed from $d'_{copy}$ and deallocated, and all of the nodes in $d_{copy}$ that were not used in constructing $d'_{copy}$ will be deallocated.

```
Function Match(d', State)
  d = State.d
  d_copy = State.d_copy

  -- Create node ⊤ and insert it into d'_copy
  d'_copy = Create⊤()

  PROVIDED(root(d)) = {d_copy} -- the entire subtree d_copy is available
  NEEDED(root(d')) = {⊤} -- a copy of a subtree representing d'_copy is needed

  -- Walk d' ∪ d to compute in-degrees for nodes
  InitializeInDegrees(d, d')

  R-QUEUE = {}
  if (in-degree(d) == 0) insert root(d) into R-QUEUE
  if (in-degree(d') == 0) insert root(d') into R-QUEUE

  BuildTree(R-QUEUE, d'_copy)

  Reinitialize(d')
  Remove⊤(d'_copy)

  State.d = d'
  State.d_copy = d'_copy
  return d'_copy
end
```

**Figure 7.4** Match – TM Algorithm

Function *Match*, shown in Figure 7.4, when invoked with the new syntax term $d'$ and the saved state *State*, extracts the values of $d$ and $d_{copy}$ from *State*, invokes the function *Create*$\top$ to create the special node $\top$ acting as the initial crown of $d'_{copy}$, initializes the PROVIDED set for the root of $d$ and the NEEDED set for the root of $d'$, invokes the function *InitializeInDegrees* to compute in-degrees for each node in $d \cup d'$, and then builds the initial queue R-QUEUE of nodes without predecessors. *Match* then invokes the procedure *BuildTree*, shown in Figure 7.5, to compute the values of each of the PROVIDED and NEEDED sets for nodes in $d \cup d'$ and to construct the new attribute tree $d'_{copy}$ from components of $d_{copy}$ during a topological traversal of the nodes in $d \cup d'$ using the computed in-degrees for each node. Finally, *Match* invokes the function *Reinitialize* to reset the values of the PROVIDED and NEEDED sets of nodes in $d'$ to be empty sets, invokes the function *Remove*$\top$ to remove the node $\top$ from $d'_{copy}$, and then stores the values $d'$ and $d'_{copy}$ back into *State*.

Procedure *Buildtree*, shown in Figure 7.5, selects nodes in $d \cup d'$ in topological order and invokes the function *Fill* to select subtrees from the set PROVIDED and to create new attribute tree nodes to fill each

[1]The term "processed" will be used throughout this chapter to refer to exactly this operation.

```
Procedure BuildTree(R-QUEUE, d'_copy)
  while R-QUEUE not empty
    dagnode = q-remove(R-QUEUE)

    needs = NEEDED(dagnode)
    provs = PROVIDED(dagnode)
    ⟨remprovs,newtreenodes⟩ = Fill(dagnode,needs,provs)

    – – make each child of each node that is still available be available
    if (|remprovs| > 0)
      for each child i of dagnode
        for each attrnode in remprovs
          union attrnode[i] into PROVIDED(dagnode[i])

    – – deallocate each unused node in PROVIDED
    for each attrnode in remprovs
      DeallocNode(attrnode)

    – – each child of a newtreenode is needed
    if (|newtreenodes| > 0)
      for each child i of dagnode
        for each attrnode in newtreenodes
          union ⟨attrnode,i⟩ into NEEDED(dagnode[i])

    – – basic queue management for topological sorting
    for each child i of dagnode
      IN-DEGREE(dagnode[i]) = IN-DEGREE(dagnode[i]) - 1
      if (IN-DEGREE(dagnode[i]) == 0)
        q-insert dagnode[i] into R-QUEUE
end
```

**Figure 7.5** BuildTree – TM Algorithm

placeholder in NEEDED for each node in $d \cup d'$. When *Fill* is invoked at a node, say *dagnode*, it returns two sets: *remprovs* and *newtreenodes*. The set *remprovs* is the set of nodes that are roots of subtrees in $d_{copy}$ that represent the subterm rooted by *dagnode* but which were not needed in constructing $d'_{copy}$. The set *newtreenodes* is the set of new attribute tree nodes that were created to fill a placeholder for the subterm rooted by *dagnode* for which no subtree in $d_{copy}$ was available. After invoking *Fill*, *BuildTree* adds the children of each node in *remprovs* to the PROVIDED sets of the children of *dagnode*, deletes each of the nodes in *remprovs*, and adds placeholders for each of the children of an attribute tree node in *newtreenodes* to the NEEDED sets of the children of *dagnode*.

The function *Fill*, shown in Figure 7.6, is invoked with *dagnode* and the NEEDED and PROVIDED sets for *dagnode*, and fills each of the placeholders in NEEDED in two steps. First, it uses subtrees from PROVIDED to fill as many placeholders in NEEDED as possible. Then, if there are any placeholders remaining, it creates new attribute nodes to fill each of those placeholders. The function *Choose* selects and removes an element from a set. Subtrees and new nodes are attached to the new attribute tree by invoking the function *Graft*.

### 7.2.3 A Topological Matching Example

The example shown in Figure 7.7 was first presented in Chapter 6 as a counter-example to Teitelbaum and Chapman's matching algorithm. Figure 7.8 depicts the values of the PROVIDED and NEEDED sets for each node in $d \cup d'$ as computed according to the topological matching algorithm just outlined, and Figure 7.9 presents the sequence of operations and the attribute tree resulting from applying the TM algorithm. As before, filled nodes represent new attribute tree nodes. The operations are chosen from the instruction set:

- Install $R/S$ into ⟨X/Y,Z⟩: Install the subtree whose root is labeled $R/S$ of $d'_{copy}$ into the $Z$th child of the node labeled $X/Y$.
- $X/Y \leftarrow$ NEW $S$: Create a new attribute tree node to represent dagnode $S$. The new attribute tree node is labeled $X/Y$.

```
Function Fill(dagnode,needs,provides)
  while (|needs| > 0 and |provides| > 0)
    need = Choose(needs)
    child = Choose(provides)
    Graft(child,need.parent,need.slot)

  newtreenodes = {}
  while (|needs| > 0)
    need = Choose(needs)
    child = CreateAttrNode(dagnode)
    Graft(child,need.parent,need.slot)
    insert child into newtreenodes

  return ⟨provides,newtreenodes⟩
end
```

**Figure 7.6** Fill – TM Algorithm

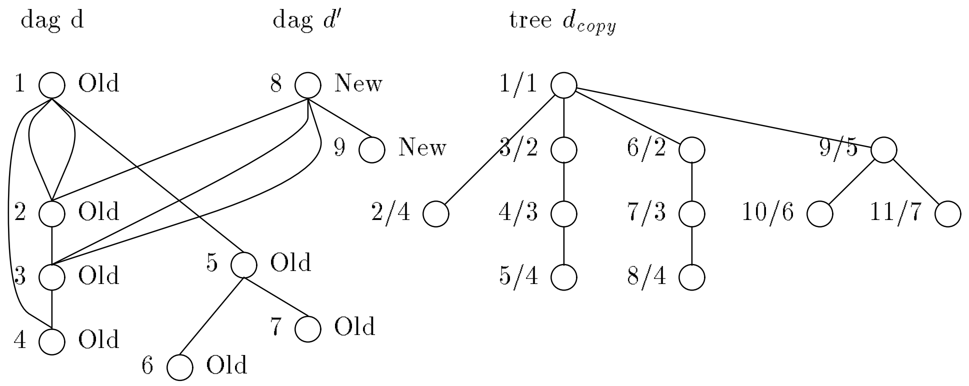


**Figure 7.7** The dags $d$ and $d'$, and the tree $d_{copy}$

### 7.2.4 Proofs for the TM Algorithm

**Theorem 7.2** The TM algorithm, when applied to a dag $d'$, and a saved state *State* containing the dag $d$ and the attribute tree $d_{copy}$, constructs the complete attribute tree $d'_{copy}$ using maximal components from $d_{copy}$.

**Proof** The TM algorithm implements the pseudo-algorithm proved correct in Theorem 7.1. □

**Theorem 7.3** The TM algorithm performs matching in $O(|d \cap d'| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ time.

**Proof** The total cost of matching is dominated by the sum of the costs of function *InitializeInDegrees* and procedure *BuildTree*. Clearly, *InitializeInDegrees* requires $O(|d \cup d'|)$ time.

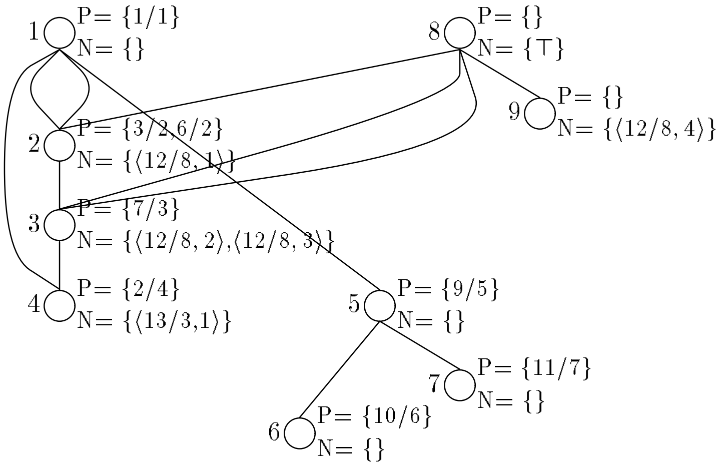


**Figure 7.8** $d \cup d'$ Annotated with PROVIDED and NEEDED Sets

1) 12/8 ← NEW 8
2) Install 12/8 into ⊤
3) Install 3/2 into ⟨12/8,1⟩
4) Install 7/3 into ⟨12/8,2⟩
5) 13/3 ← NEW 3
6) Install 13/3 into ⟨12/8,3⟩
7) Install 2/4 into ⟨13/3,1⟩
8) 14/9 ← NEW 9
9) Install 14/9 into ⟨12/8,4⟩
10) Remove ⊤

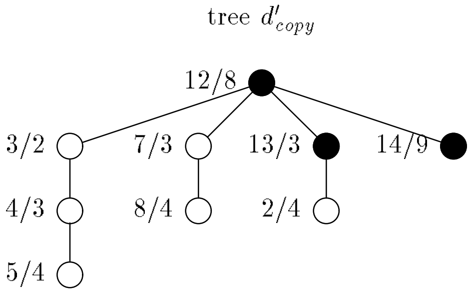


**Figure 7.9** Operations Performed During Topological Matching and the Resulting Attribute Tree

*BuildTree* is responsible for: (1) topologically sorting $d \cup d'$, (2) constructing the PROVIDED and NEEDED sets for each node in $d \cup d'$, (3) invoking *Fill* with each node in $d \cup d'$ to construct $d'_{copy}$, and (4) deallocating all of the unused attribute tree nodes in $d_{copy}$. The cost of topological sorting is $O(|d \cup d'|)$.

The NEEDED set for each syntax node $n$ in $d \cup d'$ other than $root(d')$ contains a placeholder $p$ for each new attribute tree node in $d'_{copy} - d_{copy}$ that represents a parent of syntax tree node $n$. The single element $\top$ is included in the NEEDED set for $root(d')$. The PROVIDED set for each syntax node $n$ in $d \cup d'$ other than $root(d)$ contains an attribute subtree for each attribute tree node in $d_{copy} - d'_{copy}$ that represents a parent of $n$. The single element $d_{copy}$ is included in the PROVIDED set for $root(d)$. Hence, $\sum |\text{NEEDED}| = O(|d'_{copy} - d_{copy}|)$, and $\sum |\text{PROVIDED}| = O(|d_{copy} - d'_{copy}|)$. Each invocation of *Fill* at a node $n$ in $d \cup d'$ takes $O(|\text{NEEDED}(n)|)$ time, and, therefore, the total cost of all of the evaluations of *Fill* is $O(\sum |\text{NEEDED}|)$ which is $O(|d'_{copy} - d_{copy}|)$. Finally, deallocating attribute tree nodes left unused after *Fill* is invoked at a node $n$ in $d \cup d'$ takes $O(|\text{PROVIDED}(n)|)$ time, and, therefore, the total cost of deallocation is $O(|d_{copy} - d'_{copy}|)$.

Summing together the running times of each of these components gives a running time of $O(|d \cup d'| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$. Since $|d \cup d'| = |d - d'| + |d \cap d'| + |d' - d|$, and since $|d' - d|$ is dominated by $|d'_{copy} - d_{copy}|$ and $|d - d'|$ is dominated by $|d_{copy} - d'_{copy}|$, then the cost of the TM algorithm is $O(|d \cap d'| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$. $\square$

The TM algorithm explicitly traverses the entire dag $d \cup d'$ in order to construct the new attribute tree $d'_{copy}$, in the process performing $O(|d \cap d'|)$ more work than allowed by Teitelbaum and Chapman's suggested optimal time bound. It may seem that a matching algorithm that traverses all of $d \cup d'$ must be too slow to use as the basis for an efficient incremental evaluator. In the hierarchical context, however, this is not the case, since taking the time to select the appropriate attribute subtrees for use in the new attribute tree for a module instance, may drastically reduce the cost of evaluating module instances recursively invoked by that module instance. In addition, the cost of the TM algorithm never exceeds the cost of batch evaluation for any module instance, at least in an amortized sense. The running time of the TM algorithm can be written as $O(|d_{copy}| + |d'_{copy}|)$. By charging the $|d_{copy}|$ term to the previous matching operation for each module instance, the amortized running time becomes $O(|d'_{copy}|)$ which is exactly the cost of batch evaluation of the module instance with input $d'$.

## 7.3 The Retaining Topological Matching Algorithm

This section extends the TM algorithm to form the *Retaining Topological Matching algorithm (RTM)* which satisfies both the Maximal Reuse Heuristic and the Retention Heuristic. The TM algorithm selects attribute subtrees from a single complete attribute tree $d_{copy}$ for use in constructing the new attribute tree $d'_{copy}$ and then deallocates all of the unused attribute tree nodes in $d_{copy}$. In contrast, the RTM algorithm will retain the complete attribute subtrees in $d_{copy}$ that remain after the new attribute tree $d'_{copy}$ has been built from subtrees of $d_{copy}$ and will then use those retained subtrees for use in later matching operations. The notation $d^*_{copy}$ will be used to refer to the forest of retained attribute trees that are available at the start of matching and the notation $d'^*_{copy}$ will be used to refer to the forest of retained attribute tree that remain after matching has completed.

The RTM algorithm is actually a family of algorithms, which will be referred to as RTM-0, RTM-k, and RTM-$\infty$, where the meaning of the suffix on the name of each algorithm will be described below. The name "RTM" will refer to the RTM-$\infty$ algorithm.

In Theorem 7.1, we proved that, by selecting subtrees from $d_{copy}$ in topological order with respect to $d \cup d'$, it is guaranteed that maximal subtrees of $d_{copy}$ will be used in constructing the new attribute tree $d'_{copy}$. The proof extends directly to prove that the new attribute tree $d'_{copy}$ is constructed from maximal subtrees of the attribute trees in $d^*_{copy}$ by selecting subtrees in topological order with respect to the dag $d^* \cup d'$, where $d^*$ is the dag which is the union of all dags ever evaluated by each module instance. Actually, it is only necessary to consider nodes in $d^*$ that are represented by attribute tree nodes in $d^*_{copy}$, since only those nodes can actually contribute subtrees of $d^*_{copy}$ to the construction of the attribute tree $d'_{copy}$. Henceforth, $d^*$ is considered to contain only those nodes that are represented by attribute tree nodes in $d^*_{copy}$, and $d'^*$ is considered to contain only those nodes in $d^* \cup d'$ that are still represented by attribute tree nodes in $d'^*_{copy}$ on completion of matching.

### 7.3.1 The RTM Approach

The RTM algorithms will be invoked with the new dag $d'$ and the saved state *State* containing a list of the roots of all of the retained attribute trees in $d^*_{copy}$. The RTM algorithm is essentially the TM algorithm modified to build the new attribute tree given a multi-rooted dag $d^*$ represented by a set of attribute trees $d^*_{copy}$ instead of a singly-rooted dag $d$ represented by a single attribute tree $d_{copy}$. The key modifications include the following:

- The RTM algorithm will explicitly insert each of the attribute trees in $d^*_{copy}$ into the PROVIDED sets for the appropriate syntax nodes in $d^*$ to make sure that all attribute trees in $d^*_{copy}$ are available for use in $d'_{copy}$.
- In order for the RTM algorithm to be able to traverse all of $d^*$ in topological order, the function *InitializeInDegrees* will be invoked with the list of roots of the terms in $d^* \cup d'$, and the function *Match* will explicitly insert each of the attribute tree roots having an *in-degree* of 0 into R-QUEUE.
- Each time function *Fill* is invoked by the RTM algorithm to move an attribute subtree $T$ from $d^*_{copy}$ to $d'_{copy}$, it will explicitly invoke the function *DeallocAncestors* to deallocate all of the ancestors of the root of $T$ in $d^*_{copy}$, instead of blindly deallocating all of the unused attribute tree nodes.
- The functions *Prune*, *Graft* and *DeallocNode* will be modified to explicitly maintain the lists of roots of $d^*_{copy}$.

Although the matching technique resulting from these modifications will correctly construct the new attribute tree $d'_{copy}$ using maximal subtrees from the database of retained attribute trees, the time required to perform this matching operation can easily exceed the time required to perform batch evaluation of the module instance. In fact, the running time for the RTM algorithm can be shown to be $O(|d^* \cap d'| + |d'_{copy} - d^*_{copy}| + |d^*_{copy} - d'_{copy}|)$ which can be rewritten as $O(|d'| + |d'_{copy}| + |d^*_{copy} - d'_{copy}|)$ since $|d^* \cap d'|$ is $O(|d'|)$, and $|d'_{copy} - d^*_{copy}|$ is $O(|d'_{copy}|)$. The size of $d^*_{copy} - d'_{copy}$ can cause the cost of matching to exceed the cost of batch evaluation as we now demonstrate.

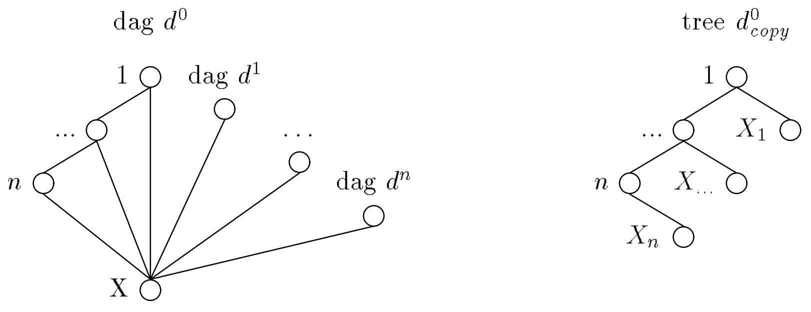


**Figure 7.10** An Example

Consider the sequence of dags $d^0$, $d^1$, $\ldots$, $d^n$ of length $n+1$ as depicted in Figure 7.10, where the initial dag $d^0$ consists of a backbone of $n$ nodes each sharing a child node $X$, and the remaining dags in the sequence each consist of a single new node with node $X$ as its only child. If evaluation of this sequence of dags is performed non-incrementally, then the entire sequence of $n$ module evaluations can be performed in $O(n)$ time since evaluation of $d^0$ requires $O(n)$ time and evaluation of each of the other dags requires only $O(1)$ time.

Now consider the application of the RTM matching algorithm to this sequence of dags. Figure 7.11 shows the dag $d^*$ and the attribute tree forest $d^*_{copy}$ that remain after the first and the last matching operations if attribute tree node $X_1$ is repeatedly chosen to represent $X$ in the $n$ input dags $d^1, \ldots, d^n$. The RTM algorithm will require $O(n)$ time to perform matching for the $i$th dag in the sequence, since the algorithm will traverse the subtree of size $O(n)$ that is retained in the attribute tree forest. Since matching for each

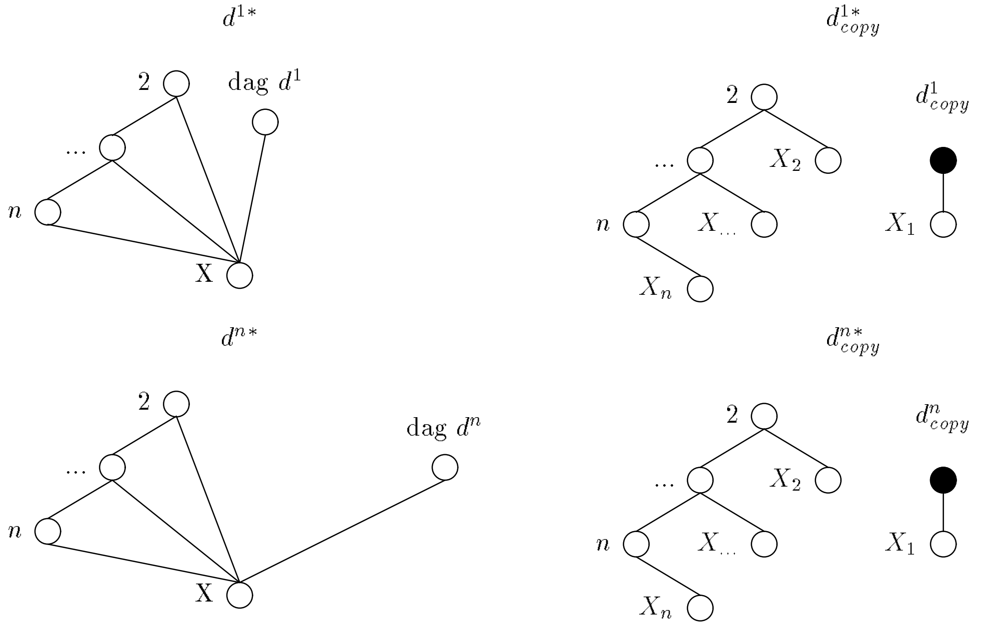


**Figure 7.11** Retained Dags and Attribute Trees if $X_1$ is Repeatedly Selected for $X$

of the $n$ dags in the sequence takes $O(n)$ time, the total time required for incremental evaluation of the sequence using the RTM algorithm is $O(n^2)$ which obviously exceeds the $O(n)$ cost of batch evaluation for the entire sequence of dags.

### 7.3.2 The RTM-k Approach

We now describe an approach to retention of unused attribute subtrees that guarantees that the amortized cost of matching will never exceed the cost of batch evaluation. Very few modifications to the approach just described are actually required. Whereas the RTM (RTM-$\infty$) approach permits a retained attribute tree node to be considered an unlimited number of times for use in constructing the new attribute tree, the RTM-k approach only allows each retained attribute node to be considered $k$ times before the node is forced to be deallocated. The RTM-0 algorithm is just the TM algorithm.

The goal of the RTM-k approach is to limit the amount of effort the matching algorithm expends at any attribute tree node that is retained by the RTM-k algorithm but which would not have been retained by the TM algorithm. In fact, the goal of the RTM-k algorithm is to limit the additional cost of processing these retained attribute tree nodes to a constant factor which can be charged to the matching operation responsible for first retaining the additional attribute tree nodes. Therefore, the TM and RTM-k algorithms will have asymptotically the same cost. To achieve this goal, a counter will be maintained on each attribute tree node. The counter on attribute tree node $n$ will be decremented each time the matching algorithm considers node $n$ for use in constructing $d'_{copy}$ but then chooses not to use the subtree rooted at $n$ in the new attribute tree, i.e., when $n$ appears in the PROVIDED set for the dag node represented by $n$, but is not used in constructing the new attribute tree.

Attribute tree nodes with counter values of 0 and $k$ will be referred to as *empty* and *full* nodes, respectively. Any node with a counter value less than $k$ will be referred to as being *depleted*, and the act of setting the counter for a node to the value $k$ will be referred to as *replenishing.* Counter manipulation will be performed by the following protocol:

- Whenever a new attribute node is created by the matching algorithm, its counter will be set to 0. Once *BuildTree* has finished selecting subtrees for $d'_{copy}$, the procedure *Replenish* will be invoked to replenish each depleted node in the new attribute tree $d'_{copy}$, including newly allocated nodes. The depleted nodes form a crown of $d'_{copy}$, hence, replenishing can be performed during a depth-first traversal of the depleted nodes using the values of the counter to identify those nodes occurring in the crown of depleted nodes.

- Whenever procedure *BuildTree* processes a dag node whose PROVIDED set is larger than its NEEDED set, the counter on the root of each subtree in PROVIDED that is still available after filling all of the outstanding needs in NEEDED is decremented. Whenever the counter on an attribute tree node $n$ reaches 0, each edge from $n$ to a child of $n$ is removed by invoking the function *Prune*, and $n$ is deallocated.

### 7.3.3 The RTM-k Algorithm

We now describe the functions implementing the RTM-k algorithm. The majority of the changes from the function definitions of the TM algorithm support the processing of the forest of attribute trees $d^*_{copy}$ and the multi-rooted dag $d^*$.

```
Function Match(d', State)
  attrroots = State.attrroots

  dagroots = {root(d')}
  for each root in attrroots
    insert root.dagnode into dagroots

  d'_copy = Create⊤() -- Create node ⊤ and insert it into d'_copy

  -- Add each subtree in attrroots into the appropriate PROVIDED set
  for each root in attrroots
    insert root into PROVIDED(root.dagnode)
  NEEDED(root(d')) = {⊤} -- a copy of a subtree representing d'_copy is needed

  -- Walk d* ∪ d' to compute in-degrees for nodes
  InitializeInDegrees(dagroots)

  -- Build initial R-QUEUE
  for each root in dagroots
    if (IN-DEGREE(root) == 0) insert root into R-QUEUE

  BuildTree(R-QUEUE, d'_copy, attrroots)

  Reinitialize(dagroots)
  Replenish(root(d'_copy))
  Remove⊤(d'_copy)

  State.attrroots = attrroots
  return d'_copy
end
```

**Figure 7.12** Match – RTM-k Algorithm

Function *Match*, shown in Figure 7.12, retrieves the list of roots of attribute trees in $d^*_{copy}$, and then builds the list *dagroots* of the roots of the terms in $d^*$. *Match* then invokes the function *Create*$\top$ as before, explicitly inserts each of the roots of $d^*_{copy}$ into the appropriate PROVIDED sets, and initializes the NEEDED set for the root of $d'$. *Match* then invokes *InitializeInDegrees* with the roots of the terms in $d^*$ and inserts the roots of terms in $d^*$ having *in-degree* counters with value 0 into R-QUEUE for processing. As before, *Match* invokes *BuildTree* to build $d'_{copy}$, however, the list of roots of $d^*_{copy}$ must be passed as an additional argument so that the list may be updated by the functions *Prune*, *Graft* and *DeallocNode*. Finally, *Match* invokes the function *Reinitialize* to reset the PROVIDED and NEEDED sets of nodes in $d'^*$ to be empty, invokes the procedure *Replenish* to replenish the depleted nodes in the crown of $d'_{copy}$, invokes the function *Remove*$\top$ to remove the node $\top$ from $d'_{copy}$, and then stores the list of attribute tree roots back into the saved state.

```
Procedure Replenish(attrnode)
  if (attrnode.counter == k))
    return
  else
    attrnode.counter = k
    for each child i of attrnode
      Replenish(attrnode[i])
  fi
end
```

**Figure 7.13** Replenish – RTM-k Algorithm

Procedure *Replenish*, shown in Figure 7.13, when invoked with the root of the new attribute tree $d_{copy}$, performs a straightforward depth-first traversal over the depleted nodes in the attribute tree.

Procedure *BuildTree*, shown in Figure 7.14, is essentially identical to the function described for the TM algorithm except for the additional argument *attrroots* which is passed along to function *Fill*, and the decrementing of the *counter* field of each of the unused attribute tree nodes with the subsequent deallocation of attribute tree nodes which become empty as a result of having their counters decremented.

Function *Fill*, shown in Figure 7.15, chooses complete attribute subtrees and creates new attribute tree nodes to fill placeholders for *dagnode*. Each time a complete subtree is selected, *Fill* invokes *DeallocAncestors* to deallocate all of the attribute tree nodes which are ancestors of the root of the selected attribute subtree. *Fill* is, otherwise, identical to the function presented for the TM algorithm.

### 7.3.4 A Retaining Topological Matching Example

We now present a simple example of the application of the RTM-k algorithm to the sequence of three dags shown in Figure 7.16, consisting of the dags $d$ and $d'$ from the running example and a new dag $d''$.

On invocation of the RTM-k algorithm to construct the new attribute tree $d'_{copy}$, the *attrroots* list for the module instance will contain only the root of $d_{copy}$. On completion of matching, the attribute tree forest $d'^{*}_{copy}$ will contain the new attribute tree $d'_{copy}$ rooted by the node labeled 12/8 and one additional attribute tree rooted by the node labeled 9/5 as shown in Figure 7.17, and, therefore, the *attrroots* list will contain nodes 12/8 and 9/5. Nodes 12/8, 13/3 and 14/9 in $d'^{*}_{copy}$ are new nodes as indicated by the filled circles.

The invocation of the RTM-k algorithm to construct the new attribute tree $d''_{copy}$ from the trees in $d'^{*}_{copy}$ constructs the attribute tree forest $d''^{*}_{copy}$ shown in Figure 7.18 with an *attrroots* list containing nodes 15/10, 13/3 and 14/9. Nodes 15/10 and 16/11 in $d''^{*}_{copy}$ are new nodes as indicated by the filled circles. The complete subtrees rooted by 3/2, 7/3 and 9/5 have been reused in $d''^{*}_{copy}$. Node 12/8 has been deallocated since it was the root of an incomplete subtree after the attribute subtree rooted by node 3/2 was reused. Retention has made it possible to use the attribute tree whose root is labeled 9/5 in constructing $d''_{copy}$. Retention also causes the two subtrees rooted by 13/3 and 14/9 to be saved for later matching operations.

### 7.3.5 Proofs for the RTM-k Algorithm

**Theorem 7.4** The RTM-k algorithm, when applied to a dag $d'$, and a saved state containing a list of the roots of retained attribute trees in $d^{*}_{copy}$, constructs the complete attribute tree $d'_{copy}$ using maximal subtrees of $d^{*}_{copy}$.

**Proof** The proof of Theorem 7.1 placed no restrictions on the dag $d$ requiring $d$ to be a singly-rooted term. Therefore, it also proves that selecting subtrees from the attribute tree forest $d^{*}_{copy}$ for use in $d'_{copy}$ during a topological traversal of the dag $d^{*} \cup d'$ uses maximal subtrees from $d^{*}_{copy}$. Therefore, the RTM-k algorithm uses maximal subtrees of $d^{*}_{copy}$ to construct $d'_{copy}$. □

**Theorem 7.5** The RTM-k algorithm performs matching for a module instance with input term $d'$ in $O(|d \cap d'| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ amortized time, where $d_{copy}$ is the attribute tree

```
Procedure BuildTree(R-QUEUE, d'_copy, attrroots)
  while R-QUEUE not empty
    dagnode = q-remove(R-QUEUE)

    needs = NEEDED(dagnode)
    provs = PROVIDED(dagnode)
    ⟨remprovs,newtreenodes⟩ = Fill(dagnode,needs,provs,attrroots)

    – – make each child of each node that is still available be available
    if (|remprovs| > 0)
      for each child i of dagnode
        for each attrnode in remprovs
          union attrnode[i] into PROVIDED(dagnode[i])

    – – Decrement the counter on each provided subtree remaining in remprovs.
    – – Expire nodes whose counter reaches the value 0.
    for each prov in remprovs
      prov.counter = prov.counter - 1
      if (prov.counter == 0)
        for each child i of prov
          Prune(prov,i,prov[i],attrroots)
        DeallocNode(prov,attrroots)

    – – each child of a newtreenode is needed
    if (|newtreenodes| > 0)
      for each child i of dagnode
        for each attrnode in newtreenodes
          union ⟨attrnode,i⟩ into NEEDED(dagnode[i])

    – – basic queue management for topological sorting
    for each child i of dagnode
      IN-DEGREE(dagnode[i]) = IN-DEGREE(dagnode[i]) - 1
      if (IN-DEGREE(dagnode[i]) == 0)
        q-insert dagnode[i] into R-QUEUE
end
```

**Figure 7.14** BuildTree – RTM-k Algorithm

```
Function Fill(dagnode,needs,provides,attrroots)
  while (|needs| > 0 and |provides| > 0)
    need = Choose(needs)
    child = Choose(provides)

    – – Deallocate all of child's ancestors by following parent pointers
    – – in attribute tree nodes.
    DeallocAncestors(child,attrroots)
    Graft(child, need.parent, need.slot, attrroots)

  newtreenodes = {}
  while (|needs| > 0)
    need = Choose(needs)
    child = CreateAttrNode(dagnode) – with counter of 0
    Graft(child,need.parent,need.slot,attrroots)
    insert child into newtreenodes

  return ⟨provides,newtreenodes⟩
end
```

**Figure 7.15** Fill – RTM-k Algorithm

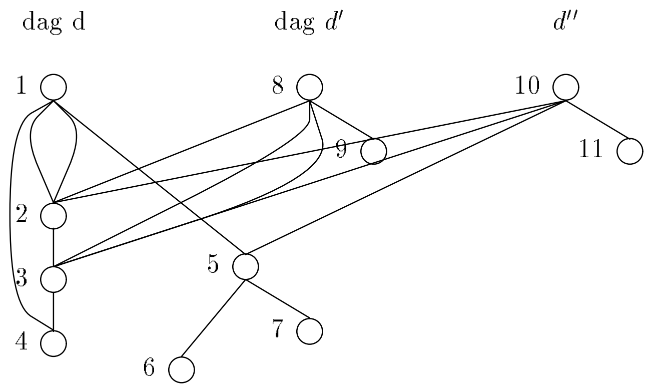


**Figure 7.16** $d$, $d'$, and $d''$

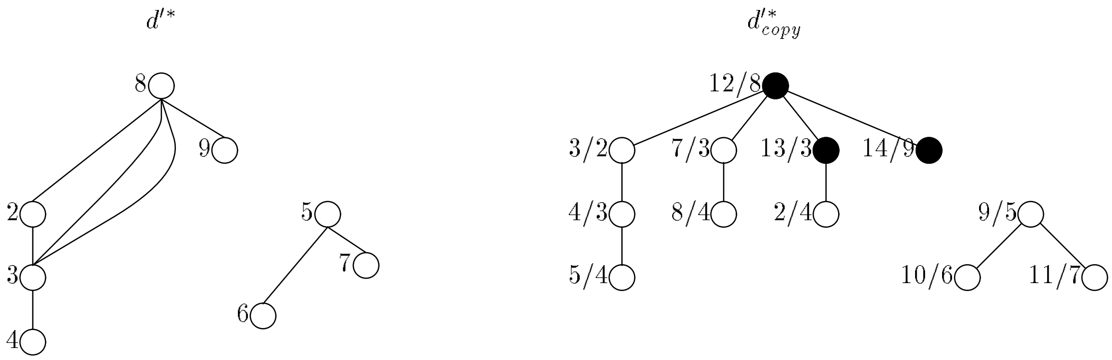


**Figure 7.17** $d'^*$ and $d'^*_{copy}$

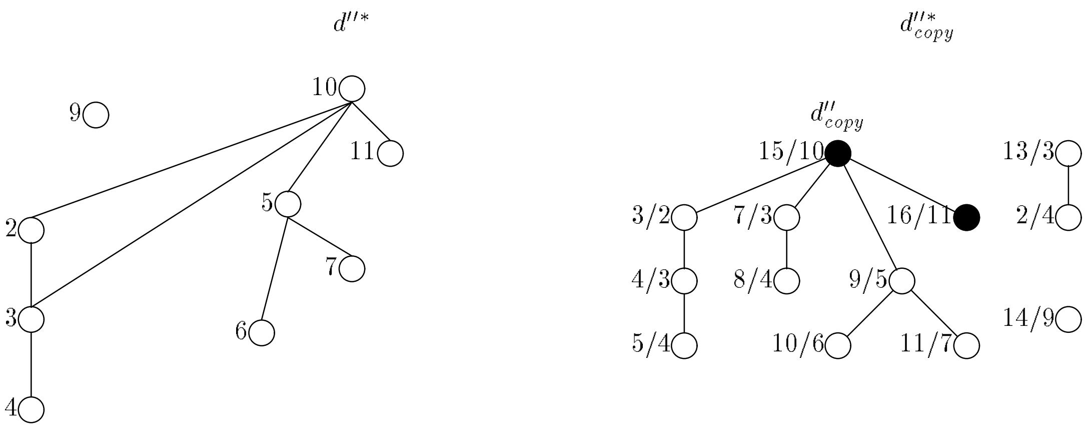


**Figure 7.18** $d''^*$ and $d''^*_{copy}$

that was built by the RTM-k algorithm the last time this module instance was invoked, and $d$ is the syntax term that was evaluated by that invocation of the RTM-k algorithm.

**Proof** As described above, the use of the counter $k$ is intended to guarantee that only a constant amount of work is expended by the RTM-k algorithm to retain and then later reuse or deallocate an attribute tree node which would have been deallocated by the TM algorithm. Between the time that an attribute tree node *attrnode* is first retained and then either moved into the new attribute tree $d'_{copy}$ by *Fill* and then replenished by *Replenish* or deallocated by *BuildTree*, *attrnode* can be inserted into the PROVIDED set for *attrnode.dagnode* at most $k$ times and can have its counter decremented at most $k$ times. Since only these $k$ operations, each taking only constant time, can be performed on *attrnode*, the total amount of effort that can be expended for *attrnode* is bounded by a constant, and can, therefore, be charged to the matching operation that was responsible for retaining that node. Since all of the work done by the RTM-k algorithm that was not done by the TM algorithm can be charged to earlier matching operations, the amortized cost of the RTM-k algorithm is shown to be the same as the cost of the TM algorithm. □

The advantage of the RTM-k algorithm over the TM algorithm lies in the fact that it maintains a larger database of consistent attribute subtrees to use in constructing the new attribute tree $d'_{copy}$. Complete subtrees that would have been deallocated by the TM algorithm, are retained for use in any of the $k$ subsequent invocations of the RTM-k algorithm. The retention of complete attribute subtrees can act as a buffer in cases where the input to a module instance exhibits a "pruning" and "grafting" behavior in which a subterm that was evaluated by a module instance is temporarily pruned from the input to that module instance, and then later grafted back into that module instance's input. This pruning and grafting behavior can occur directly as a result of the user's editing of the input to the root module instance. The behavior can also occur in the input to any module instance in a hierarchical system as an indirect result of the user's editing operations. If the pruned subterm is grafted back into the input term within $k$ invocations of the module instance, then the attribute subtrees representing the pruned subterm will still be available for use in the new attribute tree.

We now present versions of the TM and RTM-k algorithms that construct the new attribute tree for each module instance during a lazy topological traversal of a modeled region of $d \cup d'$ and $d^* \cup d'$, respectively. The goal of the lazy approach to matching is to reduce the total amount of work required to construct $d'_{copy}$. In addition, in the lazy version of the RTM-k algorithm, $k$ will continue to represent the number of times attribute nodes in retained subtrees can be considered by the matching algorithm, but nodes will only be considered if they represent modeled syntax nodes. This means that retained attribute tree nodes can be retained even longer.

## 7.4 The Lazy Topological Matching Algorithm

We new present a lazy version of the TM algorithm from Section 7.2. The goal of the lazy approach to matching is to avoid traversing all of the nodes in $d \cap d'$ while constructing $d'_{copy}$.

The *Lazy Topological Matching Algorithm (LTM)* will construct $d'_{copy}$ using maximal components from the retained attribute tree $d_{copy}$ during a traversal over a modeled region of $d \cup d'$ in time proportional to $O(|M_{P=N=\emptyset}(d \cap d')| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$, where $M_{P=N=\emptyset}(d \cap d')$ is the set of nodes in $d \cap d'$ that are in the model and whose NEEDED and PROVIDED sets both would have been determined to be empty by the TM algorithm.

### 7.4.1 The Model

Let TM-NEEDED be the set of nodes in $d \cup d'$ whose NEEDED sets would have been determined to be non-empty by the TM algorithm. Of course, the contents of TM-NEEDED is not known, a priori. During matching, the LTM algorithm will lazily construct a model that includes every node in TM-NEEDED and every ancestor in $d$ of nodes in TM-NEEDED. The nodes in TM-NEEDED are included in the model because it is at exactly those nodes that the function *Fill* will be invoked to extend the new attribute tree with new attribute tree nodes and old attribute subtrees to fill each of the placeholders in the NEEDED sets of those nodes. The ancestors of nodes in TM-NEEDED are included in the model in order to guarantee that nodes can be selected from the model in an order that is consistent with the topological order defined by the dag $d \cup d'$, thereby guaranteeing that whenever a node in the model is selected for processing, the PROVIDED and NEEDED sets at that node will have the same values as they would have if matching was being performed by the TM algorithm.

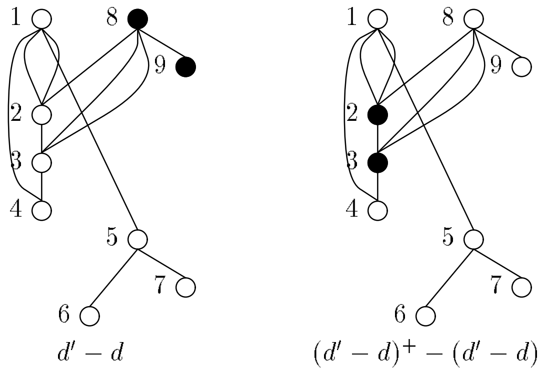


**Figure 7.19** Regions $d' - d$ and $(d' - d)^+ - (d' - d)$

Let $(d' - d)^+$ be the union of the nodes in $d' - d$ and direct descendents of nodes in $d' - d$. The set $(d' - d)^+ - (d' - d)$ will be referred to as the "fringe" of $d' - d$. Each node in the fringe of $d' - d$ is in $d$ and has a parent in $d' - d$. Figure 7.19 depicts $d' - d$ and $(d' - d)^+ - (d' - d)$ for the running example.

The LTM algorithm begins by inserting each node in $d' - d$ and the fringe of $d' - d$ into an initial model, since each node in this set of nodes can be guaranteed, a priori, to be in TM-NEEDED. As a side-effect of inserting each node $n$ in the fringe of $d' - d$ into the initial model, every ancestor of the node $n$ in $d$ will be added to the model. As matching progresses, whenever a placeholder is inserted into the NEEDED set for a node in $d$ that is not yet in the model, the model will be expanded to include the new node. Again, as a side-effect of inserting the node into the model, every unmodeled ancestor of the node will be inserted into the model.

The model can be efficiently constructed, expanded, and topologically sorted by the LTM algorithm if the following facts are maintained for each node $n$ in $d$:

- *AttrNodesFor* – the set of attribute tree nodes in $d_{copy}$ representing node $n$ in $d$, and
- *InEdges* – the set of edges incident on node $n$ whose sources are nodes in $d$.

These facts must be maintained on a per-module instance basis. For example, the *AttrNodesFor* set should contain only those attribute tree nodes appearing in the attribute tree $d_{copy}$ for a particular module instance and the *InEdges* set should contain only those edges incident on node $n$ from nodes in $d$. As before, each attribute tree node contains the pair of fields *parent* and *dagnode*.

To maintain *AttrNodesFor* and *InEdges* on a per-module instance basis, an association map will be stored as part of the saved state for each module instance and will be used to map each syntax node in $d$ to the appropriate pair of sets. The association map will be explicitly passed to each of the functions responsible for managing the pair of sets. It is convenient to use a unique integer *id* associated with each syntax node as the key in the association map. The unique id is assigned to each syntax node when the node is allocated. The functions *CreateAttrNode* and *DeleteAttrNode* will update the *AttrNodesFor* set for each syntax node and will be responsible for inserting and deleting *Map* entries. Whenever the first attribute tree node is inserted into the map for a particular syntax node with an id *id*, a new map entry for *id* will be created and entered into the map. When the last attribute tree node for that syntax node is deleted, the map entry will be removed automatically from the map and deallocated.

On invocation of the matching algorithm with a saved state containing the dag $d$, the attribute tree $d_{copy}$, and the association map *Map*, the following conditions will hold:

- *Map* contains an entry for the id of each node in $d$.
- If attribute tree node $X$ is in $d_{copy}$, then $X$ is in the *AttrNodesFor* set of node $X.dagnode$.
- If attribute tree node $X$ is in $d_{copy}$ and $Y$ is the $i$th child of $X$, then the *InEdges* set for $Y.dagnode$ includes an element representing the edge connecting $Y.dagnode$ as the $i$th child of $X.dagnode$.

On completion of the matching algorithm, the association map will only contain entries for nodes in $d'$, and the sets *AttrNodesFor* and *InEdges* will correctly represent the contents of $d'$ and $d'_{copy}$. The mechanism by which the *InEdges* set is manipulated is described in the next section.

```
Function inD(n, Map)
  return (Map(id(n)) != NULL)
end
```

**Figure 7.20** inD – LTM Algorithm

The function *inD*, shown in Figure 7.20, uses the association map to determine if a node $n$ which has been reached by a traversal from the root of $d'$ is a member of $d$. Node $n$ is in $d$ if and only if it has an entry in *Map*. Otherwise, node $n$ must be in $d' - d$.

### 7.4.2 Managing the *InEdges* Sets

The set of *InEdges* for each syntax node serves two purposes:

- The elements of the *InEdges* set for a node $n$ will be used as parent pointers so that, when $n$ is inserted into the model, all of $n$'s unmodeled ancestors can be efficiently identified and inserted into the model.
- The number of edges in the *InEdges* set for each node will be used in computing the in-degree of the node for use in topological sorting of the model.

The notation $\langle X, i, Y \rangle$ will be used to represent the fact that node $Y$ is the $i$th child of node $X$ in either an attribute tree or a syntax term. Similarly, an edge incident on a particular node $S$ in an attribute tree or a syntax dag will be represented by a pair $\langle P, i \rangle$ where $P$ is a node and $i$ is the index of the slot in $P$ occupied by $S$.

The *InEdges* set for each syntax node will be maintained by the functions *AddEdge* and *DeleteEdge* as a side-effect of the functions *Prune* and *Graft* which are responsible, respectively, for detaching attribute tree nodes from their parents and for connecting attribute tree nodes to their new parents. The following simple protocol guarantees that the *InEdges* sets for $d'$ properly represent the structure of $d'$ when matching completes, assuming that the *InEdges* sets for nodes in $d$ properly represent the structure of $d$ when matching commences.

The *InEdges* set for each syntax node will be implemented as a doubly-linked list of *InEdge* nodes each containing a tuple $\langle dagnode, i \rangle$ and a reference count, so that the LTM algorithm can enumerate the edges incident on a syntax node in time proportional to the size of the set. The reference count is required since there may be multiple attribute tree edges representing the edge $\langle dagnode, i \rangle$.

An association map, *EdgeMap*, will be used to map from an edge tuple $\langle dagnode, i \rangle$ to the *InEdge* node containing that edge tuple. The *EdgeMap* for each module instance will be stored as part of a module instance's saved state. Each time a graft operation links an old attribute tree node $S$ to slot $i$ of a new attribute tree node $P$, the *EdgeMap* will be used to locate the appropriate *InEdge* node for the edge tuple $\langle P.dagnode, i \rangle$ if such a node exists. If an *InEdge* node is located, then its reference count is incremented. Otherwise, a new *InEdge* node with a reference count of 1 must be created to represent the edge tuple. The *InEdge* node is then inserted into the *InEdges* set for $S.dagnode$, and an entry is inserted into *EdgeMap* for the edge tuple.

### 7.4.3 Bookkeeping for Lazy Topological Matching

Nodes will be selected from the model based on the values of two counters on each node in $d \cup d'$. The *in-degree* counter on a node $n$ will record the number of edges from nodes in $d \cup d'$ that are incident on node $n$, and the *del-degree* counter will record the number of edges incident on node $n$ whose sources have been processed by the matching algorithm. The *in-degree* counter for each node will be set correctly when that node is first inserted into the model. During the construction of the initial model, any modeled node whose *in-degree* is found to be equal to its *del-degree* will be inserted into the queue R-QUEUE. During matching, modeled nodes will be inserted into R-QUEUE whenever incrementing the *del-degree* counter on a modeled node causes the *del-degree* and the *in-degree* counters for a node to become equal.

Both the *in-degree* and *del-degree* counters on nodes in $d \cup d'$ will have the value 0 when matching commences. After the new attribute tree has been constructed, the *in-degree* and *del-degree* counters on the nodes in the model and on the fringe of the model will be explicitly reinitialized to 0.

### 7.4.4 The LTM Algorithm

Function *Match*, shown in Figure 7.21, retrieves $d$ and $d_{copy}$ from the saved state *State*, creates the initial crown of the new attribute tree by invoking *Create*$\top$, explicitly initializes the PROVIDED and NEEDED sets of the root of $d$ and $d'$, and then invokes procedure *InitializeModel* to build the initial model of $d \cup d'$. *Match* then invokes the procedure *BuildTree* to process the modeled region of $d$ and to build the complete attribute tree $d'_{copy}$. Once the new attribute tree has been completed, *Match* invokes function *Reinitialize* to reset the NEEDED, PROVIDED, *in-degree* and *del-degree* fields for each node in the model and for each node in the model's fringe. Finally, *Match* invokes *Remove*$\top$ to remove the node $\top$ from $d'_{copy}$, stores the values $d'$ and $d'_{copy}$ back into *State* for use in the next evaluation of this module instance, and returns the new attribute tree.

Procedure *InitializeModel*, shown in Figure 7.22, is responsible for creating the initial model of $d \cup d'$ containing all of the nodes in $d' - d$, the nodes in the fringe of $d' - d$, and every ancestor in $d$ of the nodes in the fringe of $d' - d$ during a depth-first traversal over the nodes in $d' - d$. When invoked with an unmodeled node in $d' - d$, *InitializeModel* inserts the node into the model, and then recursively invokes *InitializeModel* to add each of the node's children to the model. When invoked with an unmodeled node in the fringe of

```
Function Match(d',State)
  d = State.d
  d_copy = State.d_copy

  d'_copy = Create⊤()

  PROVIDED(root(d)) = {d_copy}
  NEEDED(root(d')) = {⊤}

  -- Build the crown of the new attribute tree d'_copy
  model = {}
  R-QUEUE = {}
  InitializeModel(root(d'), R-QUEUE, model, State.Map, State.EdgeMap)

  -- Build the rest of the attribute tree d'_copy
  BuildTree(R-QUEUE, model, d'_copy, State.Map, State.EdgeMap)

  Reinitialize(model)
  Remove⊤(d'_copy)

  State.d = d'
  State.d_copy = d'_copy
  return d'_copy
end
```

**Figure 7.21** Match – LTM Algorithm

```
Procedure InitializeModel(dagnode, R-QUEUE, model, Map, EdgeMap)
  if (dagnode ∈ model))
    return
  -- if there is no map entry for dagnode in Map, then dagnode should be added
  -- to the initial model
  if (!inD(dagnode,Map))
  then -- dagnode is in the d′ − d
    insert dagnode into model
    for each child i of dagnode
      IN-DEGREE(dagnode[i]) = IN-DEGREE(dagnode[i]) + 1
      InitializeModel(dagnode[i], R-QUEUE, model, Map, EdgeMap)
  else -- dagnode is in the fringe of d′ − d
    ExpandModel(dagnode, R-QUEUE, model, Map, EdgeMap)
  fi
end
```

**Figure 7.22** InitializeModel – LTM Algorithm

$d' - d$, *InitializeModel* invokes *ExpandModel* to insert the node and all of its unmodeled ancestors into the model. On completion of *InitializeModel*, the *in-degree* counter on each node in the model will be correct as a consequence of incrementing the *in-degree* counter for all nodes that are found to be targets of edges from a node in $d' - d$ and as a result of procedure *ExpandModel* incrementing the *in-degree* counters on each node $n$ in $d$ to include each of the edges from nodes in $d$ to node $n$, as node $n$ is inserted into the model.

```
Procedure ExpandModel(dagnode, R-QUEUE, model, Map, EdgeMap)
  if (dagnode ∈ model)
    return

  insert dagnode into model

  inedges = InEdges(dagnode,Map)

  -- attempt to add each ancestor of dagnode to the model
  for each ⟨parent,index⟩ in inedges
    ExpandModel(parent,R-QUEUE, model, Map, EdgeMap)

  IN-DEGREE(dagnode) = IN-DEGREE(dagnode) + |InEdges(dagnode,Map)|

  -- if all edges from parents of dagnode to dagnode have been traversed,
  -- then add dagnode to R-QUEUE
  if (IN-DEGREE(dagnode) == DEL-DEGREE(dagnode))
      q-insert dagnode into R-QUEUE
end
```

**Figure 7.23** ExpandModel – LTM Algorithm

Procedure *ExpandModel*, shown in Figure 7.23, is invoked to insert the node *dagnode* and all of its ancestors in $d$ into the model. The ancestors of *dagnode* are then inserted into the model by recursively invoking *ExpandModel*. As described in the previous section, the parents of *dagnode* are identified by traversing the contents of the *InEdges* list for *dagnode*. Each element in the set of *InEdges* is a tuple containing an edge ⟨parent,index⟩ and a reference count, where *parent* is one of the syntax nodes that is a parent of *dagnode*, and the *index*th slot of *parent* is occupied by *dagnode*. The same parent of *dagnode* may appear in the set *InEdges* with different *index* values. To account for the entire set of edges from nodes in $d$ to *dagnode*, *ExpandModel* adds the size of the *InEdges* set for *dagnode* to the *in-degree* counter for *dagnode*. If upon inserting *dagnode* into the model, the *in-degree* counter becomes equal to its *del-degree* counter, then the node is inserted into R-QUEUE.

Procedure *BuildTree*, shown in Figure 7.24, has been modified in three ways from the function presented for the TM algorithm: (1) it invokes procedure *ExpandModel* to add a node and its ancestors to the model whenever a placeholder is inserted into the NEEDED set for an unmodeled node, (2) it increments the *del-degree*

```
Procedure BuildTree(R-QUEUE, model, d'_copy, Map, EdgeMap)
  while R-QUEUE not empty
    dagnode = q-remove(R-QUEUE)

    needs = NEEDED(dagnode)
    provs = PROVIDED(dagnode)
    ⟨remprovs,newtreenodes⟩ = Fill(dagnode,needs,provs,Map,EdgeMap)

    – – make each child of each node that is still available be available
    if (|remprovs| > 0)
      for each child i of dagnode
        for each attrnode in remprovs
          union attrnode[i] into PROVIDED(dagnode[i])

    – – deallocate each unused node in PROVIDED
    for each attrnode in remprovs
      DeallocNode(attrnode, Map, EdgeMap)

    – – each child of a newtreenode is needed
    if (|newtreenodes| > 0)
      for each child i of dagnode
        for each attrnode in newtreenodes
          union ⟨attrnode,i⟩ into NEEDED(dagnode[i])
        ExpandModel(dagnode[i], R-QUEUE, model, Map, EdgeMap)

    – – basic queue management for topological sorting
    for each child i of dagnode
      DEL-DEGREE(dagnode[i]) = DEL-DEGREE(dagnode[i]) + 1
      if (dagnode[i] ∈ model and
          IN-DEGREE(dagnode[i]) == DEL-DEGREE(dagnode[i]))
        q-insert dagnode[i] into R-QUEUE
end
```

**Figure 7.24** BuildTree – LTM Algorithm

```
Function Fill(dagnode,needs,provides,Map,EdgeMap)
  while (|needs| > 0 and |provides| > 0)
    need = Choose(needs)
    child = Choose(provides)
    Graft(child,need.parent,need.slot,Map,EdgeMap)

  newtreenodes = {}
  while (|needs| > 0)
    need = Choose(needs)
    child = CreateAttrNode(dagnode,Map)
    Graft(child,need.parent,need.slot,Map,EdgeMap)
    insert child into newtreenodes

  return <provides,newtreenodes>
end
```

**Figure 7.25** Fill – LTM Algorithm

counter for each child of a processed node instead of decrementing its *in-degree* counter, and (3) it inserts each modeled child with equal *in-degree* and *del-degree* counters into R-QUEUE.

Function *Fill*, shown in Figure 7.25, is essentially identical to the function presented for the TM algorithm.

### 7.4.5 A Lazy Topological Matching Example

We now present an illustration of the LTM algorithm for the running example. Figure 7.26 depicts $d$, $d'$ and $d_{copy}$ with each node in $d \cup d'$ labeled with a four-tuple $\langle$in-degree, del-degree, PROVIDED, NEEDED$\rangle$. Before invoking *Match*, each node in $d$ has *in-degree* and *del-degree* values that are both 0 and NEEDED and PROVIDED sets that are both empty. Figures 7.27 depicts the initial model and the labeling of nodes in $d \cup d'$, and the initial crown of $d'_{copy}$ on invocation of procedure *BuildTree*. Figure 7.28 through Figure 7.33 present the labeling of nodes in $d \cup d'$ and the portion of $d'_{copy}$ that has been constructed after each modeled node has been selected in topological order with respect to the model of $d \cup d'$. As nodes are processed by *BuildTree*, they are eliminated from the figures. Filled circles in $d \cup d'$ represent unprocessed nodes in the model, and filled circles in $d'_{copy}$ represent new attribute tree nodes.

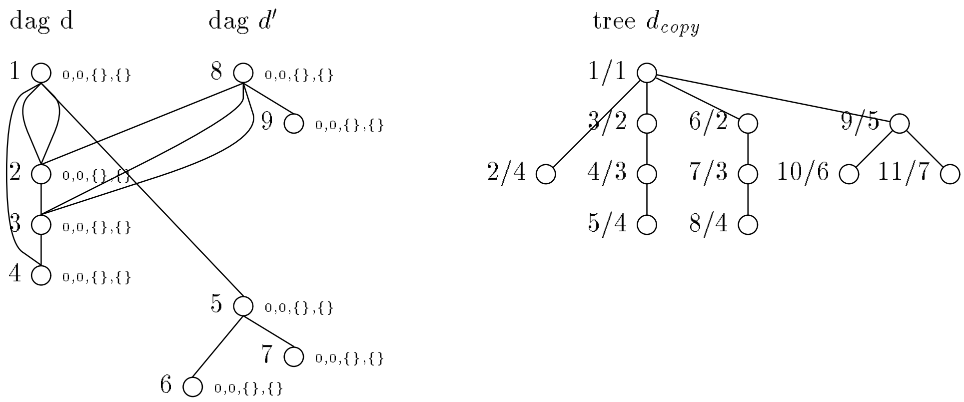


**Figure 7.26** Inputs to LTM Algorithm

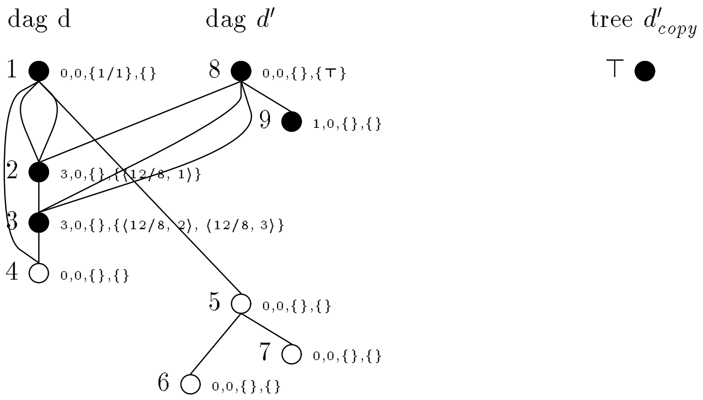


**Figure 7.27** On Invocation of *BuildTree*

### 7.4.6 Proofs for the LTM Algorithm

We now prove informally that the LTM algorithm constructs the new attribute tree $d'_{copy}$ using maximal subtrees of the attribute tree $d_{copy}$ in $O(|M_{P=N=\emptyset}(d \cap d')| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ time.

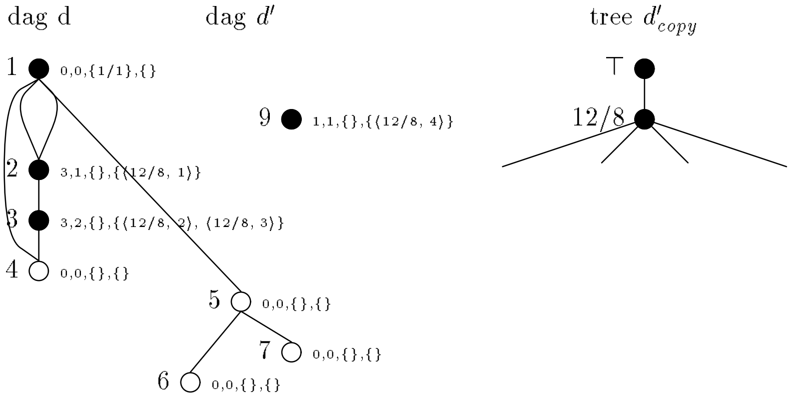


**Figure 7.28** After Processing node 8

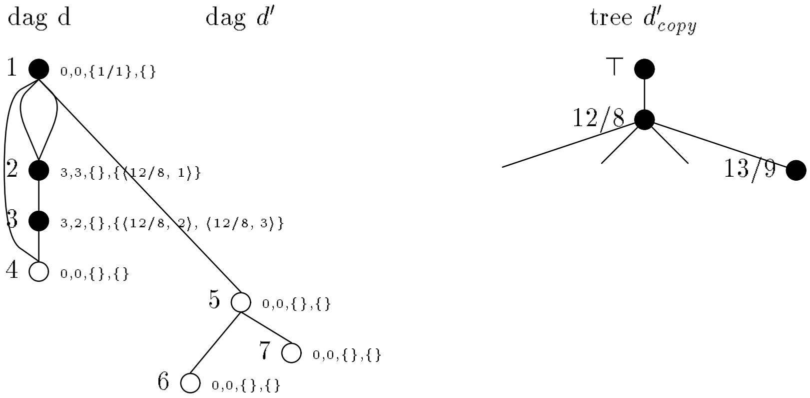


**Figure 7.29** After Processing node 9

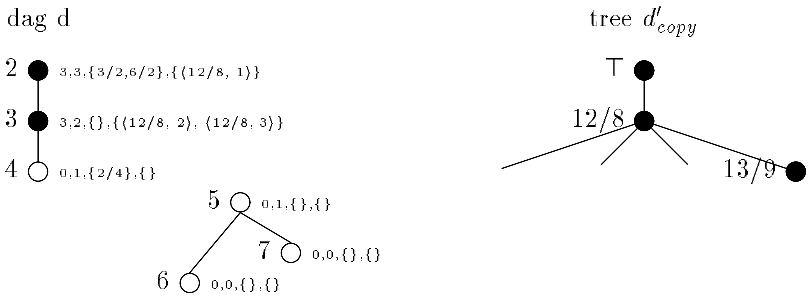


**Figure 7.30** After Processing node 1

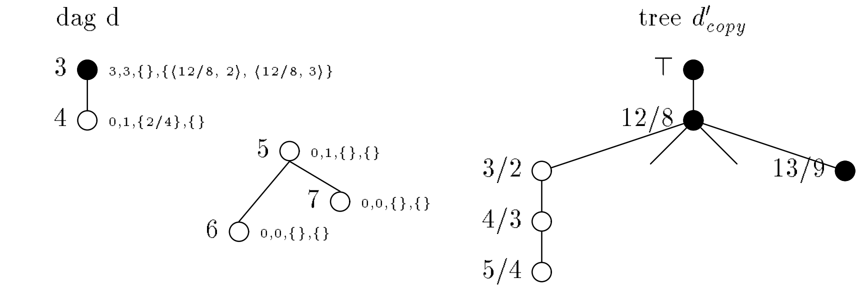


**Figure 7.31** After Processing node 2

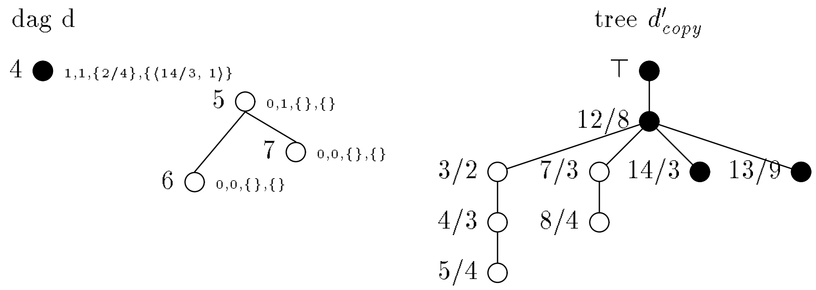


**Figure 7.32** After Processing node 3

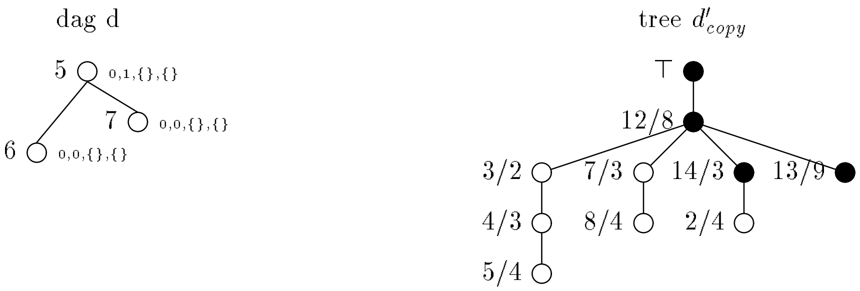


**Figure 7.33** After Processing node 4

It has already been assumed that, on invocation of the LTM algorithm, the association maps $Map$ and $EdgeMap$ correctly represent the dag $d$ and the attribute tree $d_{copy}$ as described above in Section 7.4.1. This implies that, on completion of matching, the maps $Map$ and $EdgeMap$ will correctly represent the dag $d'$ and the attribute tree $d'_{copy}$, since the functions *CreateAttrNode* and *DeleteAttrNode* are responsible for maintaining $Map$ and the functions *AddEdge* and *DeleteEdge* are responsible for maintaining $EdgeMap$. As the LTM algorithm deallocates edges and nodes in $d_{copy}$ and allocates new edges and nodes for $d'_{copy}$, the contents of $Map$ and $EdgeMap$ will automatically be updated to reflect the contents of $d'$ and $d'_{copy}$.

To establish correctness of the LTM algorithm, it is necessary to demonstrate that the LTM algorithm processes a model of the nodes in $d \cup d'$ in an order that respects the topological order defined by $d \cup d'$, and that, after processing the nodes in the model, the new attribute tree $d'_{copy}$ will be complete. Satisfying the topological ordering ensures that the values assigned to the PROVIDED and NEEDED sets associated with nodes in the model by the LTM algorithm will be identical to the values assigned to the sets associated with those nodes by the TM algorithm, and, therefore, that the invocations of function *Fill* will build an attribute tree identical to the tree constructed by the TM algorithm. The proof in Theorem 7.6 that the LTM algorithm is correct is based on the following premises:

- On completion of *InitializeModel*, the *in-degree* counter of every node $n$ in the initial model is equal to the number of edges incident on node $n$ from nodes in $d \cup d'$.
- Whenever *ExpandModel* is invoked by *BuildTree* to insert an unmodeled node $n$ into the model, the *in-degree* counter on node $n$ is set to be equal to the number of edges incident on node $n$ from nodes in $d \cup d'$.
- The *del-degree* counter on each child of a node $n$ in $d \cup d'$ is correctly incremented after node $n$ has been processed by *BuildTree*.
- Whenever all of the predecessors of a modeled node have been processed, the modeled node is inserted into R-QUEUE.

**Theorem 7.6** The LTM algorithm, when applied to a dag $d'$ and a saved state containing the dag $d$, the attribute tree $d_{copy}$, and the association maps $Map$ and $EdgeMap$, constructs the new attribute tree $d'_{copy}$ using maximal subtrees from $d_{copy}$.

**Proof** The claims listed above ensure that procedure *BuildTree* will process all of the nodes in the model in topological order. On completion of *BuildTree*, since all of the nodes in the model will have been processed, and, since no unmodeled node can possibly have a non-empty NEEDED set, the new attribute tree $d'_{copy}$ will be complete. Since at all times during matching, the nodes in the model of $d \cup d'$ will "topologically precede" the nodes in $d \cup d'$ that are not in the model, the order in which *BuildTree* selects nodes from the model respects the partial order defined by $d \cup d'$. Therefore, $d'_{copy}$ selects maximal subtrees from $d_{copy}$ for use in $d'_{copy}$, since the attribute tree constructed by the LTM algorithm will be identical to the attribute tree built by the TM algorithm. □

**Theorem 7.7** The LTM algorithm requires $O(|M_{P=N=\emptyset}(d \cap d')| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ time to perform matching.

**Proof** The cost of the LTM algorithm is dominated by the sum of the costs required to traverse $M(d \cup d')$, and to deallocate and allocate nodes in $d_{copy} - d'_{copy}$ and $d'_{copy} - d_{copy}$.

Divide $M(d \cup d')$ into $M(d' - d)$, $M(d \cap d')$, and $M(d - d')$, and note that $|M(d' - d)|$ is $O(|d'_{copy} - d_{copy}|)$, and $|M(d - d')|$ is $O(|d_{copy} - d'_{copy}|)$. $M(d \cap d')$ is the union of four components $M_{P>N}(d \cap d')$, $M_{P<N}(d \cap d')$, $M_{P=N>\emptyset}(d \cap d')$, and $M_{P=N=\emptyset}(d \cap d')$, where $M_{P>N}(d \cap d')$ is the set of modeled nodes in $d \cap d'$ whose PROVIDED sets are larger than their NEEDED sets, $M_{P<N}(d \cap d')$ is the set of modeled nodes whose PROVIDED sets are smaller than their NEEDED sets, $M_{P=N>\emptyset}(d \cap d')$ is the set of modeled nodes whose PROVIDED and NEEDED sets have the same size and are non-empty, and $M_{P=N=\emptyset}(d \cap d')$ is the set of modeled nodes whose PROVIDED and NEEDED sets are both empty. The cost of processing three of these four components,

$M_{P>N}(d \cap d')$, $M_{P<N}(d \cap d')$, and $M_{P=N>\emptyset}(d \cap d')$ is $O(|d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}|)$, hence, the running time for the LTM algorithm is $O(|M_{P=N=\emptyset}(d \cap d')| + |d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}|)$. □

The LTM algorithm exceeds Teitelbaum and Chapman's suggested optimal time bound by a factor of $|M_{P=N=\emptyset}(d \cap d')|$. How large can $M_{P=N=\emptyset}(d \cap d')$ be? The simple example, shown in Figure 7.34, unfortunately, demonstrates that the cost of processing nodes in $M_{P=N=\emptyset}(d \cap d')$ can be quite large. If the LTM algorithm is applied to this example to construct the new attribute tree $d'_{copy}$, then the model will include the root of $d'$ and the nodes labeled 1,2,3,...,$N-1$ and $N$ in $d$. Deallocation and allocation of nodes to create the new attribute $d'_{copy}$ takes only $O(1)$ time, since the number of nodes in $d_{copy}$ and $d'_{copy}$ that must be deallocated and allocated is independent of the size of $d$ and $d'$. $M_{P=N=\emptyset}(d \cap d')$, however, has size $O(n)$ since the PROVIDED and NEEDED set for each node between nodes 3 and $N-1$, inclusive, will be empty.

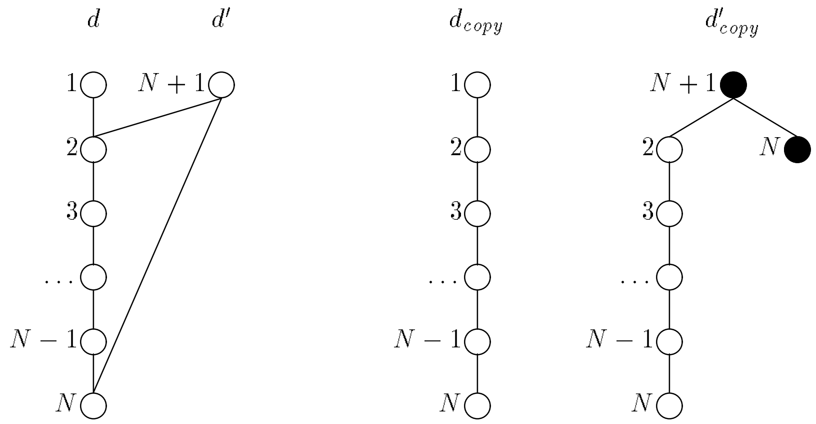


**Figure 7.34** An Example

We conjecture that a non-constant amount of work must be performed at nodes in $M_{P=N=\emptyset}(d \cap d')$ in order to determine the proper topological ordering of nodes in the model in order to guarantee optimal construction of $d'_{copy}$ from $d_{copy}$. It, therefore, remains an open problem to either find an algorithm that can perform matching within Teitelbaum and Chapman's suggested bound of $O(|d_{copy} - d'_{copy}| + |d'_{copy} - d_{copy}|)$, or to prove that such an algorithm cannot exist.

## 7.5 The Lazy Retaining Topological Matching Algorithm

We now combine the LTM and RTM-k algorithms from the previous to sections to form the *Lazy Retaining Topological Matching Algorithm (LRTM-k)* which constructs the new attribute tree $d'_{copy}$ by selecting maximal subtrees from the attribute trees in the forest $d^*_{copy}$ during a lazy topological traversal of the dag $d^* \cup d'$.

### 7.5.1 The Lazy Retaining Topological Matching Approach

The LRTM-k algorithm will be invoked with the new dag $d'$ and a saved state containing only the association maps *Map* and *EdgeMap*. Unlike the RTM-k algorithm, it is unnecessary to explicitly maintain references to the roots of $d^*_{copy}$ as part of the saved state for each module instance, and it is unnecessary for *Prune*, *Graft* and *DeallocNode* to update the lists of roots.

The LRTM-k algorithm will apply the same protocols as used by the LTM algorithm to maintain the *InEdges* sets for each of the nodes in $d^*$. The initial model construction, topological processing of nodes in the model, and expansion of the model will also be performed in exactly the same manner as in the LTM algorithm.

Instead of explicitly initializing the PROVIDED sets for the roots of $d^*$ with attribute trees from $d^*_{copy}$, the LRTM-k algorithm will insert each attribute tree in the forest of retained attribute trees into the PROVIDED set of a node in $d^*$ each time the procedure *ExpandModel* is invoked to insert a node that is the root of a subterm represented by an attribute tree in the forest into the model. This process guarantees that all retained attribute trees which may contribute components to $d'_{copy}$ will be available to the matching algorithm and requires that the matching algorithm be able to identify, for each node in $d^*$, the set of attribute trees in $d^*_{copy}$ that represent the subterm rooted by that node.

Prior to matching, each retained attribute tree in $d^*_{copy}$ represents some subterm of the dag $d^*$, and after matching, each attribute tree in the database of trees $d'^*_{copy}$ represents a subterm of $d'^*$. The root of the subterm $d^*$ that is represented by a retained attribute tree in $d^*_{copy}$ will be said to be the *birthpoint* of that attribute tree, and the attribute tree will be said to be *born* at that dag node. Every attribute tree in $d^*_{copy}$ is born at some node in $d^*$.

The LRTM-k algorithm will explicitly maintain the set of attribute trees in $d^*_{copy}$ that are born at each node in $d^*$. Instead of associating a single *AttrNodesFor* set with each syntax node $n$, as in the LTM algorithm, the LRTM-k algorithm will divide the set of attribute tree nodes for $n$ into *BornHere* and *BornAbove* sets. The *BornHere* set will contain all of the attribute tree nodes representing node $n$ which are roots of attribute subtrees born at $n$. The *BornAbove* set will contain all of the attribute tree nodes representing node $n$ which are not the roots of attribute trees born at $n$.

The functions *AddEdge* and *DeleteEdge* of the LRTM-k algorithm will move attribute tree nodes between the *BornHere* and *BornAbove* lists as edges are inserted and deleted between attribute tree nodes. Whenever a new attribute tree node is created by *CreateAttrNode*, the new node will be inserted into the *BornHere* list of the syntax node represented by the new attribute tree node, since the new attribute tree node will have no parent initially. The function *AddEdge* will then ensure that an attribute tree node $S$ will be moved from *S.dagnode.BornHere* to *S.dagnode.BornAbove*, whenever an edge is inserted to connect a child attribute tree node $S$ to its parent node $P$. Whenever an edge connecting a parent attribute tree node $P$ to a child attribute tree node $S$ is deleted, *DeleteEdge* will move $S$ from *S.dagnode.BornAbove* to *S.dagnode.BornHere*. As before, the functions *AddEdge* and *DeleteEdge* are responsible for maintaining the *InEdges* set for syntax nodes.

### 7.5.2 The LRTM-k Algorithm

We now describe the functions implementing the LRTM-k algorithm. There are few actual differences between these new functions and the functions for the LTM algorithm. The functions *AddEdge* and *DeleteEdge* have been modified to maintain the *BornHere* and *BornAbove* sets for each dagnode as just described.

```
Function Match(d', State)

  d'_copy = Create⊤() -- Create node ⊤ and insert it into d'_copy

  NEEDED(root(d')) = {⊤} -- a copy of a subtree representing d'_copy is needed

  -- Walk d* ∪ d' to compute in-degrees for nodes
  model = {}
  R-QUEUE = {}
  InitializeModel(root(d'), R-QUEUE, model, State.Map, State.EdgeMap)

  -- Build the rest of the attribute tree d'_copy
  BuildTree(R-QUEUE, model, d'_copy, State.Map, State.EdgeMap)

  Reinitialize(model)
  Replenish(root(d'_copy))
  Remove⊤(d'_copy)

  return d'_copy
end
```

**Figure 7.35** Match – LRTM-k Algorithm

Function *Match*, shown in Figure 7.35, creates the new attribute tree by invoking *Create*$\top$ and then initializes the NEEDED set for the root of $d'$. The PROVIDED set is no longer initialized explicitly by *Match*, but, is instead initialized as nodes are inserted into the model by calls to *ExpandModel*. Exactly as in the LTM algorithm, *Match* invokes procedure *InitializeModel* to build the initial model, and then invokes procedure *BuildTree* to build the new attribute tree $d'_{copy}$. Procedure *InitializeModel* is identical to the procedure presented for the LTM algorithm, except function $inD^*$ which checks for membership in $d^*$ is invoked instead of function $inD$ which checks for membership in $d$. The implementations of functions $inD^*$ and $inD$ are identical, however. Once $d'_{copy}$ has been built, the function *Reinitialize* is invoked to reset the values of the PROVIDED, NEEDED, *in-degree* and *del-degree* fields for each node in the model and for each node in the model's fringe. *Match* then invokes function *Replenish* to reset the counters on nodes in the crown of depleted nodes in $d'_{copy}$ to the value $k$ during a depth-first traversal over the depleted nodes in $d'_{copy}$. Finally, *Match* invokes $Remove\top$ and then returns the complete attribute tree $d'_{copy}$.

```
Procedure BuildTree(R-QUEUE, model, d'_copy, Map, EdgeMap)
  while R-QUEUE not empty
    dagnode = q-remove(R-QUEUE)

    needs = NEEDED(dagnode)
    provs = PROVIDED(dagnode)
    ⟨remprovs,newtreenodes⟩ = Fill(dagnode,needs,provs,Map,EdgeMap)

    – – make each child of each node that is still available be available
    if (|remprovs| > 0)
      for each child i of dagnode
        for each attrnode in remprovs
          union attrnode[i] into PROVIDED(dagnode[i])

    – – Decrement the counter on the root node in each subtree in remprovs.
    – – Expire nodes whose counter reaches the value 0.
    for each prov in remprovs
      prov.counter = prov.counter - 1
      if (prov.counter == 0)
        for each child i of prov
          Prune(prov, i, prov[i], Map, EdgeMap)
        DeallocNode(prov, Map, EdgeMap)

    – – each child of a newtreenode is needed
    if (|newtreenodes| > 0)
      for each child i of dagnode
        for each attrnode in newtreenodes
          union ⟨attrnode,i⟩ into NEEDED(dagnode[i])
        ExpandModel(dagnode[i],R-QUEUE, model, Map, EdgeMap)

    – – basic queue management for topological sorting
    for each child i of dagnode
      DEL-DEGREE(dagnode[i]) = DEL-DEGREE(dagnode[i]) + 1
      if (dagnode[i] ∈ model and
        IN-DEGREE(dagnode[i]) == DEL-DEGREE(dagnode[i]))
        q-insert dagnode[i] into R-QUEUE
end
```

**Figure 7.36** BuildTree – LRTM Algorithm

Procedure *BuildTree*, shown in Figure 7.36, is the same as the function presented for the LTM algorithm, except, unused attribute tree nodes in PROVIDED are not automatically deallocated. As in the RTM-k algorithm, *BuildTree* will decrement the counter on each unused attribute tree node and will deallocate each of the nodes which, thereby, becomes empty.

Procedure *ExpandModel*, shown in Figure 7.37, inserts the node *dagnode* into the model exactly as in the LTM algorithm, and then inserts the contents of the *BornHere* set for *dagnode* into the PROVIDED set for *dagnode* to indicate that all of the attribute trees born at *dagnode* are available for use in building $d'_{copy}$.

Function *Fill*, shown in Figure 7.38, invokes the function *DeallocAncestors* to deallocate the ancestors of the root of each attribute subtree in PROVIDED that is used to fill a need in NEEDED.

```
Procedure ExpandModel(dagnode, R-QUEUE, model, Map, EdgeMap)
  if (dagnode ∈ model)
    return

  insert dagnode into model

  inedges = InEdges(dagnode,Map)

  – – attempt to add each ancestor of dagnode to the model
  for each ⟨parent, index⟩ in inedges
    ExpandModel(parent,R-QUEUE, model, Map, EdgeMap)

  IN-DEGREE(dagnode) = IN-DEGREE(dagnode) + |InEdges(dagnode,Map)|

  – – if all edges from parents of dagnode to dagnode have been traversed,
  – – then add dagnode to R-QUEUE
  if (IN-DEGREE(dagnode) == DEL-DEGREE(dagnode))
      q-insert dagnode into R-QUEUE

  – – add all rooted subtrees for this dagnode into PROVIDED
  union BornHere(dagnode,Map) into PROVIDED(dagnode) end
```

**Figure 7.37** ExpandModel – LTM Algorithm

```
Function Fill(dagnode,needs,provides,Map,EdgeMap)
  while (|needs| > 0 and |provides| > 0)
    need = Choose(needs)
    child = Choose(provides)

    – – Deallocate all of child's ancestors by following parent pointers
    – – in attribute tree nodes.
    DeallocAncestors(child,Map,EdgeMap)

    Graft(child, need.parent, need.slot, Map, EdgeMap)

  newtreenodes = {}
  while (|needs| > 0)
    need = Choose(needs)
    child = CreateAttrNode(dagnode,Map)
    child.counter = 0

    Graft(child,need.parent,need.slot,Map,EdgeMap)
    insert child into newtreenodes

  return ⟨provides,newtreenodes⟩
end
```

**Figure 7.38** Fill – LRTM Algorithm

### 7.5.3 Proofs for the LRTM-k Algorithm

By viewing the LRTM-k algorithm as a lazy version of the RTM-k algorithm it can be seen that the algorithm runs in $O(|M_{P=N=\emptyset}(d \cap d')| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ amortized time, where $d_{copy}$ is the attribute tree that was built by the LRTM-k algorithm the last time this module instance was invoked, and $d$ is the syntax term that was evaluated by that invocation of the LRTM-k algorithm, and where $M_{P=N=\emptyset}(d \cap d')$ is the set of nodes in $d \cap d'$ that are in the model and whose NEEDED and PROVIDED sets both would have been determined to be empty by the TM algorithm.

## 7.6 A Simple Extension to Matching

This section concludes the presentation of purely syntax-based matching algorithms by presenting a simple extension to the matching algorithms that may significantly decrease the work required to incrementally update a large class of hierarchical specifications.

### 7.6.1 The Modification

Each of the matching algorithms that have been described in this chapter deallocate a set of retained attribute tree nodes as a side-effect of constructing the new attribute tree $d'_{copy}$. As attribute tree nodes are deallocated, all of the information associated with those nodes becomes unavailable, including the values of the attributes associated with each node, and, most importantly, the contents of the node's saved state array. A small modification to the matching algorithm and to the change propagator permits the saved state information associated with each of these deallocated nodes to be retained and then later reused by an invocation of a module instance with an otherwise NULL-valued saved state.

In addition to *Map* and *EdgeMap*, the saved state for each module instance will contain an array *ChildStates* having an entry for each of the modules in a hierarchical specification. Each entry in *ChildStates* will contain a queue of retained saved states. Whenever function *DeallocNode* is invoked to deallocate an attribute tree node during matching for a module instance $MI$, each of the non-NULL-valued elements of the saved state array for that node will be inserted into the appropriate queues in the *ChildStates* array. When the change propagator encounters an invocation of a module instance associated with an attribute tree node $n$, if the saved state for the invoked module instance is NULL, and if the queue for the invoked module of retained saved states in *ChildStates* is non-empty, then a saved state will be removed from the queue, stored into the saved state array of node $n$, and then passed to function *ModuleEvaluate*.

Assuming that the operations for retaining and then later reusing these saved states are implemented as constant time operations, the retention and reuse of otherwise unusable saved state values will have no impact on the amortized cost of incremental evaluation of a hierarchical specification. However, as in the analysis of the RTM-k and LRTM-K algorithms, retention and reuse of previously computed attributes may dramatically reduce the time required to update modules in a hierarchical system. We now present an example which greatly benefits from this additional source of retained attribute values.

### 7.6.2 An Example

Consider a simple "cascaded" hierarchical attribute grammar specification shown in Figure 7.39. The specification consists of three modules *Root*, *BtoD* and *CtoD* and describes the mapping of a syntax tree of type $A$-tree into a value of type $D$-tree. *Root* takes an $A$-tree as input and returns a $D$-tree. When invoked with an $A$-tree, *Root* synthesizes a $B$-tree that is structurally identical to the $A$-tree but with nodes labeled with operators for $B$-trees, and invokes the module *BtoD* to evaluate the $B$-tree. Module *BtoD* then creates a $C$-tree that is structurally identical to the $B$-tree but with nodes labeled with operators for $C$-trees, and invokes the module *CtoD* to evaluate the $C$-tree. *CtoD* then constructs the $D$-tree by performing evaluation over the $C$-tree.

Figure 7.40 depicts an initial input term $A$ to be evaluated by *Root* and the terms $B$ and $C$ constructed during batch evaluation of the simple specification. Each node in $A$, $B$ and $C$ is labeled with its unique id.

To simplify the example, only tree-structured syntax terms will be used in this example. In the following, since all of the terms are tree-structured, we will implicitly use the id of a syntax node as the id of the attribute tree node which represents that syntax node, i.e., instead of having to refer to the attribute tree

| | | |
|---|---|---|
| R ::= RA(A) | R ::= RB(B) | R ::= RC(C) |
| R.d = BtoD(RB(A.b)) | R.d = CtoD(RC(B.c)) | R.d = . . . |
| A ::= AA(A A) | B ::= BB(B) | |
| A1.b = BB(A2.b, A3.b) | B1.c = CC(B2.c, B3.c) | |
| A ::= A() | B ::= B() | |
| A1.b = B() | B1.c = C() | |
| Root | BtoD | CtoD |

**Figure 7.39** A Simple Cascaded Specification

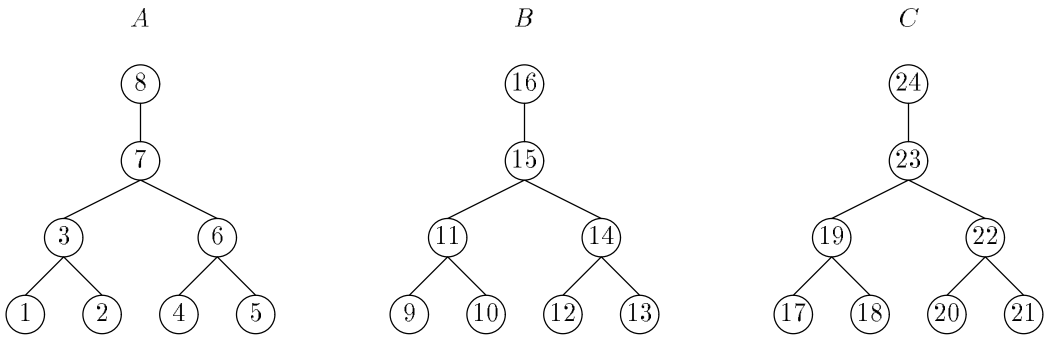


**Figure 7.40** Terms $A$, $B$ and $C$

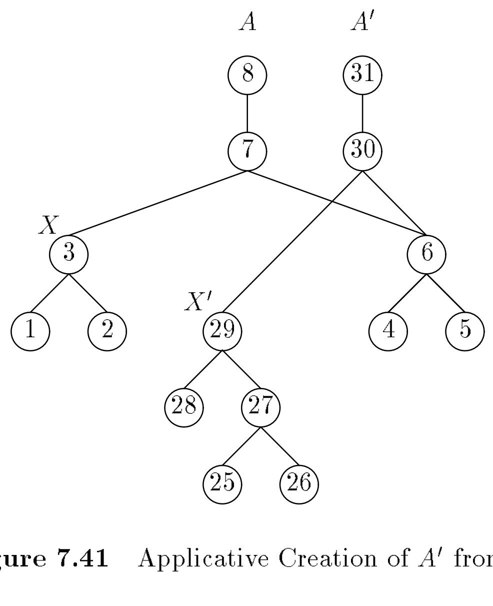


**Figure 7.41** Applicative Creation of $A'$ from $A$

node which represents the syntax node labeled 5, we are able to directly refer to the attribute tree node labeled 5. In addition, since the hierarchical specification describes a simple three module pipeline, the module instance tree for this simple specification will consist of exactly one module instance for each of the modules *Root*, *BtoD*, and *CtoD*. Now consider a syntax change to $A$ that applicatively replaces the subtree labeled $X$ with the entirely new subtree labeled $X'$ as shown in Figure 7.41.

If the original LRTM-k algorithm is applied to construct the new attribute tree for each module instance, then incremental evaluation of $A'$ begins by constructing the new attribute tree $A'_{copy}$, in the process, deallocating the attribute tree nodes labeled 7 and 8 since they were ancestors of the reused attribute subtree whose root is labeled 6. Since attribute node 8 contained the saved state for the instance of module *BtoD*, the deallocation of node 8 means that the instance of module *BtoD*, and transitively, the instance of module *CtoD*, will be evaluated non-incrementally.

If the modified LRTM-k algorithm is applied to construct the new attribute tree for each module instance, then incremental evaluation of $A'$ begins by constructing the new attribute tree $A'_{copy}$, in the process, deallocating the attribute tree nodes labeled 7 and 8 and retaining the saved state for the instance of module *BtoD*. When module *BtoD* is invoked with the $B$-tree $B'$ created by *Root*, the change propagator will reuse the retained saved state to guide incremental evaluation of *BtoD*. The original tree $B$ and the new tree $B'$ are shown in Figure 7.42. Incremental evaluation of $B'$ then begins by constructing the new attribute tree $B'_{copy}$, in the process, deallocating the attribute tree nodes labeled 15 and 16 and retaining the saved state for the instance of module *CtoD*. When module *BtoD* invokes module *CtoD* with the $C$-tree $C'$ created by *BtoD*, the change propagator will reuse the retained saved state to guide incremental evaluation of *CtoD*. The original tree $C$ and the new tree $C'$ are shown in Figure 7.43.

In this example, retaining and reusing the saved state associated with deallocated attribute tree nodes means that the instances of *BtoD* and *CtoD* will be evaluated incrementally.

## 7.7 Summary

This chapter has presented a series of new matching algorithms including:

- the Topological Matching Algorithm (TM) which selects subtrees from $d_{copy}$ to create $d'_{copy}$ during a topological traversal of $d \cup d'$ in $O(|d \cap d'| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ time,

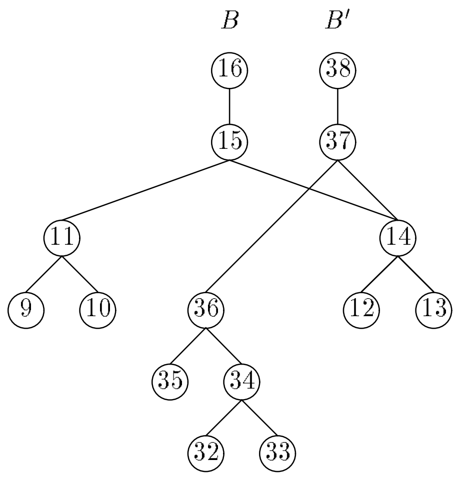


**Figure 7.42** Applicative Creation of $B'$ from $B$ Resulting from Evaluation of *Root*

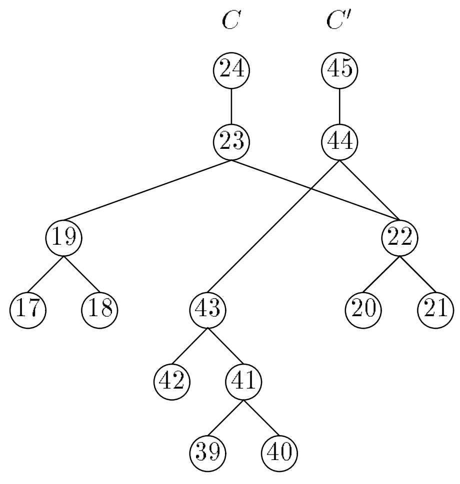


**Figure 7.43** Applicative Creation of $C'$ from $C$ Resulting from Evaluation of *BtoD*

- the Retaining Topological Matching Algorithm (RTM) which selects subtrees from a set of retained attribute subtrees $d^*_{copy}$ to create $d'_{copy}$ during a topological traversal of $d^* \cup d'$ in $O(|d \cap d'| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ amortized time,
- the Lazy Topological Matching Algorithm (LTM) which selects subtrees from $d_{copy}$ to create $d'_{copy}$ during a topological traversal of a modeled region of $d \cup d'$ in $O(|M_{P=N=\emptyset}(d \cap d')| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ time, and
- the Lazy Retaining Topological Matching Algorithm (LRTM) which selects subtrees from a set of retained attribute subtrees $d^*_{copy}$ to create $d'_{copy}$ during a topological traversal of a modeled region of $d^* \cup d'$ in $O(|M_{P=N=\emptyset}(d \cap d')| + |d'_{copy} - d_{copy}| + |d_{copy} - d'_{copy}|)$ amortized time.

A common theme in these four matching algorithms is the aggressive use of space to reduce the time required to evaluate module instances in a hierarchical system. Space is used to store syntax trees, attribute trees, attributes and the various association maps needed to support matching. The extensive space requirements arise from a sequence of deliberate decisions to use space to store previously computed facts that may later prove to be useful. Unfortunately, the decision to use a hash-consed implementation of syntactic values implies that none of this retained information is ever truly "garbage" that could be deallocated by a garbage collector. We, therefore, suggest the use of a more aggressive storage reclamation mechanism that deallocates attribute tree nodes and the attributes and saved states associated with those attribute tree nodes based on a "least recently used" policy. Even though "useful" information may be deallocated under this policy, if that information is needed later, it can be recomputed, although, potentially, at high expense.

# Chapter 8

# Incremental Evaluation: Context-Based Tree-Walking and Applicative Approaches

In this chapter, we present a new incremental evaluation algorithm, referred to as the *Context-Based Ordered Evaluator (CBOE)*, which is designed to take attribute context into account in its selection of old attribute subtrees in an attempt to minimize the cost of evaluating module instances in a hierarchical system. The new algorithm intermingles the process of matching with the process of evaluation so that attribute values will be available to guide the selection of old attribute subtrees. The CBOE algorithm is proved to evaluate entire hierarchical specifications optimally with respect to any evaluation algorithm that explicitly constructs a consistent attribute tree for each module instance and requires that the attribute tree be traversed from its root by following parent and child links between adjacent attribute tree nodes. The chapter concludes by describing the applicative approach to evaluation of hierarchical attribute grammars and by comparing that approach to the CBOE algorithm.

## 8.1 The Context-Based Approach

The context-based evaluator is based on a surprisingly simple modification to the visit-sequence interpreter for the non-incremental ordered attribute grammar evaluator presented in Chapter 2. The new evaluator will perform evaluation over a complete attribute tree consisting of a crown of new attribute tree nodes and a tentatively chosen set of retained attribute subtrees. During evaluation, the crown of new attribute tree nodes will be extended and the choices of retained attribute subtrees will be revised based on computed attribute values. Retained attribute subtrees will be selected in a manner that permits the context-based evaluator to skip visits to those subtrees in order to minimize the cost of evaluating module instances.

### 8.1.1 Applicative Reuse of Retained Attribute Subtrees

In each of the matching algorithms described in the previous chapter, reused attribute subtrees were destructively pruned from retained attribute trees and then grafted into the new attribute tree. The context-based evaluator will always access reused attribute subtrees "read-only," therefore, attribute subtree reuse can be implemented applicatively rather than destructively. To reuse an attribute subtree, the context-based evaluator will simply place a pointer to the subtree into a child field of a new attribute tree node.

Applicative reuse of attribute subtrees significantly simplifies the matching component of the context-based evaluator by permitting attribute subtrees to be selected for reuse without concern for any needs for attribute subtrees that might arise later during the evaluation process, i.e., the reuse of an attribute subtree $s$ will never prevent a larger attribute subtree $t$, which contains subtree $s$, from being reused elsewhere in the new attribute tree. Although applicative reuse of old attribute subtrees implies that attribute trees will not necessarily be tree-structured, we will continue to use the term "attribute tree" to refer to these structures.

### 8.1.2 Identification of Appropriate Subtrees

The context-based evaluator relies on the ability to map a "context" consisting of a triple $\langle n,\ visitNum,\ valList\rangle$, where $n$ is the root of a syntax term, $visitNum$ is an integer in the range 1 to $NumVisits(phylum(n))$, and $valList$ is an ordered sequence of inherited attribute values of length $|AvailInhAttrs(phylum(n), visitNum)|$, into an "appropriate" attribute subtree $t$ such that $t.dagnode == n$, and the values of the inherited attributes in the sequence $AvailInhAttrs(phylum(n),\ visitNum)$ of $root(t)$ are identical to the values in $valList$.

Unsurprisingly, an association map containing entries of the form

$$Context \rightarrow AttrNode$$

can be used to locate an attribute subtree that is appropriate for a given context.

All access to retained attribute subtrees will be provided by this association map. Entries will be inserted into the association map by invoking the procedure *Enter* with a map, a context, and an appropriate attribute subtree for that context. The function *Lookup* will be used to retrieve appropriate attribute subtrees when invoked with a map and a context.

Instead of maintaining an association map for each module instance, we will maintain an association map for each module, thereby making all of the attribute subtrees attributed by an instance of a module be available for use during incremental evaluation of any instance of that module. Since old attribute subtrees are reused applicatively, increasing the size of the database of retained attribute trees by maintaining a single database for each module will serve to decrease the time required to update module instances throughout a hierarchical system. The notation *Map(Module)* will refer to the association map for a module *Module*.

### 8.1.3 Matching with Evaluation

The context-based evaluator constructs an initial attribute tree representing an input term $d'$ consisting of a set of new attribute tree nodes, a set of interface nodes, and a set of retained attribute subtrees. The new attribute tree nodes will be identified by a boolean field *new* which will be set to *true* when a new attribute tree node is created. The new attribute tree nodes will form a crown of the initial attribute tree. The fringe of the new crown is a set of interface nodes, each of which contains most of the fields typically associated with an attribute tree node, including the fields *parent*, *dagnode*, and *operator*, and, in addition, contains a field *AppropSubtree* which will reference an attribute subtree that has been identified as being appropriate by using the association map for the invoked module. Each interface node contains a set of inherited attributes, but not synthesized attributes. An attempt to retrieve the value of synthesized attribute $a$ of interface node $f$ will, instead, return the value of synthesized attribute $a$ of the root of the subtree referenced by $f.ApropSubtree$.

The initial attribute tree will be constructed during a depth-first traversal over a crown of $d'$. At each node $n$ in $d'$, the function *Lookup* will be invoked to locate a retained attribute subtree that is appropriate for the context $\langle n,\ 0,\ \{\}\rangle$. An attribute subtree $t$ is appropriate for the context $\langle n,\ 0,\ \{\}\rangle$ if $t.dagnode == n$, i.e., if the syntax subterm represented by $t$ is rooted by node $n$. If the *Lookup* operation returns the value NULL, then a new attribute tree node is inserted into the crown of the new attribute tree. If the *Lookup* operation returns an attribute subtree $t$, then an interface node whose *ApropSubtree* field is initialized to $t$ is inserted into the crown of the new attribute tree.

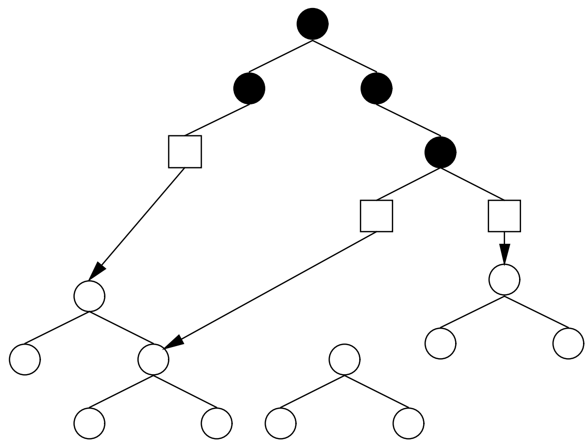

**Figure 8.1** An Initial Attribute Tree

Figure 8.1 depicts an initial attribute tree for an unspecified input term and an unspecified module. Filled circles represent new attribute tree nodes, boxes represent interface nodes, and unfilled circles represent old attribute tree nodes.

The visit-sequence interpreter for the context-based evaluator will be invoked after the initial attribute tree has been constructed. The interpreter will execute EVAL(i,a) instructions and VISIT-PARENT(r) instructions exactly as in the batch ordered evaluator. VISIT(i,r) instructions will be performed exactly as before if the node to be visited is a new attribute tree node. If the node to be visited by a VISIT(i,r) instruction is an interface node $f$, then an attempt will be made to identify an attribute subtree that is appropriate for the $r$th visit to $f$ by invoking the function *Lookup* with a context $\langle$*f.dagnode, r, valList*$\rangle$, where *valList* contains the values of the inherited attributes in *AvailInhAttrs(phylum(f),r)* of interface node $f$. If an appropriate subtree $t$ can be found for the $r$th visit to $f$ then the visit will be skipped, and $t$ will be attached to the new attribute tree. If an appropriate attribute subtree cannot be located for the $r$th visit to $f$, then interface node $f$ will be expanded into a new attribute tree node by replacing $f$ with a new attribute tree node that has interface nodes for each of its children, and the new attribute tree node will be visited. Visit-sequence interpretation terminates when the final VISIT-PARENT(r) instruction is executed at the root of the attribute tree.

As a final step after visit-sequence interpretation has completed, the context-based evaluator will traverse the crown of new attribute tree nodes in the attribute tree, setting the *new* flag on each of the nodes to *false*, and invoking procedure *Enter* to insert all of the new $Context \rightarrow AttrNode$ entries into the association map for the invoked module.

### 8.1.4 The CBOE Algorithm

Function *HagEvaluate*, shown in Figure 8.2, first initializes *NewRoots* to the empty set, and then invokes function *Evaluate* to apply the root module to an input term using the association map for the root module. Each time *Evaluate* is invoked to evaluate a module instance, it will insert a pair containing the root of the new attribute tree for that module instance and the name of the module into the set *NewRoots*. When control returns to *HagEvaluate*, procedure *EnterAll* will be invoked with the set *NewRoots* to insert new entries into each of the association maps for each module in the hierarchical specification.

Function *Evaluate*, shown in Figure 8.3, is invoked with the name of a module, the syntax term $d'$, and the set *NewRoots*. *Evaluate* first retrieves the association map for *Module* and then invokes function *BuildInitial* to construct the initial attribute tree representing $d'$. The tree built by *BuildInitial* is then stored into *NewAttrTree*. If the root of *NewAttrTree* is a new attribute tree node, then the function *MatchEval* is invoked to evaluate the new initial attribute tree. On completion of *MatchEval*, a pair $\langle$*root(NewAttrTree),Module*$\rangle$ is inserted into *NewRoots*. Finally, the value of the result attribute associated with the root of *NewAttrTree* is returned as the result of evaluating the term $d'$.

Function *BuildInitial*, shown in Figure 8.4, builds the initial attribute tree during a depth-first traversal of a crown of $d'$. First, *BuildInitial* allocates a new attribute tree node. It then invokes the recursive procedure *BuildInitialRec* for each of the children of the root of $d'$. Whenever *BuildInitialRec*, also shown in Figure 8.4, is unable to locate an appropriate subtree for a node $n$ in $d'$, it allocates a new attribute tree node, inserts the new node into the crown of the initial attribute tree, and then recursively invokes itself with each of the children of $n$. Whenever *BuildInitialRec* is able to locate an appropriate subtree for a syntax node $n$ in $d'$, it allocates an interface node, inserts it into the crown of the initial attribute tree, and then assigns the subtree to the *ApropSubtree* field of the interface node.

```
Function HagEvaluate(d')
  NewRoots = {}
  result = Evaluate(Root,d',NewRoots)
  EnterAll(NewRoots)
  return result
end
```

**Figure 8.2** HagEvaluate – CBOE Algorithm

```
Function Evaluate(Module, d′, NewRoots)
  Map = MapFor(Module)

  NewAttrTree = BuildInitial(d′,Map)
  if not(new(root(NewAttrTree)))
    MatchEval(root(NewAttrTree),Map)
    insert ⟨root(NewAttrTree),Module⟩ into NewRoots

  return ResultAttr(root(NewAttrTree))
end
```

**Figure 8.3** Evaluate – CBOE Algorithm

```
Function BuildInitial(d′, Map)
  NewAttrRoot = CreateAttrNode(root(d′))
  newAttrRoot.new = true
  for each child i of root(d′)
    BuildInitialRec(root(d′)[i],NewAttrRoot,i,Map)
  return NewAttrRoot
end

Procedure BuildInitialRec(dagnode,ParentAttrNode,i,Map)
  AppropSubtree = Lookup(Map,0,{})
  if AppropSubtree == NULL
  then
    – insert a new attribute tree node
    newAttrNode = CreateAttrNode(dagnode)
    ParentAttrNode[i] = newAttrNode
    newAttrNode.parent = ParentAttrNode
    newAttrNode.son = i
    newAttrNode.new = true
    – – recurse
    for each child i of dagnode
      BuildInitialRec(dagnode[i],newAttrNode,i,Map)
  else
    – – found an appropriate subtree
    newInterfaceNode = CreateInterfaceNode(dagnode)
    ParentAttrNode[i] = newInterfaceNode
    newInterfaceNode.parent = ParentAttrNode
    newInterfaceNode.son = i
    newInterfaceNode.AppropSubtree = AppropSubtree
  fi
end
```

**Figure 8.4** BuildInitial and BuildInitialRec – CBOE Algorithm

```
Procedure MatchEval(AttrRoot,Map)
 AttrNode = AttrRoot
 index = 1
 forever do
   if plan[AttrNode.op][index] is EVAL(i,a)
     args = ArgList(AttrNode,i,a)
     AttrNode[i].a = apply semfunc[AttrNode.op,i,a] to args
     index = index + 1
   else if plan[AttrNode.op][index] is VISIT(i,r)
     if not(isInterface(AttrNode[i]))
     then – – we are visiting an attribute tree node in the crown of d'_copy
       index = MapVisitToIndex(AttrNode[i].op,0,r)
       AttrNode = AttrNode[i]
     else – – attempt to locate an appropriate subtree for this interface node
       AttrVals = retrieve AvailInhAttrs(phylum(Attrnode[i]),r) from AttrNode[i]
       ApropSubtree = Lookup(Map,⟨AttrNode[i].dagnode,r,AttrVals⟩)

       if ApropSubtree != NULL
       then
         Install(ApropSubtree,AttrNode,i,r)
         – – skip the visit
         index = index + 1
       else – – no appropriate subtree found, so expand the tree and do the visit
         Expand(AttrNode[i],r)
         index = MapVisitToIndex(AttrNode[i].op,0,r)
         AttrNode = AttrNode[i]
       fi
     fi
   elseif plan[AttrNode.op][index] is VISIT-PARENT(r)
     if AttrNode == AttrRoot
       return

     index = MapVisitToIndex(AttrNode.parent.op,AttrNode.son,r)
     AttrNode = AttrNode.parent
   fi
 od
end
```

**Figure 8.5** MatchEval – CBOE Algorithm

Procedure *MatchEval*, shown in Figure 8.5, is the visit-sequence interpreter for the context-based evaluator. *MatchEval* restricts its traversal to the crown of new nodes in the new attribute tree. The crown of new nodes expands as *MatchEval* interprets visit-sequences. *MatchEval* executes EVAL, VISIT and VISIT-PARENT instructions in the visit-sequence for an attribute tree node *AttrNode* using the following procedures:

- Each EVAL(i,a) instruction is executed exactly as in the batch ordered evaluator.
- Each VISIT(i,r) instruction whose target is an attribute tree node in the crown of the new attribute tree is executed unconditionally. If the target of the VISIT instruction is an interface node, the inherited attribute values that are available for the $r$th visit to the interface node will retrieved from the interface node and function *Lookup* will be invoked in an attempt to identify an old attribute subtree that is appropriate for the $r$th visit to the interface node. If an appropriate attribute subtree is located by *Lookup*, then procedure *Install* will be invoked to attach the subtree to the new attribute tree, and then the visit will be skipped. If an appropriate subtree cannot be located by *Lookup*, then the interface node will be expanded into a new attribute tree with interface nodes for its children by invoking *Expand*. The new attribute tree node is then visited.
- Each VISIT-PARENT(i,a) instruction is executed exactly as in the batch ordered evaluator.

Procedure *Install*, shown in Figure 8.6, is invoked with a retained attribute subtree *ApropSubtree*, an attribute tree node *AttrNode*, a child index $i$, and a visit number $r$, and is responsible for attaching the subtree *ApropSubtree* to the new attribute tree. Whenever *Install* is invoked, *AttrNode[i]* contains a reference to an interface node. If the $r$th visit to *AttrNode[i]* is not the final visit to the interface node, then *ApropSubtree* will be assigned to the *ApropSubtree* field of the interface node. If the $r$th visit is the final visit to the interface node, then the interface node will be deallocated, and *ApropSubtree* will be inserted in its place. In this manner, a retained attribute subtree that is appropriate for the last visit to an interface node becomes attached to a new attribute tree node without an intervening interface node.

Procedure *Expand*, shown in Figure 8.7, is invoked whenever an appropriate attribute subtree cannot be located for an interface node. It is called with an interface node *Interface* and a visit number $r$ and is responsible for allocating a new attribute tree node containing interface nodes for each of its children. After allocating the new attribute tree node, *Expand* copies the inherited attributes from *Interface* and the synthesized attributes of *Interface.ApropSubtree* into the new attribute tree node. *Expand* then allocates a new interface node for each child of the new attribute tree node, inserts each of the new interface nodes into the new attribute tree node, and then copies the inherited attributes of the $i$th child of *Interface.ApropSubtree* into the $i$th new interface node. In addition, *Expand* assigns the $i$th child of *Interface.ApropSubtree* into the *ApropSubtree* field of the $i$th new interface node. Finally, *Expand* deallocates *Interface*.

Procedure *EnterAll*, shown in Figure 8.8, invokes procedure *EnterNewNode*, also shown in Figure 8.8, to perform a depth-first traversal of each of the crowns of new attribute trees built during evaluation of

```
Procedure Install(ApropSubtree,AttrNode,i,r)
  – – AttrNode[i] is an interface node
  if r != NumVisits(phylum(AttrNode[i]))
  then
    AttrNode[i].ApropSubtree = ApropSubtree
  else
    – – remove the attribute tree node from the tree, since ApropSubtree
    – – fits exactly into the new attribute tree
    Dealloc(AttrNode[i])
    AttrNode[i] = ApropSubtree
  fi
end
```

**Figure 8.6** Install – CBOE Algorithm

```
Procedure Expand(Interface,r)
  AttrNode = CreateAttrNode(Interface.dagnode)
  AttrNode.new = true
  AttrNode.parent = Interface.parent
  AttrNode.son = Interface.son

  copy the inherited attributes of Interface into AttrNode
  copy the synthesized attributes of Interface.AppropSubtree into AttrNode

  for each child i of AttrNode
    newInterface = CreateInterface(AttrNode.dagnode[i])
    AttrNode[i] = newInterface
    newInterface.parent = AttrNode
    newInterface.son = i
    copy the inherited attributes of Interface.AppropSubtree[i] to newInterface
    newInterface.AppropSubtree = Interface.AppropSubtree[i]

  Dealloc(Interface)
end
```

**Figure 8.7** Expand – CBOE Algorithm

```
Procedure EnterAll(NewRoots)
  for each pair ⟨AttrRoot,Module⟩ in NewRoots
    EnterNewNodes(AttrRoot,Module)
end

Procedure EnterNewNode(AttrNode,Module)
  if not(AttrNode.new)
    return

  AttrNode.new = false

  phy = phylum(AttrNode)
  for each visit number i in 1 to |NumVisits(phy)|
    AttrVals = retrieve AvailInhAttrs(phy,i) from AttrNode
    Enter(Map(Module),⟨AttrNode.dagnode,i,AttrVals⟩, AttrNode)

  for each child i of AttrNode
    EnterNewNode(AttrNode[i],Module)
end
```

**Figure 8.8** EnterAll and EnterNewNode – CBOE Algorithm the hierarchical attribute grammar. For each new attribute tree node *AttrNode* and for each visit number $i$, *EnterNewNode* constructs a context $\langle$*AttrNode.dagnode,i,AttrVals*$\rangle$, where *AttrVals* contains the values of the inherited attributes in the sequence *AvailInhAttrs(phylum(AttrNode),i)* and then invokes *Enter* to insert this context and *AttrNode* into the association map for $Module$.

### 8.1.5 Proofs for the CBOE Algorithm

We now prove that the CBOE algorithm correctly updates a complete hierarchical specification and does so optimally with respect to any evaluation algorithm that explicitly constructs a consistent attribute tree for each module instance and requires that the attribute tree be traversed from its root by following parent and child links between adjacent attribute tree nodes.

The proof of correctness of the CBOE algorithm requires that we be able to make claims about the values of attributes throughout the attribute tree being constructed and evaluated by the context-based evaluator. An abstract view of attribute trees must be adopted, though, since the actual nodes in the attribute tree may change as the result of assigning a retained attribute subtree to the *AppropSubtree* field of an interface node or as a result of expanding an interface node into a new attribute tree node having new interface nodes for each of their children. Each attribute, therefore, will be considered to be a pair $\langle$*Path,Attr*$\rangle$ where *Path* is a sequence of child indices leading from the root of the attribute tree to a specific attribute tree node, and *Attr* is an attribute associated with that node. An interface node $f$ and the root of the subtree referenced by $f$.*AppropSubtree* will be abstractly treated as a single attribute tree node, i.e., if the node reached via *Path* is an interface node $f$ and *Attr* is an inherited attribute, then the actual value of the attribute can be found in $f$, otherwise, if *Attr* is a synthesized attribute, then the actual value of the attribute can be found in the root of $f$.*AppropSubtree*.

Instead of referring to the operations EVAL, VISIT and VISIT-PARENT as instructions, it is useful to define an instruction to be a pair $\langle$*Path, index*$\rangle$, where *Path* identifies an attribute tree node in the new attribute tree, and *index* is the index of an operation in the visit sequence for the attribute tree node identified by *Path*. The function *PathTo* will be used below to convert an attribute tree node into a path to that node.

To prove that the CBOE algorithm is correct we will prove that *Evaluate* correctly updates each module in a hierarchical specification.

> **Theorem 8.1** Function *Evaluate* of the CBOE algorithm will correctly compute the result of applying a module $Module$ to a term $d'$.

**Proof**

Let $ConsBatch(Inst)$ be the set of attributes in an attribute tree being evaluated by the batch evaluator that are guaranteed to be consistent just before executing $Inst$, and let $ConsInc(Inst)$ be the set of attributes in an attribute tree being evaluated by the context-based evaluator that are guaranteed to be consistent just before executing $Inst$.

To prove Theorem 8.1 we will demonstrate that the following invariant is satisfied after each EVAL, VISIT and VISIT-PARENT operation is executed by the context-based evaluator:

> If the set of attributes in $ConsBatch(Inst)$ is identical to the set of attributes $ConsInc(Inst)$, and if instruction $Inst'$ is the instruction that will be evaluated by the context-based evaluator after executing instruction $Inst$, then $ConsBatch(Inst')$ will be identical to $ConsInc(Inst')$.

Satisfaction of this invariant implies that the result attribute associated with the root of the new attribute tree will be assigned a consistent value, and, therefore, *Evaluate* will return the correct value. Notice that the instruction that will be executed by the batch evaluator after executing instruction $Inst$ will not necessarily be the same as the instruction that will be executed by the incremental evaluator after executing instruction $Inst$, since the incremental evaluator may skip entire visits to attribute subtrees.

We now examine the processing of operations by the batch evaluator and the context-based evaluator for an instruction $Inst = \langle$*PathTo(AttrNode),index*$\rangle$ under the assumption that $ConsBatch(Inst) == ConsInc(Inst)$.

- If $Inst$ = EVAL(i,a).

  The instruction to be executed after $Inst$ by both evaluators will be

  $$Inst' = \langle \text{PathTo(AttrNode),index+1} \rangle.$$

  Both the batch evaluator and the context-based evaluator will evaluate attribute $a$ of *AttrNode[i]*, hence

  $$ConsBatch(Inst') = ConsBatch(Inst) \cup \langle PathTo(AttrNode[i]), a \rangle$$

  and

  $$ConsInc(Inst') = ConsInc(Inst) \cup \langle PathTo(AttrNode[i]), a \rangle.$$

  Therefore,

  $$ConsBatch(Inst') == ConsInc(Inst').$$

- If $Inst$ = VISIT(i,r) and if *AttrNode[i]* is a new attribute tree node, then the instruction to be executed after $Inst$ by both evaluators will be

  $$Inst' = \langle PathTo(AttrNode[i]), MapVisitToIndex(AttrNode[i].op, 0, r) \rangle.$$

  No attributes are evaluated by either evaluator when executing a VISIT(i,r) operation. Therefore,

  $$ConsBatch(Inst') == ConsInc(Inst').$$

- If $Inst$ = VISIT(i,r) and if *AttrNode[i]* is an interface node for which an appropriate retained subtree can be found, then the visit will be skipped, and, therefore, the instruction to be executed after $Inst$ by the context-based evaluator will be

  $$Inst' = \langle PathTo(AttrNode[i]), index+1 \rangle.$$

  Let *oldRoot* be the root of the attribute subtree that was last referenced by the *ApropSubtree* field of *AttrNode[i]* and which must, therefore, be appropriate for the $r-1$ visit to the interface node *AttrNode[i]*, and let *newRoot* be the new attribute subtree that is appropriate for the $r$th visit to *AttrNode[i]*. *AttrNode[i].dagnode*, *oldRoot.dagnode* and *newRoot.dagnode* must have the same value, say *dagnode*.

  By the time that the batch evaluator reaches instruction $Inst'$, it will have computed consistent values for every attribute in *EvalWithVisit(dagnode,r)*, the set of attributes in any attribute subtree representing the syntax subterm rooted by node *dagnode* that will be assigned consistent values during the $r$th visit to its root. Therefore,

  $$ConsBatch(Inst') = ConsBatch(Inst) \cup EvalWithVisit(dagnode, r).$$

  Since *oldRoot* was appropriate for the $r-1$ visit to *AttrNode[i]*, every attribute of the subtree rooted by *oldRoot* in $\bigcup_{1 \le j \le r-1} EvalWithVisit(dagnode, j)$ must be consistent, and since *newRoot* is appropriate for the $r$th visit to *AttrNode[i]*, every attribute in $\bigcup_{1 \le j \le r} EvalWithVisit(dagnode, j)$ of the subtree rooted by *newRoot* must be consistent. Therefore, as a result of replacing *oldApprop* with *newApprop*,

  $$\begin{aligned} ConsInc(Inst') &= (ConsInc(Inst) - \bigcup_{1 \le j \le r-1} EvalWithVisit(dagnode, j)) \\ &\quad \cup \bigcup_{1 \le j \le r} EvalWithVisit(dagnode, j), \end{aligned}$$

  which can be simplified to

  $$ConsInc(Inst') = ConsInc(Inst) \cup EvalWithVisit(dagnode, r),$$

  and, therefore,

  $$ConsBatch(Inst') == ConsInc(Inst').$$

- If $Inst$ = VISIT(i,r) and if $AttrNode[i]$ is an interface node for which an appropriate retained subtree cannot be found, the instruction to be executed by both evaluators will be

  $$Inst' = \langle PathTo(AttrNode[i]), MapVisitToIndex(AttrNode[i].op,0,r)\rangle.$$

  The context-based evaluator, however, will first expand the interface node $AttrNode$ into a new attribute tree node with interface nodes for each of its children by invoking the procedure *Expand*. No attributes are modified by the batch evaluator or the context-based evaluator. Therefore,

  $$ConsBatch(Inst') == ConsInc(Inst').$$

- If $Inst$ = VISIT-PARENT(r) and $AttrNode$ is not the root of the new attribute tree, then the next instruction to be executed by both evaluators will be

  $$Inst' = \langle PathTo(AttrNode.parent), \; MapVisitToIndex(AttrNode.parent.op,\ AttrNode.son,\ r)\rangle.$$

  No attributes are evaluated by either evaluator during the execution of a VISIT-PARENT(r) operation. Therefore,

  $$ConsBatch(Inst') == ConsInc(Inst').$$

- If $Inst$ = VISIT-PARENT(r) and if $AttrNode$ is the root of the new attribute tree, then both evaluators are done, and there is no next instruction $Inst'$.

Since, each of the attributes in the new attribute tree constructed by the context-based evaluator will have consistent values once the final VISIT-PARENT instruction has been executed at the root of the new attribute tree, the attribute result returned by function *Evaluate* will be correct. □

**Theorem 8.2** The CBOE algorithm evaluates hierarchical specifications optimally under the restriction that the incremental evaluation algorithm must visit all of the attribute nodes on paths from the root of the new attribute tree to nodes with attributes in AFFECTED in order to update all of the inconsistent attributes in the new attribute tree.

**Proof**

Since an incremental evaluator must traverse the crown of new attribute tree nodes reaching from the root of the attribute tree to each node with an attribute in AFFECTED, we will refer to the set of attribute tree nodes that will be traversed by the incremental evaluator as the AFFCROWN. The nodes in the attribute tree that are not traversed by the incremental evaluator will be referred to as the $\overline{\text{AFFCROWN}}$ set. An attribute tree node is in $\overline{\text{AFFCROWN}}$ if and only if all of its attributes are in RETAINED.

Since an incremental evaluator is required to traverse all of the nodes in the set AFFCROWN, if only a constant amount of work is required at each attribute tree node in AFFCROWN, then minimizing the cost of updating a module instance will require selecting old attribute subtrees in a manner that minimizes the size of AFFCROWN, or, equivalently, maximizing the size of $\overline{\text{AFFCROWN}}$. Let AFFCROWN$'$ be the set of attribute tree nodes in the minimum-sized AFFCROWN.

Under the assumption that entries can be inserted and retrieved from the association map in constant time, the context-based evaluator can be shown to perform only a constant amount of work at each node in AFFCROWN and at the fringe of AFFCROWN. Since each of the productions in the phylum/operator grammar underlying each module specification has a fixed number of children, evaluation of a module by the context-based evaluator requires $O(|\text{AFFCROWN}|)$ time. We now argue that the context-based evaluator only requires $O(|\text{AFFCROWN}'|)$ time.

The goal of the entire process of selecting appropriate attribute subtrees in the context-based evaluator is to ensure that the minimum possible number of new attribute tree nodes are created in the new attribute tree since AFFCROWN is just the set of new attribute tree nodes. The number of new attribute tree nodes is minimized by ensuring that new attribute tree nodes are inserted into the new attribute tree only when it can be guaranteed that no retained attribute subtree could be used instead. New nodes are inserted into the new tree by *BuildInitial* and by *Expand*. *BuildInitial* inserts a new attribute tree node *AttrNode* only if there does not exist a retained attribute subtree that represents the syntax subterm rooted by *AttrNode.dagnode*. *Expand* inserts a new attribute tree only after *Lookup* has failed to find an appropriate retained attribute subtree for an interface node. Failure to identify an appropriate attribute subtree means that no retained attribute subtree can be used to represent that interface node without needing to recompute some of its attributes, and therefore, the new attribute tree node represents a node that must be in AFFCROWN. Therefore, the size of AFFCROWN is minimized, and the cost of evaluation by the context-based evaluator is $O(|\text{AFFCROWN}'|)$.

Since each module instance that is invoked during evaluation of a hierarchical system will be updated optimally, and since applicative reuse of retained attribute subtrees guarantees that the evaluation of a module instance can not interfere with the evaluation of another module instance, the complete hierarchical system will be updated optimally. □

The cost of applying the CBOE algorithm to apply a hierarchical specification to a modified input term is the sum of the costs of evaluating each of the module instances in AFF-MOD-INSTS, where AFF-MOD-INSTS is the set of module instances that are invoked by the incremental evaluator during evaluation of the modified input term. The total cost of evaluation is, therefore, $O(\sum_{aff \in \text{AFF-MOD-INSTS}} |\text{AFFCROWN}'_{aff}|)$.

### 8.1.6 A Simple Improvement to the CBOE Algorithm

As described above, new entries are inserted into the association map for each module in a hierarchical system by invoking procedure *EnterAll* only after all of the modules instances in the hierarchical system have been incrementally updated. More effective use of previously computed attribute values can be made by inserting a new entry into the map whenever a visit to a new attribute tree node completes. When the $k$th visit to a new attribute tree node *AttrNode* completes, an entry should be inserted into the map for context ⟨*AttrNode.dagnode, k, valList*⟩, with *valList = AvailInhAttrs(phylum(AttrNode),k)*, and attribute tree node *AttrNode*. As soon as this new map entry is inserted into the association map, the attribute subtree rooted by *AttrNode* becomes available for use as a retained attribute subtree that is appropriate for the $k$th visit to an interface node.

## 8.2 The Applicative Approach

We now describe the applicative approach to incremental evaluation of hierarchical attribute grammars and compare it to the context-based evaluator.

### 8.2.1 The Approach

The applicative approach to incremental evaluation of non-hierarchical attribute grammars employs a function cache to store the database of previously computed attribute values, with reuse occurring whenever a function is invoked and a matching tuple is retrieved from the function cache. The same approach can be applied to incremental evaluation of hierarchical attribute grammars by retaining the contents of the function cache across a series of executions of the functional program constructed for a hierarchical attribute grammar as described in Chapter 5.

The cost of applying the applicative evaluator to apply a hierarchical specification to a modified input term is the sum of the costs of evaluating each of the module instances in AFF-MOD-INSTS, where AFF-MOD-INSTS is the set of module instances that are invoked by the incremental evaluator during evaluation of the modified input term. Let $TREE(S_{aff})$, for $aff \in \text{AFF-MOD-INSTS}$, be an attribute tree representing the syntax term $S_{aff}$ that is the input to module instance $aff$. The cost of updating a single module instance $aff \in \text{AFF-MOD-INSTS}$ is $O(|\text{AFFECTED}_{aff}| + |\text{PATHS}_{aff}|)$, where $\text{AFFECTED}_{aff}$ is the set of attributes associated with nodes in $TREE(S_{aff})$ that must be assigned new attribute values during evaluation of module instance *aff*, and $\text{PATHS}_{aff}$ is a set of nodes forming a crown of $TREE(S_{aff})$. The next section attempts to provide a basis for comparing this cryptic running time to the running time of the context-based evaluator.

### 8.2.2 Comparison of Context-Based and Applicative Approaches

Although the essentially data-driven context-based incremental evaluator may seem quite different than the demand-driven applicative evaluator, in essence, the two approaches are quite similar in the way they identify and reuse previously computed attribute values. In some sense, the context-based evaluator acts as a bridge between the well-understood tree-walking evaluation algorithms and the applicative algorithms based on function caching.

The context-based evaluator evaluates attributes associated with nodes in a crown of the new attribute tree and skips visits to those subtrees for which old attribute subtrees can be tentatively selected based on the values of inherited attributes associated with the roots of those subtrees. As discussed above, the context-based evaluator minimizes the size of the crown of nodes traversed by the context-based evaluator during the evaluation of a module instance. Therefore, the size of the crown of nodes traversed by the context-based evaluator during the evaluation of module instance *aff* is $|\text{AFFCROWN}'_{aff}|$.

It is relatively easy to correlate the function evaluations performed by the applicative evaluator during evaluation of a term $d$ with the nodes of an attribute tree representing $d$, even though the attribute tree is not actually constructed. Each time a function *M–N–s* is invoked to compute the value of a synthesized attribute $s$ associated with a nonterminal $N$ in a module $M$, its first argument will be a subterm *T[i]* of the term $T$ passed as the first argument to the invoking function *M–O–t*, the function responsible for computing the value of attribute $t$ of nonterminal $O$ in module $M$. The invocation of a function computing a synthesized attribute value with a term *T[i]* can be considered to be a visit to the $i$th child of the current attribute tree node, and the return from that function can be considered to be the visit back to the original attribute tree node. Therefore, the applicative evaluator can be seen as evaluating attributes associated with a set of nodes in the crown of an attribute tree, skipping visits to attribute tree nodes whenever the function cache can be used to retrieve the value of a previously computed synthesized attribute associated with those nodes. In the notation used above, the crown of the attribute tree traversed by the applicative evaluator when applied to evaluate a module instance *aff* has size $|\text{PATHS}_{aff}|$.

Consider the entry in the function cache for a function *M–N–s*. The function *M–N–s* takes as arguments a syntax subterm and a set of inherited attributes which comprise the argument selector for inherited attribute $s$. An inherited attribute *inh* is in the argument selector for $s$ if there exists some term $T'$, such that the value of *inh* is needed in computing the value of $s$. The entry in the function cache can be used to determine the value of attribute $s$ given an invocation of *M–N–s* with a specific syntax subterm, and specific values for each of the inherited attributes in the argument selector for $s$.

Now consider an entry *Context* $\rightarrow$ *AttrNode* in the context-based evaluator's association map for a module, where *Context* is $\langle$*AttrNode.dagnode, k, valList*$\rangle$, with *valList* the list of values of the attributes in *AvailInhAttrs(phylum(AttrNode),k)*. This entry can be used to locate an attribute subtree whose root is annotated with consistent values for each of the synthesized attributes in *AvailSynAttrs(phylum(AttrNode),k)* given the values of the inherited attributes in *AvailInhAttrs(phylum(AttrNode),k)*.

```
...
N ::= M(S S)
  N.S1 = N.I1 * N.I1
  N.S2 = N.I2 * N.I2
...
```

**Figure 8.9** A Fragment of a Simple Specification

We now present a simple example demonstrating that, in general, the set $\text{PATHS}_{aff}$ will be smaller than the set $\text{AFFCROWN}'_{aff}$. Figure 8.9 presents a single production from a larger specification in which the semantic functions for an operator $M$ define the value of a synthesized attribute *S1* to be the square of the value of inherited attribute *I1*, and the value of a synthesized attribute *S2* to be the square of the value of inherited attribute *I2*. Assume that *AvailInhAttrs(N,1)* contains *N.I1* and *N.I2*, and *AvailSynAttrs(N,1)* contains *N.S1* and *N.S2*.

Figure 8.10 presents a pair of retained attribute subtrees, *R1* and *R2*, whose roots are both labeled with operator $M$, with *R1.dagnode == R2.dagnode == dagnode*. Now consider what happens when the context-based evaluator attempts to locate an attribute subtree that is appropriate for the first visit to an interface


**Figure 8.10** A Pair of Retained Attribute Subtrees

node $f$ with $f.dagnode == R1.dagnode == dagnode$, and with $I1 == 3$ and $I2 == 2$. Neither $R1$ nor $R2$ is appropriate for $f$, and, therefore, the interface node $f$ must be expanded to create a new attribute tree node. This, of course, is the correct action for the context-based evaluator, since the evaluator will use the retained attribute subtrees that it finds to construct the new consistent attribute tree, and since neither $R1$ nor $R2$ can be used in the new tree without updating some of its attributes.

The applicative evaluator, however, when applied to this same problem, initially contain a function cache containing the set of tuples:

$$\begin{gathered}\langle \text{dagnode, "N.S1"}, 1\rangle \rightarrow 1,\\ \langle \text{dagnode, "N.S2"}, 2\rangle \rightarrow 4,\\ \langle \text{dagnode, "N.S1"}, 3\rangle \rightarrow 9,\\ \langle \text{dagnode, "N.S2"}, 4\rangle \rightarrow 25.\end{gathered}$$

During evaluation, the function cache will be queried to find an entry for the tuple ⟨R1.dagnode, "N.S1", 3⟩ and to then find an entry for ⟨R1.dagnode, "N.S2", 2⟩. The function cache will return the results 9 and 4, respectively, for these two queries. The function caching algorithm, therefore, is able to skip visits to attribute subtrees that the context-based evaluator must visit. Therefore, the set $\text{PATHS}_{aff}$ will be smaller than the set $\text{AFFCROWN}'_{aff}$.

## 8.3 Summary

This chapter has described the Context-Based Ordered Evaluation algorithm, and has proven that the algorithm evaluates entire hierarchical attribute grammar specifications optimally under the assumption that an incremental evaluator for modules of a hierarchical specification must traverse all of the nodes in the new attribute tree on paths from the root of the tree to each attribute tree node having an inconsistent attribute. The applicative approach to incremental evaluation of hierarchical attribute grammars was then presented and then shown to outperform the context-based evaluator.

# Chapter 9

# Conclusions

This dissertation has examined *Hierarchical Attribute Grammars*, a new class of attribute grammar dialects that supports composition of modules specified by attribute grammars. Composition of independently specified modules enables hierarchical attribute grammars to be applied to the description of complex language translators such as source-to-source compilers, optimizers and parallelizers. When coupled with an efficient batch or incremental evaluator, hierarchical attribute grammars provide a practical means for implementing these complex tools. We now present an outline of the dissertation and a list of contributions, and then describe future work.

## 9.1 An Overview of the Work

Chapter 2 reviewed the attribute grammar formalism and described both tree-walking and applicative approaches to batch and incremental evaluation.

Chapter 3 introduced hierarchical attribute grammars by first presenting two of Knuth's early informal applications of the hierarchical approach; the first, a pipelined specification of the meaning of Turningol programs, and the second, a recursive specification of a simple lambda expression language.

A set of attribute grammar dialects that we consider to be hierarchical was then surveyed. Each of these dialects takes advantage of the notion of constructing complex systems by composing a set of non-procedurally specified modules. The set of dialects includes Schulz's attributed transformations, the Attribute Coupled Grammars of Ganzinger and Giegerich, SSL, the specification language of the Synthesizer Generator of Reps and Teitelbaum, and the Higher Order Attribute Grammars of Vogt, Swierstra and Kuiper. Based on the deficiencies in the surveyed dialects, a new dialect, *Modular Attribute Grammars*, was introduced. The Modular Attribute Grammar dialect is based on an improved notion of annotated abstract syntax that we believe to be suitable for describing a collection of independently specified modules.

In order to give a flavor of the use of Modular Attribute Grammars in the description of real language processing systems, Chapter 4 presented a set of Modular Attribute Grammar specifications of modules of two source-to-source optimizers for a simple source language. Several specifications of live data flow analysis were presented, with the explicit goal of demonstrating how recursion between modules of a hierarchical specification can be used to describe problems typically specified as a circular attribute grammar specification.

Chapter 5 presented a tree-walking and a applicative batch evaluation algorithm for hierarchical attribute grammars. These batch evaluation algorithms for hierarchical dialects follow directly from the batch evaluation algorithms for non-hierarchical dialects.

Chapter 6 motivated the need for tree-walking incremental evaluation algorithms for hierarchical attribute grammars to explicitly determine the syntactic relationship between a module instance's new input term and its old input term by performing *Matching*. It also presented and critiqued Teitelbaum and Chapman's algorithm for matching in the context of Higher Order Attribute Grammars.

Chapter 7 described a closely related set of matching algorithms that constructed the new attribute tree for an invoked module instance from the old attribute tree for that module instance without using semantic information to guide the construction of the new attribute tree. The algorithms were designed to satisfy two heuristics: the *retention heuristic*, and *the maximal reuse heuristic.*

Chapter 8 described a context-based tree-walking evaluator for modules of a hierarchical attribute grammar that does take semantic information into account during the construction of the new attribute tree by combining the process of matching and evaluation. The resulting algorithm is optimal under the assumption that a tree-walking evaluator for a module of a hierarchical attribute grammar must traverse a crown of the new attribute tree starting with the root of the tree and then following child and parent links between adjacent attribute tree nodes. Chapter 8 concluded by describing the applicative approach to incremental evaluation of hierarchical attribute grammars and then comparing that approach to the context-based tree-walking evaluator.

## 9.2 Contributions

The key contributions presented in this dissertation include the following:

- The survey of a collection of existing hierarchical attribute grammar dialects.
- The introduction of a new hierarchical attribute grammar dialect, Modular Attribute Grammars.
- The description of a translation from Higher Order Attribute Grammars to hierarchical attribute grammars demonstrating the close relationship between Higher Order Attribute Grammars and the hierarchical and non-hierarchical attribute grammar dialects.
- The description of an automatic translation from hierarchical attribute grammars to functional programs. The translation is the basis of the applicative approach to batch and incremental evaluation of hierarchical attribute grammars.
- The description of several modules from a complete hierarchical attribute grammar specification of two prototype source-to-source optimizers, including the specification of a live data flow analysis module in which it is demonstrated that recursion between modules of a hierarchical specification can be used very effectively to describe problems typically described by circular attribute grammars.
- The description of tree-walking and applicative approaches to batch evaluation of hierarchical attribute grammars.
- The motivation for the need for *Matching* in tree-walking incremental evaluation algorithms, the critique of Teitelbaum and Chapman's incremental evaluator for Higher Order Attribute Grammars, and the development of four new matching algorithms.
- The development of an optimal evaluation algorithm for hierarchical attribute grammars.

Several contributions that were not reported in this dissertation include:

- The description of an automatic translation from hierarchical attribute grammars to non-hierarchical attribute grammars.
- The implementation of a prototype tree-walking incremental evaluator for hierarchical attribute grammars based on a matching algorithm similar to the ones that were presented.
- The implementation of a prototype applicative evaluator for hierarchical attribute grammars.

## 9.3 Future Work

The dissertation has presented a new specification language for complex language translators and new batch and incremental evaluation algorithms for the new language. Validation of the language and the algorithms will require constructing a specification of a non-toy programming language that describes a significantly complex and useful translation, and building efficient implementations of the batch and incremental evaluators. We are working on identifying a problem that fits this criterion.

Chapter 4 described a set of specifications of a live data flow analysis module, including a circular non-hierarchical attribute grammar specification, a non-circular non-hierarchical specification derived by ad-hoc means from the circular specification, and a recursive hierarchical attribute grammar specification. From the standpoint of efficiency of batch and incremental evaluation, which of these specifications is the best description of live data flow analysis? Several groups have examined incremental evaluation of circular attribute grammars [Jon90, WJ88, JS86]. How does applying an incremental evaluator for circular attribute grammars to the circular specification of live data flow analysis compare to the application of the traditional incremental evaluation algorithms to the non-circular specification, or the application of the incremental evaluation algorithms for hierarchical attribute grammars to the recursive specification?

# Appendix A

# Translation of HOAGs to HAGs

This appendix describes a method for automatically translating Higher Order Attribute Grammars into hierarchical attribute grammars. It is assumed that the initial specification is normalized and that syntactic references have been replaced with the new attributes and semantic rules. In the following construction, it will be necessary to apply the technique, described in Section 3.2.2, for invoking modules with a set of values to be treated as inherited attributes of the root of the term to be evaluated.

Figure A.1 depicts a simple Higher Order Attribute Grammar specification and the dependence graph constructed for evaluation of a simple input tree. The specification contains four productions with operators $opRA\bar{B}$, $opACC$, $opBCC$, and $opC$ where the name of each operator is formed by concatenating the names of the nonterminals appearing in each production, and where $\bar{B}$ is a nonterminal attribute.

This translation is based on the technique described in Section 2.2.2 for converting an attribute grammar into a functional program. The translation of a Higher Order Attribute Grammar to a hierarchical attribute grammar creates a collection of new module specifications. One module is created for each synthesized attribute associated with a nonterminal from the context free grammar underlying the Higher Order Attribute Grammar that was used as a nonterminal attribute in the original higher order specification. Each non-

R = opRA$\bar{B}$(A $\bar{B}$)
  $\bar{B}$ = A.S1
  $\bar{B}$.I1 = A.S2
  $\bar{B}$.I2 = A.S3
  R.S1 = $\bar{B}$.S3

A = opACC(C C)
  A.S1 = BCC(C$1 C$2)
  C$1.I1 = A.I1
  C$2.I1 = C$1.S1
  A.S2 = C$2.S1
  C$1.I2 = A.I2
  C$2.I2 = C$1.S2
  A.S3 = C$2.S2

B = opBCC(C C)
  B.S1 = 0
  C$1.I1 = B.I1
  C$2.I1 = C$1.S1
  B.S2 = C$2.S1
  C$1.I2 = B.I2
  C$2.I2 = C$1.S2
  B.S3 = C$2.S2

C = opC()
  C.S1 = C.I1
  C.S2 = C.I1 + C.I2

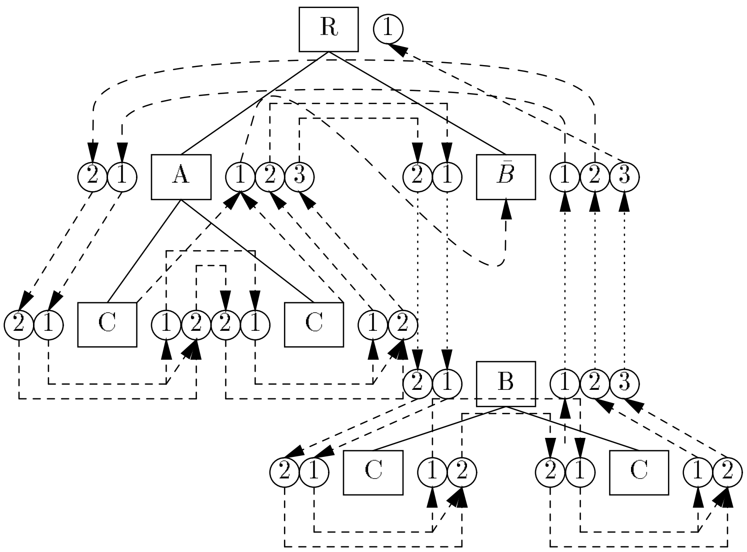


**Figure A.1** The Higher Order Attribute Grammar Example and Graph terminal attribute in a production will be replaced by a new nonterminal defined by a single new *nullary* production. Each new nonterminal represents a single old nonterminal that was previously used as a nonterminal attribute. Each new nonterminal will carry the same attributes as did the old nonterminal, but will have an additional inherited attribute that will be assigned the term value that was previously assigned to the nonterminal attribute. In the example, the nonterminal attribute $\bar{B}$ will be replaced by a new nonterminal $B'$ defined by the single production

$$B' = newopB'(),$$

where $newopB'$ is a nullary operator. Nonterminal $B'$ has the same two inherited attributes and three synthesized attributes as nonterminal $\bar{B}$, but, in addition, $B'$ has a single new inherited attribute $T$ that will be assigned the appropriate term value.

The translation is completed by associating semantic rules with each of the synthesized attributes of each of the new nonterminals and by defining each of the modules that is invoked by each of these new semantic rules. The semantic rule defining synthesized attribute $s$ of nonterminal $N$, invokes the module $M$–$N$–$s$ with the term assigned to the inherited attribute $T$ and with some subset of the inherited attributes associated with $N$. In the example, the synthesized attribute $S1$ originally associated with nonterminal attribute $\bar{B}$ and now associated with the new operator $newopB'$ is computed by invoking the function $M$–$B'$–*S1*. As in the technique for translating an attribute grammar to a functional program described above, the subset of inherited attributes passed to a function to compute a particular synthesized attribute corresponds to the *argument selector* for that attribute. Remember that the argument selector is the set of inherited attributes that *may* be needed to compute the value of the synthesized attribute.

It is assumed that all nonterminal attributes have been replaced as described above, and that all that is left in the translation of Higher Order Attribute Grammar to hierarchical attribute grammars is to define each of the new modules. To construct a new module, $M$–$N$–$s$, defining the value of synthesized attribute $s$ of nonterminal $N$, initially let module $M$ contain the subgrammar of the translated Higher Order Attribute Grammar that is necessary to perform evaluation over any tree whose root is labeled with the nonterminal $N$. Module $M$ can then be reduced to include only those semantic rules and attributes that are necessary to compute a value for $s$. Reduction of attribute grammars is described in [Fil83]. Figure A.2 through Figure A.5 depict the four modules generated by translating the simple example from Figure A.1.

Module ROOT
R = opRA$B'$(A$B'$)
  $B'$.T = A.S1
  $B'$.I1 = A.S2
  $B'$.I2 = A.S3
  R.S1 = $B'$.S3

A = opACC(C C)
  A.S1 = opBCC(C\$1 C\$2)
  C\$1.I1 = A.I1
  C\$2.I1 = C\$1.S1
  A.S2 = C\$2.S1
  C\$1.I2 = A.I2
  C\$2.I2 = C\$1.S2
  A.S3 = C\$2.S2

C = opC()
  C.S1 = C.I1
  C.S2 = C.I1 + C.I2

$B'$ = newop$B'$()
  $B'$.S1 = *M–B'–S1*($B'$.T)
  $B'$.S2 = *M–B'–S2*($B'$.T,$B'$.I1)
  $B'$.S3 = *M–B'–S3*($B'$.T,$B'$.I1,$B'$.I2)

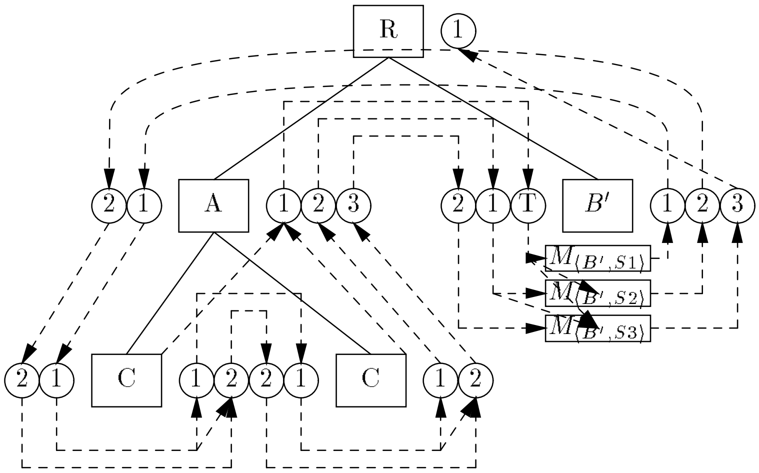


**Figure A.2** Module ROOT

Module *M–B'–S1*
B = opBCC(C C)
  B.S1 = 0
C = opC()

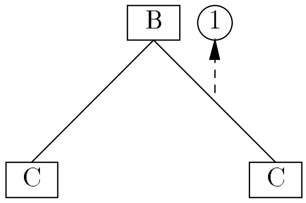


**Figure A.3** Module *M–B–S1*

Module *M–B'–S2*
B = opBCC(C C)
  C$1.I1 = B.I1
  B.S2 = C$2.S1
  C$1.I2 = B.I2

C = opC()
  C.S1 = C.I1

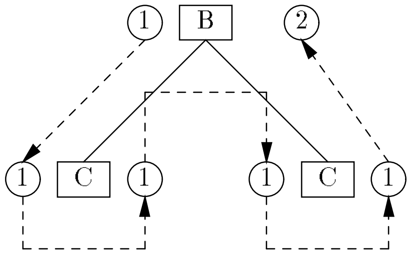


**Figure A.4** Module *M–B–S2*

Module *M–B'–S3*
B = opBCC(C C)
  C$1.I1 = B.I1
  C$2.I1 = C$1.S1
  C$1.I2 = B.I2
  C$2.I2 = C$1.S2
  B.S3 = C$2.S2

C = opC()
  C.S1 = C.I1
  C.S2 = C.I1 + C.I2

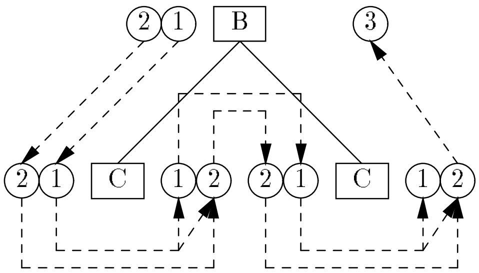


**Figure A.5** Module *M–B–S3*

There are two problems with using the translation process just described. First, the translation requires *reducing* an attribute grammar to remove useless attributes, semantic equations, and productions. Unfortunately, the reducibility problem has been shown by Filé to be of exponential time complexity [Fil83]. Second, even if we could efficiently reduce the attribute grammars for each module, then two modules defining different synthesized attributes of the same nonterminal may share attributes. Actually, these attributes should be considered to be *replicated* attributes, since they will be evaluated once per module in which they appear and the values of each of the copies of the replicated attributes will be identical. In fact, whereas the original Higher Order Attribute Grammar computes values for only a linear number of attributes in the size of the final syntax tree, the equivalent hierarchical attribute grammar might compute an exponential number of attributes.

# Appendix B

# Tradeoffs in Matching-Based Evaluation

Figure B.1 through Figure B.6 demonstrate the tradeoff between using maximal subtrees from $d_{copy}$ in constructing $d'_{copy}$ and selecting old attribute subtrees based on attribute context. The simple specification in Figure B.1 describes the attribution of input terms that each consist of a root node having three subterms: the first subterm, a single node labeled with one of the two operators $Y$ or $Z$, and the second and third subterms, a pair of subterms consisting of nodes labeled with the operators $F$, $G$ and $H$. If the first child of the root node is labeled with the operator $Y$ then the value 1 is passed to the root node via the synthesized attribute *Syn*, and the semantic rules associated with the root node force the value 1 (i.e., 1 mod 2) to be passed to each of the nodes in the root's second subterm and the value 0 (i.e., $(1 + 1) \bmod 2$) to be passed to each of the nodes in the root's third subterm via the inherited attribute *Inh*. If, instead, the first subterm of the root node is labeled with the operator $Z$ then the value 2 will be passed to the root node, the value 0 (i.e., 2 mod 2) will be passed to each node in the root's second subterm, and the value 1 (i.e., $(2+1) \bmod 2$) will be passed to each node in the root's third subterm.

Figure B.2 depicts a subterm $d$ whose first subterm is labeled $Y$ and which is suitable for evaluation by the simple specification. Figure B.3 depicts the attribute tree $d_{copy}$ representing the dag $d$ after evaluation. Nodes in $d$ and $d_{copy}$ are labeled exactly as described in Section 6.2. Boxed values are attributes at each attribute tree node.

Now consider the dag $d'$ shown with dag $d$ in Figure B.4. The dag $d'$ has a node labeled $Z$ as its first subterm. If a matching algorithm constructs $d'_{copy}$ using maximal subtrees from $d_{copy}$, then the attribute tree shown in Figure B.5 will be the result. Notice that the second subtree of the root of $d_{copy}$ is reused as the second subtree of the root of $d'_{copy}$, and the third subtree of the root of $d_{copy}$ is reused as the third subtree of the root of $d'_{copy}$. Filled circles in $d'_{copy}$ represent new attribute tree nodes. The boxed values correspond to the values of all of the attributes in the new attribute tree after change propagation has been performed, and the doubly-boxed boxes represent attributes determined to be in AFFECTED by the change propagator. In this example, every attribute of the new attribute tree will be reevaluated.

Figure B.6 depicts the attribute tree $d'_{copy}$ that results from selecting old attribute subtrees from $d_{copy}$ taking into account the attribute context in which those subtrees will be placed in $d'_{copy}$. In this attribute tree, the third subtree of $d'_{copy}$ consists of a single new attribute tree node and the second subtree from $d_{copy}$, and the second subtree of $d'_{copy}$ is a proper subtree from the third subtree of $d_{copy}$. Attributes in AFFECTED

```
S ::= R(X A1 A2)
  A1.Inh = X.Syn mod 2
  A2.Inh = (X.Syn + 1) mod 2
X ::= Y()
  X.Syn = 1
X ::= Z()
  X.Syn = 2
A1 ::= F(A2 A3)
  A2.Inh = A1.Inh
  A3.Inh = A1.Inh
A1 ::= G(A2)
  A2.Inh = A1.Inh
A ::= H()
```

**Figure B.1** A Simple Specification

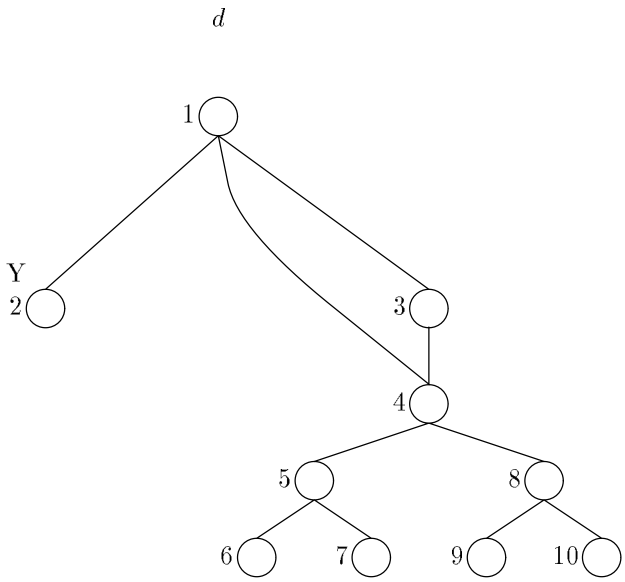


**Figure B.2** An Input Term $d$

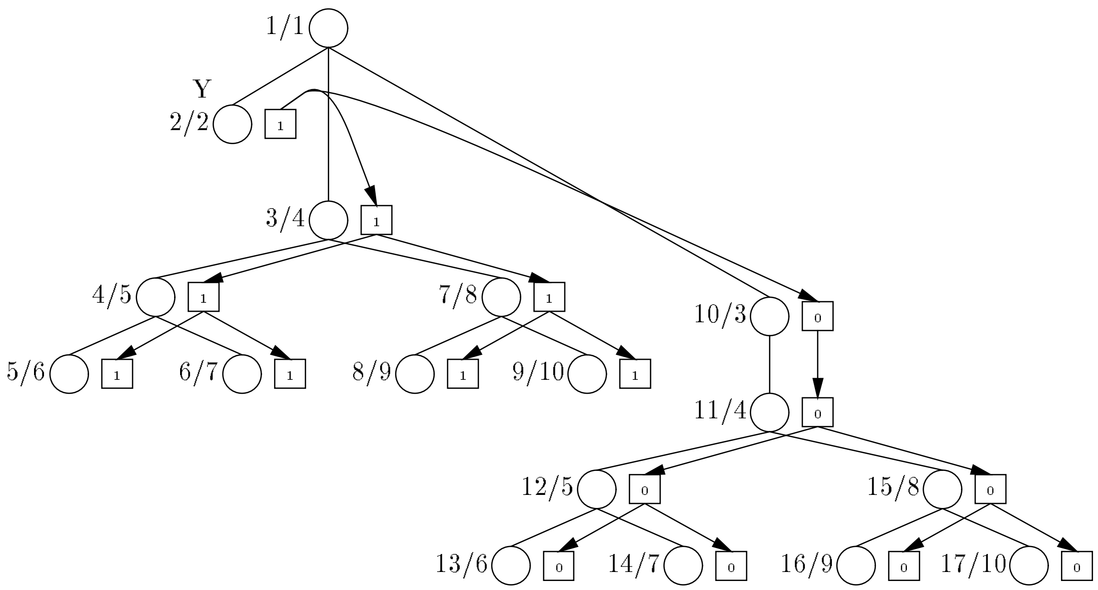


**Figure B.3** The tree $d_{copy}$

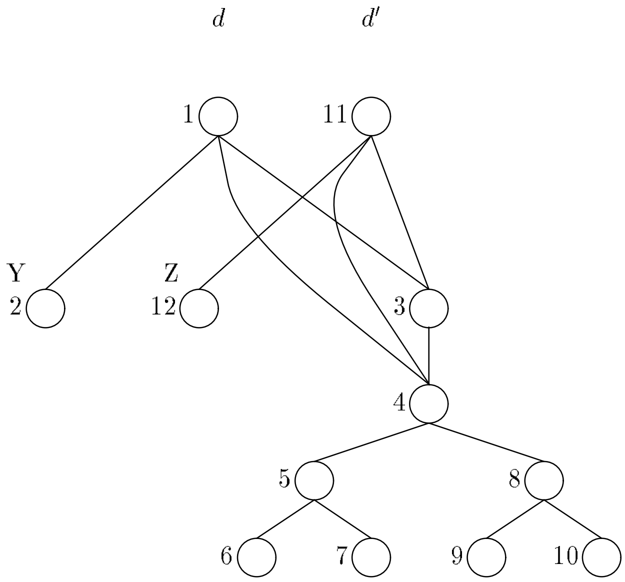


**Figure B.4** Two terms $d$ and $d'$

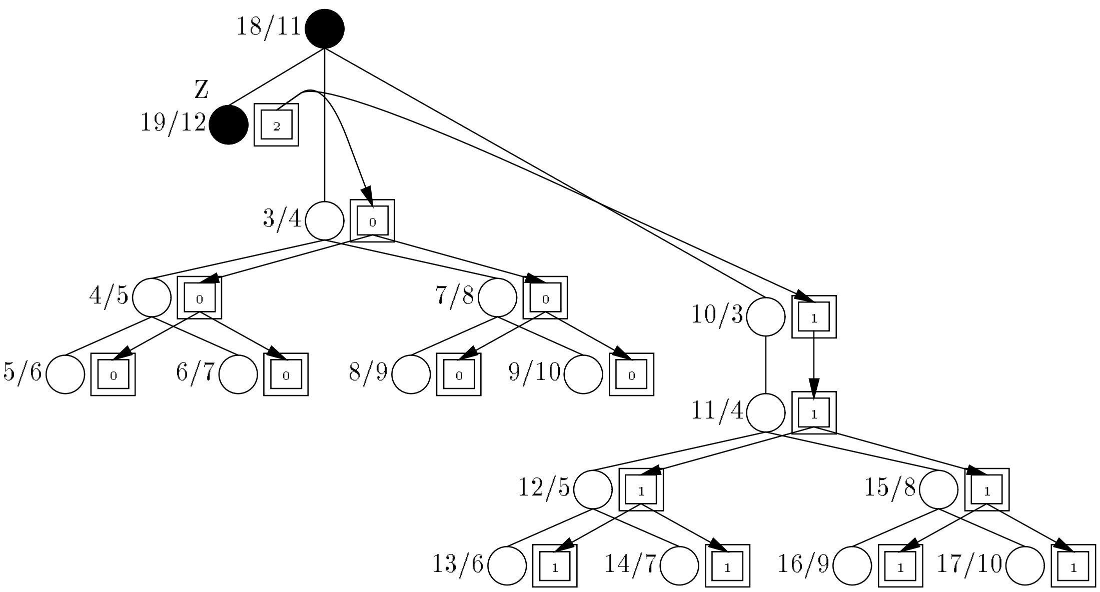


**Figure B.5** The tree $d'_{copy}$

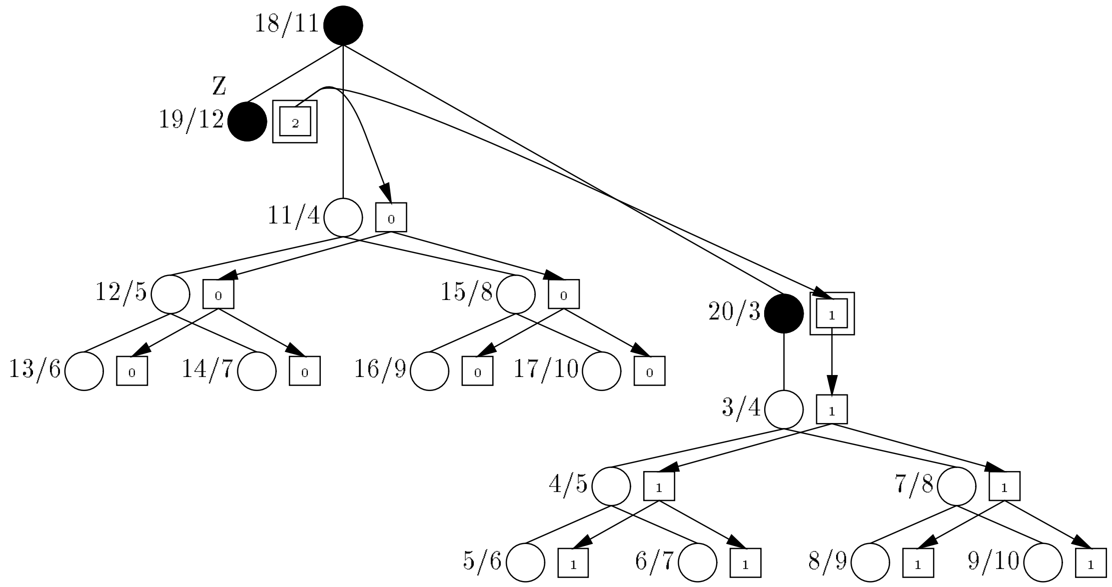


**Figure B.6** The Optimal Tree $d'_{copy}$

are again drawn as doubly-boxed boxes. Since the values of incoming inherited attributes at the roots of the two reused subtrees in $d'_{copy}$ are identical to those values in $d_{copy}$, none of those attributes will be AFFECTED.

# Bibliography


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

[CFZ82] B. Courcelle and P. Franchi-Zannettacci.
Attribute grammars and recursive program schemas I and II.
*Theoretical Computer Science*, 17(2):163–191, 235–257, 1982.

[CM79] L. M. Chirica and D. F. Martin.
An order-algebraic definition of Knuthian semantics.
*Mathematical Systems Theory*, 13(1):1–27, 1979.

[Cou84] B. Courcelle.
Attribute grammars: Definitions, analysis of dependencies, proof methods.
In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 81–102. Cambridge University Press, 1984.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho.
*Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*.
Springer-Verlag, New York, 1988.

[Eng84] J. Engelfriet.
Attribute grammars: Attribute evaluation methods.
In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, 1984.

[ES86] Orit Edelstein and Shmuel Sagiv.
Machine independent optimizations via attribute grammars.
Technical Report 88.187, IBM Israel, 1986.

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

[Fil83] Gilberto Filé.
Interpretation and reduction of attribute grammars.
*Acta Informatica*, 19:115–150, 1983.

[FZ82] P. Franchi-Zannettacci.
*Attributs Sémantiques et Schémas de Programmes.*
PhD thesis, Université de Bordeaux I, March 1982.

[GG84] Harald Ganzinger and R. Giegerich.
Attribute Coupled Grammars.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 157–170, Montreal, Canada, June 1984.

[GGV86] Harald Ganzinger, Robert Giegerich, and Martin Vach.
MARVIN: A tool for applicative and modular compiler specifications.
Forschungsbericht Nr. 220, July 1986.

[Gie88] R. Giegerich.
Composition and evaluation of attribute coupled grammars.
*Acta Informatica*, 25:355–423, 1988.

[Got74] Eiichi Goto.
Monocopy and associative algorithms in an extended LISP.
Technical Report 74–03, Information Science Laboratory, University of Tokyo, Tokyo, Japan, May 1974.

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

[Jaz81] M. Jazayeri.
A simpler construction for showing the intrinsically exponential complexity of the circularity problem for attribute grammars.
*Journal of the ACM*, 28(4):715–720, October 1981.

[Jon90] Larry G. Jones.
Efficient evaluation of circular attribute grammars.
*ACM Transactions on Programming Languages and Systems*, 12(3):429–462, July 1990.

[JOR75] M. Jazayeri, W.F. Ogden, and W.C. Rounds.
The intrinsically exponential complexity of the circularity problem for attribute grammars.
*Communications of the ACM*, 18(12):679–706, December 1975.

[JQ85] Xiaoping Jia and Jiahua Qian.
Incremental evaluation of attributed grammars for incremental programming environments.
In *IEEE COMPSAC'85*, pages 342–349, Chicago, Illinois, October 1985.

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

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

[Kat84] Takuya Katayama.
Translation of attribute grammars into procedures.
*ACM Transactions on Programming Languages and Systems*, 6(3):345–369, July 1984.

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

[Knu71a] Donald E. Knuth.
Examples of formal semantics.
In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, pages 212–235. Springer-Verlag, 1971.

[Knu71b] Donald E. Knuth.
Semantics of context-free languages: Correction.
*Mathematical Systems Theory Journal*, 5(1):95–96, March 1971.

[Mic68] Donald Michie.
"Memo" functions and machine learning.
*Nature*, 218:19–22, April 1968.

[Pug88] William Worthington Pugh, Jr.
*Incremental Computation and the Incremental Evaluation of Function Programs.*

PhD thesis, Cornell University, August 1988.
TR 88–936.

[Rep82] Thomas Reps.
Optimal-time incremental semantic analysis for syntax-directed editors.
In *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*, pages 169–176, January 1982.

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

[RT87] Thomas Reps and Tim Teitelbaum.
*The Synthesizer Generator: Reference Manual.*
Cambridge, Massachusetts, second edition, July 1987.

[RT89] Thomas W. Reps and Tim Teitelbaum.
*The Synthesizer Generator: A System for Constructing Language-Based Editors.*
Springer-Verlag, New York, 1989.

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

[Saa78] Mikko Saarinen.
On constructing efficient evaluators for attribute grammars.
In G. Ausiello and C. Böhm, editors, *5th ICALP*, pages 382–397. Springer-Verlag, July 1978.

[SEFR89] S. Sagiv, O. Edelstein, N. Francez, and M. Rodeh.
Resolving circularity in attribute grammars with applications to data flow analysis.
In *Proceedings of the Sixteenth Annual Symposium on Principles of Programming Languages*, pages 36–48, Austin, Texas, January 1989.

[TC90] Tim Teitelbaum and Richard Chapman.
Higher–order attribute grammars and editing environments.
In *Proceedings of the SIGPLAN '90 Symposium on Programming Language Design and Implementation*, pages 197–208, June 1990.

[VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper.
Higher order attribute grammars.
In *Proceedings of the SIGPLAN '89 Symposium on Programming Language Design and Implementation*, pages 131–145, June 1989.

[WJ88] Janet A. Walz and Gregory F. Johnson.
Incremental evaluation for a general class of circular attribute grammars.
In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 209–221, June 1988.

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

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