A Survey of
Tree-Walk Evaluation Strategies
for Attribute Grammars

by

Daniel Yellin

CUCS-134-84

Computer Science Department
Columbia University
New York, New York, 10027

September 1984

List of Figures

Abstract

Since the time that Knuth's seminal paper on attribute grammars (AGs) first appeared [24], the AG formalism for specifying the translation process has received much attention and has become the basis for several real translator-writing systems [12, 15, 19, 20, 25]. It seems to be a promising vehicle for automating the construction of compilers, in addition to its various other uses currently under investigation which includes syntax directed editors [5], attributed parsing [18, 39], interactive proof checking [35], waveform analysis [30], natural language translation into SQL data base code [31] and SQL data base code translation into natural language [27].

Given a sentence in a source language, an AG specifies a unique semantic tree for that sentence. The translation is found by *evaluating* the attribute-instances of the semantic tree. Upon completion, certain distinguished attributes of the root contain the translated string. Due to the declarative nature of the AG, there is no unique way to evaluate the semantic tree. Determining strategies to evaluate the semantic trees of an AG has been the focus of much discussion in the literature. It has both theoretical and practical importance: Bad strategies of evaluating semantic trees can be very inefficient in terms of both space and time making AGs less attractive for real translator-writing systems.

In this paper we survey static tree-walk evaluator strategies that have been developed for evaluating semantic trees. We compare the evaluators to each other on the basis of several performance criteria. We develop the notion of *optimality* of an evaluator and determine how close a given evaluation strategy comes to optimal. By considering different strategies we find a natural taxonomy of AGs: as not every AG will give rise to semantic trees evaluable by a given strategy S, we can define the class of S-evaluable AGs as exactly those AGs whose semantic trees are evaluable by strategy S. In this way we will discover a hierarchy of strategies of increasing power, the strongest strategy being one capable of evaluating the semantic trees of *any* well-defined AG.

In section 1 we present a brief introduction to AGs and set forth some terminology to be used in the rest of the paper. In section 2 we introduce the general idea of tree-walk evaluators and we formalize the concept of an optimal tree-walk evaluator for a semantic tree. In section 3 we define an especially useful class of evaluators called *static evaluators*. In the next three sections we survey three types of static evaluators: pass-oriented evaluators, uniform evaluators and multi-protocol evaluators. In section 7 we summarize our conclusions and review the taxonomy of AGs we have developed. The appendix presents the proofs of two NP-complete problems cited in the survey.

1. A Brief Introduction to Attribute Grammars

Attribute grammars were first proposed by Knuth [24] as a way to specify the semantics of context-free languages. A *context-free grammar* is a 4-tuple (N,Σ,S,P), where N is a set of *non-terminal* symbols, Σ is a set of *terminal* symbols, S $\in$ N is the *start symbol*, and P is a set of *productions*. A production is of the form [p: X_0 ::= X_1 ··· X_{np}]. $X_0 \in N$ is the *left-part* of p; X_1 X_2 ... X_{np} is the *right-part* of p and for i > 0, $X_i \in N \cup \Sigma$. Sometimes the expression "p[i]" is used to denote X_i.

The basis of an attribute grammar is a context-free grammar. This describes the context-free language that is the domain of the translation; i.e., those strings on which the translation is defined. This context-free grammar is augmented with *attributes* and *semantic functions*. Attributes are associated with the nonterminal symbols of the grammar. We write "X.A" to denote attribute A of symbol X, and $A(X)$ to denote the set of attributes associated with X. Semantic functions are associated with productions: they describe how the values of some attributes of the production are defined in terms of the values of other attributes of the production. Given a production [p: X_0 ::= X_1 ··· X_{np}], we call $A(p) = A(X_0) \cup ... \cup A(X_{np})$ the attributes of p.

Below is an attribute grammar that describes binary numerals and the decimal values they denote.

```
binary_number:  (synthesized attributes: binary_number.val;
                 inherited atts: ∅)
digits:         (synthesized atts: digits.val;
                 inherited atts: digits.place)
digit:          (synthesized atts: digit.val;
                 inherited atts: digit.place)
1:              ∅
0:              ∅
```

Context free symbols of the attribute grammar and their attributes

```
p1: binary_number ::= digits.
        binary_num.val = digits.val;
        digits.place = 0;

p2: digits0 ::= digits1 digit.
        digits0.val = digits1.val + digit.val;
        digit.place = digits0.place;
        digits1.place = digits0.place + 1;

p3: digits ::= digit.
        digits.val = digit.val;
        digit.place = digits.place;

p4: digit ::= 0.
        digit.val = 0;

p5: digit ::= 1.
        digit.val = 2 ^ digit.place;
```

Productions of the attribute grammar and their semantic functions

Figure 1: *An attribute grammar example*

In this AG there are 5 productions and each production has associated attributes and semantic functions. In production 2, <digits0> and <digits1> denote separate occurrences of the same symbol, <digits>; the numeric suffixes distinguish these different occurrences.

A semantic function specifies the value of an attribute-occurrence of the production, e.g. the value of digits0.val in production 2 above is defined to be equal to the sum of digits1.val and digit.val. Semantic functions are pure functions, they have no side-effects. Their only arguments are either constants or other attribute-occurrences of the production.

How an attribute grammar specifies a translation can be most easily explained by an operational description. The underlying context-free grammar of an attribute grammar describes a language. Any string in this language has a parse tree associated with it by the grammar. The nodes of this parse tree can be labelled with symbols of the grammar. Each interior node of this tree, N, has two productions associated with it. The left-part production (LP) is the production that applies at N, say p, deriving N's children. The right-part production (RP) is the production that applies at the parent of N, say p', deriving N and its siblings. The production instances of p and p' are *adjacent* productions of the semantic tree. Leaves of the tree don't have LP productions; the root doesn't have an RP production.

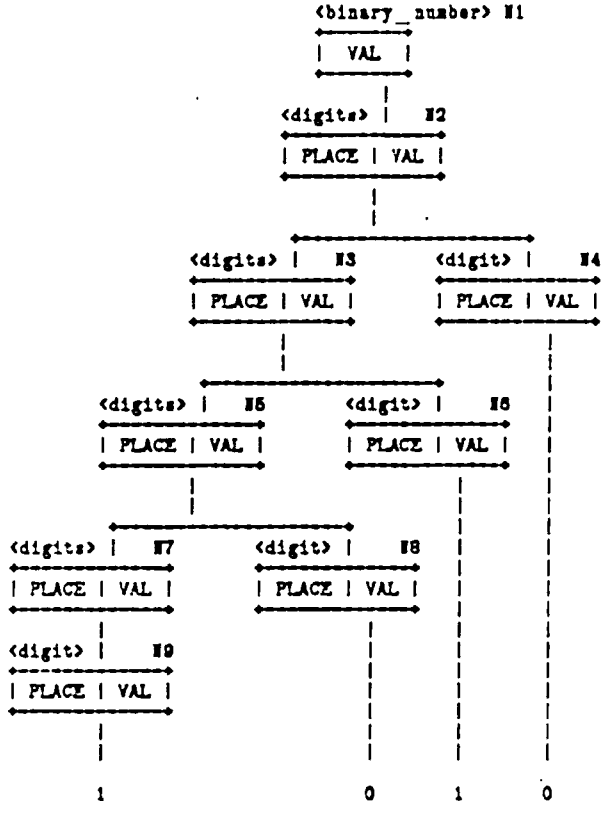


Figure 2: *A semantic tree for the attribute grammar of figure 1*

A *semantic tree* is a parse tree in which each node contains fields that correspond to the attributes of its labelling grammar symbol. Each of these fields is an *attribute-instance.* Associated with each attribute is a set of possible values that instances of this attribute can be assigned. This is analogous to the "type" of a variable in a programming language.

However, each attribute-instance takes on precisely one such value; attribute-instances are not variables. The values of attribute-instances are specified by the semantic functions. Figure 2 shows a semantic tree for the string 1010 of the attribute grammar given in figure 1. Each node in this tree is labelled with its associated grammar symbol. To distinguish between different instances of the same grammar symbol each node is also assigned a number.

The semantic functions of a production represent a template for specifying the values of attribute-instances in the semantic tree. Consider figure 2 again. N2 is a semantic tree node labeled by the context-free symbol <digits>. It has two productions associated with it: [p_1: binary_number ::= digits] (its RP production) and [p_2: digits0 ::= digits1 digit] (its LP production). These two production instances are adjacent in the tree. The semantic function <digits.place = 0> of production p_1 indicates that the value of attribute-instance N2.place will be set equal to the constant 0. Similarly, the semantic function <binary_number.val = digits.val> of that production indicates that the value of attribute-instance N1.VAL should be copied from the value of N2.val.

Since two different productions are associated with each attribute-instance, there could be two semantic functions that independently specify its value, one from the LP production and one from the RP production. If we assume that each attribute-instance is defined by only one semantic function, either from the LP production xor from the RP production, then we must guard against an attribute-instance not being defined at all because the LP production assumed that the RP production would define it and vice versa. These difficulties are avoided in attribute grammars by adopting the convention that for every attribute, X.A, either: (1) every instance of X.A is defined by a semantic function associated with its LP production, or (2) every instance of X.A is defined by a semantic function associated with its RP production. We refer to the semantic function which evaluates X.a as $f_{X.a}$. Attributes whose instances are all defined in their LP production are called *synthesized* attributes; attributes whose instances are all defined in their RP production are called *inherited* attributes. Every attribute is either inherited or synthesized[1]. Inherited attributes propagate information down the tree, towards the leaves. Synthesized attributes propagate information up the tree, toward the root. The inherited attributes of a non-terminal X are denoted by $\mathcal{I}(X)$, the synthesized attriubtes by $\mathcal{S}(X)$; $\mathcal{A}(X) = \mathcal{I}(X) \uplus \mathcal{S}(X)$. The start symbol has no inherited attributes. From the point of view of an individual production the above conditions require that the semantic functions of a production MUST define EXACTLY all the inherited attributes of the right-part symbols and all synthesized attributes of the left-

[1]Some systems [10] have employed a third type of attribute, called an *intrinsic attribute*, initially introduced by [37]. An intrinsic attribute has its value defined during the parsing phase, as the semantic tree is being built. These attributes are evaluated during parsing and before actual semantic evaluation of the tree due to both conceptual and practical reasons. Conceptually these attributes directly relate to the input being parsed. Practically, using intrinsic attributes often helps to eliminate an extra pass in a pass-oriented evaluator. It has similar benefits for other evaluation strategies. In this paper, however, we shall restrict ourselves to just inherited and synthesized attributes.

part symbol. We call these attributes the *defining* attributes of p, $DEF(p) = S(X_0) \cup I(X_1) \ldots \cup I(X_{np})$.

We shall not allow all attributes of $A(p)$ to be used in defining the attributes of $DEF(p)$; we only allow the *applied* attributes $APP(p) = A(p) - DEF(p)$ to be used as arguments in the semantic functions of p. That is, no attribute defined in p can itself be used to define another attribute in p. An attribute grammar which obeys such a condition is said to be in Bochmann normal form [1]. In the remainder of this paper we assume that all attribute grammars are in Bochmann Normal Form unless stated explicitly otherwise. This does not impinge upon the power of the attribute grammar formalism; indeed it is not hard to see that *any* attribute grammar can be easily converted to Bochmann Normal Form [1]. Nonetheless, for reasons of efficiency and convenience, many real systems (such as Linguist [10] and GAG [20]) drop this constraint and allow for attribute grammars not in Bochmann Normal Form[2].

Thus the semantic functions of an attribute grammar specify a unique value for each attribute-instance. However, in order to actually compute the value of attribute-instance X.att we must first have available the values of those other attribute-instances that are arguments of the semantic function $f_{X.att}$ that defines X.att. In the example of figure 2, before N1.VAL can be computed the value of N2.VAL must have already been computed. Such *dependency relations* restrict the order in which attribute-instances can be evaluated. In extreme cases an attribute-instance can depend on itself; such a situation is called a circularity and by definition such situations are forbidden from occuring in well-defined attribute grammars. In general, it is an exponentially hard problem [16] to determine that an attribute grammar is *non-circular*; i.e. that no semantic tree that can be generated by the attribute grammar contains a circularly defined attribute-instance. Fortunately there are several interesting and widely applicable sufficient conditions that can be checked in polynomial time [1, 17, 21, 23].

The result of the translation specified by an attribute grammar is realized as the value of one or more (necessarily synthesized) attribute-instances of the root of the semantic tree. In order to compute these values the other attribute-instances must be computed. Figure 3 shows the semantic tree of the previous example with *dependency arcs* drawn in. If an attribute Ni.att of a node in the tree depends on the value of another attribute Nj.att in the tree, then there exists an arc from Nj.att to Ni.att. These arcs create a partial ordering on the attribute instances in the tree. It is important to note that since this ordering is only partial, there is no unique order of evaluating the attribute instances in the tree.

---

[2] Consider, for example, a production [p: $X_0 ::= X_1\ X_2$] with semantic functions $X_1.a = f(X_0.a)$ and $X_2.a = f(X_0.a)$. A literal implementation of these functions would require 2 evaluations of $f(X_0)$, one to evaluate $X_1.a$ and one to evaluate $X_2.a$. If these functions were to be rewritten into the equivalent form $X_1.a = f(X_0.a)$ and $X_2.a = X_1.a$ then $f(X_0)$ would only have to be evaluated once but Bochmann Normal Form constraints would be violated

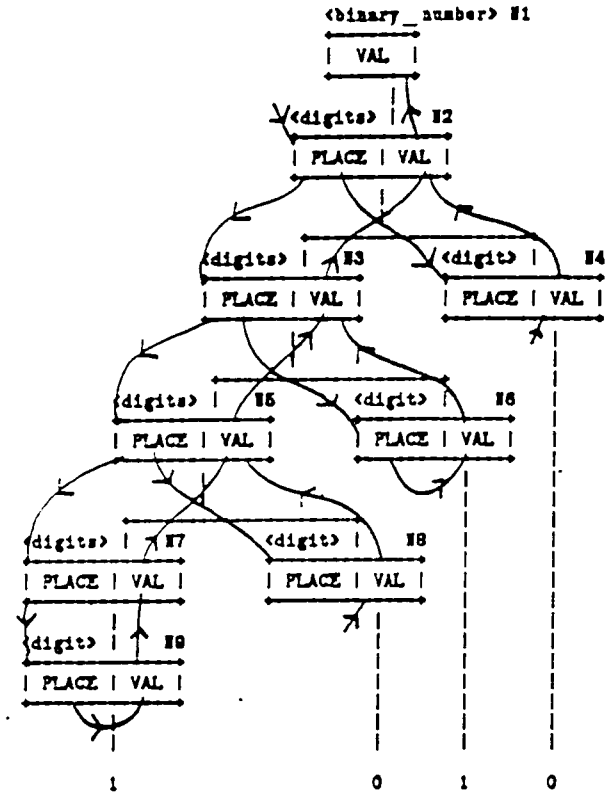


Figure 3: *The semantic tree of figure 2 with dependency arcs drawn in*

An attribute grammar evaluator for a given attribute grammar G is an algorithm which takes as input a semantic tree constructed from any valid sentence in G and computes the distinguished attribute-instances of the root (this normally involves computing all attribute-instances of the tree and unless we say so explicitly, we shall make no further distinctions between the two[3]). Since the evaluator must be capable of evaluating *any* semantic tree of the grammar G, embodied in its control structure must be some general mechanism describing how to evaluate semantic trees. An *attribute grammar evaluation strategy* describes how to build the control structure of an evaluator. It is a meta-algorithm for building an algorithm that will compute attribute-instances in an order such that no attribute-instance is computed before all dependent attribute-instances are available and such that all attribute-instances of the root are computed. An attribute grammar evaluation strategy may work correctly only on a subset of all well-defined attribute grammars, but it must work correctly

[3]Attributes in a semantic tree which are not needed to compute the distinguished attribute-instances of the root are called *useless attributes*. Normally evaluators are built which compute all attribute-instances of the tree including useless ones. Some evaluators, however, are built to evaluate only needed attributes, claiming that this can save a substantial amount of computation in the semantic tree. See [19, 22, 38] for instance. In [14], Filè discusses how to convert an attribute grammar into an equivalent attribute grammar which has the property that none of its semantic trees contain useless attributes. In general this construction can take an exponential amount of time.

on any semantic tree of an acceptable attribute grammar. The focus of this paper is to survey various strategies which have been developed for efficient evaluation of semantic trees.

Attribute grammars are attractive specification tools. Two principal reasons for this are their *locality of reference* and their *non-procedural nature*. We say that an attribute grammar has locality of reference in that the values it defines (i.e. the attribute-instances) are specified exclusively in terms of other attribute-instances local to a production. An attribute grammar does not contain any global variables or implicit state information that can affect the translation. Each local piece of an attribute grammar, i.e. each production, communicates with the rest of the attribute grammar only through strictly defined interfaces: the attributes of the symbols occurring in this production.

Like a context-free grammar, an attribute grammar is a description rather than an algorithm. Just as a context-free grammar specifies phrase-structure independently of a parsing algorithm, so does an attribute grammar specify semantics or translation without presuming an evaluation order. Because semantic functions are pure functions, the definition of an attribute-instance is determined by the attribute grammar and the semantic tree: not by the algorithms of the evaluator or those of the semantic functions. Thus, an attribute grammar is a locally described, non-procedural specification of values, rather than an algorithm for computing those values.

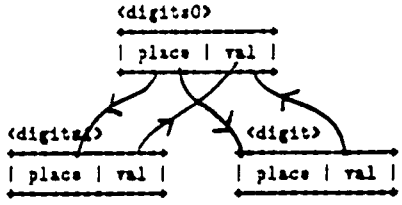
The modular nature of attribute grammars also allows for dependency information between attributes to be expressed in a modular form. The *dependency graph* Dp of a production p contains one vertex for each attribute of p and has an edge from v to w iff v is an argument of the semantic function defining w. Any production-instance of p in *any* semantic tree will exhibit the dependencies between its attribute-instances as given in the graph Dp; if $N_0,N_1,...,N_{np}$ are nodes of a tree corresponding to the symbols $X_0,X_1,...,X_{np}$ of $[p: X_0 ::= X_1 \cdots X_{np}]$, then an edge $(X_j.att_1, X_k.att_2)$ in Dp means that the attribute-instance $N_j.att_1$ must be evaluated before the attribute-instance $N_k.att_2$.

Just because there is no edge (v,w) in Dp does not mean, however, that w can be evaluated before v; it may be that w is indirectly dependent upon v. Consider a subtree rooted at a node N in a semantic tree. Information flows *into* N's subtree by way of the inherited attributes of N and information flows *out* of N's subtree by way of the synthesized attributes of N. Hence the dependency information of N's subtree as it relates to N can be summarized by a graph G_N whose vertices are attributes of N and whose edges are between inherited and synthesized attributes. An edge from the inherited attribute $N.att_i$ to the synthesized attribute $N.att_s$ implies that $N.att_i$'s value must already be computed before $N.att_s$'s value can be evaluated. Given a production $[p: X_0 ::= X_1 \cdots X_{np}]$ and the graphs G_{Xk}, $k = 1...n_p$, summarizing the dependency information for any subtree rooted at X_k, the *augmented dependency graph* $Dp[G_{X1},...,G_{Xnp}]$ augments the dependency information of the Dp graph by adding dependency information concerning the subtrees rooted at the right-part symbols of the production p. It is obtained from the Dp graph by adding an edge

$(X_k.att_1, X_k.att_2)$ to Dp whenever such an edge exists in G_{Xk}. Assuming that in a given semantic tree we have nodes $N_0,...,N_{np}$ corresponding to the context-free symbols of the production $[p: X_0 ::= X_1 \cdots X_{np}]$ and graphs G_{Xk} $(1 \leq k \leq np)$ summarizing the dependency information of the subtrees rooted at N_k $(1 \leq k \leq np)$ then a *path* from $X_i.att_1$ to $X_j.att_2$ in $Dp[G_{X1},...,G_{Xnp}]$ means that $N_i.att_1$ must be evaluated before $N_j.att_2$. Figure 4 gives a production p (production$_2$ of Figure 1), its Dp graph, G_k graphs for p's RHS symbols, and the associated augmented dependency graph. Note that the graph G_{digit} contains an edge (place,val) since in some subtrees rooted at an instance of digits the attribute-instance digit.val will depend upon the attribute-instance digit.place (see production p_5 of figure 1).

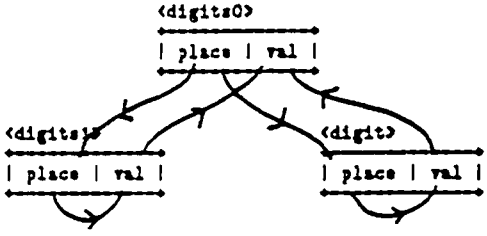
```
p: digits0 ::= digits1 digit.
       digits0.val = digits1.val * digit.val;
       digit.place = digits0.place;
       digits1.place = digits0.place + 1;
```

A production p



G_k graphs for digits and digit

The dependency graph Dp for p



The augmented dependency graph $Dp[G_{digits1}, G_{digit}]$

Figure 4: *Dependency and augmented dependency graphs*

Dependency and augmented dependency graphs are useful in determining the order in which attributes of a production must be evaluated. For example, the augmented dependency graph given in the above figure indicates a partial evaluation order of <digits0.place, digit.place, digit.val, digits0.val> and of <digits0.place, digits1.place, digits1.val, digits0.val>. In sections 5 and 6 we shall make use of a slightly extended version of augmented dependency graphs of the form $Dp[G_{X0}, G_{X1},...,G_{Xnp}]$ where the composite graph in this case is the same as before only now edges in G_{X0} are included as well. In this case an edge from $G_{Xi}.a$ to $G_{Xi}.b$ $(0 \leq i \leq np)$ indicates that according to a given strategy, $X_i.a$ must be evaluated before $X_i.b$.

We close this section by presenting an attribute grammar which translates simple English descriptions of mathmatical expressions into post-fix Polish notation. Although for the sake of brevity we have greatly simplified the attribute grammar, we hope that it nonetheless illustrates the power and simplicity of the attribute grammar formalism in translating strings from one language into strings of another language. The grammar distinguishes between expressions involving only integer values (in which case operators of the form $+_i$ and $*_i$ are required) and those involving a decimal point value (in which case operators of the form $+_r$ and $*_r$ are required). Figures 5 and 6 give the attribute grammar and a typical semantic tree for the grammar which translates the string 'Multiply 80 by 5.8' into the mathematical post-fix expression $(80,5.8,*_r)$. The attribute-instances of the tree have been filled in to reflect the values they will acquire upon computation.

```
p1: S ::= Op Number1 Preposition Number2.

          S.translation = If (Preposition.msg = ok) then
              Concatenate('(',Number1.translation,',',Number2.translation,',',Op.translation,')')
                        else 'error: preposition violates context sensitivities';

          Op.type = If (Number1.type = decimal_point) or (Number2.type = decimal_point)
                    then decimal_point else integer;

          Preposition.type = If (Op.translation = '*r') or (Op.translation = '*i')
                             then multiply else add;

p2: Number ::= Integer.

          Number.translation = Integer.translation;

          Number.type = Int;

p3: Number ::= Decimal_num.

          Number.translation = Decimal_num.translation;

          Number.type = decimal_point;

p4: Op ::= 'add'.

          Op.translation = If (Op.type = decimal_point) then '+r' else '+i';

p5: Op ::= 'multiply'.

          Op.translation = If (Op.type = decimal_point) then '*r' else '*i';

p6: Integer ::= digits.

          Integer.translation = digits.value;

p7: Decimal_num ::= digits1 '.' digits2.

          Decimal_num.translation = Concatenate(digits1.value,'.',digits2.value);

p8: Preposition ::= 'by'.

          Preposition.msg = if (Preposition.type = multiply) then ok else notok;

p9: Preposition ::= 'to'.

          Preposition.msg = if (Preposition.type = add) then ok else notok;

p10: Preposition ::= 'with'.

          Preposition.msg = if (Preposition.type = multiply) then ok else notok;
```

Figure 5: *An attribute grammar for translating mathematical expressions into post-fix Polish notation*

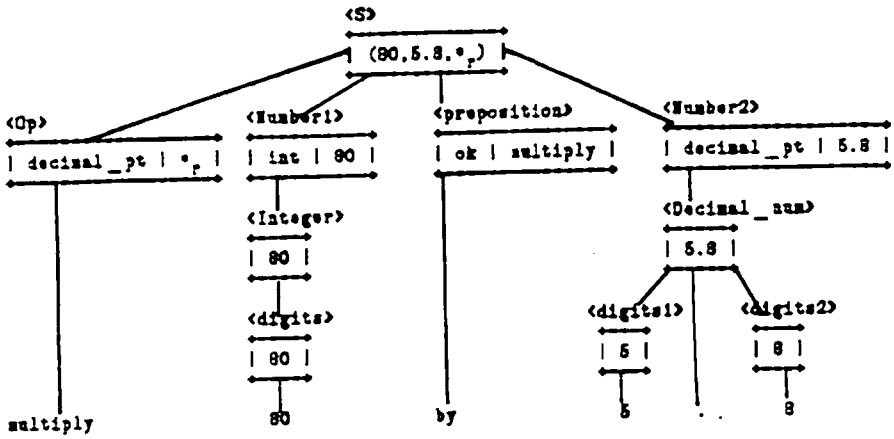


Figure 6: *A semantic tree for the grammar of figure 5*

2. Tree-walk Evaluators and Optimal Visit Sequences

An obvious way to evaluate the attributes in a semantic tree T would be to build an acyclic graph G=(V,E) where V = attribute-instances of T and [v,w] $\in$ E iff w is dependent upon v in T. We could then do a topological sort on the vertices of G and evaluate the attributes according to their topological order. This strategy has the pleasant property that a single evaluator would be able to evaluate *any* semantic tree for *any* well-defined attribute grammar. This strategy has actually been implemented, along with several optimizations, in the Delta system [26]. Unfortunately, the large number of attributes and dependencies in a semantic tree makes this approach expensive and unweildy to use [26]. Furthermore, it also has the drawback of doing most of the semantic analysis during run-time, precluding many time and space optimizations [13]. Various other strategies relying upon run-time semantic analysis have also been considered [8, 19, 34]. Due to the reasons cited above, however, most evaluators that have been devised are *tree-walk evaluators*. A tree-walk evaluator has a single *locus of control* that moves around an explicit semantic tree. The locus of control is always at some production instance in the tree. We will often say that the locus of control is at a node N when we really mean that it is at the LP production associated with N. So, for example, if the locus of control is at a production-instance of [p: $X_0 ::= X_1 \cdots X_{np}$] in some semantic tree, we will say that it is at N, where N is the node labeled by this occurrence of X_0. The evaluator executes a sequence of $EVAL_{att}$ and $VISIT_k$ instructions. An $EVAL_{att}$ instruction says to evaluate the attribute att of the production that applies at the current node by invoking its associated semantic function f_{att}. Arguments supplied to this function are the (previously computed) values of appropriate applied attribute-instances of the current node and its children. A $VISIT_k$ instruction causes the evaluator's locus of control to move from the current node to either its parent or one of its children; $VISIT_0$ moves it to the parent, if $k > 0$ then $VISIT_k$ moves the locus of control to the k^{th} child. Hence the locus of control always moves between adjacent productions in the semantic tree. The locus of control is never moved to a leaf as terminal grammar symbols have no attributes; a $VISIT_0$ executed at the root indicates the end of attribute evaluation.

It is helpful to view a tree-walk evaluator as 'roaming' over the semantic tree. Upon reaching a node, it decides to evaluate some attributes, visit some children, return to the node, evaluate some more attributes, etc., and then leave the node. How it decides which attributes to evaluate and which children to visit depends upon the control structure, which embodies some attribute grammar evaluation strategy.

What computational costs are associated with evaluating a semantic tree using a *tree-walk strategy*? Certainly an evaluator will incur the cost of evaluating each attribute in the semantic tree, corresponding to the execution of the $EVAL_{att}$ instructions. This cost cannot be increased or decreased depending upon the evaluation method; it is constant regardless of the evaluation strategy chosen. The second cost incurred by a tree-walk evaluator is the cost of transferring the locus of control from one production-instance in the tree to another adjacent production-instance in the tree, corresponding to the execution of the $VISIT_k$

instructions. This cost is dynamic and can vary widely in different evaluators. It might be useful for the reader at this point to glance back at figure 2 and notice that starting at the root, there are many ways one can traverse the tree, evaluating attributes and visiting adjacent productions (see figure 7). A third cost incurred during the evaluation of a semantic tree is intimately connected with the control structure of the evaluator. Upon visiting a node a tree-walk evaluator decides upon some sequence of $VISIT_k$ and $EVAL_{att}$ instructions to perform. If this decision involves a lot of computation it can also make the evaluator very inefficient. For the static evaluators we shall study (to be introduced in the next section) deciding what instruction sequence to execute has little overhead (it involves the execution of a case statement for each $VISIT_k$ instruction executed) and its cost can be tacked onto the cost of executing the $VISIT_k$ instruction. Therefore an important criterion by which to judge a tree-walk evaluator is by how many $VISIT_k$ instructions it uses to evaluate a semantic tree. An *optimal tree-walk evaluator* for a semantic tree T is, by definition, one which uses the minimum number of $VISIT_k$ instructions possible to evaluate every attribute instance in the tree.

If we examine the behavior of a tree-walk evaluator E when evaluating a given semantic tree T, we can write down the sequence of $EVAL_{att}$ and $VISIT_k$ instructions it executes. Upon looking at this *computation sequence* [28] we can retrace the evaluator's traversal of the tree and note exactly when it evaluated each attribute-instance. If we remove all of the $EVAL_{att}$ instructions from this sequence, the remaining $VISIT_k$ instructions are called a *visit sequence* of E for T. An optimal tree-walk evaluator for a semantic tree will give rise to the shortest visit sequence possible. The problem of determining whether or not there exists a visit sequence of length $\leq N$ evaluating all the attributes in a semantic tree is shown in the appendix to be NP-complete in the number of attributes in the tree. This means that building an optimal evaluator for a single semantic tree, let alone one which will be optimal for every semantic tree of the attribute grammar, is an intractable problem.

Whenever a tree-walk evaluator visits the child of a node it must eventually return to the parent, as the evaluator ends its evaluation with the locus of control at the root. We adopt the convention of referring to a $VISIT_0$ instruction as a RETURN instruction. Note that for every $VISIT_k$ instruction ($k > 0$) in a computation sequence there will exist exactly one RETURN instruction. Figure 7 shows two different computation sequences an evaluator might employ to evaluate the semantic tree of figure 2. The notation $VISIT_k$ {Ni} is used to indicate that the visit to the k^{th} child moves the locus of control of the evaluator to the node Ni of the tree.

$EVAL_{N2.place}$, $VISIT_1$ (N2), $EVAL_{N3.place}$,
$EVAL_{N4.place}$, $VISIT_2$ (N4), $EVAL_{N4.val}$,
RETURN (N2), $VISIT_1$ (N3), $EVAL_{N5.place}$,
$EVAL_{N6.place}$, $VISIT_2$ (N6), $EVAL_{N6.val}$,
RETURN (N3), $VISIT_1$ (N5), $EVAL_{N7.place}$,
$EVAL_{N8.place}$, $VISIT_2$ (N8), $EVAL_{N8.val}$,
RETURN (N5), $VISIT_1$ (N7), $EVAL_{N9.place}$,
$VISIT_1$ (N9), $EVAL_{N9.val}$, RETURN (N7),
$EVAL_{N7.val}$, RETURN (N5)), $EVAL_{N5.val}$,
RETURN (N3), $EVAL_{N3.val}$, RETURN (N2),
$EVAL_{N2.val}$, RETURN (N1), $EVAL_{N1.val}$

$EVAL_{N2.place}$, $VISIT_1$ (N2), $EVAL_{N3.place}$,
$VISIT_1$ (N3), $EVAL_{N5.place}$, $VISIT_1$ (N5),
$EVAL_{N8.place}$, $VISIT_2$ (N8), $EVAL_{N8.val}$,
RETURN (N5), RETURN (N3), $EVAL_{N6.place}$,
$VISIT_2$ (N6), $EVAL_{N6.val}$, RETURN (N3),
RETURN (N2), $EVAL_{N4.place}$, $VISIT_2$ (N4),
$EVAL_{N4.val}$, RETURN (N2), $VISIT_1$ (N3),
$VISIT_1$ (N5), $EVAL_{N7.place}$, $VISIT_1$ (N7),
$EVAL_{N9.place}$, $VISIT_1$ (N9), $EVAL_{N9.val}$,
RETURN (N7), $EVAL_{N7.val}$, RETURN (N5),
$EVAL_{N5.val}$, RETURN (N3), $EVAL_{N3.val}$,
RETURN (N2), $EVAL_{N2.val}$, RETURN (N1),
$EVAL_{N1.val}$

Figure 7: *Two different computation sequences for the semantic tree of figure 2*

Asymptotically the optimal evaluator, topological sort, and tree-walk evaluators all perform work proportional to the number of attributes in the tree. For real systems constant factors become important and in this survey of tree-walk evaluators one criterion that shall be considered is how close a given evaluator is to the optimal one. It is not that we actually wish to build optimal evaluators- as mentioned above that is too hard and unnecessary. But it is important for the evaluator not to spend *too* much time on simply visiting nodes, lest most of its time is spent executing $VISIT_k$ instructions instead of actually evaluating the attribute instances of the tree[4]. Some tree-walk evaluators we shall look at, for example, in the worst case must visit each production-instance in the entire tree to evaluate a single attribute. When this occurs the cost of visiting nodes in the tree drastically increases, dominating the cost of evaluating a semantic tree. Minimizing the number of $VISIT_k$ instructions an evaluator uses has been a major motivation behind the development of the tree-walk evaluators described in this paper.

[4]This is especially true if the entire semantic tree cannot fit into the working memory of the machine. In such a case additional visits often mean additional IO operations (as that part of the tree may not be in working memory but must be fetched from secondary memory) and can be quite costly. Hence minimizing the number of visits becomes very important.

3. Static Tree-Walk Evaluators

The remainder of this paper will deal with *static* tree-walk evaluators. This class includes most tree-walk evaluators which have been proposed in the literature and almost all which have been implemented to date. This is due to the fact that most static evaluators recognize a large class of attribute grammars, are not overly complex to construct, and are farily efficient in their evaluation of semantic trees. The difference between the various static evaluators themselves lie in how they choose to balance the tradeoffs in efficiency versus complexity; the simplest evaluators to build are not as efficient as those requiring a more complicated construction and cannot be built for as many attribute grammars as their more complex counterparts.

Static tree-walk evaluators are *static* in the sense that they do not *dynamically* generate a sequence of $EVAL_{att}$ and $VISIT_k$ instructions to execute upon visiting a node but they always execute a precomputed sequence of instructions. That is, upon arriving at a node they do not examine the attribute dependencies and then decide which attributes to evaluate and which children to visit but they merely retrieve and execute some sequence of instructions which has been precomputed to obey the appropriate attribute dependencies. These evaluators are not allowed to use global information concerning the nature of the semantic tree (such as dependency relations found in a particular subtree of a node) in deciding which sequence of instructions to execute but only *local* information concerning which production applies at the locus of control (the current production-instance) and which productions apply at adjacent production-instances in the semantic tree. Some strategies which violate these constraints have been proposed in order to extend the power of the evaluator but they will not be discussed in any detail in this paper.

Formally, a static tree-walk evaluator consists of a set of mutually recursive procedures where each procedure is associated with a production $[p: X_0 ::= X_1 \cdots X_{np}]$ and consists of a sequence of $EVAL_{att}$ and $CASEVISIT_k$ instructions. The execution of a $EVAL_{att}$ instruction will cause the appropriate attribute of the production to be evaluated. The execution of a $CASEVISIT_k$ instruction ($k = 1...n_p$) will cause a visit to X_k, the k^{th} child of the production and also determines which procedure will begin execution upon arrival at that node. A $CASEVISIT_k$ instruction has the form:

Case LP production of X_k = p_1: $VISIT_k$ and call $proc_1$;
p_2: $VISIT_k$ and call $proc_2$;
.
.
p_n: $VISIT_k$ and call $proc_n$;

Note that the code executed upon arriving at X_k is determined soley by which production applies there. If X_k is the LHS of m unique productions then there will be m conditional parts to the case expression. Each $proc_j$ procedure of the $CASEVISIT_k$ statement is itself one of the recursive routines of the evaluator and hence will consist of a sequence of $EVAL_{att}$ and $CASEVISIT_k$ instructions. After the last instruction of the procedure is executed a RETURN instruction is automatically performed, the locus of control returns to the adjacent production and the execution of instructions in the calling procedure resumes.

For example, if a procedure contains the subsequence $<EVAL_{att1}, CASEVISIT_i, EVAL_{att2}>$, after att1 is evaluated the $CASEVISIT_i$ instruction is executed causing a visit to X_i and the execution of a sequence of instructions in some procedure $proc_j$. After execution of the last instruction in $proc_j$ the locus of control returns to X_i's RP production and the $EVAL_{att2}$ instruction is executed. Evaluation of the semantic tree begins by initially invoking a special procedure to visit the root of the semantic tree and evaluation finishes when control returns from this procedure.

Any *correct* evaluator for an attribute grammar G which can be built in the above form is said to be a *static tree-walk evaluator* for G. By a *correct* evaluator we mean that for *any* valid semantic tree of G the evaluator must finish evaluation with the special attributes of the root computed and without having computed any attribute of the semantic tree twice.

Although we have defined a static tree-walk evaluator in terms of recursive procedures, we could just as well have defined it in terms of coroutines, stack automata, or finite automata. Our choice in couching the definition in terms of a specific implementation is based upon our desire to present concrete examples of evaluators and a complete description of the construction process. In [21], Kastens shows how his evaluator could be implemented by any of the above methods. Similar implementations could be built for any static tree-walk evaluator as well.

In the next 3 sections, as we survey *static tree-walk evaluators*, we shall attempt to find the unifying concepts behind them and a general criteria by which to judge them. One criterion we shall use is how close a given strategy comes to an optimal one in the sense mentioned in section 2. But this shall not be our only consideration; we shall also ask the following questions about a strategy: Can it be used to build an evaluator for any well-defined attribute grammar? If not, for what class of attribute grammars will such a strategy work? How hard is it to build such an evaluator? (Not surprisingly, the more efficient the evaluator is, the harder it will be to construct it). How large is the evaluator? And finally, how do we automate the building of an attribute grammar evaluator using this strategy; that is, given an attribute grammar, how do we automatically generate an evaluator which will evaluate any semantic tree of the grammar based on the given strategy?

The first type of static evaluators we shall discuss are the *pass-oriented evaluators*. These are obtained by restricting the nature of the instruction sequences allowed in the recursive procedures of the evaluator. These restrictions insure that the evaluator will always evaluate semantic trees by making left-to-right (LR) or right-to-left (RL) depth-first passes over the tree. The next type of static tree-walk evaluators we shall look at are the *uniform evaluators*. These evaluators assign a unique order to the attributes of a context-free symbol called a *protocol* such that for any instance of the symbol in a semantic tree the attribute-instances are evaluated in the given order. As we shall see, however, for many attribute grammars it is not possible to find a *unique* protocol for every context-free symbol and still evaluate every semantic tree. This will lead us to the *multi-protocol evaluator* which assigns a *set* of protocols to each context-free symbol. The order of computing attributes in this

evaluator must be consistent with some protocol of each context-free symbol but not with every one.

The order in which we present these evaluators corresponds to their increasing complexity and size. Pass-oriented evaluators are easy to construct but can be far from optimal in their evaluation of semantic trees. Furthermore, they can be built for only a small subset of all attribute grammars. Uniform evaluators can be constructed for a larger class of attribute grammars and are much more efficient in their evaluation of semantic trees. Multi-protocol evaluators can be built for an even larger subset of attribute grammars yet are still as close to optimal as uniform evaluators. The size of the resulting evaluator, however, can be significantly larger than other static tree-walk evaluators.

4. Pass-oriented Evaluators

In a pass-oriented evaluator, every attribute X.a of the grammar is assigned a pass number, PN(X.a). Let K_{max} be the highest numbered pass number assigned to any attribute. The visit sequence resulting from the evaluation of any semantic tree will consist of $VISIT_k$ instructions defining K_{max} consecutive left-to-right or right-to-left depth-first passes over the tree. On the i^{th} pass over the tree, before visiting any node labeled X, all the inherited attributes of X with a pass number = i are evaluated. Before returning from X (i.e., before transferring the locus-of-control from the X's left-part production to X's right-part production) all the synthesized attributes of X with a pass number = i are evaluated. After the K^{th}_{max} pass over the semantic tree all the attributes of the tree have been computed and evaluation is complete. It is important to note that if an attribute X.a has pass number = i then *every* occurrence of X.a in *every* semantic tree will be evaluated on the i^{th} pass.

The start procedure of a pass-oriented evaluator is given below. It consists of K_{max} visits to the root node labeled S.

```
proc_S
begin
CASEVISIT_S^1, ... ,CASEVISIT_S^Kmax
end;
```

In addition to this procedure there exists K_{max} procedures for every production of the grammar, one for each pass over the tree. They shall be referred to $proc_{p(1)}, proc_{p(2)}, \ldots, proc_{p(Kmax)}$ where $proc_{p(i)}$ is the procedure associated with the i^{th} visit to the production p occurring during the i^{th} pass over the tree. The figure below gives the form of a typical procedure $proc_{p(i)}$ where the i^{th} pass over the tree is left-to-right and where p is of the form $[p: X_0 ::= X_1 \cdots X_{np}]$.

```
proc_p(i)
begin
EVAL_att instructions for each inherited attribute of X_1 with pass number = i);
CASEVISIT_1;
EVAL_att instructions for each inherited attribute of X_2 with pass number = i);
CASEVISIT_2;
   .
   .
EVAL_att instructions for each inherited attribute of X_np with pass number = i);
CASEVISIT_np;
EVAL_att instructions for each synthesized attribute of X_0 with pass number = i);
end;
```

Each $CASEVISIT_k$ instruction is of the form :

```
Case LP production of X_k = p_1: VISIT_k and call proc_p1(i);
                            p_2: VISIT_k and call proc_p2(i);
                                  .
                                  .
                            p_n: VISIT_k and call proc_pn(i);
```

A pass-oriented evaluator may make LR passes over a tree, RL passes, or a mixture of LR and RL passes. Such an evaluator is called a LR-pass, RL-pass, and alternating-pass evaluator respectively. If the i^{th} pass over the tree is to be RL instead of LR then the procedure $proc_{p(i)}$ differs from the one given above in that the order of evaluating attributes and visiting nodes is reversed. First X_n's inherited attributes are evaluated and X_n is visited, then X_{n-1}'s inherited attributes are evaluated and X_{n-1} is visited and so on until X_1 is visited. Once again the synthesized attributes of X_0 are evaluated last.

Figure 8 gives a small attribute grammar and a pass assignment PN. Figure 9 presents the pass-oriented evaluator constructed for this attribute grammar using the pass assignment PN. The evaluator will make 2 passes over any semantic tree, the first being a LR pass and the second a RL pass. Due to space considerations, the procedures $proc_{p2(1)}$, $proc_{p2(2)}$, $proc_{p4(1)}$, and $proc_{p4(2)}$ are omitted.

```
PN(S.translation) = 2.
PN(A.inhatt) = 1.
PN(A.synatt) = 1.
PN(B.inhatt) = 2.
PN(B.synatt) = 2.
```

Pass numbers for the attributes

```
p0: S ::= A B.
        A.inhatt = constant1;
        B.inhatt = constant2;
        S.translation = f(A.synatt,B.synatt);

p1: A0 ::= A1 A2.
        A1.inhatt = g(A0.inhatt);
        A2.inhatt = h(A1.synatt);
        A0.synatt = l(A2.synatt);

p2: B0 ::= B1 B2.
        B1.inhatt = q(B2.synatt);
        B2.inhatt = r(B0.inhatt);
        B0.synatt = s(B1.synatt);

p3: A ::= a.
        A.synatt = k(A.inhatt);

p4: B ::= b.
        B.synatt = t(B.inhatt);
```

Figure 8: *An attribute grammar and pass assignment PN*

Pass-oriented evaluators were first presented by Bochmann as making only left-to-right passes over a semantic tree [1]. It soon became apparent that the method could be enhanced by allowing right-to-left passes over the tree, as information often flows in that direction in semantic trees [17, 32]. If we look at the attribute grammar of figure 8, for instance, we see that by only allowing LR passes we cannot bound the number of passes that will be needed to evaluate a semantic tree: the deeper the tree becomes, more passes will be required. In particular, we cannot assign a pass number N to B.inhatt such that *every* instance of B.att in *every* semantic tree could be evaluated on the N^{th} pass. Since no evaluator making only LR passes could be constructed for this attribute grammar, it is not in the class of *LR-pass attribute grammars*. It is, however, in the class of *alternating pass attribute grammars* as the evaluator given in figure 9 bears witness.

```
proc_S
        begin
        Case LP production of S = p0: VISIT_S and call proc_p0(1);
        Case LP production of S = p0: VISIT_S and call proc_p0(2);
        end;

proc_p0(1) begin
        EVAL_A.inhatt;
        Case LP production of A = p1: VISIT_1 and call proc_p1(1),
                                  p3: VISIT_1 and call proc_p3(1);
        Case LP production of B = p2: VISIT_2 and call proc_p2(1),
                                  p4: VISIT_2 and call proc_p4(1);
        end;

proc_p0(2) begin
        EVAL_B.inhatt;
        Case LP production of B = p2: VISIT_2 and call proc_p2(2),
                                  p4: VISIT_2 and call proc_p4(2);
        Case LP production of A = p1: VISIT_1 and call proc_p1(2),
                                  p3: VISIT_1 and call proc_p3(2);
        EVAL_S.translation;
        end;

proc_p1(1) begin
        EVAL_A1.inhatt;
        Case LP production of A1 = p1: VISIT_1 and call proc_p1(1),
                                   p3: VISIT_1 and call proc_p3(1);
        EVAL_A2.inhatt;
        Case LP production of A2 = p1: VISIT_1 and call proc_p1(1),
                                   p3: VISIT_1 and call proc_p3(1);
        EVAL_A0.synatt;
        end;

proc_p1(2) begin
        Case LP production of A2 = p1: VISIT_1 and call proc_p1(2),
                                   p3: VISIT_1 and call proc_p3(2);
        Case LP production of A1 = p1: VISIT_1 and call proc_p1(2),
                                   p3: VISIT_1 and call proc_p3(2);
        end;

proc_p3(1) begin                    proc_p3(2) begin
        EVAL_A.synatt                       {null}
        end;                                end;
```

Figure 9: *The pass-oriented evaluator for the attribute grammar of figure 8*

Let $D = <D_1,....,D_{Kmax}>$ be a sequence of *directions* where each D_i is either LR or RL indicating that the i^{th} pass is either LR or RL and there are to be k_{max} passes. If PN is

an assignment of pass numbers to the attributes of the grammar such that *i*) for every attribute X.a, $1 \leq PN(X.a) \leq k_{max}$ and *ii*) for *any* instance of X.a in *any* semantic tree of the grammar, X.a can be evaluated on the $PN(X.a)^{th}$ pass, then PN is said to be *valid* for D.

Given a sequence D and a valid assignment PN for D, it is easy to create an evaluator for the grammar. First construct the procedure $proc_S$. Then, for each production p and for each i, $1 \leq i \leq k_{max}$, add a procedure $proc_{p(i)}$ which visits each child and evaluates the attributes of the production with pass number = i, as was done for the grammar of Figure 8. The main challenge in building the evaluator becomes one of finding such a sequence D and valid assignment PN. This is done incrementally: First a pass direction D_1 and a set of attributes S_1 are found such that for every attribute in S_1, *any* instance of that attribute in *any* semantic tree can be evaluated on the first pass in direction D_1. Each attribute X.a $\in S_1$ is assigned a pass number PN(X.a) = 1. Assuming that all attribute-instances of the attributes in $S_1 \cup S_2 ... \cup S_{i-1}$ in any semantic tree can be evaluated before the i^{th} pass, the i^{th} step consists of finding a pass direction D_i and a set of attributes S_i such that all attribute-instances of attributes in S_i in any semantic tree can be evaluated on the i^{th} pass in direction D_i. Each attribute X.a $\in S_i$ is assigned a pass number PN(X.a) = i. The algorithm terminates upon one of two conditions:

i) All the attributes of the grammar have been given a pass number. In this case a sequence $<D_1,...,D_{Kmax}>$ and a valid assignment PN to the attributes have been found upon which to build the pass-oriented evaluator.

ii) There exists attributes which have not yet been assigned pass numbers yet no pass direction D_i can be found which will produce a non-empty set S_i of attributes to evaluate on the i^{th} pass. This means that there exists at least one attribute X.a which cannot be assigned a pass number; depending upon where the attribute instance occurs in the tree it must be evaluated on *different* passes over the tree. If this condition holds then the attribute grammar is not in the class of *alternating pass-oriented attribute grammars*. Figure 10 below gives a simple attribute grammar which derives only one tree but is not in the class of alternating pass-oriented attribute grammars. This is because in the tree which this grammar derives, some instances of list.inhatt (namely $list_1$.inhatt and $list_3$.inhatt) need to be evaluated on the pass before other instances of this attribute (namely $list_2$.inhatt)[5].

Using the incremental approach described above, depending upon which direction is chosen for D_i (either LR or RL) the set S_i may differ. Let S_{iLR} be the set chosen if D_i = LR

[5]Note, that if we were to introduce a new context-free symbol Y, change the production p_0 to p_0': S ::= $list_1$ Y $list_2$ and add a production p_2: Y ::= term making the appropriate changes to the semantic functions, the attribute grammar would be evaluable in 2 RL or 2 LR passes: Evaluate Y.inhatt, Y.synatt, and S.translation on the second pass and all other attributes on the first pass. This illustrates the fact that often an AG can be *massaged* to make it evaluable by a given strategy. In [9], Farrow explores this issue for certain classes of attribute grammars.

```
P0: S ::= list1 list2 list3.
        list1.inhatt = const;
        list3.inhatt = const;
        list2.inhatt = f(list1.synatt,list3.synatt);
        S.translation = list2.synatt;

P1: list ::= term.
        list.synatt = g(list.inhatt);
```

Figure 10: *An attribute grammar not evaluable by an alternating pass strategy*

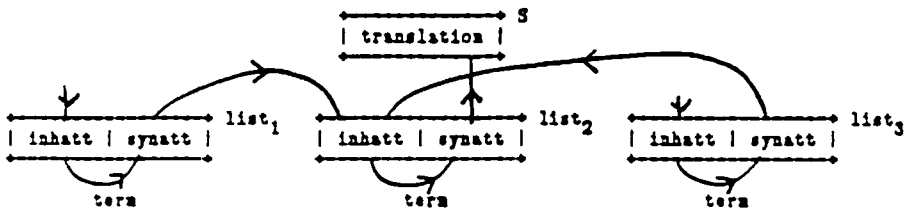


Figure 11: *The unique semantic tree that the attribute grammar of figure 10 derives*

and S_{iRL} the set chosen if D_i = RL. In deciding which direction to make D_i, both S_{iLR} and S_{iRL} are computed. If $S_{iRL} \subseteq S_{iLR}$ then D_i = LR is chosen. If $S_{iLR} \subseteq S_{iRL}$ then D_i = RL is chosen. If neither of these conditions hold then D_i is arbitrarily chosen to be the direction opposite of the one chosen for D_{i-1}. It has been shown that making this arbitrary choice can result in an evaluator which makes about twice as many passes over the semantic trees then needed [33]. Finding the direction which would minimize the total number of passes required, however, is an NP-complete problem [33]. Figure 12 gives the algorithm described above for the construction of a sequence of directions D and a valid assignment PN.

This algorithm calls for the computation of the sets S_{iLR} and S_{iRL} given the sets $S_1,\ldots,S_{i-1}$. To see how these sets can be computed note that an attribute X.a can be evaluated during pass i iff it will be the case that for any occurrence of X.a in any semantic tree dependent upon an occurrence of Y.a, either Y.a was defined on an previous pass (i.e., PN(Y.a) < i) or Y.a is defined on the same pass but *before* the time to evaluate X.a. Using the Dp graphs introduced at the end of section 1, this condition can be made more precise as follows:

Let X.a be an inherited attribute of the grammar. Then X.a is evaluable on the i^{th} pass *iff* for each production [p: $X_0 ::= X_1 \cdots X_{np}$] with $X = X_j$, $j > 0$, if $\exists$ an edge (X_k.att, X_j.a) in Dp then either

i) $PN(X_k.att) < i$ or

ii) $k = 0$ and $PN(X_k.att) = i$ or

iii) D_i = LR, $PN(X_k.att) = i$, and $1 \leq k < j$ or

iv) D_i = RL, $PN(X_k.att) = i$, and $j < k \leq n_p$.

```
begin
NOT_YET_ASSIGNED := all the attributes of the grammar;
i := 1;
Compute S_ILR and S_IRL;
While (S_ILR ≠ ∅) and (S_IRL ≠ ∅) do
        begin
        if S_IRL ⊆ S_ILR then D_i := LR and S_i := S_ILR
        else if S_ILR ⊆ S_IRL then D_i := RL and S_i := S_IRL
        else if i = 1 then D_i := LR and S_i := S_ILR
        else if D_i-1 = RL then D_i := LR and S_i = S_ILR
        else D_i := RL and S_i := S_IRL
        endif;
        For each attribute X.a ∈ S_i do PN(X.a) := i;
        NOT_YET_ASSIGNED := NOT_YET_ASSIGNED - S_i;
        k_max := i;
        i := i + 1;
        Compute S_ILR and S_IRL;
        end;
If NOT_YET_ASSIGNED ≠ ∅ then return("a pass-oriented evaluator cannot be built for the grammar")
else return(<D = D_1,....D_kmax>, PN);
/* D is the sequence of directions for the evaluator and PN defines a valid assignment for D */
end;
```

Figure 12: *An algorithm for assigning pass numbers to attributes*

If these conditions are not met then there exists some semantic tree with an occurrence of X.a dependent upon an occurrence of X_k.att where X_k.att will defined on the i^{th} pass or later and where the order of evaluation is such that X.a must be defined on a pass after X_k.att is evaluated. An example of such a case is given below in figure 13. In that example, if D_i = LR and PN(X_k.att) = i then PN(X_j.a) > i as the pass-oriented evaluator strategy calls for evaluating attributes of X_j before those of X_k during an LR pass.

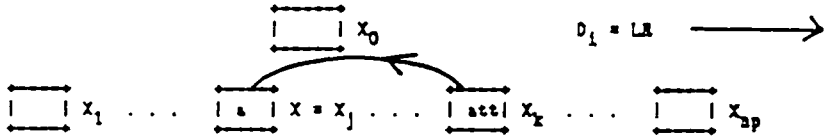


Figure 13: *Two attributes not evaluable on the same pass*

For a synthesized attribute an even simpler condition can be formulated: Let X.a be a synthesized attribute of the grammar. Then X.a is evaluable on the i^{th} pass *iff* for each production [p: X_0 ::= X_1 ··· X_{np}] with X = X_0 if ∃ an edge (X_k.att, X_0.a) in Dp then PN(X_k.att) ≤ i. This simple condition is due to the fact that the synthesized attributes of X_0 are the last to be evaluated in the procedures of the pass-oriented evaluator.

Using these conditions the algorithm of figure 12 computes the sets S_{iLR} and S_{iRL} given the sets $S_1,...,S_{i-1}$ of attributes already assigned pass numbers. Initially all the attributes in NOT_YET_ASSIGNED are candidates to be evaluated on the i^{th} pass. These attributes are examined and dismissed as candidates if they are found to violate the above conditions. As the the removal of one attribute may cause another attribute to violate the above conditions this step must be performed iteratively (taking the transitive closure) until stable sets S_{iLR} and S_{iRL} are found.

An evaluator strategy closely related to the pass-oriented strategy is one which makes *sweeps* over a semantic tree instead of passes. A sweep over a tree visits each node of the tree once in a depth-first manner, but not necessarily in a strictly left-to-right or right-to-left pattern. It may first visit the entire subtree rooted at X_k, then the subtree rooted at X_{k-1}, and finally the subtree rooted at X_{k+1}. The reader is referred to [7] for a comparison of this strategy to a pass-oriented one.

Because pass-oriented evaluators visit every node in the tree during each pass they can be extremely inefficient. If we look back at the example of figure 9, for example, we see that during the first pass over any semantic tree of the attribute grammar no attributes of nodes labeled B are evaluated. Similarly during the second pass no attributes of nodes labeled A are evaluated. As there can be an arbitrary number of these nodes in a semantic tree, the evaluator can consume much time performing $VISIT_k$ instructions which are not at all needed. This inefficient evaluation behavior was a prime motivation for the development of the evaluators to be discussed in the next two sections. In contrast to pass-oriented evaluators, these evaluators will only visit a node if doing so will cause it to evaluate at least one attribute of that node. This will not necessarily result in an optimal evaluator but does guarantee a much more efficient one.

5. Uniform Evaluators

Uniform evaluators [38], like pass-oriented evaluators, assign a number to every attribute of the grammar. As a uniform evaluator is not restricted like the pass-oriented evaluator in making passes over the tree, this number no longer corresponds to a pass number but to a *visit number*. That is, if an inherited attribute X.a has a visit number = i, then immediately before the i^{th} visit to the left-part production of an instance of X, X.a must be evaluated. Similarly, if a synthesized attribute X.b has a visit number = i, then before returning from the i^{th} visit to the left-part production of an instance of X, X.b must be evaluated. Bearing this in mind, we define the concept of a *protocol* for a context-free symbol X.

Let X be a nonterminal symbol of an attribute grammar. Then a *protocol* $\pi_X = \pi(1),....\pi(2i\text{-}1),\pi(2i),...,\pi(2n_X)$ for X is a sequence of sets of attributes of X such that

1. There exisits an even number of elements in the sequence.
2. Even elements of the sequence consist entirely of synthesized attributes, odd elements consist entirely of inherited attributes.
3. $A(X) = \pi(1) \cup ... \cup \pi(2n_X)$
4. Every set $\pi(i)$, $i > 1$, is non-empty.

A protocol for X is basically just an assignment of visit numbers to the attributes of X. Let $\pi_X = \pi(1),...,\pi(2i\text{-}1),\pi(2i),...,\pi(2n_X)$ be a protocol. The uniform evaluator will visit any instance of X in any semantic tree n_X times. Before the i^{th} visit it will evaluate the inherited attributes of $\pi(2i\text{-}1)$ [henceforth referred to as $\pi_X(2i\text{-}1)$]. Before returning from the i^{th} visit it will evaluate the synthesized attributes of $\pi_X(2i)$. n_X is called the *length* of the protocol π_X. For every production [p: $X_0 ::= X_1 \cdots X_{np}$] with $X = X_0$, the evaluator will have n_X routines, one for each of the n_X visits to X. The procedure associated with the i^{th} visit to a production p will be called $proc_{p(i)}$. It must evaluate the synthesized attributes in $\pi_{X0}(2i)$. What other instructions are in the procedure depends upon the dependencies of p and the protocols for $X_1,....,X_{np}$. This is best illustrated by an example. Let [p: $X_0 ::= X_1\ X_2$] be a production with dependencies as given in the graph Dp of figure 14 and let π_{X0}, π_{X1}, π_{X2} be protocols for X_0, X_1, and X_2 respectively.

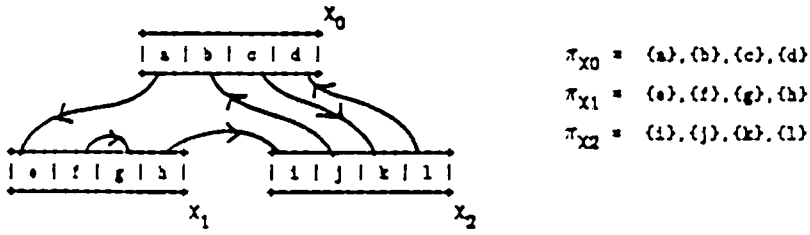


Figure 14: *A Dp graph and protocols for X_0, X_1, and X_2*

The protocol for X_0 calls for the procedure $proc_{p(1)}$ to evaluate X_0.b. (X_0.a, being an inherited attribute of the LHS node, would be evaluated before visiting X_0 and activating

this procedure). As this attribute is dependent upon X_2.j, the procedure must visit X_2 evaluating the attributes in $\pi_{X2}(2) = \{X_2.j\}$. But the protocol for X_2 requires that X_2.i be evaluated before visiting X_2 for the first time so this attribute must also be evaluated in $proc_{p(1)}$. This in turn requires that X_1 be visited, X_1.h be evaluated, and so forth. In this way production dependencies and protocols for the context-free symbols of the production interact to determine the procedures of the evaluator. In this example we can determine that on the first visit to p the attributes { X_1.e, X_1.g, X_2.i, X_0.b } must be evaluated and on the second visit { X_2.k, X_0.d } must be evaluated. In addition some of the RHS nodes must be visited in between evaluating these attributes according to protocol specifications. These considerations completely determine the form of the procedures $proc_{p(1)}$ and $proc_{p(2)}$, corresponding to the 1st and 2nd visits to p, to be:

```
proc_p(1)
    Begin
    EVAL_X1.e;
    CASEVISIT_1^1;
    EVAL_X1.g;
    CASEVISIT_1^2;
    EVAL_X2.i;
    CASEVISIT_2;
    EVAL_X0.b;
    End;
```

```
proc_p(2)
    Begin
    EVAL_X2.k;
    CASEVISIT_2;
    EVAL_X0.d;
    End;
```

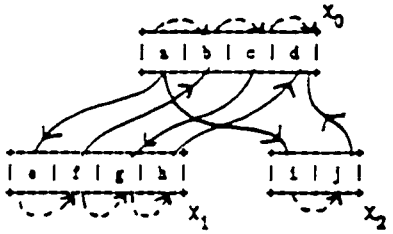
In $proc_{p(1)}$ we are guaranteed that after the execution of the $CASEVISIT_1^1$ instruction control will transfer to some procedure $proc_{p'(1)}$ where p': $Y_0 ::= Y_1 \ldots Y_m$ with $Y_0 = X_1$ and in that procedure X_1.f will be evaluated. This is because in constructing the procedure $proc_{p'(1)}$ we make use of the same protocol π_{X1} as we did in constructing the procedure $proc_{p(1)}$ and this protocol demands that X_1.f (= Y_0.f) be evaluated on the 1st visit to its left-part production; the protocols guarantee that the procedures will *fit together*.

The procedures of a uniform evaluator allow a single procedure to visit a node any number of times and in any order. Furthermore, a node will only be visited if such a visit will cause an attribute of the node to be evaluated. Contrast this to the procedures of a pass-oriented evaluator where a node is visited exactly once per procedure whether or not such a visit will cause an additional attribute evaluation.

It is crucial to note that not *any* protocols for the nonterminals will allow us to create a uniform evaluator. To see why this is true, consider the previous example again, only substitute $\pi'_{X0} = \{c\}, \{d\}, \{a\}, \{b\}$ in place of π_{X0} as the protocol for X_0. On the first visit to p, this protocol requires that X_0.d be evaluated. Because of the dependencies in p and the semantics of the protocols, X_2.l and hence all of X_2's attributes must also be evaluated during this visit. As some of them are dependent upon X_0.a, this in turn means that X_0.a must be evaluated before visiting this production and in particular, before evaluating X_0.d. But this *violates* the protocol π'_{X0} which calls for the evaluation of X_0.d

before X_0.a! Hence from the protocols π'_{X0}, π_{X1}, and π_{X2} it is not possible to create a uniform evaluator. A set of protocols (one for each nonterminal of the grammar) is said to be *consistent* iff a uniform evaluator can be built from them. Similarly a protocol for a symbol X is said to be *consistent* if there exists some uniform evaluator using that protocol for X.

Assuming that a consistent set of protocols for an attribute grammar has been found, it is an easy task to create the uniform evaluator for it. The evaluator will consist of the start procedure $proc_S$ and n_X procedures for each production [p: X_0 ::= $X_1 \cdots X_{np}$] where n_X is the length of the protocol for X_0. The procedure $proc_S$ will contain just one $CASEVISIT_k$ instruction. Each procedure $proc_{p(i)}$, $1 \leq i \leq n_X$ evaluates the synthesized attributes of X_0 in $\pi_{X0}(2i)$ as well as other inherited attributes of the right-part symbols. Which inherited attributes of the right-part symbols are evaluated and which $CASEVISIT_k$ instructions are performed is determined by the dependencies of p and the other protocols of the production, as illustrated in the example above. Actually, the construction is slightly more complicated than that example would indicate. This is because the dependencies of p and the protocols for the symbols of p will often only indicate a *partial order* of evaluation of the attributes of p and to unambiguously construct the procedures of p a *total order* on the attributes is required. This complication is easily resolved by choosing *any* total order for the attributes of p compatible with the partial order. This can best be illustrated by the following example:



π_{X0} = (a), (b), (c), (d)
π_{X1} = (e), (f), (g), (h)
π_{X2} = (i), (j)

Protocols for X_0, X_1, and X_2

Here the dependencies of p and the protocols π_{X0},π_{X1}, and π_{X2} describe only a partial order on the attributes of p. <X_0.a, X_1.e, X_1.f, X_0.b, X_0.c, X_1.g, X_1.h, X_2.i, X_2.j, X_0.d> and <X_0.a, X_1.e, X_1.f, X_2.i, X_2.j, X_0.b, X_0.c, X_1.g, X_1.h, X_0.d>, for example, are both total orders compatible with the partial order. On the first visit to this production either total order calls for the procedure to evaluate X_1.e, visit X_1, and evaluate X_0.b. On the second visit both must evaluate X_1.g, visit X_1, and evaluate X_0.d. On *one* of these visits X_2.i must be evaluated and X_2 visited. This must be done before evaluating X_0.d but exactly when (in either the first or second visit) makes no difference and does not significantly effect the evaluator. Hence this ambiguity can be resolved arbitrarily by choosing some total order compatible with the partial order and appropriately dividing it up into visit sequences.

How are a consistent set of protocols for the nonterminals of the grammar found? Unfortunately, Engelfreit and Filè show in [6] that finding these protocols, if they exist at all, is an NP-complete problem. That is, it is an Np-complete problem to determine if an

attribute grammar is in the class of *uniform attribute grammars* and if it is, it is just as hard to determine a consistent set of protocols for the grammar. Still, for some subsets of this class a consistent set of protocols can be found in polynomial time. The rest of this section will examine how this is done but first a a more precise method of determining the consistency of protocols needs to be formulated.

For any protocol π_X, one can form the *protocol graph* $\delta_X = (V,E)$ where $V = A(X)$ and $\exists$ a path from X.a to X.b iff $X.a \in \pi_X(m)$ and $X.b \in \pi_X(m+k)$ for some $k > 0$[6]. A protocol graph is always acyclic since if there exists a path in δ_X from X.a to X.b then the protocol π_X specifies that X.a is to be evaluated before X.b. Given a protocol graph δ_X it is always possible to reconstruct the protocol π_X from which it was derived. Note that in δ_X, for every inherited attribute I and synthesized attribute S either $I \Rightarrow S$ or $S \Rightarrow I$. Moreover, these edges alone are sufficient to reconstruct the protocol associated with this protocol graph. Using protocol graphs it is possible to formalize when a set of protocols are consistent for a given attribute grammar; the following theorem is proved in [6].

> **Theorem 1:** A set of protocols for an attribute grammar G is consistent iff for each production $[p: X_0 ::= X_1 \cdots X_{np}]$ in G, $Dp[\delta_{X0},...,\delta_{Xnp}]$ is acylic, where δ_{Xi} is the protocol graph for X_i's protocol.

Intuitively this is so is because an edge $(X_j.a,X_k.b)$ in a graph $Dp[\delta_{X0},...,\delta_{Xnp}]$ means that for any instance of p in any semantic tree, the occurrence of $X_j.a$ must be evaluated before the occurrence of $X_k.b$. If the edge is from the Dp graph then this requirement stems from the fact that $X_k.b$ is dependent upon $X_j.a$. If the edge is from a δ_X graph then this requirement is due to the semantics of the protocol. If the graph $Dp[\delta_{X0},...,\delta_{Xnp}]$ contains a circularity then this requirement states that some attribute X.a must be evaluated before X.a, which is clearly not possible. If there are no circularities, however, then we can create a *total order* from the *partial order* given in $Dp[\delta_{X0},...,\delta_{Xnp}]$. This total order, together with the protocols $\pi_{X0},...,\pi_{Xnp}$ will unambiguously determine the procedures $proc_{p(1)},...,proc_{p(m)}$. We know that all these procedures will *fit together* as they all make use of the same protocols for the symbols of the grammar.

In figure 15 we give the graphs $Dp[\delta_{X0},\delta_{X1},\delta_{X2}]$ and $Dp[\delta'_{X0},\delta_{X1},\delta_{X2}]$ for the example of figure 14 above. The graph $Dp[\delta'_{X0},\delta_{X1},\delta_{X2}]$ contains a cycle since the protocols π'_{X0},π_{X1}, and π_{X2} are not consistent. The graph $Dp[\delta_{X0},\delta_{X1},\delta_{X2}]$, however, is acyclic since the protocols π_{X0},π_{X1}, and π_{X2} are consistent.

The above theorem also suggests a method of computing consistent protocols for the nonterminals of an attribute grammar. For each nonterminal X of the grammar one starts with a graph G_X having attributes of X as vertices and no edges. Edges are added

---

[6] According to this definition many different protocol graphs can be constructed for a given protocol. All of these graphs, however, have the same *transitive reduction* graph. As this transitive reduction graph is itself a protocol graph (with the fewest possible number of edges), we shall take it to be the unique protocol graph for a given protocol.

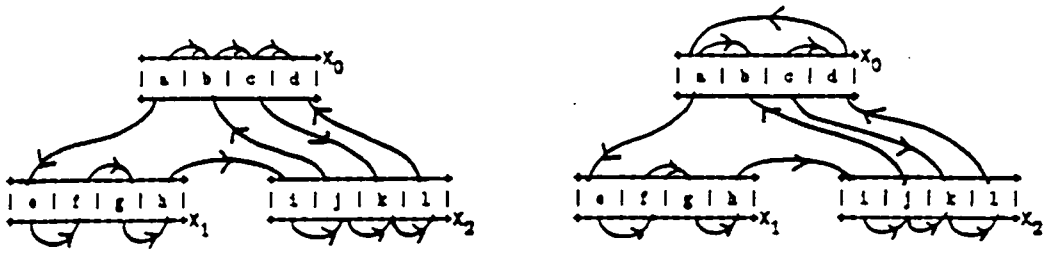


Figure 15: *The acyclic graph $Dp[\delta_{X0}\delta_{X1}\delta_{X2}]$ and the cyclic graph $Dp[\delta_{X0}\delta_{X1}\delta_{X2}]$*

incrementally from $X.a_1$ to $X.a_2$ if it is determined (by examining an augmented dependency graph) that an instance of $X.a_1$ must be evaluated before an instance of $X.a_2$ in some semantic tree using a uniform evaluation strategy. This process continues until no more edges can be added to any G_X graph. At each step in the process, each G_X graph can be considered as a rough draft of a protocol graph δ_X. The reason it is only a rough draft and not an actual protocol graph is because it may not have the property that for every inherited attribute I and synthesized attribute S of X, either $I \Rightarrow S$ or $S \Rightarrow I$. Even upon termination when no more edges can be added to any G_X graph, these graphs may still be only rough drafts and not actual protocol graphs. However, if at termination of this stage there exists an edge from $X.a_1$ to $X.a_2$ then $X.a_1$ must be evaluated before $X.a_2$ using a uniform evaluation strategy. Hence any edge in a G_X graph must exist in a consistent protocol graph for X. We shall refer to the $\{G_X\}$ graphs upon termination of this process as *pseudo-protocol graphs.*

The following two stage algorithm can be employed to find a consistent set of protocols for an AG (if one exists): first form the pseudo-protocol graphs $\{G_X\}$ as above, so that any edge from $X.a_1$ to $X.a_2$ in one of these graphs means that any uniform evaluator for the grammar must evaluate $X.a_1$ before $X.a_2$. Check and see if these pseudo-protocol graphs are consistent- the addition of these edges has not caused a circularity in an augmented dependency graph. If they are not consistent then the attribute grammar is not uniform. If they are consistent then stage two calls for *completing* these G_X graphs into actual protocol graphs such that for every inherited attribute I and synthesized attribute S of X, either $I \Rightarrow S$ or $S \Rightarrow I$. The addition of edges in stage two of the algorithm may also cause an inconsistency to arise in the augmented dependency graphs.

Stage 1 uses the *pseudo-protocol creation algorithm* [21] given in figure 16 which finds pseudo-protocol graphs $\{G_X\}$ for each nonterminal X of the grammar, adding an edge to G_X only if such an edge must exist in any consistent protocol graph for X. It begins by creating the graph $G_X = (V,E)$ for each nonterminal X of the grammar, with $V = A(X)$ and $E = \emptyset$. It then considers the various augmented dependency graphs $Dp[G_{X0},G_{X1},\ldots,G_{Xnp}]$ adding an edge $(X_i.a,X_i.b)$ to the graph G_{Xi} if $\exists$ a path from $X_i.a$ to $X_i.b$ in $Dp[G_{X0},G_{X1},\ldots,G_{Xnp}]$ and this edge is not already in G_{Xi}. After all such edges have been added, each augmented dependency graph $Dp[G_{X0},G_{X1},\ldots,G_{Xnp}]$ is examined. If any one is cyclic then the algorithm halts; in such a case the AG is not in the class of uniform attribute grammars. This algorithm was initially formulated by Kastens [21] as the computation of his IDS

relation. It is similar but not equivalent to Kennedy and Warren's computation of IO_X graphs which is presented in the next section. It can be implemented in terms of a transitive closure algorithm.

```
For each nonterminal X create the graph G_X = (V,E) where V = A(X) and E = ∅;
While an edge can be added to some graph G_X do
        Begin
        Choose a production [p: X_0 ::= X_1 ··· X_np] of the grammar;
        If ∃ a path from X_k.a to X_k.b in Dp[G_X0,G_X1,....,G_Xnp] and (X_k.a,X_k.b) is not in G_Xk
        Then add the edge (X_k.a,X_k.b) to G_Xk;
        End;
If for any production p of the grammar the graph Dp[G_X0,G_X1,....,G_Xnp] contains a cycle
Then return('grammar is not evaluable by a uniform evaluator')
Else return('the pseudo-protocol graphs:' {G_X} 'have been computed');
```

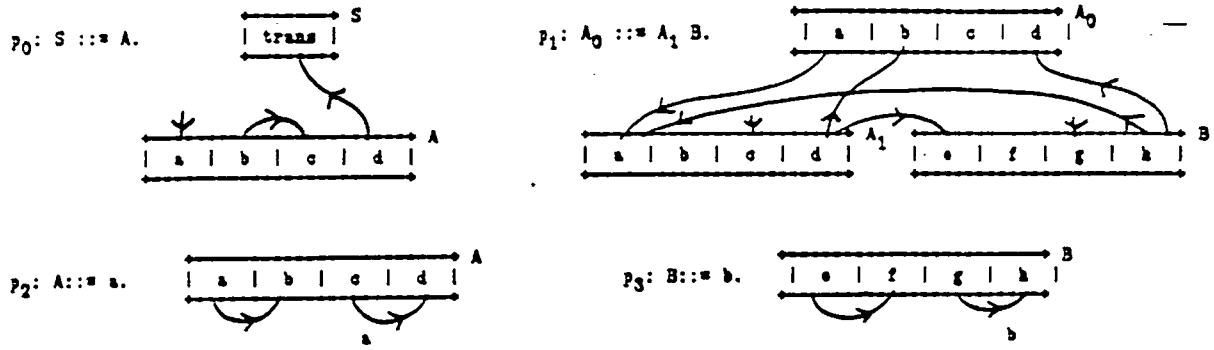
Figure 16: *The pseudo-protocol creation algorithm*

Assume this algorithm terminates returning the pseudo-protocol graphs $\{G_X\}$. If we look at these graphs, we may find that in the process of adding edges some of them have become actual protocol graphs whereas others remain only pseudo-protocol graphs; i.e., they need to be completed further in order to be actual protocol graphs. In any case, for *any* resulting G_X graph, if ∃ an edge (X.a,X.b) in this graph then any consistent protocol π_X for X must have X.a $\in \pi_X(m)$ and X.b $\in \pi_X(m+k)$ for some $k > 0$. This is due to the fact that an edge (X.a,X.b) was added only if some production required X.a to be evaluated before X.b. (See [11, 29]).

In the next figure we present an attribute grammar (giving the Dp graphs to describe dependencies instead of the semantic functions themselves) having 3 nonterminals S, A, and B. We give the graphs G_A and G_B that are formed by the pseudo-protocol creation algorithm.

The graph G_A is a protocol graph for A describing the protocol $\pi_A = \{a\},\{b\},\{c\},\{d\}$ whereas G_B is only a pseudo-protocol graph and could still be completed into 3 different protocol graphs corresponding to 3 different protocols: $\pi_B = \{g\},\{h\},\{e\},\{f\}$, $\pi'_B = \{e\},\{f\},\{g\},\{h\}$, and $\pi''_B = \{e,g\},\{f,h\}$. The first of these is a consistent protocol for B. The last two, however, introduce a cycle in the augmented dependency graph for Dp_1; i.e., the graphs $Dp_1[\delta_A,\delta_A,\delta'_B]$ and $Dp_1[\delta_A,\delta_A,\delta''_B]$ are cyclic, where δ_A is the graph G_A, δ'_B is the protocol graph formed from the protocol π'_B, and δ''_B is the protocol graph formed from the protocol π''_B. The graph $Dp_1[\delta_A,\delta_A,\delta''_B]$ is given below in figure 18.

The application of the pseudo-protocol creation algorithm completes the first stage in finding a consistent set of protocols for the AG. After this step it may have determined that the grammar is not in the class of uniform attribute grammars. If this is not the case, then a collection of pseudo-protocol graphs have been found which contain edges essential to any set of consistent protocols for the grammar. The second stage of the strategy is



The productions of the grammar and their Dp graphs

Figure 17: *Pseudo-protocol graphs formed by the algorithm of figure 16*

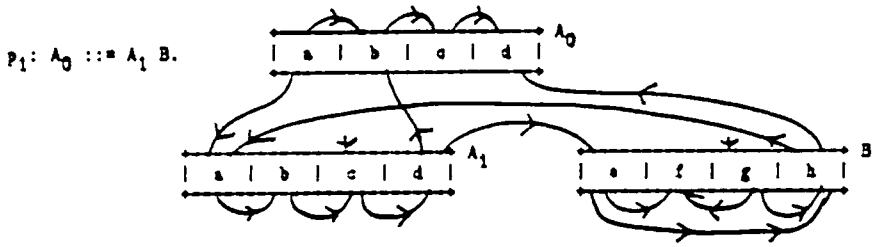


Figure 18: *The cyclic augmented dependency graph* $Dp_1[\delta_A, \delta_A, \delta''_B]$

concerned with further completing the $\{G_X\}$ pseudo-protocol graphs so that each graph becomes an actual protocol graph instead of a pseudo one and such that the resulting graphs make up a consistent set of protocols for the attribute grammar. In general, as stated earlier, this is an intractable problem. Instead an algorithm is chosen which completes the graphs according to some heuristic H. On applying heuristic H to the pseudo-protocol graphs $\{G_X\}$ a set of actual protocol graphs $\{\delta_X\}$ for the grammar is obtained. If these graphs determine a consistent set of protocols (which can be checked by examining whether or not the augmented dependency graphs are cyclic or not) then a uniform evaluator can be built based upon them. If they are found not to be consistent, however, then these cannot serve as the basis of a uniform evaluator. We cannot be sure (without exhaustive search) whether there is *some* way to complete the graphs to a consistent set or whether the grammar is not in the class of uniform attribute grammars. We could attempt to use a different heuristic H' to complete the graphs and once again check the resulting protocols for consistency. A better suggestion is to build a multi-protocol evaluator (as described in the next section) instead. Figure 19 gives an AG which 'passes' the pseudo-

protocol creation algorithm test; i.e., all the dependency graphs augmented by the pseudo-protocol graphs are acyclic yet this AG is not in the class of uniform attribute grammars. In particular, the dependencies of p_1 indicate that stmt.a must be evaluated before stmt.c but the dependencies of p_2 indicate that stmt.c must be evaluated before stmt.a. Hence a consistent protocol for stmt cannot be formed even though the dependency graphs augmented by the pseudo-protocol graphs are all acyclic. This attribute grammar would pass the first stage but fail in the second stage no matter which heuristic was used to complete the pseudo-protocol graphs. Any uniform attribute grammar would pass the first stage but would pass or fail the second stage depending upon which heuristic was used to complete the graphs.

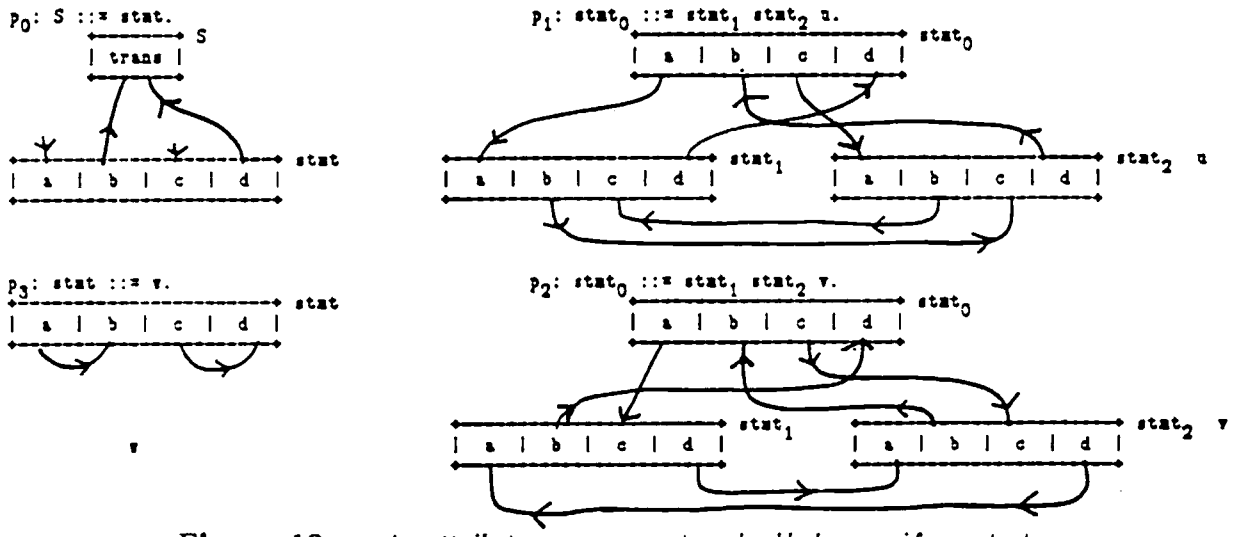


Figure 19: *An attribute grammar not evaluable by a uniform strategy*

Several heuristics for completing the pseudo-protocol graphs into actual protocol graphs have been discussed in the literature [9, 11]. Here we describe a heuristic similar to the one devised by Kastens [21]. It follows a *greedy* strategy in that it completes the pseudo-protocol graphs so as to make as small a protocol as possible[7] and to evaluate attributes as early in the protocol as possible. Given the graph G_X, we define $T_1 = \{i \mid i \in I(X) \mid \nexists$ edge (v,i) in $G_X\}$. T_1 contains those vertices of G_X corresponding to inherited attributes of X which have no entering arcs. Given the sets $T_1,...,T_{j-1}$ we create the set T_j in the following way:

$T_{2j} = \{s \mid s \in S(X) \mid s \notin T_h,\ h < 2j \mid$ if $\exists$ an edge (v,s) in G_X then $v \in T_k,\ k < 2j\}$

$T_{2j+1} = \{i \mid i \in I(X) \mid i \notin T_h,\ h < 2j+1 \mid$ if $\exists$ an edge (v,i) in G_X then $v \in T_k,\ k < 2j+1\}$

T_{2j} are those vertices s of G_X corresponding to synthesized attributes of X such that if $\exists$

[7] By making $p_X = p(1),...,p(n_X)$ as small as possible we mean making the length n_X of the protocol as small as possible. This is advantageous for several reasons. Most importantly, the smaller p_X is, the fewer $VISIT_k$ instructions required to nodes labeled X in the tree.

an edge (v,s) entering s, then v is in an dq"earlier" set T_k. T_{2j+1} has a similar interpetation for the inherited attributes of X. Note that if there is a path from v to w in G_X then $v \in T_h$ and $w \in T_{h+k}$ for some $k > 0$. We create the non-empty sets $T_1,...,T_m$ in this way forming a partition of the attributes of X. The protocol completion strategy calls for introducing an edge (v,w) in G_X between every 2 vertices v and w such that $v \in T_j$ and $w \in T_{j+1}$. This heuristic certainly completes the pseudo-protocol graphs into actual protocol graphs as after adding these arcs, either $I \Rightarrow S$ or $S \Rightarrow I$ for each inherited attribute I and synthesized attribute S of X. Figure 20 gives a pseudo-protocol graph and the protocol graph obtained by using this protocol completion strategy. It is not hard to see that this heuristic will not always result in the creation of a consistent protocol. For example, this heuristic would complete the pseudo-protocol graph G_B of figure 17 into the protocol graph δ''_B corresponding to the protocol π''_B. We already saw that this protocol is not consistent (see figure 18) even though there are ways to complete G_B into a protocol graph corresponding to a consistent protocol for B.

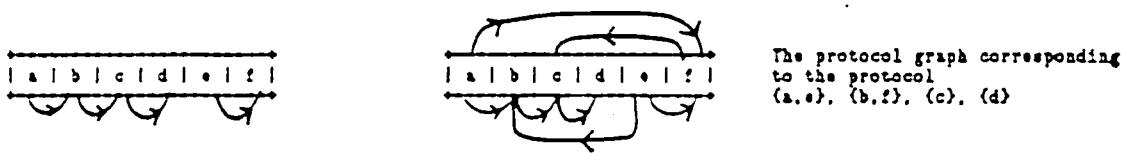


Figure 20: *A protocol graph obtained from a pseudo-protocol graph using the greedy strategy*

Kasten's heuristic [21] is similar to the one given here except that he forms protocols in the reverse direction; he first creates a set T_m of those synthesized attributes s such that $\nexists$ an edge (s,v) in G_X. He then creates the set T_{m-1} of those inherited attributes i such that if $\exists$ the edge (i,v) in G_X then $v \in T_m$. He continues in this fashion to form the sets $T_1,...,T_m$ and then completes the pseudo-protocol graphs based on these sets as we did above. If the resulting set of protocols turns out to be consistent, the attribute grammar is said to be an *ordered attribute grammar*. For the pseudo-protocol graph given in figure 20 Kasten's strategy would compute the protocol {a},{b},{c,e},{d,f}. Contrast this protocol to the one produced by the greedy strategy given above (see figure 20). It is easy to construct grammars for which the greedy strategy would find a consistent set of protocols but Kasten's strategy would not and vice versa. Both heuristics have the desireable property of producing protocols which are as small as possible[8].

It is not hard to see that a pass-oriented evaluator is just a special case of a uniform evaluator. Given a k-pass evaluator for an AG, we can easily construct a consistent set of protocols for the grammar. To each nonterminal X we assign the protocol: {X.i $\in$ $\mathcal{I}$(X) | PN(X.i) = 1}, {X.s $\in$ $\mathcal{S}$(X) | PN(X.s) = 1}, {X.i $\in$ $\mathcal{I}$(X) | PN(X.i) = 2}, {X.s $\in$ $\mathcal{S}$(X) |

[8]This does not mean that the resulting evaluator will be optimal as the optimal evaluator may not be uniform at all. It does mean that the resulting evaluator will be the most efficient *uniform* evaluator which can be constructed for the attribute grammar.

PN(X.s) = 2}...., {X.i ∈ $\mathcal{I}$(X) | PN(X.i) = k}, {X.s ∈ $\mathcal{S}$(X) | PN(X.s) = k}[9]. This set will be consistent or we would not have been able to build the pass-oriented evaluator for the attribute grammar. Not only are uniform evaluators more efficient than pass-oriented evaluators, they can be built for a larger class of attribute grammars. For example, the reader should have no problem in recognizing {π_S = $\emptyset$,{translation}, π_{list} = {inhatt}, {synatt}} as a consistent set of protocols for the attribute grammar of figure 10. Thus although we cannot construct a pass-oriented evaluator for that grammar, we can construct a uniform evaluator for it.

[9]This may not be a protocol in the strict sense of the definition as some of these sets may be empty. This can be corrected by deleting any set other than the first which is empty.

6. Multi-protocol Evaluators

In the construction of a uniform evaluator given in the last section, a set of pseudo-protocol graphs were obtained, one for each nonterminal grammar symbol. Each of these graphs was then used as a *model* from which to build a unique protocol for each nonterminal. The construction of a multi-protocol evaluator proceeds along similar lines. Initially a set of model graphs is formed, one for each nonterminal grammar symbol. However, instead of using these graphs to form a *unique* protocol for each nonterminal, they will be used to form a *set* of protocols for each nonterminal. From this family of protocol sets the procedures of the multi-protocol evaluator will be built in a manner analogous to the construction of the procedures of the uniform evaluator.

The evaluator presented in this section follows, to a large degree, the *sub-protocol evaluator* of Farrow [9]. It has many aspects in common with the *tree-walk evaluator* of Kennedy and Warren [23] and the *direct evaluator* of Nielson [28]. We have chosen Farrow's paradigm as it is a natural extension of the uniform evaluator of the last section.

We begin by showing how to form the model graphs of the multi-protocol evaluator. For each nonterminal X, a graph called IO_X is formed[10]. This graph has only edges from inherited attributes to synthesized attributes. If in *any* semantic tree ∃ a node N labeled X with the synthesized attribute N.s dependent upon the inherited attribute N.i then there will be an edge (i,s) in IO_X. The converse is not true however: there may exist an edge (i,s) in IO_X even though there does not exist a subtree with a node N labeled X and with N.s dependent upon N.i. The *IO_X closure algorithm* which finds the set of IO_X graphs for an attribute grammar is given in Figure 21.

```
For each nonterminal X create the graph IO_X = (V,E) where V = A(X) and E = ∅;
While an edge can be added to some graph IO_X do
        Begin
        Choose a production [p: X_0 ::= X_1 ··· X_np] of the grammar;
        If ∃ a path from X_0.a to X_0.b in Dp[IO_X1,....,IO_Xnp] and (X_0a.X_0.b) is not in IO_X0
        Then add the edge (X_0a.X_0.b) to IO_X0;
        End;
If for any production p of the grammar the graph Dp[IO_X1,....,IO_Xnp] contains a cycle
Then return("grammar is not evaluable by a multi-protocol evaluator")
Else return("the graphs:" {IO_X} "have been computed");
```

Figure 21: *The IO_X closure algorithm*

Note the similarity between the pseudo-protocol creation algorithm of the last section and

[10] IO_X graphs were first introduced by Knuth [24] and subsequentially used by Kennedy and Warren [23], who gave them the name IO_X graphs. IO stands for input-output as the graph shows the dependencies between information *input* into a subtree rooted at a node labeled X and information *output* from the subtree. The information enters via the inherited attributes of the root and leaves via the synthesized attributes of the root.

the IO_X closure algorithm. In fact, for any nonterminal X, the graph IO_X is a subgraph of the pseudo-protocol graph G_X created by the pseudo-protocol creation algorithm. Any edge in IO_X certainly exists in G_X but not every edge in G_X exists in IO_X. The main difference between the algorithms is that the IO_X closure algorithm only adds edges to the graph IO_{X0} and not to graphs IO_{Xi}, $i > 0$, when paths are found in $Dp[IO_{X1},...,IO_{Xnp}]$ while the pseudo-protocol creation algorithm adds edges to any G_{Xi} graph. Note that because the grammar is in Bochmann Normal Form, any path in $Dp[IO_{X1},...,IO_{Xnp}]$ from $X_0.a$ to $X_0.b$ will be from an inherited to a synthesized attribute of X_0.

If for every production p in the attribute grammar the graph $Dp[IO_{X1},...,IO_{Xnp}]$ is acyclic, then the grammar is called *absolutely non-circular* (ANC) [23]. A multi-protocol evaluator can be constructed for *any* ANC attribute grammar. Hence if we intend to build a uniform evaluator but find that in the last step of completing the pseudo-protocol graphs we do not end up with a consistent set of protocols we can still construct a multi-protocol evaluator for the grammar[11].

Given an ANC attribute grammar, using the set of IO_X graphs a *set* of protocols Π_X is constructed for each nonterminal X, except for the start symbol S which will have only one protocol associated with it. This set of protocols will be such that for *each* production $[p: X_0 ::= X_1 \cdots X_{np}]$ of the grammar, for *each* protocol $\pi_{X0} \in \Pi_{X0}$, $\exists$ protocols $\pi_{X1} \in \Pi_{X1},...,\pi_{np} \in \Pi_{Xnp}$ such that $Dp[\delta_{X0},\delta_{X1}, ..., \delta_{Xnp}]$ is acyclic, where δ_{Xi} is the protocol graph corresponding to π_{Xi}. Let $\Pi = \{\Pi_X \mid X$ a nonterminal of the grammar$\}$ be a family of protocol sets obeying the above property. Then Π is said to be a *consistent* family of protocol sets. Just as a uniform evaluator can be built for any consistent set of protocols, a multi-protocol evaluator can be built for any consistent family of protocol sets.

Given a family of consistent protocol sets the multi-protocol evaluator can be built in a fashion similar to the way the uniform evaluator was built. The main difference is that in the uniform evaluator one set of procedures was constructed for a production p based on the unique protocols for $X_0,...,X_{np}$, whereas in the multi-protocol evaluator a set of procedures is created for each protocol of X_0. That is, for each π_{X0} in Π_{X0} a set of procedures is built based on the protocols $\pi_{X1} \in \Pi_{X1},...,\pi_{Xnp} \in \Pi_{Xnp}$ such that $Dp[\delta_{X0},\delta_{X1}, ..., \delta_{Xnp}]$ is acyclic. These procedures will be similar to the procedures of the uniform evaluator with the exception of some additional bookeeping to keep track of which

---

[11]Recall that our algorithm for finding a consistent set of protocols for an attribute grammar consists of two stages: we first build pseudo-protocol graphs for the nonterminals and check whether the augmented dependency graphs are acyclic. If so, we proceed by attempting to complete the pseudo-protocol graphs into a consistent set of actual protocols. If we were successful in stage 1, i.e., all the dependency graphs augmented by the pseudo-protocol graphs were acyclic, then we can always build a multi-protocol evaluator as these graphs being acyclic implies that the attribute grammar is ANC. But even if we failed at stage 1, i.e., some $Dp[G_{X0},G_{X1},...,G_{Xnp}]$ graph is cyclic, the grammar may still be ANC and we could create a multi-protocol evaluator for it. This is because a cycle in $Dp[G_{X0},G_{X1},...,G_{Xnp}]$ does not necessarily mean that $Dp[IO_{X1},...,IO_{Xnp}]$ will contain a cycle.

protocol is being used for each nonterminal. So, for example, a CASEVISIT$_k$ statement now takes on the following form:

```
Case LP production of X_k = p_1: VISIT_k and call proc_p1(i)[π];
                            p_2: VISIT_k and call proc_p2(i)[π];
                                 .
                                 .
                            p_m: VISIT_k and call proc_pm(i)[π];
```

Here $proc_{p(i)}[\pi]$ is a procedure to visit production p for the i^{th} time, using π as a protocol for X_k. π is not a parameter to the procedure but part of the procedure name. Unlike the uniform evaluator where there was only one procedure $proc_{p(i)}$ corresponding to the i^{th} visit to an instance of p in a semantic tree, in the multi-protocol evaluator there may be h procedures $proc_{p(i)}[\pi_1]$,..., $proc_{p(i)}[\pi_h]$ each corresponding to the i^{th} visit to an instance of p. Which of the procedures is used depends upon which of the h possible protocols is being used for the node labeled X_k. It does not require any extra overhead during run-time, however, to determine which one to use; that is determined by the calling procedure. This can best be demonstrated by an example. Figure 22 gives the sets of protocols for the nonterminals of the attribute grammar presented in figure 19. Although this attribute grammar is not a uniform attribute grammar it is a multi-protocol attribute grammar. Figure 23 presents the procedures of the multi-protocol evaluator built for it. In addition to the procedures $proc_S$ and $proc_{p0}$ there exists 2 sets of procedures for each of the productions p_1, p_2 and p_3. For example, since p_1 has the symbol stmt as its LHS, we need one set of procedures for p_1 using π_{stmt} as the protocol for $stmt_0$ and one set using π'_{stmt} as the protocol for $stmt_0$. The set of procedures using π_{stmt} as the protocol for $stmt_0$, namely $proc_{p1(1)}[\pi_{stmt}]$ and $proc_{p1(2)}[\pi_{stmt}]$, uses π_{stmt} as the protocol for $stmt_1$ and π'_{stmt} as the protocol for $stmt_2$ since these protocols cause the augmented dependency graph to be acyclic; should π_{stmt} be used as protocols for both $stmt_1$ and $stmt_2$ the augmented dependency graph would contain a cycle. Similarly the set of procedures for p_1 using π'_{stmt} as the protocol for $stmt_0$, namely $proc_{p1(1)}[\pi'_{stmt}]$ and $proc_{p1(2)}[\pi'_{stmt}]$, uses π'_{stmt} as the protocol for $stmt_1$ and π_{stmt} as the protocol for $stmt_2$ since these protocols cause the augmented dependency graph to be acyclic; should π'_{stmt} be used as protocols for both $stmt_1$ and $stmt_2$ the augmented dependency graph would contain a cycle. Due to space considerations, figure 23 does not give the procedures for production p_2.

Π_S = { π_S = ∅, {trans} }

Π_{stmt} = { π_{stmt} = {a},{b},{c},{d}, π'_{stmt} = {c},{d},{a},{b} }

Figure 22: *Sets of protocols for the symbols of the attribute grammar of figure 19*

At this point it may be beneficial for the reader to trace the execution of the evaluator on a typical semantic tree for this attribute grammar. Figure 24 gives such a tree. In this figure we have labeled each nonterminal node by the protocol that would be (automatically) used by the evaluator to evaluate the attributes of that node. Evaluation of the tree would begin by calling $proc_S[\pi_S]$. This procedure would in turn call $proc_{p0}$. This first evaluates N2.a, then visits N2's LP production invoking the procedure $proc_{p1(1)}[\pi_{stmt}]$ (indicating that

```
proc_S[π_S]          /* The start procedure */
    begin
    Case LP production of S = p_0: VISIT_stmt and call proc_p0;
    end;

proc_p0              /* p_0: S ::= stmt.  */
    begin
    EVAL_stmt.a
    Case LP production of stmt = p_1: VISIT_stmt and call proc_p1(1)[π_stmt];
                                p_2: VISIT_stmt and call proc_p2(1)[π_stmt];
                                p_3: VISIT_stmt and call proc_p3(1)[π_stmt];
    EVAL_stmt.c
    Case LP production of stmt = p_1: VISIT_stmt and call proc_p1(2)[π_stmt];
                                p_2: VISIT_stmt and call proc_p2(2)[π_stmt];
                                p_3: VISIT_stmt and call proc_p3(2)[π_stmt];
    EVAL_S.trans;
    end;

proc_p1(1)[π_stmt] = proc_p1(2)[π'_stmt]      /* p_1: stmt_0 ::= stmt_1 stmt_2 a.  */
    begin
    EVAL_stmt_1.a;
    Case LP production of stmt_1 = p_1: VISIT_stmt1 and call proc_p1(1)[π_stmt];
                                p_2: VISIT_stmt1 and call proc_p2(1)[π_stmt];
                                p_3: VISIT_stmt1 and call proc_p3(1)[π_stmt];
    EVAL_stmt_2.c;
    Case LP production of stmt_2 = p_1: VISIT_stmt2 and call proc_p1(1)[π'_stmt];
                                p_2: VISIT_stmt2 and call proc_p2(1)[π'_stmt];
                                p_3: VISIT_stmt2 and call proc_p3(1)[π'_stmt];
    EVAL_stmt_0.b
    end;

proc_p1(2)[π_stmt] = proc_p1(1)[π'_stmt]      /* p_1: stmt_0 ::= stmt_1 stmt_2 a.  */
    begin
    EVAL_stmt_2.a;
    Case LP production of stmt_2 = p_1: VISIT_stmt2 and call proc_p1(2)[π'_stmt];
                                p_2: VISIT_stmt2 and call proc_p2(2)[π'_stmt];
                                p_3: VISIT_stmt2 and call proc_p3(2)[π'_stmt];
    EVAL_stmt_1.c;
    Case LP production of stmt_1 = p_1: VISIT_stmt1 and call proc_p1(2)[π_stmt];
                                p_2: VISIT_stmt1 and call proc_p2(2)[π_stmt];
                                p_3: VISIT_stmt1 and call proc_p3(2)[π_stmt];
    EVAL_stmt_0.d
    end;

proc_p3(1)[π_stmt] = proc_p3(2)[π'_stmt]      proc_p3(2)[π_stmt] = proc_p3(1)[π'_stmt]   /* stmt ::= x.  */
    begin                                         begin
    EVAL_stmt.b                                   EVAL_stmt.d
    end;                                          end;
```

Figure 23: *The multi-protocol evaluator constructed for the grammar of figure 19*

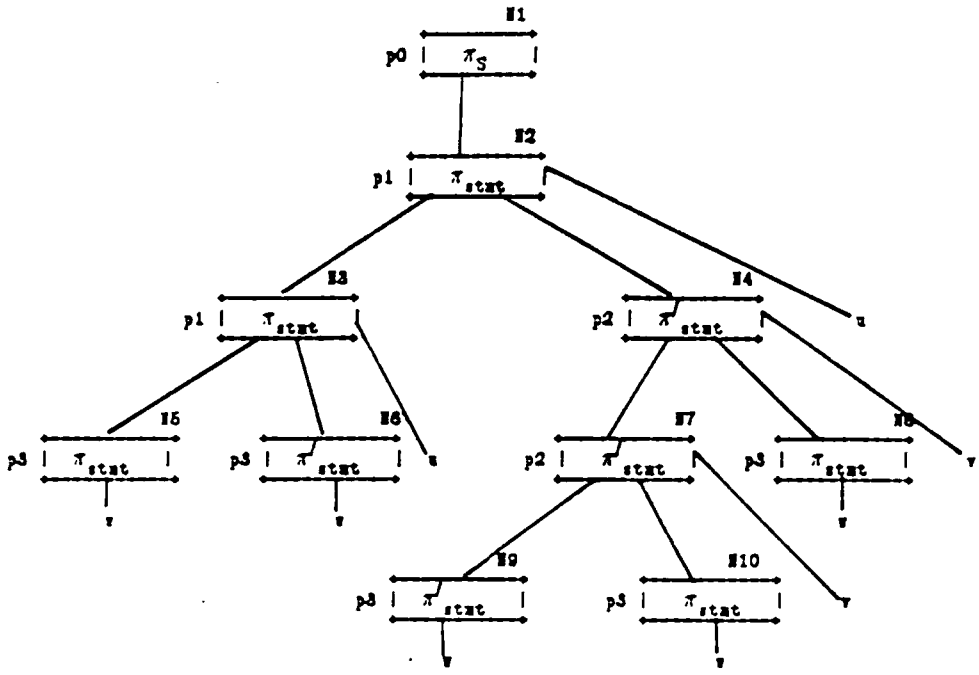


Figure 24: *A semantic tree for the attribute grammar of figure 19*

N2's LP production is p_1, that N2 is associated with the protocol π_{stmt} and that N2 is being visited for the first time). Upon returning from this visit N2.c is evaluated; since N2's protocol is π_{stmt} we know that N2.b was evaluated during the visit. Next N2 is visited for the second time invoking the procedure $proc_{p1(2)}[\pi_{stmt}]$. Upon returning from this visit we know that N2.d has been evaluated and that N1.trans can be evaluated. We see that although the evaluator never explicitly gave N2 the protocol π_{stmt} this assignment is implicit in the procedure $proc_{p0}$. If the reader continues the simulation of the evaluator as it visits the other nodes in the tree, he will see that multi-protocol evaluator always implicitly assigns to a node a protocol by choosing the appropriate procedure calls.

Note that several different procedures (procedures for the same production but using different protocols for the nonterminals) can have the same code. $proc_{p1(2)}[\pi_{stmt}]$ and $proc_{p1(1)}[\pi'_{stmt}]$, for example, have the same one instruction: $EVAL_{stmt.b}$. The procedure $proc_{p1(2)}[\pi_{stmt}]$ corresponds to the second visit to a production-instance of p_1 where the LHS node stmt has the protocol π_{stmt}. The procedure $proc_{p1(1)}[\pi'_{stmt}]$ corresponds to the first visit to a production-instance of p_1 where the LHS node stmt has the protocol π'_{stmt}. Nonetheless this procedure needs to be written only once; during compiler construction time the compiler generator needs to notice that two or more procedures are the same and to eliminate duplicates. This can significantly reduce the amount of code needed for the evaluator[12].

[12]This idea of eliminating duplicate procedures which have the same code is due to Farrow [9]. We actually have generalized his technique somewhat. We allow more procedures to be eliminated, but for this to be done we need a more complicated technique- such as actually checking the procedures for equivalent code.

How many procedures are required by the multi-protocol evaluator? The uniform evaluator required construction of n_{X0} procedures for each production [p: $X_0 ::= X_1 \cdots X_{np}$], where n_{X0} is the length of the protocol for X_0. A multi-protocol evaluator requires the construction of $n^1_{X0} + n^2_{X0} + \ldots + n^h_{X0}$ procedures for each production p, where $\Pi_{X0} = \{\pi^1_{X0},\ldots,\pi^h_{X0}\}$ are the set of protocols for X_0 and n^i_{X0} is the length of π^i_{X0}. Since the number of protocols per nonterminal can be exponential in the number of attributes of the symbol [9], this can result in an extremely large evaluator. Fortunately this is only in the worst case. If, for instance, a multi-protocol evaluator for the attribute grammar can be built by assigning a *singleton* set of protocols to every nonterminal except one, to whom a set containing 2 protocols must be assigned, then the resulting multi-protocol evaluator constructed will be only slightly larger than the uniform evaluator would have been. Furthermore, as illustrated in the last example, many of the procedures constructed have the same code and can be shared. For these reasons it seems probable that the size of a multi-protocol evaluator constructed for an attribute grammar will be on the same order as a uniform evaluator would have been (had we been able to construct one for the attribute grammar).

We have seen how to construct a multi-protocol evaluator given a consistent family of protocol sets. But how do we initially find such a set? In figure 25 we give the *protocol closure algorithm* which, given an ANC attribute grammar, finds a set of protocols for each nonterminal so that the multi-protocol evaluator can be built. Here we shall not concentrate on finding a *small* set of protocols for each nonterminal, although this is obviously desireable in order to build as small an evaluator as possible. The interested reader should consult [9] which presents heuristics for this purpose. The protocol closure algorithm starts with a unique protocol for the start symbol S. As S has no inherited attributes, this protocol is simply $\pi_S = \emptyset,\{\text{synthesized attributes of S}\}$. It then generates protocol sets for all of the nonterminals of the attribute grammar. The protocol closure algorithm will terminate as there are only a finite number of possible protocols for a given nonterminal. It will always be able to find a set of protocols such that $Dp[\delta_{X0},\delta_{X1}, \ldots, \delta_{Xnp}]$ is acyclic because the grammar is ANC. (See [28, 29]).

The protocol closure algorithm uses the function COMPUTE_PROTOCOLS given in figure 26. This function can be viewed as a heuristic for completing protocols (see end of section 5). Given the dependency graph Dp, the IO_X graphs for $X_1,\ldots,X_{np}$, and a protocol for X_0, it completes the model IO_X graphs to form protocols for $X_1,\ldots,X_{np}$. It does so by determining which attributes will be evaluated and which children visited on each visit to the production, based on the protocol for X_0. From the resultant protocols $\pi_1,\ldots,\pi_{np}$ together with π_0, a set of procedures for the multi-protocol evaluator can be constructed; i.e., the graph $Dp[\delta_{X0},\delta_{X1}, \ldots, \delta_{Xnp}]$ is acyclic. The function READY_TO_EVALUATE_INH(W,G,j) used by the function COMPUTE_PROTOCOLS takes a set of already evaluated attributes W, an augmented dependency graph G, and a subscript j. It returns the set $\{att \in I(X_j) - W \mid \text{for each arc } (w, att) \text{ in } G, w \in W\}$; i.e., those inherited attributes of X_j which can be evaluated immediately as all of the attributes it depends upon are in the set W of already evaluated attributes. The function

```
Let π_S = ∅.(synthesized attributes of S);
For each nonterminal X ≠ S let Π_X = ∅ endFor;
Repeat until no more protocols need to be added
        For each production [p: X_0 ::= X_1 ··· X_np]
                For each protocol π_X0 ∈ Π_X0
                        π_1,....,π_np ::= COMPUTE_PROTOCOLS(Dp,π_X0,IO_X1,....,IO_Xnp);
                                                /* Dp[δ_X0,δ_X1, .... δ_Xnp] is acyclic */
                        For i ∈ [1..np]
                                if π_i ∉ Π_Xi then add π_i to Π_Xi;
                        endFor;
                endFor;
        endFor;
endRepeat;
```

Figure 25: *The protocol closure algorithm*

READY_TO_EVALUATE_SYN(W,G,j) similarly returns those synthesized attributes of X_j which can be evaluated immediately. The function, HAS_NON_EMPTY_YIELD(W,G) returns a *set* of right-part context-free symbols $\{X_k \mid k > 0 \mid \exists$ synthesized attribute $X_k.att \in S(X_k) - W \mid$ for each arc $(w,X_k.att)$ in G, either $w \in W$ or $w \in$ READY_TO_EVALUATE_INH(W,Dp[δ_{X0},IO_{X1},...,IO_{Xnp}])}; i.e., those right-part symbols which would *yield* newly evaluated synthesized attributes by evaluating any of their inherited attributes ready to be evaluated and then visiting their subtrees. So if $X_k \in$ HAS_NON_EMPTY_YIELD(W,G) then we know that after evaluating the inherited attributes of X_k which are ready to be evaluated immediately, we can visit X_k and during that visit evaluate some synthesized attribute of X_k which has not yet been evaluated. The protocol closure algorithm and the COMPUTE_PROTOCOLS function are based on a similar construction in [28, 29]. They also have features in common with the *Kennedy-Warren Planning Algorithm* [23].

Notice the nondeterminism in the function COMPUTE_PROTOCOLS. On each step through the WHILE loop the set HAS_NON_EMPTY_YIELD(W,Dp[δ_{X0},IO_{X1},...,IO_{Xnp}]) is computed. This is a set of RHS nodes X_i, $1 \leq i \leq np$, such that if X_k is in this set then X_k can be visited allowing at least one synthesized attribute to be evaluated (we say that the visit *yields* an attribute). The function arbitrarily chooses *one* of these nodes to visit and then recomputes the set. This continues until no more possible visits can be made; i.e., until there does not exist a node which will yield an attribute by visiting it. Depending upon how this nondeterminism is implemented different protocols may result for the children $X_1,...,X_{np}$. It is desireable to restrict this nondetermism so that the protocols produced are as small as possible. In the appendix it is shown, however, that producing the set of protocols for $X_1,...,X_{np}$ (given the protocol π_{X0} and the graphs $IO_{X1},...,IO_{Xnp}$) such that their total length is minimal is an NP-complete problem.

```
func COMPUTE_PROTOCOLS(Dp: a dependency graph, π_X0: a protocol for X_0, IO_X1,....,IO_Xnp: IO_X graphs
                                                                    for RHS nonterminals);
/* Forms protocols for the RHS nonterminals of p by completing the IO_X graphs based on the protocol π_X0 */
Begin
Let δ_X0 be the protocol graph for π_X0 = π(1)...π(2a_X0);
W ::= ∅;
For j ::= 1 to a_p do π_j ::= empty sequence; endFor;
For i ::= 1 to a_X0 do
        W ::= W ∪ π_X0(2i-1);
        While HAS_NON_EMPTY_YIELD(W,Dp[δ_X0,IO_X1,....,IO_Xnp]) ≠ ∅ do
                Choose some X_j ∈ HAS_NON_EMPTY_YIELD(W,Dp[δ_X0,IO_X1,....,IO_Xnp]);
                I_j ::= READY_TO_EVALUATE_INH(W,Dp[δ_X0,IO_X1,....,IO_Xnp],j);
                π_j ::= π_j.I_j;          /* Concantenate I_j to the end of π_j */
                W ::= W ∪ I_j;
                S_j ::= READY_TO_EVALUATE_SYN(W,Dp[δ_X0,IO_X1,....,IO_Xnp]);
                π_j ::= π_j.S_j;          /* Concantenate S_j to the end of π_j */
                W ::= W ∪ S_j;
                endWhile;
        W ::= W ∪ π_X0(2i);
        endFor;
Return(π_1,....,π_ap);
End;
```

Figure 26: *The function COMPUTE_PROTOCOLS*

Although the multi-protocol evaluator outlined in this section is based on the ideas in [9], it is very similar to the *tree-walk evaluator* of Kennedy and Warren [23] and the *direct evaluator* of Nielson [28]. A comparison of these evaluators is presented in [9]. A different interesting approach, based on an algebraic formulation of attribute grammars [2, 4], can be found in [19, 22]. These evaluators are not really tree-walk evaluators; their underlying scheme is to build one recursive function per synthesized attribute. In this way they avoid construction of protocols and the need for flags (as found in the Kennedy-Warren evaluator).

Although multi-protocol evaluators can be built for a large class of attribute grammars, if an evaluator for a attribute grammar which is not ANC is required then we must go beyond static evaluators and build a *dynamic* evaluator. We need to use more run-time information than the static evaluator allows. Assume, for example, that we have a attribute grammar which is not ANC. Then there is some production p such that $Dp[IO_{X1},...,IO_{Xnp}]$ is cyclic. Assuming that the grammar is actually well defined, this circularity must be due to a *spurious* edge. That is, there are 2 edges $(X_k.a,X_k.b)$ and $(X_k.c,X_k.d)$ along the cycle in $Dp[IO_{X1},...,IO_{Xnp}]$ such that one will appear in some subtree rooted at a node labeled X_k and the other will appear in a *different* subtree rooted at a node labeled X_k but they will never appear in the same subtree. In order to know in which order to visit the

children one must know which edge is spurious. A static evaluator cannot deal with this situation: the only information it can use when at an instance of a production p is which productions apply at its children. This information is not sufficient to detect which edge is spurious and hence cannot determine in what order to visit the children nodes. A dynamic evaluator, however, can use an arbitrary amount of knowlege concerning the semantic tree. It could, for example, make a prepass over the tree passing information to each node on the nature of its subtree. This sort of information is sufficient to evaluate any semantic tree for any well-formed AG. Many dynamic evaluators have been formulated [3, 23, 28].

7. Conclusion

In this paper we have developed the notion of *static tree-walk evaluators*. These evaluators represent an important subclass of all tree-walk evaluator strategies. We presented three types of static evaluators: pass-oriented, uniform and multi-protocol evaluators. Pass-oriented evaluators evaluate semantic trees by traversing each tree a fixed number of times in right-to-left or left-to-right depth-first passes. They are easy to construct and reasonably small. They have two significant weaknesses however. First of all, a pass-oriented strategy can be very inefficient in its evaluation of semantic trees. Secondly, many attribute grammars cannot be evaluated by such a strategy. Uniform evaluators do significantly better in both of these areas: as they only visit a node in order to evaluate an attribute they are much more efficient in their evaluation of semantic trees. Also, they can be constructed for a larger subset of well-defined attribute grammars. But uniform evaluators have a weak point as well: in general it is too hard (Np-complete) to construct such an evaluator for every attribute grammar which could have one constructed for it. Most compiler generators based on a uniform evaluation strategy will therefore only construct evaluators for a subset of all uniform attribute grammars. Multi-protocol evaluators are not as restricted as uniform ones: for any absolutely non-circular AG a multi-protocol evaluator can be built. Moreover, this evaluator evaluates semantic trees as efficiently as the uniform evaluator, only visiting a node of a tree if doing so would result in the evaluation of an attribute of the node. Unfortunately, in the worst case the resultant evaluator can be exponentially large.

ACKNOWLEDGEMENT

I would like to thank Rodney Farrow for serving as my advisor and introducing me to most of the material presented in this survey. Our conversations always led to new and exciting ideas. He also read several drafts of this paper and offered many useful suggestions. I would also like to thank Rich Korf and Kathy Mckeown for their helpful recomendations.

Appendix

In this appendix we show that two related problems are NP-Complete. The first concerns finding minimal length protocols, the second concerns optimal time evaluation of semantic trees.

Given a production [p: X_0 ::= X_1 ··· X_{np}] and an augmented dependency graph $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ the algorithm of figure 26 computes valid protocols for the symbols of $X_1,...,X_{np}$. By valid protocols we mean protocols whose graphs $\delta_{X1},...,\delta_{Xnp}$ are such that the graph $Dp[\delta_{X0},\delta_{X1}, ..., \delta_{Xnp}]$ is acyclic and such that every edge in $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ is in $Dp[\delta_{X0},\delta_{X1}, ..., \delta_{Xnp}]$. It would be desireable to create these protocols such that their *total length* length (i.e., $n_{X1} + n_{X2} + ... + n_{Xnp}$) is as small as possible; a smaller total length means that less visits to children need to made for any such production-instance in a semantic tree and this would lead to more efficient evaluators.

> Theorem 2: Given an augmented dependency graph $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ and an integer k finding whether there exists valid protocols $\pi_{X1},...,\pi_{Xnp}$ for the grammar symbols $X_1,...,X_{np}$ having lengths $n_{X1},...,n_{Xnp}$ such that $n_{X1} + n_{X2} + ... + n_{Xnp} \leq k$ is NP-complete.

Proof: Certainly the problem is in NP. Guess valid protocols $\pi_{X1},...,\pi_{Xnp}$ for the grammar symbols. Verification can be done in p-time. To show that it is NP-complete we will show that if this problem could be solved in p-time, the *shortest common supersequence problem*, described below, could also be solved in p-time.

Given a string S over an alphabet Σ, a *supersequence* S' of S is any string $S' = w_0 s_1 w_1 s_2 w_2 ... s_m w_m$ over Σ such that $S = s_1 s_2 ... s_m$ and each w_i belongs to Σ^*; we also say that S is a subsequence of S'. A common supersequence of a set of strings $\varsigma = \{S^1,...,S^r\}$ is a string S over Σ such that S is a supersequence of each S^j. We shall refer to the k^{th} symbol of S^j as S^j_k. The shortest common supersequence problem is defined as follows: given an alphabet Σ, a finite set ς of strings from Σ^*, and a positive integer k, is there a common supersequence of ς of length $\leq$ k? This problem was shown to be NP-complete by Maier (3), provided that the size of the alphabet Σ is $\geq$ 5. The result was sharpened to include any alphabet with at least 2 elements in (4).

The reduction will be as follows: given an instance of the common supersequence problem (a set of strings $\varsigma = \{S^1,...,S^r\}$ over $\Sigma = \{a_1,...,a_n\}$) we will create a graph $Dp[\delta_{X0},IO_{X1},...,IO_{Xn}]$ such that there will exist a common supersequence for ς of length $\leq$ k iff there exists valid protocols for $X_1,...,X_n$ of total length $\leq$ k. The graph $Dp[\delta_{X0},IO_{X1},...,IO_{Xn}]$ can be specified by giving the attributes of each symbol X_i, $0 \leq i \leq n$, the edges in δ_{X0}, the edges in IO_{Xi}, $1 \leq i \leq n$, and the edges in Dp.

Let $\varsigma = \{S^1,...,S^r\}$ be given. Create $Dp[\delta_{X0},IO_{X1},...,IO_{Xn}]$ such that X_0 has 2 attributes X_0.inh and X_0.syn. The protocol graph δ_{X0} will contain the single edge (inh,syn). Every other symbol X_i, $1 \leq i \leq n$, will have an inherited and a synthesized attribute (called an *attribute pair*) for *each* occurence of a_i in any string S^j. Hence if a_i appears 5 times in

strings of ς then the context-free symbol X_i will have 5 inherited and 5 synthesized attributes. If a_i appears as the k^{th} symbol of the string S^j then the corresponding inherited and synthesized attributes added to X_i will be $X_i.S^j{}_k$inh and $X_i.S^j{}_k$syn. For example, if $S^j = a_5\ a_2\ a_3\ a_5$ then S^j would contribute an attribute pair to each of the context-free symbols X_2 and X_3 and two attribute pairs to X_5 (namely, $X_5.S^j{}_1$inh, $X_5.S^j{}_1$syn, $X_5.S^j{}_4$inh and $X_5.S^j{}_4$syn). For any attribute pair ($X_i.S^j{}_k$inh , $X_i.S^j{}_k$syn) the graph IO_{X_i} will contain the edge ($X_i.S^j{}_k$inh , $X_i.S^j{}_k$syn). These are the only edges in the IO_{X_i} graphs. The graph Dp will have one edge for each two consecutive symbols $S^j{}_k\ S^j{}_{k+1}$ in any string S^j: Say that $S^j{}_k = a_e$ and $S^j{}_{k+1} = a_f$. Then Dp will have an edge from $X_e.S^j{}_k$syn to $X_f.S^j{}_{k+1}$inh. The only other edges in the Dp graph are edges from X_0.inh to the inherited attributes corresponding to the first character of S^j ($1 \leq j \leq r$), and from the synthesized attributes corresponding to the last character of S^j ($1 \leq j \leq r$), to X_0.syn. So, for example, if $S^j = S^j{}_1,\ldots,S^j{}_{final}$ with $S^j{}_1 = a_e$ and $S^j{}_{final} = a_f$ then Dp will have the edges (X_0.inh , $X_e.S^j{}_1$) and ($X_f.S^j{}_{final}$syn , X_0.syn).

The above description shows how to form, in p-time, the graph $Dp[\delta_{X0},IO_{X1},\ldots,IO_{Xn}]$ from an instance of the common supersequence problem ς. Figure 27 gives the augmented dependency graph $Dp[\delta_{X0},IO_{X1},IO_{X2},IO_{X3}]$ for the grammar formed from strings $\varsigma = \{S^1, S^2\}$ over the alphabet $\Sigma = \{a_1, a_2, a_3\}$, where $S^1 = a_1\ a_2\ a_3\ a_1$ and $S^2 = a_2\ a_3\ a_2$.

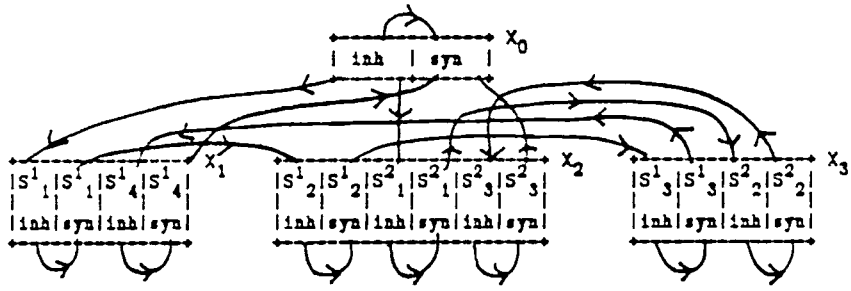


Figure 27: *An augmented dependency graph formed from a set of strings*

We now must show that there exists valid protocols for $X_1,\ldots,X_n$ (protocols whose graphs $\delta_{X1},\ldots,\delta_{Xn}$ are such that the graph $Dp[\delta_{X0},\delta_{X1},\ldots,\delta_{Xn}]$ is acyclic and such that every edge in $Dp[\delta_{X0},IO_{X1},\ldots,IO_{Xn}]$ is in $Dp[\delta_{X0},\delta_{X1},\ldots,\delta_{Xn}]$) and of total length $\leq$ k iff there exists a common supersequence for ς of length $\leq$ k. Assume that we had valid protocols for $X_1,\ldots,X_n$ of total length k. Then we could write a procedure to evaluate the attributes of p based on these protocols (this would be done as given in sections 5 and 6). This procedure would mix evaluation of the attributes of p with visits to the children nodes (these visits would *yield* synthesized attributes of the node) and would obey the dependency relations given in $Dp[\delta_{X0},IO_{X1},\ldots,IO_{Xn}]$[13] . There would be EXACTLY k visit instructions in this

[13] A procedure obeys the dependency relations of an augmented dependency graph if before an attribute $X_i.i$ is evaluated all of its dependencies have already been evaluated. If it is dependent upon a synthesized attribute $X_k.s$ ($k > 0$) then X_k must be visited before evaluating $X_i.i$ and at the time of visiting X_k all of $X_k.s$'s dependencies must have already been evaluated. This last condition means that if the attribute is dependent upon a synthesized attribute of X_k then that synthesized attribute has been yielded on a previous visit to X_k.

procedure. This is because a protocol for X of length m requires m visits to X in order to evaluate all of its attributes. Similarly if we are given a procedure containing k VISIT instructions which evaluates all the attributes of p and obeys the dependencies of $Dp[\delta_{X0},IO_{X1},...,IO_{Xn}]$ then we can find the valid protocols for $X_1,...,X_n$ of total length = k corresponding to this procedure. This fact is fairly intuitive: from valid protocols we can construct procedures which do not violate any dependencies and from procedures which obey dependencies we can extract valid protocols. This fact allows us to express our proof a little differently: we will show that there exists a procedure to evaluate all the attributes of p (assuming only that X_0.inh has already been evaluated upon entry into the procedure) using k visit instructions and obeying all the dependencies of $Dp[\delta_{X0},IO_{X1},...,IO_{Xn}]$ *iff* there exists a common supersequence of ς of length k. In particular, if the procedure contains the sequence of instructions $VISIT_j$ {EVAL instructions} $VISIT_k$ {EVAL instructions} ... $VISIT_m$ then $a_j\ a_k\ ...\ a_m$ will be a common supersequence of ς. As there exists a 1-1 correspondance between the visit sequence of a procedure and strings in Σ^* we shall refer to the visit sequence simply by the string it corresponds to; e.g., if a procedure contains a sequence of visit instructions $VISIT_j,VISIT_k,VISIT_m$ we shall refer to it as the string $a_j a_k a_m$.

i) Say a procedure obeying the dependencies of the augmented dependency graph $Dp[\delta_{X0},IO_{X1},...,IO_{Xn}]$ has a visit sequence $V = a_{i1}a_{i2}...a_{ih}$. Let $S^j = S^j_1\ S^j_2...S^j_m \in \varsigma$. We must show that $S^j_1\ S^j_2...S^j_m = a_{u1}a_{u2}...a_{um}$ is a subsequence of V. Let $S^j_k = a_e$ and $S^j_{k+1} = a_f$. From the way that the Dp and IO_{Xi} graphs were constructed, *any* procedure evaluating the attributes of p according to the dependencies of the augmented dependency graph will evaluate $X_f.S^j_{k+1}$inh only after visiting X_e yielding $X_e.S^j_k$syn and cannot visit X_e yielding that synthesized attribute until after evaluating $X_e.S^j_{k-1}$inh. Hence the procedure must evaluate the inherited attributes $X_{u1}.S^j_1$inh, $X_{u2}.S^j_2$inh,.... $X_{um}.S^j_m$inh sequentially, visiting $X_{u1},X_{u2},...,X_{um}$ sequentially. Hence S^j is a subsequence of V.

ii) Let $V = a_{i1}a_{i2}...a_{ih}$ be a common supersequence of ς. We must show that there exists a procedure evaluating all the attributes of p (assuming X_0.inh has already been evaluated), obeying the dependencies of the augmented dependency graph $Dp[\delta_{X0},IO_{X1},...,IO_{Xn}]$ and having a visit sequence of V. Create a procedure whose visit sequence is V and evaluates attributes in between visits according to the following method: Looking at each subsequence $S^j = a_{u1}a_{u2}...a_{um}$ $(1 \le j \le r)$ of V, have the procedure evaluate $X_{uk+1}.S^j_{k+1}$inh before visiting X_{uk+1} $(= S^j_{k+1})$ but after visiting X_{uk} $(= S^j_k)$. Finally evaluate X_0.syn. This procedure will obey the dependencies of the augmented dependency graph and will evaluate all of the attributes of p.

End of proof

We now turn to the question of optimal tree-walk evaluators and show that it is an NP-complete problem to construct optimal evaluators. Recall from section 2 that a visit sequence for a tree T is the sequence of $VISIT_k$ instructions used to evaluate T.

Corollary 3: Given a tree T, it is an NP-complete problem to determine

whether or not there exists a visit sequence for T of length $\leq$ k.

proof: Certainly the problem is in NP. Guess a sequence of $VISIT_k$ instructions. Verification can be done in p-time. To show that it is NP-complete we will show that if this problem could be solved in p-time, then the problem of finding minimal length protocols could also be solved in p-time.

Our reduction will be as follows: given an augmented dependency graph $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ where δ_{X0} has length n_{X0} we will construct a semantic tree T and show that there exists a visit sequence for T of length $\leq (2^*n_{X0} + 2^*k)$ iff there exists valid protocols for $X_1,...,X_{np}$ of total length $\leq$ k. As pointed out in the proof of the last theorem, we can state this a little differently: given $Dp[\delta_{X0},IO_{X1},...IO_{Xnp}]$ we will construct a semantic tree T and show that there exists a visit sequence for T of length $\leq (2^*n_{X0} + 2^*k)$ iff there exists procedures to evaluate all the attributes of p obeying the dependencies of $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ and using $\leq$ k visit instructions.

Given the augmented dependency graph $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ construct a semantic tree as follows. Let $<p_0$: S ::= $X_0>$ be the production applying at the root. Let $<$p: X_0 ::= $X_1...X_{np}>$ apply at X_0 and let a production $<p_i$: X_i ::= terminal$_i>$ apply at X_i ($1 \leq i \leq np$). The semantic functions of p have dependencies as given in Dp. the semantic functions of p_0 have dependencies as extracted from δ_{X0} and the semantic functions of p_i have dependencies as given in IO_{Xi}. Figure 28 gives the form of a semantic tree T constructed by this method.

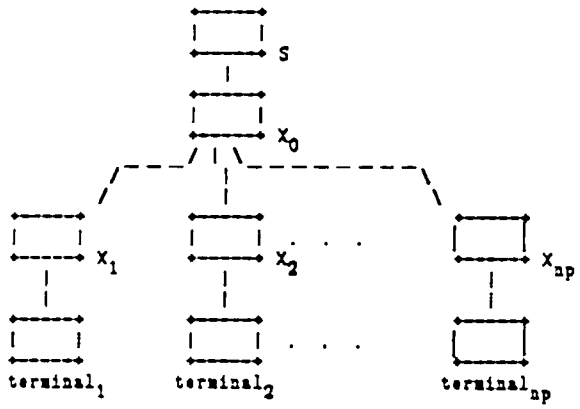


Figure 28: *The semantic tree T constructed from the augmented dependency graph* $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$

Say that there exists a visit sequence for T of length $2^*n_{X0} + 2^*k$. We will show that there exists procedures to evaluate all the attributes of p and obeying the dependencies of $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ of length k. Any visit sequence to evaluate T must have n_{X0} visit instructions executed at the root. For each of these there is one $VISIT_0$ instruction executed at p to return control to the root node. Say that in addition to these visit

instructions only H are needed to evaluate the rest of the tree. For each $VISIT_j$ instruction executed at the production p to visit the node X_j one $VISIT_0$ instruction is needed at p_j to return back to p. Hence $H/2 = k$ visit instructions are executed at p. Since T has all the dependencies of $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ this means that procedures obeying the dependencies of $Dp[\delta_{X0},IO_{X1},...,IO_{Xnp}]$ and using k visit instructions can be constructed to evaluate the attributes of p. The only if part of the proof is left for the reader.

End of proof

References

[1] G.V. Bochmann.
Semantic evaluation from left to right.
Communications of the ACM 19, 1976.
pp. 55-62.

[2] L. M. Chirica and D. F. Martin.
An algebraic formulation of Knuthian semantics.
In *Proceedings of the 17th IEEE Symposium on the Foundations of Computer Science.* IEEE, 1976.

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

[4] B. Courcelle and Franchi-Zannettacci.
Attribute Grammars and Recursive Program Schemes.
Theoretical Computer Science 17:163-191 and 235-257, 1982.

[5] Alan Demers, Thomas Reps and Tim Tietelbaum.
Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors.
In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages.* ACM, January, 1981.

[6] Joost Engelfriet and Gilberto Filè
Simple Multi-Visit Attribute Grammars.
Technical Report, Department of Applied Mathematics, Twente University of Technology, August, 1980.

[7] Joost Engelfriet and Gilberto Filè
Passes,Sweeps and Visits in Attribute Grammars.
Technical Report, Department of Applied Mathematics, Twente University of Technology, August, 1982.

[8] I. Fang.
FOLDS, a declarative formal language definition system.
Technical Report STAN-CS-72-329, Stanford University, 1972.

[9] Rodney Farrow.
Covers of Attribute Grammars and Sub-Protocol Attribute Evaluators.
Technical Report, Department of Computer Science, Columbia University, New York, New York 10027, September, 1983.

[10] Rodney Farrow.
Experience with a Production Compiler Automatically Generated from an Attribute Grammar.
Technical Report, Department of Computer Science, Columbia University, New York, New York 10027, March, 1984.

[11] Rodney Farrow.
Sub-Protocol Evaluators for Attribute Grammars.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.* ACM-SIGPLAN, June, 1984.
Published as Volume 19, Number 6, of *SIGPLAN Notices.*

[12] Rodney Farrow.
Experience with an attribute grammar based compiler.
In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages.* ACM. January, 1982.

[13] Rodney Farrow and Daniel Yellin.
A Comparison of Storage Optimizations in Automatically-Generated Attribute Evaluators.
Technical Report, Department of Computer Science, Columbia University, New York, New York 10027, January, 1985.

[14] G. Filè
Interpetation and reduction of attribute grammars.
Acta Informatica 19, 1980.

[15] H. Ganzinger, R. Giegerich, U. Moncke and R. Wilhelm.
A Truly Generative Semantics-Directed Compiler Generator.
In *Proceedings of the SIGPLAN Symposium on compiler construction.* ACM, June, 1982.

[16] 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, 1975.

[17] M. Jazayeri and K.G. Walter.
Alternating semantic evaluator.
In *Proceedings of ACM 1975 Annual Conference.* ACM, 1975.

[18] Neil D. Jones and C. Michael Madsen.
Attribute-Influenced LR Parsing.
In *Lecture Notes in Computer Science 94,* . Springer-Verlag, 1980.

[19] Martin Jourdan.
Strongly Non-Circular Attribute Grammars and their Recursive Evaluation.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.* ACM-SIGPLAN, June, 1984.
Published as Volume 19, Number 6, of *SIGPLAN Notices.*

[20] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann.
GAG: A Practical Compiler Generator.
Spring-Verlag, Berlin-Heidelberg-New York, 1982.

[21] U. Kastens.
Ordered attribute grammars.
Acta Informatica 13, 1980.

[22] T. Katayama.
Translation of Attribute Grammars into Procedures.
ACM TOPLAS 6(3), July, 1984.

[23] K.. Kennedy and S. K. Warren.
Automatic generation of efficient evaluators for attribute grammars.
In *Conference Record of the Third ACM symposium on Principles of Programming Languages.* ACM. 1976.

[24] D. E. Knuth.
Semantics of context-free languages.
Mathematical Systems Theory 2, 1968.
correction in volume 5, number 1.

[25] K. Koskimies, K-J. Raiha, and M. Sarjakoski.
Compiler Construction Using Attribute Grammars.
In *Proceedings of the SIGPLAN Symposium on compiler construction.* ACM, June, 1982.

[26] B. Lorho.
Semantic attribute processing in the system DELTA.
In A. Ershov and C.H.A. Koster (editor), *Methods of Algorithmic Language Implementation.* Springer-Verlag, Berlin-Heidelberg-New York, 1977.

[27] Eva-Maria M. Mueckstein.
Q-TRANS: Query Translation Into English.
In *Proceedings of the Eight International Joint Conference on Artificial Intelligence,* pages 660-662. IJCAI-83, August, 1983.

[28] H.R.Nielson.
Computation sequences: A way to characterize subclasses of attribute grammars.
Technical Report, Aarhus University, Denmark, 1981.

[29] H.R.Nielson.
Using computation sequences to define attribute evaluators.
Technical Report, Aarhus University, Denmark, 1981.

[30] George K. Papakonstantinou.
An Interpeter of Attribute Grammars ans its Application to Waveform Analysis.
IEEE Transactions On Software Engineering Se-7(3), May, 1981.

[31] S. R. Petrick.
Semantic Interpetation in the Request System.
Technical Report RC4457, IBM, JULY, 1973.

[32] Diane Pozefsky and M. Jazayeri.
A Family of Pass-Oriented Attribute Grammar Evaluators.
In *Proceedings of ACM 1978 Annual Conference.* ACM, 1978.

[33] K-J. Raiha and E. Ukkonen.
Minimizing the number of evaluation passes for attribute grammars.
SIAM Journal of Computing 10(4), NOVEMBER, 1981.

[34] Thomas W. Reps.
Generating Language-Based Environments.
PhD thesis, Cornell University, Ithaca, New York, December, 1983.

[35] Thomas Reps and Bowen Alpern.
Interactive Proof Checking.
In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages.* ACM, January, 1984.

[36] M. Saarinen.
On constructing efficient evaluators for attribute grammars.
In C. Ausiello and C. Bohm (editor), *Automata, Languages, and Programming: 5th Colloquium.* Springer-Verlag, Springer-Verlag, New York, 1978.

[37] W.A. Schulz.
Semantic analysis and target language synthesis in a translator.
PhD thesis, University of Colorado, Boulder, Colorado, July, 1976.

[38] S. K. Warren.
The coroutine model of attribute grammar evaluation.
PhD thesis, Rice University, May, 1976.

[39] David A. Watt.
Rule splitting and attribute-directed parsing.
In *Lecture Notes in Computer Science 94,* . Springer-Verlag, 1980.