Proc. Natl. Sci. Counc. ROC(A)
Vol. 21, No. 6, 1997. pp. 601-610

Conditional Evaluation in Simple Multi-Visit Attribute-Grammar Evaluators

WUU YANG

Department of Computer and Information Science
National Chiao-Tung University
Hsin-Chu, Taiwan, R.O.C.

(Received January 30, 1997; Accepted May 5, 1997)

ABSTRACT

Attribute grammars are a formalism for specifying computations on context-free languages. Due to the nonstrictness of the *if* constructs in attribution equations, it is possible to avoid evaluating certain attribute instances in a syntax tree. A dynamic evaluator can easily avoid such useless computations with a demand-driven approach. However, dynamic evaluators are not efficient because they need to keep the attribute dependence graph during evaluation, and they need to decide an evaluation order for each syntax tree. In contrast, a visit-oriented (static) evaluator can carefully re-arrange the evaluation order and still avoid unnecessary computations. We propose such a technique in this paper.

Key Words: attribute grammars, conditional evaluation, ordered attribute grammars, simple multi-visit attribute grammars

I. Introduction

Since their introduction in 1968 (Knuth, 1968), attribute grammars have attracted much research interest. Attribute grammars are a very convenient and powerful framework for specifying computations on context-free languages (Deransart *et al.*, 1988). Attribute evaluation has been extensively studied since then. In this paper, we propose a new conditional evaluation technique for static evaluators that may potentially save evaluation time and space.

Consider the attribution equation *A.a := if g (B.b) then C.c else D.d* in an attribute grammar. If the predicate *g (B.b)* evaluates to true, it may be possible to avoid evaluating the attribute *D.d* (and other attributes on which *D.d* depends directly or indirectly). Similarly, if *g (B.b)* evaluates to false, it may be possible to avoid evaluating *C.c* (and other attributes on which *C.c* depends directly or indirectly).

A demand-driven dynamic evaluator can easily avoid such useless computations. However, a dynamic evaluator needs to maintain the attribute dependence graph of a syntax tree and to decide an evaluation order for every syntax tree. This incurs significant overhead both in time and in storage space. By contrast, a static evaluator may decide an evaluation scheme from an attribute grammar that is applicable to all syntax trees. During evaluation, the static evaluator simply follows the pre-determined scheme. Thus, static evaluators are usually more efficient than dynamic ones. Although static evaluators exist only for certain subclasses of attribute grammars, such as ordered attribute grammars (Kastens, 1980; Engelfriet and Filè, 1982), it is commonly agreed that these subclasses of grammars are sufficiently large for practical use.

We propose in this paper a conditional evaluation scheme for static evaluators that may avoid useless computations. Avoiding useless computations not only speeds up the evaluator, but makes possible savings in storage space. This is because much storage is allocated dynamically during evaluation of attribute instances. If an attribute instance is not evaluated at all, then there is no need to allocate storage for it. On the other hand, the storage overhead in our conditional evaluator is a one-bit guard for each attribute instance. (This space overhead may be further reduced. See Section III.3.) An attribute instance is evaluated only if its guard is true. Furthermore, whenever a downward visit to a child is about to be performed, the evaluator checks whether the guards of the attribute instances that are computed during that downward visit are true. If none is true, the evaluator may simply skip that visit. The time overhead in the worst case is linear in the number of attribute instances in the syntax tree.

For a static evaluator to evaluate attribute instances in a syntax tree, it is usually necessary to perform the evaluation in an order that is consistent with the dependencies among the attribute instances.

For the above example *A.a := if g (B.b) then C.c else D.d*, the three attributes *B.b*, *C.c*, and *D.d* are evaluated before the *if* expression in a traditional static evaluator because *A.a* depends on the other three attributes. In a traditional static evaluator, the relative evaluation order of *B.b*, *C.c*, and *D.d* is left unspecified. Observe that it is possible to evaluate the predicate *g (B.b) before* the attributes *C.c* and *D.d* . Furthermore, it is also possible to selectively evaluate only one of *C.c* and *D.d* based on the outcome of evaluating *g (B.b)*. Note that when g *(B.b)* evaluates to true (or false), *D.d* (or *C.c*, respectively) may still need to be evaluated if it is used in computation of other attributes. Our technique of conditional evaluation establishes a sufficient condition for avoiding attribute evaluation.

Our conditional evaluation scheme can be applied to handle exceptions. Though attribute evaluation usually terminates on serious errors, for minor errors or exceptions, such as undeclared variables or mismatched data types, evaluation should skip unnecessary or unevaluable attributes rather than terminates. These exceptions can be specified in terms of the nonstrict conditional equations and be handled by our conditional evaluation scheme. Such an example appears in Kastens's paper (Kastens, 1980). Boyland and Graham (1994) also pointed out that nonstrict conditional equations are useful for subexpression ordering (a code generation task in a compiler) and in computing "offsets" in a compiler for a series of declarations (Boyland, 1996).

Our technique of conditional evaluation is based on the visit-oriented evaluation scheme (Engelfriet and Filé, 1982). There is one evaluation plan for each production rule in the grammar. A plan consists of instructions of the following four kinds: an *if-then-else* instruction, evaluation of an attribute instance, a visit to a child node, and a return to the parent node. Our technique generates plans for the production rules.

The conditional evaluation technique not only speeds up the evaluation process, but also saves dynamically allocated storage. Herndon (1986) proposed a method for generating dynamic, lazy evaluators for functional attribute grammars. Naini (1988) proposed a demand-driven approach, which does not incorporate conditional evaluation into the computation of visit sequences. Both of these methods have the same aim as does this paper: to avoid unnecessary attribute evaluation. However, their methods are for dynamic evaluation. In contrast, the method proposed in this paper is a static one. It works by carefully analyzing attribute dependencies during the generation of evaluation sequences. Though there have been many other research papers on attribute evaluation methods (Alblas, 1991; Jourdan, 1991) and on storage optimization (Akker and Sluiman, 1991; Engelfriet and Jong, 1990), these works did not address the issue of conditional evaluation.

Our conditional evaluation scheme is somewhat similar to Boyland's conditional attribute grammars (Boyland, 1996). Due to the undecidability of the predicate equivalence problem, Boyland proposed a new syntax with which a user can group together identical predicates manually. In addition to this minor syntactic difference, Boyland's technique splits a production with k conditional attribution equations into 2^k new rules (assuming that all these k equations are not nested within one another and, hence, are independent). Each of the 2^k new rules corresponds to a combination of the values (true or false) of the k predicates in the equations. Boyland proceeded to analyze the node-split grammar far various properties, such as circularity, complexity classes, evaluation methods etc. In contrast, the unique feature of our conditional evaluation scheme lies in delaying the evaluation of the attribute occurrences used in the two branches of a conditional equation until after the evaluation of the attribute occurrences used in the predicate as well as the evaluation of the predicate itself.[1] Boyland's technique did not attempt to achieve this delayed evaluation. In this respect, our scheme is closer to the (optimal) demand-driven dynamic evaluators. On the other hand, Boyland's node-splitting technique focuses on detecting the false circularity in an attribute grammar that is caused by distinct, mutually exclusive evaluation orders of attributes in the two branches of conditional equations.

The rest of this paper is organized as follows. The notations are introduced in Section II. The technique for generating conditional evaluation plans is presented in Section III. In Section IV, we describe the conditional evaluator. In the last section, we discuss the efficiency of the conditional evaluation scheme and summarize our work.

II. Notations

In this section, we will define the notations used

[1] Note that the expressions in the two branches of a conditional equation are, of course, evaluated after the predicate in all evaluators. However, the attribute occurrences used in the two expressions might be evaluated *before* the attribute occurrences used in the predicate in some evaluators. Our conditional evaluation scheme attempts to delay the evaluation of the attribute occurrences used in the two expressions.

in this paper. Basically, we adopt Kastens's notations (Kastens, 1980). An attribute grammar is built from a context-free grammar (N, T, P, S), where N is a finite set of nonterminals, T is a finite set of terminals, S is a distinguished nonterminal, called the *start symbol*, and P is a set of production rules of the form $X \rightarrow \alpha$, where X is a nonterminal, and α is a string of terminals and nonterminals. For each nonterminal X, there is at least one production rule whose left-hand-side symbol is X. As usual, we require that the sets of terminals and nonterminals be disjoint. In this paper, a *symbol* refers to a terminal or a nonterminal. A symbol may appear more than once in a production rule. We give each *occurrence* of a symbol in a production rule a unique index in order to differentiate occurrences. The symbol occurrences will be denoted as $X^{(1)}$, $X^{(2)}$ etc. The indexes may be omitted if no confusion arises. Furthermore, a production rule may be applied more than once in a syntax tree. In this case, we say that there are many *instances* of a symbol occurrence in the syntax tree.

Attached to each symbol X of the context-free grammar is a set of *attributes*. Intuitively, instances of attributes describe the properties of specific instances of symbols in a syntax tree. In order to simplify our presentation, we assume that attributes of different symbols have different names. The attributes of a symbol X are partitioned into two disjoint subsets, called the *inherited* attributes and the *synthesized* attributes. There are attribution equations defining these attributes. In a production rule, there are an attribution equation defining each synthesized attribute occurrence of the left-hand-side symbol and an attribution equation defining each inherited attribute occurrence of the right-hand-side symbol. We assume that the start symbol has no inherited attributes.

An attribute a of a symbol X is denoted by $X.a$. Since there may be many occurrences of a symbol, there are many *occurrences* of an attribute in a production rule. These attribute occurrences will be denoted as $X^{(1)}.a$, $X^{(2)}.a$ etc., where 1, 2 etc. are the unique indexes of the different occurrences of the symbol X. Similarly, since a production rule may be applied more than once in a syntax tree, there may be many *instances* of an attribute occurrence in a syntax tree.

There is still some freedom in specifying the attribution equations. Therefore, we require that, for each production rule, the attribution equations be defined in terms of the inherited attribute occurrences of the left-hand-side symbol and the synthesized attribute occurrences of the right-hand-side symbols of the production rule. This is called the *normal form* in the literature (Bochmann, 1976; Paakki, 1995). The advantage of the normal form is that it specifies the smallest number of dependencies among attributes. Since dependencies among attributes enforce an evaluation order, an attribute grammar in the normal form allows the most freedom in the evaluator.

III. Conditional Evaluation Plans

Since a conditional evaluator may avoid evaluating certain attribute instances in a syntax tree, it is important to designate the attribute instances that must be evaluated. For simplicity of presentation, we assume that all and only the attribute instances of the start symbol in a syntax tree must be evaluated. Other attribute instances may be discarded if they are not needed. We further assume that the start symbol appears exactly once in a syntax tree, and that there is a path from every attribute instance to an attribute instance of the start symbol in the attribute dependence graph corresponding to the syntax tree (for otherwise, that attribute instance would not be used, directly or indirectly, in the computation of the attribute instances of the start symbol).

1. Adding Augmented Dependence Edges

We assume that the attribution equations are of two forms: $A^{(i)}.a := f\ (B^{(j)}.b,\ C^{(k)}.c,\ ...)$ or $A^{(i)}.a :=$ *if* $g\ (B^{(j)}.b,\ ...)$ *then* $h\ (C^{(k)}.c,\ ...)$ *else* $f\ (D^{(l)}.d,\ ...)$.[2] We also assume that the functions f, g, and h etc. are strict functions. However, the *if* construct is not strict: If the predicate $g\ (B^{(j)}.b,\ ...)$ evaluates to true, it may be possible to avoid evaluating $f\ (D^{(l)}.d,\ ...)$ (and other attributes on which $D^{(l)}.d$ depends directly or indirectly). Similarly, if $g\ (B^{(j)}.b,\ ...)$ evaluates to false, it may be possible to avoid evaluating $h\ (C^{(k)}.c,\ ...)$ (and other attributes on which $C^{(k)}.c$ depends directly or indirectly). We show that, by carefully arranging the evaluation order and by making use of a guard in each attribute instance in the syntax tree, a static evaluator can indeed avoid certain useless computations.

First consider an example. Assume that there is a production rule $A \rightarrow B\ C\ D$ and an attribution equation $A.a :=$ *if* $B.b=0$ *then* $C.c+1$ *else* $D.d+2$. A traditional visit sequence for this equation might be (1) a downward visit to B, evaluating the attribute occurrence $B.b$, (2) a downward visit to C, evaluating $C.c$, (3) a downward visit to D, evaluating $D.d$, and (4) evaluation of the *if* expression, assigning the result to the attribute

[2] Nested *if* expressions can be handled by a straightforward extension of the technique presented in this paper. For the sake of simplicity, we will exclude nested *if* expressions in our discussion.

occurrence $A.a$. As we pointed out before, either step (2) or step (3) might be redundant. The visit sequence produced by the conditional evaluation technique (approximately) consists of the following steps: (1) visit symbol B, evaluating $B.b$, (2) evaluate the predicate $B.b$=0, (3) if the result in step (2) is true, reset the guard of the attribute instance $D.d$ (that is, its value becomes false); otherwise reset the guard of $C.c$, (4) if the guard of $C.c$ is true, visit C, evaluating $C.c$, (5) if the guard of $D.d$ is true, visit D, evaluating $D.d$, and (6) evaluate either $A.a$=$C.c$+1 or $A.a$=$D.d$+2, based on the outcome of step (2). In essence, evaluation of the *if* expression is distributed into three instructions: steps 2 and 6 and an instruction that resets the guards of other attribute occurrences.

We adopt an evaluator generator similar to the one discussed in Kastens (1980) to generate a correct evaluation plan. For an *if* expression in an attribution equation in a production rule P, such as $A^{(i)}.a$ = *if* g $(B^{(j)}.b, \ldots)$ *then* h $(C^{(k)}.c, \ldots)$ *else* f $(D^{(l)}.d, \ldots)$, it is important to delay the evaluation of $C^{(k)}.c$ and $D^{(l)}.d$ until after the evaluation of g $(B^{(j)}.b, \ldots)$ (hence, after $B^{(j)}.b$). Thus, we add augmented dependence edges $B^{(j)}.b \rightarrow C^{(k)}.c$ and $B^{(j)}.b \rightarrow D^{(l)}.d$ (and possibly a few other augmented dependence edges), when appropriate, to the dependence graph of production P. (See below for the definition of dependence graphs.)

We need to investigate a sufficient condition under which the augmented dependence edge $B^{(j)}.b \rightarrow C^{(k)}.c$ may be added to the dependence graph of production P. It is obvious that the evaluation of the attribute occurrence $C^{(k)}.c$ can be avoided if (1) $C^{(k)}.c$ is *used only for* the computation of $A^{(i)}.a$ and (2) $C^{(k)}.c$ is used only in one branch of the *if* expression. Furthermore, the evaluation of any attribute occurrence $X^{(m)}.x$ in production P can also be avoided if $X^{(m)}.x$ is used only for the computation of $C^{(k)}.c$ in production P.

In terms of attribute dependencies in a syntax tree, an attribute occurrence $X^{(m)}.x$ is *used only for* the computation of another attribute occurrence $Y^{(n)}.y$ in production P if and only if $Y^{(n)}.y$ occurs in every path from $X^{(m)}.x$ to an attribute instance of the start symbol in the dependence graph corresponding to a syntax tree. We will define the notion of "*used only for*" in what follows.

Definition III.1. Given a production P of an attribute grammar, the *dependence graph of* P, denoted by DP_p, is defined as follows:

DP_p= {$X^{(i)}.a \rightarrow Y^{(j)}.b$ | $X^{(i)}.a$ is used in the attribution equation defining $Y^{(j)}.b$ in production P}.

In DP_p, nodes denote attribute occurrences in production P and edges dependencies between attribute occurrences. The dependencies between attribute occurrence are derived from the attribution equations in production P. An edge $X^{(i)}.a \rightarrow Y^{(j)}.b$ means that the attribute occurrence $X^{(i)}.a$ is a parameter of the function defining the attribute occurrence $Y^{(j)}.b$ in production P.

Definition III.2. Given a production P, the *downward closed dependence graph* of production P, denoted by $DCDP_p$, is defined as follows:

$DCDP_p$= $DP_p \cup$ {$X^{(i)}.a \rightarrow X^{(i)}.b$ | $X^{(i)}$ appears on the right-hand side of production P, and there is a production Q of which the symbol X appears on the left-hand side and $X.a \rightarrow X.b$ is an edge in the transitive closure of the graph $DCDP_Q$}.

$DCDP_P$ is called IDP_P-*ANCAG* in Kastens (1980). The second part of the definition of $DCDP_P$ implies that $X.a$ may be used, directly or indirectly, in computing $X.b$ in a syntax tree rooted at symbol X. Intuitively, $DCDP_P$ includes the transitive dependencies among attribute occurrences of symbols on the right-hand side of production P.

Let S denote a set of attributes of a symbol X. $X^{(k)}.S$ is a shorthand notation for the set of attribute occurrences {$X^{(k)}.a$| $a \in S$}.

Definition III.3. Given a production P and a set α of attribute occurrences in P, $UOF(P, \alpha)$ is a set of attribute occurrences in production P that is defined as follows: Let $Y^{(m)}$ be a symbol occurrence in production P and a be an attribute of Y. $Y^{(m)}.a \in UOF(P, \alpha)$ if (1) $Y^{(m)}.a \in \alpha$; or (2) for every attribute occurrence $Z^{(n)}.b$ in production P such that there is an edge $Y^{(m)}.a \rightarrow Z^{(n)}.b$ in $DCDP_P$, we have $Z^{(n)}.b \in UOF(P, \alpha)$. If the symbol occurrence $Y^{(m)}$ is on the left-hand side of production P, we further require that, for every production Q of which the symbol Y appears on the right-hand side (let the occurrence of Y in Q be denoted by $Y^{(j)}$), either $Y^{(j)}.a$ has no outgoing edges in DP_Q or there is a set of attributes B of symbol Y such that $a \notin B$, $Y^{(j)}.a \in UOF(Q, Y^{(j)}.B)$, and $Y^{(m)}.B \subseteq UOF(P, \alpha)$.

Intuitively, $Y^{(m)}.a \in UOF(P, \alpha)$ means that, in any syntax tree T that contains an instance of production P, every path from the attribute instance $Y^{(m)}.a$ to an attribute instance of the root of T in the dependence graph corresponding to T must pass through an attribute instance of α in the same instance of production P. In the above definition, the first condition means that $\alpha \subseteq UOF(P, \alpha)$. The second condition may be explained for two cases: First consider the case where $Y^{(m)}$ appears

on the right-hand side of production P. Since $DCDP_P$ includes all downward transitive dependencies of $Y^{(m)}$, $Y^{(m)}.a \in UOF(P, \alpha)$ if all successors of $Y^{(m)}.a$ in $DCDP_P$ are members of $UOF(P, \alpha)$ (because every path from $Y^{(m)}.a$ to an attribute instance of the root in every syntax tree must pass through one of the successors). On the other hand, when $Y^{(m)}$ appears on the left-hand side of production P, we need to consider the upward transitive dependencies of $Y^{(m)}.a$, that is, all possible dependencies via the ancestor productions of P in all syntax trees. Note that this piece of information is not included in $DCDP_P$. However, it suffices to prevent $Y^{(m)}.a$ to reach an attribute instance of the root symbol without passing through an element of $UOF(P, \alpha)$ in all syntax trees. There are two ways to satisfy this requirement: in any production Q that includes the symbol Y on the right-hand side (let $Y^{(j)}$ denote an occurrence of Y on the right-hand side of Q), either $Y^{(j)}.a$ has no outgoing edges in DP_Q or there is a set of attributes B of symbol Y such that, in production Q, $Y^{(j)}.a \in UOF(Q, Y^{(j)}.B)$ and, in production P, $Y^{(m)}.B \subseteq UOF(P, \alpha)$. Because $\alpha \subseteq UOF(P, \alpha)$, we further require that $a \notin B$ in order to prevent a loophole.

We can easily verify the following four properties of the *UOF* sets: (1) $UOF(P, \phi)=\phi$; (2) $\alpha \subseteq UOF(P, \alpha)$; (3) $UOF(P, \alpha) \cup UOF(P, \beta) \subseteq UOF(P, \alpha \cup \beta)$; and (4) if $\alpha \subseteq UOF(P, \beta)$ and $\beta \subseteq UOF(P, \gamma)$, then $\alpha \subseteq UOF(P, \gamma)$. The *ComputeUOF* algorithm in Fig. 1 computes the *UOF* for each production P and each set α of attribute occurrences. The *ComputeUOF* algorithm is a straightforward translation of the definition, leaving many possibilities for improvement. In particular, *ComputeUOF* may make use of the four properties. For the sake of simplicity, we present the most straightforward algorithm in Fig. 1.

Having defined the *UOF* sets, we will describe exactly the augmented dependence edges that are added to DP_P. Consider an *if* expression in an attribution equation in a production P, say $A^{(i)}.a$ = *if* g ($B^{(j)}.b$, ...) *then* h ($C^{(k)}.c$, ...) *else* f ($D^{(l)}.d$, ...). Let α be a subset of attribute occurrences that appear in the *then* branch of the *if* expression but not in the predicate and the *else* branch satisfying $\alpha \subseteq UOF(P, \{A^{(i)}.a\})$. We add an augmented dependence edge $X^{(m)}.x \rightarrow Y^{(n)}.y$ to DP_P, for each attribute occurrence $X^{(m)}.x$ that appears in the predicate and for each attribute occurrence $Y^{(n)}.y \in UOF(P, \alpha)$. Similarly, let β be a subset of attribute occurrences that appear in the *else* branch of the *if* expression but not in the predicate and the *then* branch satisfying $\beta \subseteq UOF(P, \{A^{(i)}.a\})$. We add an augmented dependence edge $X^{(m)}.x \rightarrow Y^{(n)}.y$ to DP_P, for each attribute occurrence $X^{(m)}.x$ that appears in the predicate and for each attribute occurrence $Y^{(n)}.y \in UOF(P, \beta)$. After adding these augmented dependence edges, we may apply Kastens's algorithm (Kastens, 1980) or other similar algorithms (Yang and Cheng, 1996) to compute an evaluation order for each production rule.

2. Generating Evaluation Plans

After a feasible evaluation order is determined, we need to generate an evaluation plan for each production rule. A plan consists of a sequence of instructions. There are four kinds of instructions: an *if-then-else* instruction, (guarded or unguarded) evaluation of an attribute occurrence, an n^{th} visit to a child node, and a visit to the parent node. We modify Kastens's method (Kastens, 1980) for generating evaluation plans as follows: The instruction for evaluating an *if* predicate is inserted immediately after those for evaluating all the attribute occurrences used in the predicate. The evaluation of an attribute occurrence is also controlled by the guard of the attribute occurrence. Finally, the instruction to evaluate one of the two branches of an *if* expression is selected according to the outcome of evaluating the predicate and is placed where the attribute occurrence on the left-hand side of the attribution equation is in the evaluation order.

Example III.4. An example attribute grammar is shown in Fig. 2 (a). There are six production rules and 12 attributes. Figure 2 (b) is DP_{P2}, which will be used in our discussion. Note that there is an *if* expression defining the attribute occurrence $Y.e$ in production $P3$. We will examine the production $P3$, the downward closed dependence graph of which is shown in Fig. 2 (c). The bold edge $k \rightarrow j$ represents a downward transitive dependence. Note that the upward transitive dependence, represented by the edge $g \rightarrow f$, is not included in $DCDP_{P3}$. Consider the *else* branch of the *if* expression. $Z.j$ is the only attribute occurrence used in that branch. $Z.j \in UOF(P3, \{Y.e\})$ because the only outgoing edge from $Z.j$ in $DCDP_{P3}$ is directed toward $Y.e$ and $Y.e \in UOF(P3, \{Y.e\})$. Since $Z.j \in UOF(P3, \{Y.e\})$, we will add an augmented edge from $Z.h$ to each attribute occurrence in $UOF(P3, \{Z.j\})$. We will show how to compute $UOF(P3, \{Z.j\})$. (Computation for the *then* branch is similar.)

According to the definition of *UOF* sets, $Z.j \in UOF(P3, \{Z.j\})$. Furthermore, $Z.k \in UOF(P3, \{Z.j\})$ because $Z.k$ has only one outgoing edge in $DCDP_{P3}$, and that edge is directed toward an attribute occurrence (namely, $Z.j$) already in $UOF(P3, \{Z.j\})$, and because the symbol occurrence Z appears on the right-hand side of $P3$. Next consider whether $Y.f \in UOF(P3, \{Z.j\})$. Similarly, $Y.f$ has only one outgoing edge and that edge

is directed toward an attribute occurrence (namely, $Z.k$) already in $UOF(P3, \{Z.j\})$. Note that Y occurs on the left-hand side of production $P3$. Thus, we need to examine all productions of which Y appears on the right-hand side, that is, we need to examine production $P2$. Because $Y.f$ has no outgoing edges in DP_{P2}, $Y.f \in UOF(P3, \{Z.j\})$. Finally, we consider whether $Y.g \in UOF(P3, \{Z.j\})$. Note that $Y.g$ has no outgoing edges in $DCDP_{P3}$, but that $Y.g$ has an outgoing edge in DP_{P2}. In this case, we need to find a set B of Y's attributes such that $g \notin B$, $Y.g \in UOF(P2, Y.B)$, and $Y.B \subseteq UOF(P3, \{Z.j\})$. We may let $B=\{f\}$ and verify that $Y.g \in UOF(P2, \{Y.f\})$. Thus, we conclude that $UOF(P3, \{Z.j\})=\{Z.j, Z.k, Y.f, Y.g\}$. (Similarly, we may compute $UOF(P3, \{Y.e\})$, which is $\{Z.h, Z.i, Z.j, Z.k, Y.f, Y.g\}$.) An augmented edge from $Z.h$ to each

```
Algorithm: ComputeUOF
/* UOF is a 2-dimensional global array, indexed by P and α, */
/* where P is a production and α is a set of attribute occurrences of P. */
UOF[P, α] := α for all P and all sets α of attribute occurrences of P initially.
/* Note that if α is not a set of attribute occurrences of P, UOF[P, α] := ∅ */
repeat
   changed := false
   for each production P do
      for each set α of attribute occurrences of P do
         new := ComputeTentativeUOF(P, α)
         if new ≠ UOF[P, α] then
            changed := true
            UOF[P, α] := new
         end if
      end for
   end for
until changed = false

function ComputeTentativeUOF(P, α) return a set of attribute occurrences of production P
/* P is a production, and α is a set of attribute occurrences of P. */
new := UQF[P, α]
repeat
   result := new
   for each attribute occurrence Y(m).a in P do
      if for all attribute occurrences Z(n).b in P such that there is an edge
        Y(m).a → Z(n).b in DCDP_P, we have Z(n).b ∈ result then
         if Y(m) is on the right-hand side of P then
            new := new ∪ { Y(m).a }
         else /* Y(m) is on the left-hand side of P */
            addflag := true
            for each production Q of which the symbol Y occurs on the right-hand side do
               for each occurrence Y(j) of the symbol Y on the right-hand side of Q do
                  if not (Y(j).a has no outgoing edge in DP_Q or there is a set B of
                        attributes of symbol Y such that a ∉ B, Y(j).a ∈ UOF[Q, Y(j).B]
                        and Y(m).B ⊆ UOF[P, α])
                     then addflag := false end if
               end for
            end for
            if addflag = true then new := new ∪ { Y(m).a }
         end if
      end if
   end for
until result = new
return result
```

Fig. 1. The *ComputeUOF* algorithm.

(a) An attribute grammar

P1:	$S \rightarrow X$	$S.a := X.b$
		$X.c := X.d$
P2:	$X \rightarrow Y$	$X.b := Y.e$
		$X.d := Y.g$
		$Y.f := X.c$
P3:	$Y \rightarrow Z$	$Y.g := 3$
		$Y.e :=$ *if* $Z.h = 0$ *then* $Z.i$ *else* $Z.j$
		$Z.k := Y.f$
P4:	$Z \rightarrow W$	$Z.j := Z.k$
		$Z.i := 2$
		$Z.h := W.m$
P5:	$W \rightarrow 0$	$W.m := 0$
P6:	$W \rightarrow 1$	$W.m := 1$

(b) DP_{P2}

(c) $DCDP_{P3}$

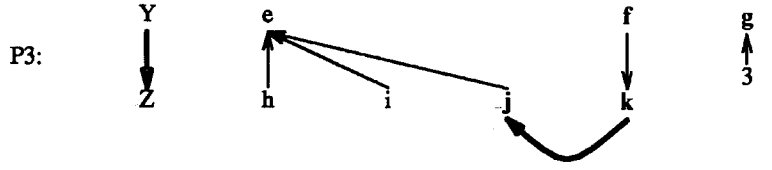

Fig. 2. An attribute grammar, DP_{P2}, and $DCDP_{P3}$.

attribute occurrence in UOF ($P3$, $\{Z.j\}$) is added to DP_{P3}.

A feasible evaluation order for production $P3$ is $Z.h$, $Z.i$, $Y.g$, $Y.f$, $Z.k$, $Z.j$, and $Y.e$. Based on this evaluation order, there are two visits to Y in production $P3$ from the parent production instance in a syntax tree:

(1) The first visit consists of the following seven steps:
  (i) Reset the guards of all attribute occurrences in $UOF(P3, F)$, where F is the set of attribute occurrences of Y whose guards are false.
  (ii) If the guard of $Z.h$ is true, visit the child Z for the first time, evaluating $Z.h$. Upon returning from the visit, reset the guards of all attribute occurrences in $UOF(P3, F)$, where F is the set of attribute occurrences of Y whose guards are false.
  (iii) If the guard of $Y.e$ is true, evaluate the predicate $Z.h$=0.
  (iv) If the guard of $Y.e$ is true and the result in step (1.iii) is true, reset the guards of $Z.j$, $Z.k$, $Y.f$, and $Y.g$; otherwise, if the guard of $Y.e$ is true (and the result in step (1.iii) is false), reset the guard of $Z.i$.
  (v) If the guard of $Z.i$ is true, visit the child Z for the second time, evaluating $Z.i$. Upon returning from the visit, reset the guards of all attribute occurrences in $UOF(P3, F)$, where F is the set of attribute occurrences of Y whose guards are false.
  (vi) If the guard of $Y.g$ is true, evaluate $Y.g$.
  (vii) Return to the parent production.
(2) The second visit consists of the following five steps:
  (i) Reset the guards of all attribute occurrences in $UOF(P3, F)$, where F is the set of attribute occurrences of Y whose guards are false.

(ii) If the guard of $Z.k$ is true, evaluate $Z.k$.
(iii) If the guard of $Z.j$ is true, visit the child Z for the third time, evaluating $Z.j$. Upon returning from the visit, reset the guards of all attribute occurrences in $UOF(P3, F)$, where F is the set of attribute occurrences of Y whose guards are false.
(iv) If the guard of $Y.e$ is true and the result in step (1.iii) is true, evaluate the *then* branch of the *if* expression and assign the result to $Y.e$; otherwise, if the guard of $Y.e$ is true (and the result in step (1.iii) is false), evaluate the *else* branch of the *if* expression and assign the result to $Y.e$.
(v) Return to the parent production.

Steps (1.i) and (2.i) are intended to propagate the false guards of the attribute occurrences of the left-hand-side symbol of a production to the guards of other attribute occurrences when a symbol instance is visited from its parent. Since the set of false guards of a symbol instance is not known until evaluation time, the two steps are implemented with a function that maps each subset of attribute occurrences of the left-hand-side symbol of a production to a set of attribute occurrences in the production. This function is computed during evaluator generation and is represented as a table for each production. (The function may be simplified. See Section III.3.)

3. Discussion

Note that we have assumed that each attribute instance comes with a guard. The guard is tested before evaluating the attribute instance. It should be obvious that the evaluator generator can compute the exact set of attribute occurrences that need the guards. Hence, there is no overhead for the attributes that are not involved in conditional evaluation. For instance, in the example in Fig. 2, the attributes $S.a$, $X.b$, $Y.e$, $Z.h$, and $W.m$ are always evaluated. They do not need guards at all. Based on this observation, the functions used to propagate the false guards may also be simplified.

Adding augmented dependencies to DP may render the attribute grammar unevaluatable by a simple multi-visit evaluator. In this case, we may apply the conditional evaluation method to only a subset of the *if* expressions in the grammar. This naturally leads to the problem of selecting a "maximal" subset of *if* expressions for conditional evaluation. This problem seems very complex since it is remotely related to a problem raised by Kastens (1980): selecting a proper set of augmented dependencies to make an l-ordered attribute grammar ordered.

IV. The Lazy Evaluator

An evaluator generator generates plans for production rules; an evaluator needs to implement the four kinds of instructions in the plans. An *if-then-else* instruction has the form *if v then reset(A) else reset(B)*, where v is a temporary boolean variable and A and B are two sets of attribute occurrences. When v is true, the guards of the attribute occurrences in the set B are reset (that is, their values become false). Similarly, when v is false, the guards of the attribute occurrences in the set A are reset. Initially, the guards of all the attribute instances in a syntax tree are set to true.

Before evaluating a guarded assignment to an attribute occurrence, say $X^{(k)}.a$, the evaluator first checks the associated guard. The assignment is evaluated only if its guard is true.

For visit-parent or visit-child instructions, we will refer to the fragment of a syntax tree in Fig. 3. Let X be the symbol instance on the left-hand side of production instance P. Let Q be the production instance of which X appears on the right-hand side. Let F be the set of attributes of X whose guards are false when the evaluator executes instructions in the production instance P. When the evaluator executes a visit-parent instruction in the production instance P, the evaluator resets the guards of the attribute occurrences in $UOF(Q, X.F)$ in production Q. On the other hand, when the evaluator executes a visit-child-X (for the nth time) instruction in the production instance Q, the evaluator knows the set of attribute instances of symbol X that are computed during the nth visit to X. (This piece of information may be computed by the evaluator generator (Engelfriet and Filé, 1982).) The evaluator skips the visit-child instruction if the guards of all the attribute instances in this set are false. Only if at least one of these guards is true will the evaluator pay a visit to child X. However, it will reset the guards

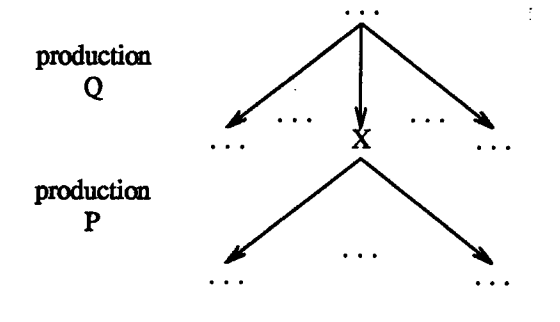

Fig. 3. Symbol X as a child and as a root.

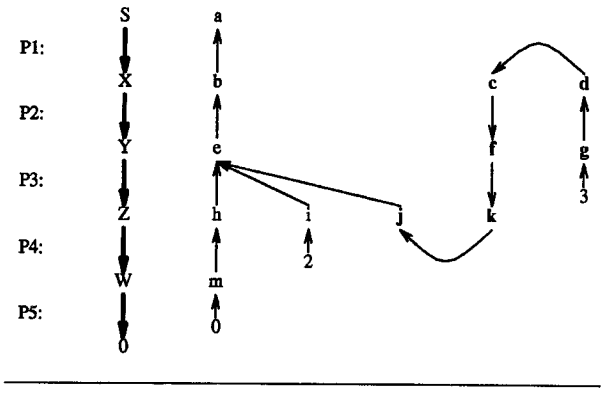

Fig. 4. The syntax tree for the sentence 0.

of the attribute instances in $UOF(P, X.F)$, where F is the set of attribute instances of X whose guards are false.

Example IV.1. The syntax tree for the sentence "0" is shown in Fig. 4 according to the grammar in Fig. 2. Since $Z.h=0$ and $Z.j \in UOF(P3, \{Y.e\})$, it is not necessary to evaluate $Z.j$ and all attribute instances in $UOF(P3, \{Z.j\})$. We have shown that $UOF(P3, \{Z.j\})=\{Z.j, Z.k, Y.f, Y.g\}$. Furthermore, $UOF(P2, \{Y.f\})=\{Y.f, Y.g, X.c, X.d\}$ and $UOF(P1, \{X.c\})=\{X.c, X.d\}$. Hence, the attribute instances $X.c$, $X.d$, $Y.f$, $Y.g$, $Z.j$, and $Z.k$ are not evaluated by the conditional evaluator. (Actually, the second visits to production instances $P1$, $P2$, and $P3$ are all skipped.) By contrast, a traditional evaluator will evaluate *all* attribute instances in the syntax tree.

V. Conclusion

We have proposed a new conditional evaluation technique for simple, multi-visit attribute-grammar evaluators. This technique makes use of the nonstrictness of the conditional equations in an attribute grammar. Unnecessary or infeasible evaluation is omitted; hence, both evaluation time and storage for attribute instances are reduced. This technique may be used to handle exceptions in an attribute grammar, subexpression ordering, and declarations.

It would be interesting to combine our technique with Boyland's conditional attribute grammars. Boyland focuses on the false circularity due to the strict treatment of the nonstrict conditional constructs in traditional attribute-grammar systems. Our technique instead attempts to re-arrange the evaluation order so that unnecessary evaluation is avoided. Combining the two techniques may yield more powerful and efficient attribute evaluators.

Acknowledgment

This work was supported in part by National Science Council, R.O.C., under grants NSC 85-2213-E009-051, NSC 86-2213-E-009-021, and NSC 86-2213-E-009-079.

References

Alblas, H. (1991) Attribute evaluation methods. *Proceedings of the International Summer School SAGA*, pp. 48-113. Prague, Czechoslovakia. Lecture Notes in Computer Science 545, Springer-Verlag, New York, NY, U.S.A.

Bochmann, G. V. (1976) Semantic evaluation from left to right. *Comm. ACM*, **19**(2), 55-62.

Boyland, J. T. and S. L. Graham (1994) Composing tree attributions. *Conference Record of the 21st ACM Symposium on Principles of Programming Languages*, pp. 375-388. Portland, OR, U.S.A.

Boyland, J. T. (1996) Conditional attribute grammars. *ACM Trans. Programming Languages and Systems*, **18**(1), 73-108.

Deransart, P., M. Jourdan, and B. Lorho (1988) *Attribute Grammars: Definitions, Systems and Bibliography*. Lecture Notes in Computer Science 323, Springer-Verlag, New York, NY, U.S.A.

Engelfriet, J. and G. Filè (1982) Simple multi-visit attribute grammars. *J. of Computer and System Sciences*, **24**, 283-314.

Engelfriet, J. and W. de Jong (1990) Attribute storage optimization by stacks. *Acta Informatica*, **27**, 567-581.

Herndon, R. M. Jr. (1986) *A Method for the Construction of Dynamic, Lazy Evaluators for Functional Attribute Grammars*. TR 86-6, Computer Sciences Dept., University of Minnesota, Minneapolis, MN, U.S.A.

Jourdan, M. (1991) A survey of parallel attribute evaluation methods. *Proceedings of the International Summer School SAGA*, pp. 234-255. Prague, Czechoslovakia. Lecture Notes in Computer Science 545, Springer-Verlag, New York, NY, U.S.A.

Kastens, U. (1980) Ordered attribute grammars. *Acta Informatica*, **13**, 229-256.

Knuth, D. E. (1968) Semantics of context-free languages. *Mathematical System Theory*, **2**(2), 127-145. Correction. *ibid.* (March, 1971), **5**(1), 95-96.

Naini, M. M. (1988) A new efficient incremental evaluator for attribute grammars allowing conditional semantic equations. *Proceedings of 1988 IEEE SOUTHEASTCON*, pp. 386-390. Knoxville, TN, U.S.A.

op den Akker, R. and E. Sluiman (1991) Storage allocation for attribute evaluators using stacks and queues. *Proceedings of the International Summer School SAGA*, pp. 234-255. Prague, Czechoslovakia. Lecture Notes in Computer Science 545, Springer-Verlag, New York, NY, U.S.A.

Paakki, J. (1995) Attribute grammar paradigms- a high-level methodology in language implementation. *ACM Computing Surveys*, **27**(2), 196-255.

Yang, W. and W. C. Cheng (1996) *A Polynomial-Time Extension to Ordered Attribute Grammars*. Technical report, Computer and Information Science Dept., National Chiao-Tung Univ., Hsin-Chu, Taiwan, R.O.C.

W. Yang

簡單式多重檢視屬性文法計算器之條件計算方法

楊　武

國立交通大學資訊科學系

摘　要

屬性文法是與前後文無關語言上之一種計算架構。針對屬性公式中的if敘述之非嚴格性，我們可以巧妙地安排計算屬性的順序，以避免計算無必要之屬性。雖然動態式屬性計算器可以達到類似的效果，但是動態式屬性計算器須針對個別的語法樹，計算其屬性之先後順序，並且須要將整棵語法樹儲存在記憶體內，因此動態式計算器比較缺乏效率。相對的，靜態式屬性計算器亦可以避免無需之計算，而無動態式計算器之缺點。我們提出一套方法，藉由仔細分析屬性之間的相依性，來安排計算屬性之順序，以避免無需之計算工作，這套方法可以用於靜態式屬性計算器。