A Classification of Noncircular Attribute Grammars Based on the Look-Ahead Behavior

Wuu Yang

Abstract—We propose a family of static evaluators for subclasses of the well-defined (i.e., noncircular) attribute grammars. These evaluators augment the evaluator for the absolutely noncircular attribute grammars with look-ahead behaviors. Because this family covers exactly the set of all well-defined attribute grammars, well-defined attribute grammars may be classified into a hierarchy, called the NC hierarchy, according to their evaluators in the family. The location of a noncircular attribute grammar in the NC hierarchy is an intrinsic property of the grammar. The NC hierarchy confirms a result of Riis and Skyum, which says that all well-defined attribute grammars allow a (static) pure multivisit evaluator by actually constructing such an evaluator. We also show that, for any finite m, an $NC(m)$ attribute grammar can be transformed to an equivalent $NC(0)$ grammar.

Index Terms—Attribute grammars, noncircular attribute grammars, ordered attribute grammars, pure multivisit attribute grammars, simple multivisit attribute grammars, well-defined attribute grammars, grammar classification.

1 Introduction

Since their introduction in 1968 [19], attribute grammars have attracted much research interest. Attribute grammars are a very convenient and powerful framework for specifying computations on context-free languages [5]. Attribute evaluation has been extensively studied since then.

In an attribute grammar, there are attribution equations that specify the rules for computing values of attribute instances in a syntax tree derived from the attribute grammar. The attribution equations, thus, induce the dependencies among attribute instances in syntax trees. The order of attribute evaluation must be consistent with the dependencies among attribute instances because an attribute instance a can be evaluated only if all the attribute instances used in a's defining equation are already evaluated. A circular dependence in a syntax tree, that is, an attribute instance transitively depending on itself, implies the impossibility of attribute evaluation in general. It becomes an important question whether there are circular dependencies among attribute instances in any syntax tree derived from a given grammar. This is the well-known *circularity problem* of attribute grammars and has been proven to be a complex problem that takes time exponential in the size of the grammar in general [12].

A *well-defined* attribute grammar is one from which no syntax trees with circular dependencies can be derived. Much research effort has been devoted to the discovery of efficient evaluation methods for (subclasses of) the well-defined attribute grammars [19], [7], [1], [22]. There are two categories of evaluation methods: static and dynamic. Static evaluators compute evaluation orders based on the grammars alone, whereas dynamic evaluators establish evaluation orders based on the dependency information in a particular syntax tree. Since static evaluation methods make use of information gathered by analyzing the attribute grammars, they are usually more efficient than dynamic methods in general [13].

In general, an evaluator for the well-defined attribute grammars must take into account the exact dependencies in a particular syntax tree implicitly or explicitly in order to find an evaluation order. In contrast, an evaluator for the class of *absolutely* (or *strongly*) *noncircular attribute grammars* ($ANCAG$) [18] limits its consideration to the direct dependencies within a single production and makes a conservative assumption about transitive dependencies (that is, dependencies that involve two or more production instances in a syntax tree). Thus, $ANCAG$ is a proper subclass of the class of the well-defined attribute grammars but it allows more efficient evaluators [18].

In this paper, we propose a new family of static evaluators for the well-defined attribute grammars. These evaluators turn out to be generalizations of the evaluator for the absolutely noncircular attribute grammars [18] in that they look ahead one or more generations of descendants during evaluation. In contrast, the evaluator for the absolutely noncircular attribute grammars does not look ahead to any descendants during evaluation. Intuitively, evaluators that look ahead more generations of descendants can evaluate larger subclasses of attribute grammars than those that look ahead fewer generations. For instance, circular dependencies that involve a production instance p, p's parent production instance, and p's child production instances in a syntax tree can be detected by an evaluator that looks ahead two generations of descendants, but not by one that looks ahead only one generation.

- *The author is with the Computer and Information Science Department, National Chiao-Tung University, HsinChu, Taiwan, R.O.C. E-mail: wuuyang@cis.nctu.edu.tw.*

Manuscript received 22 Feb. 1999; revised 18 Aug. 1999; accepted 20 Jan. 2000.
Recommended for acceptance by M. Jazayeri.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 109267.

0098-5589/02/$17.00 © 2002 IEEE

Because the family of evaluators covers exactly the set of the well-defined attribute grammars, we may classify the well-defined attribute grammars into a hierarchy, called the *NC hierarchy*, according to the evaluators for individual grammars. The $NC(m)$ class, where $m \geq 0$, consists of those grammars that can be evaluated with a static, visit-oriented evaluator that is allowed to look ahead m generations of descendants in the syntax tree during evaluation. In particular, the class of the absolutely noncircular attribute grammars is equivalent to the $NC(0)$ class and the well-defined attribute grammars coincides with the $NC(\infty)$ class in our classification. Proof of this claim is included in Section 8.

It is commonly agreed that most practical attribute grammars belong to the well-known *ordered attribute grammars* (OAG) class [16]. In contrast, this paper presents efficient static evaluators for grammars that fall out of the OAG class. The algorithms in this paper can decide the least number of generations of descendants that an evaluator needs to look ahead in order to establish an appropriate evaluation order. Furthermore, we also present a method that transforms $NC(m)$ grammars, for any positive m, to equivalent $NC(0)$ grammars by expanding the grammar rules. The resulting $NC(0)$ grammars can be evaluated with the efficient evaluators of [18] or it can be further transformed to *l-ordered* grammar [6] for evaluation. The algorithms presented in this paper tells us the least amount of expansion needed.

Note that, though the NC hierarchy is defined in terms of the attribute evaluators, the location of a noncircular attribute grammar in the NC hierarchy is an intrinsic property of the grammar. Therefore, the NC hierarchy is a good way to classify noncircular attribute grammars.

To evaluate attribute instances in a syntax tree, the evaluation order of attribute instances in a production instance must be consistent with three kinds of dependencies:

1. the direct dependencies among attribute instances in the production instance (enforced by the attribution equations of the production),
2. the upward transitive dependencies among attribute instances of the left-hand-side nonterminal of the production instance (enforced by the context of the production instance in the syntax tree), and
3. the downward transitive dependencies among attribute instances of the right-hand-side nonterminals of the production instance (enforced by the subtrees rooted at the right-hand-side nonterminals in the syntax tree).

The crux of the NC evaluation algorithms is that we use the downward transitive dependencies (as well as the direct dependencies) to determine a set of evaluation plans for each production. One of the plans is chosen according to the upward transitive dependencies in the syntax tree during attribute evaluation.

Deciding whether a given grammar belongs to a class in the NC hierarchy takes time that is exponential in the size of the grammar. There are five factors that contribute to the size of a grammar: the number of terminals, the number of nonterminals, the number of productions, the number of attributes associated with a terminal or a nonterminal, and the length of the right-hand side of a production. We will give detailed analysis of the time complexity of the algorithms involving the NC hierarchy in terms of these five factors.

A characteristic of attribute grammars in all the classes in the NC hierarchy is that they allow static, visit-oriented evaluators. From this point of view, all the classes in the NC hierarchy can be viewed as extensions to Kastens's *ordered attribute grammars* (OAG) [16] and *l-ordered* AG [15]. In all the NC classes, each production is associated with one or more evaluation plans. A plan consists of a sequence of instructions of three kinds: evaluating an attribution equation, visiting the parent production, or visiting one of the child productions. *l-ordered* AG consists of those well-defined attribute grammars for which there is a way to associate exactly one evaluation plan, rather than a set of plans, for each production. The attempt to find exactly one plan for each production needs to examine a lot of possibilities (that is, the choice of a linear order from a topological order in the *ComputeOrder* function in Fig. 6) and, hence, becomes an *NP*-complete problem [6]. Kastens proposes a polynomial-time procedure that is capable of finding exactly one evaluation plan for each production for a subclass of *l-ordered* grammars. This is called the OAG class. In a previous paper [26], we have further improved Kastens's OAG algorithm to cover a larger subclass of *l-ordered* AG which still takes polynomial time.

Kastens's *l-ordered* AG is also called the class of the *simple multivisit* (SMV) attribute grammars [6]. The word *simple* implies that every attribute of a nonterminal has a fixed visit number in the evaluation plans. Dropping this restriction, we obtain the class of the *pure multivisit* (PMV) attribute grammars [6]. The pure multivisit property coincides with the noncircularity property; in fact, Riis and Skyum prove that, for every well-defined attribute grammar, there is a tree-walking evaluator that makes a bounded number of visits to any node of a syntax tree [24]. This paper confirms Riis and Skyum's result by showing how to actually construct such a tree-walking evaluator for all well-defined grammars. As far as we know, there are no such evaluators in the published literature. Furthermore, our work has refined PMV into a hierarchy of classes.

The classification discussed in this paper is a characterization of attribute grammars based on the look-ahead behavior of their static evaluators. Nielson [21] characterizes attribute grammars based on the computation sequences of their evaluators. Because Nielson did not consider the look-head behaviors of the evaluators, her characterization is restricted to subclasses of $NC(0)$ (i.e., $ANCAG$). Within the capability of Nielson's framework, her approach is similar to ours in that the *top-down assignment of partitions* in her framework is similar to the *traverse* procedure in Fig. 7 without the look-ahead behavior (actually, this is a common characteristic of all $ANCAG$ evaluators, including the one in [18].) Our work extends Nielson's in that it characterizes all well-defined attribute grammars and proposes algorithms to construct evaluation plans and to perform evaluation.

In this paper, we will first present the plan generator and the evaluator for the $NC(1)$ class. Analysis of time

complexity of each algorithm follows immediately after the algorithm is presented. Generalizations of $NC(1)$ to $NC(m)$ should be obvious. Finally, the plan generator and the evaluator for the $NC(\infty)$ class is proposed. We also show that, for any finite m, an $NC(m)$ attribute grammar can be transformed to an equivalent $NC(0)$ grammar. The rest of this paper is organized as follows: The notations are introduced in Section 2. Two graph representations for downward transitive dependencies among attribute occurrences in productions are proposed in Sections 3 and 4. The $NC(1)$ class is defined in Section 4. In Section 5, an algorithm for computing the evaluation plans for the $NC(1)$ class is presented together with a discussion of the correctness of the algorithm. The $NC(1)$ evaluator based on the plans computed by the algorithm in Section 5 is described in Section 6. In Section 7, we generalize $NC(1)$ to other $NC(m)$ classes. The $NC(\infty)$ class is discussed in Section 8. We conclude this paper in the last section, together with a discussion of related works.

2 Notations

In this section, we define the notations used in this paper. Basically, we adopt Kastens's notations [16]. 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 productions 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 whose left-hand-side symbol is X. In this paper, a production q will be written as

$$X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k,$$

where $X_0, X_1, X_2, \ldots, X_k$ are nonterminal symbols and $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_k$ are (possibly empty) strings of terminal symbols. Furthermore, we assume that the start symbol does not appear in the right-hand side of any production.

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. There may be several *occurrences* of a symbol in a production. Furthermore, a production 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 are partitioned into two disjoint subsets, called the *inherited* attributes and the *synthesized* attributes. We will assume that the start symbol has no inherited attributes and that a terminal has only a synthesized attribute that represents the character string comprising the terminal symbol.

An attribute a of a symbol X is denoted by $X.a$. Since there may be many occurrences of a symbol, there may be many *occurrences* of an attribute in a production. Similarly, since a production may be applied more than once in a syntax tree, there may be many *instances* of an attribute occurrence in a syntax tree.

There are attribution equations defining these attributes. In a production, there is exactly one attribution equation defining each synthesized attribute occurrence of the left-hand-side symbol and each inherited attribute occurrence of the right-hand-side symbols.

There is still some freedom in specifying the attribution equations. Therefore, we require that, for each production, the attribution equations are 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. This is called the *normal form* in the literature [2], [22]. The advantage of the normal form is that it specifies the fewest 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. In this paper, we will assume that all attribute grammars under discussion are in normal form. An example attribute grammar is shown in Fig. 1a.

Attribution equations indicate dependencies among attribute occurrences in a production. The dependency relations in a production q may be represented in the *dependence graph* of q, denoted by $DP(q)$, in which nodes denote attribute occurrences in production q and edges dependencies between attribute occurrences. An edge $X.a \rightarrow Y.b$ means that the attribute occurrence $X.a$ is a parameter to the function defining the attribute occurrence $Y.b$ in production p. Fig. 1b shows the DP graphs for the example grammar in Fig. 1a. Fig. 1c and Fig. 1d will be discussed in the next section.

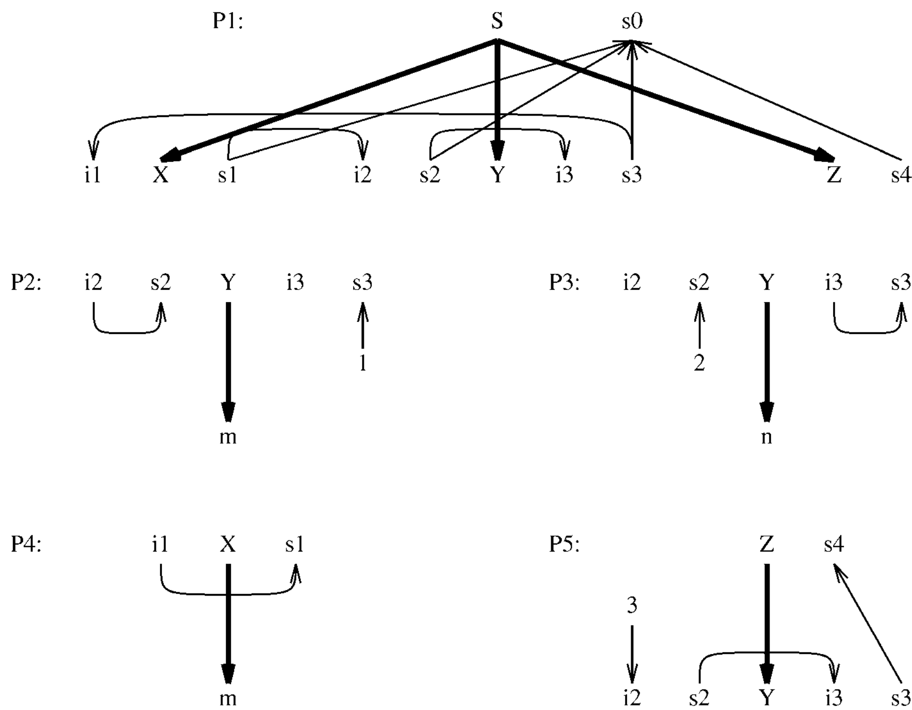
3 The *Down* Graphs

In order to evaluate attribute instances in a syntax tree, the attribute instances must be evaluated in an order that is consistent with the dependencies among the attribute instances. Dependencies among attribute instances in a syntax tree are derived from the dependencies among attribute occurrences in individual productions. The attribution equations in an attribute grammar indicate the direct dependencies among attribute occurrences in the productions. Based on the direct dependencies, we may calculate the transitive dependencies among attributes of a symbol. The transitive dependencies among attributes of a symbol arise from two different sources: The transitive dependencies may arise due to the *context* of the symbol or they may arise due to the *derived structure* of the symbol.

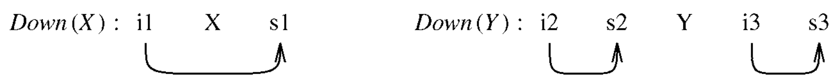
An instance of a nonterminal X in an attributed syntax tree T is the interface between the context and the derived structure of the instance X in T. The context of X in T is obtained from T by removing the subtree (but retaining the node representing X) rooted at X; the derived structure of X is the subtree rooted at X. The transitive dependencies among X's attributes due to the context of X are called the *upward* transitive dependencies. Similarly, the transitive dependencies among X's attributes due to the derived

P1:	$S \longrightarrow XYZ$	$S.s0 := X.s1 + Y.s2 + Y.s3 + Z.s4$
		$X.i1 := Y.s3$
		$Y.i2 := X.s1$
		$Y.i3 := Y.s2$
P2:	$Y \longrightarrow m$	$Y.s2 := Y.i2$
		$Y.s3 := 1$
P3:	$Y \longrightarrow n$	$Y.s2 := 2$
		$Y.s3 := Y.i3$
P4:	$X \longrightarrow m$	$X.s1 := X.i1$
P5:	$Z \longrightarrow Y$	$Z.s4 := Y.s3$
		$Y.i2 := 3$
		$Y.i3 := Y.s2$

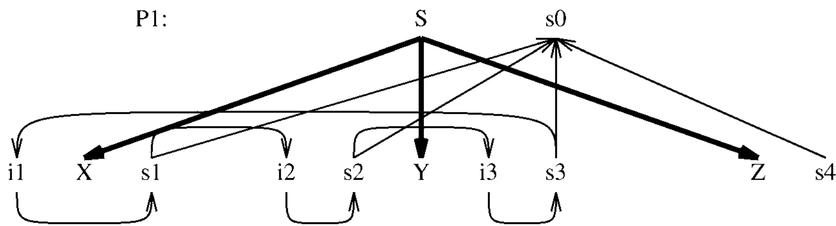
(a)



(b)



(c)



(d)

Fig. 1. An attribute grammar, its DP and $Down$ graphs. This grammar belongs to $NC(1)$ but not to $ANCAG$ (or $NC(0)$). (a) An attribute grammar. (b) The *DP* graphs. (c) The *Down* graphs. (d) The *IDP-ANCAG(P1)* graph.

structure of X are called the *downward* transitive dependencies. The crux of our work is that downward transitive dependencies are used to construct evaluation plans for productions (in Sections 3, 4, and 5) and upward transitive dependencies are used for selecting evaluation plans during attribute evaluation (in Section 6).

```
Algorithm: ComputeDownGraph
/* Initially, Down(X) = a graph with nodes only, for every symbol X. */
for each symbol X do
   Down(X) := a graph whose nodes are attributes of symbol X and which has no edges
end for
repeat
   changed := false
   for each production p: X0 ⟶ α0 X1 α1 X2 α2 ... Xk αk do
      /* Augment DP(p) with Down(X1), Down(X2), ..., Down(Xk). */
      G := DP(p) ∪ Down(X1) ∪ Down(X2) ∪ ... ∪ Down(Xk)
      /* Compute NewDown(X0). */
      NewDown(X0) := project(G, { attributes of X0 })
      /* Check if NewDown(X0) contains new dependencies. */
      if NewDown(X0) ⊈ Down(X0) then
         changed := true
         Down(X0) := Down(X0) ∪ NewDown(X0)
      end if
   end for
until changed = false

function project(G, N) return a new graph
/* G is a graph. N is a set of nodes in G. */
G := the transitive closure of G
Remove, from G, all the nodes not in N and all edges incident on the deleted nodes.
return the resulting graph G
end function project
```

Fig. 2. The *ComputeDownGraph* algorithm.

Definition. *The* downward transitive dependency graph *of a symbol X is a graph in which the nodes are X's attributes and the edges, say $X.a \rightarrow X.b$, denote a (transitive) dependency of $X.b$ on $X.a$ that exists in a subtree derived from X.*

From every subtree derived from a nonterminal X, we may find a transitive dependence relation among the attributes of X. The downward transitive dependence graph of X is the union of the transitive dependence relations of the subtrees derived from X.

The downward transitive dependency graph of a symbol is expensive to compute. In the literature, there are several proposals to compute approximations to the downward transitive dependency graphs. In particular, Kennedy and Warren [18] use the $Down(X)$ graph, for each symbol X, to approximate the downward transitive dependency graphs. Their $Down$ graphs are defined essentially by the following mutually recursive equations:

$$IDP\text{-}ANCAG(p) = DP(p) \cup Down(X_1) \cup Down(X_2) \cup ... \cup Down(X_k),$$

where p is a production written as $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 ... X_k \alpha_k$.

$$Down(X) = \bigcup \{project(IDP\text{-}ANCAG(q), \{X's\ attributes\}) | X\ is\ the\ left\text{-}hand\text{-}side\ nonterminal\ of\ production\ q\}.$$

The term IDP means *induced dependency graph for a production* [16]. The $project$ function in the above equations retains a subset of the nodes in a given dependency graph and computes the transitive dependencies among nodes in the retained subset based on the dependency graph. The $Down(X)$ graph is a safe approximation in the sense that the $Down(X)$ graph includes all possible transitive dependencies among X's attributes in all possible syntax trees; in addition, it may also include a few spurious dependencies.

In Fig. 2, an algorithm is presented that computes the $Down(X)$ graph. This algorithm is a variation of the standard algorithm available in the literature [18]. Due to the assumption that the start symbol and the terminal symbols have only synthesized attributes, their $Down$ graphs contain only nodes but no edges. In the algorithm in Fig. 2, the $Down(X)$ graph for each symbol X is assumed to be a graph with no edges initially. For each production p, the $DP(p)$ graph augmented with the $Down(X)$ graph for each nonterminal X on the right-hand side of p is examined repeatedly in order to discover new edges in the $Down$ graphs. The algorithm terminates when no more edges can be added to the $Down$ graphs.

Example. Fig. 1a is an example attribute grammar and Fig. 1b is the DP graphs for the productions. The attribute grammar does not belong to the class of the absolutely noncircular attribute grammars according to the characterization in [18]. Fig. 1c shows the $Down$ graphs for this example. Since the symbols S and Z contain only one synthesized attribute each, their $Down$ graphs contain only a node each and no edges at all. Hence, they are omitted in Fig. 1c. Fig. 1d is the *IDP-ANCAG(P1)* graph.

In the discussion of the time complexity of the $ComputeDownGraph$ algorithm, let $|N|$, $|T|$, and $|P|$ be the numbers of nonterminals, terminals, and productions, respectively. Let h be the maximum number of attributes per symbol and l be the maximum number of (nonterminal) symbols on the right-hand side of a

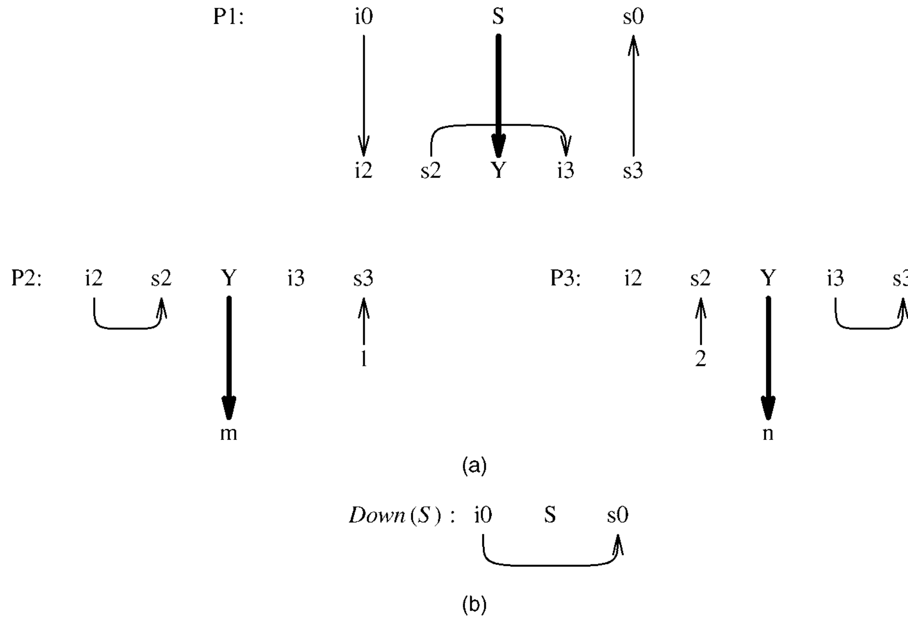


Fig. 3. An example in which the $Down(S)$ graph contains a spurious transitive dependency edge.

production. The sizes of the dependency graph $DP(p)$ for a production p is $O(h^2(l+1)^2)$ since there are at most $h(l+1)$ attribute occurrences in a production p. Hence, computing the $DP(p)$ graph takes $O(h^2(l+1)^2)$ time. We will use a linked-list representation for $DP(p)$. Each element of the linked list represents an edge. Each $Down(X)$ graph, for a nonterminal X, is represented both as a linked list (of edges) and as a Boolean matrix. The size of the graph $Down(X)$ for each symbol X is $O(h^2)$ since there are at most h attributes associated with the symbol X.

To compute the union of the $DP(p)$ graph and a $Down(X)$ graph, we simply build a node with two pointers, each of which points to one of the two graphs. Thus, a single union operation takes one unit of time. We need to perform k union operations, where $k \leq l$, in each iteration of the *for* loop, which takes $O(l)$ time. To project a total or partial order onto a subset of its nodes, we first transform the linked-list representation into a Boolean matrix, and then calculate the transitive closure of the Boolean matrix, and finally eliminate unnecessary rows and columns. Computing transitive closure dominates the computation time, which is $O(h^3(l+1)^3)$ with the Floyd-Warshall algorithm. (A slightly better algorithm for computing transitive closures takes $O(n^{\log 7}\ \log n)$ time, for an n-node graph [3].) Comparing the old and the new *Down* graphs takes $O(h^2)$ time. As soon as a new edge is found in the $NewDown$ graph, it is added to both the linked-list representation and the Boolean-matrix representation of the *Down* graph. Therefore, each iteration of the inner *for* loop in Fig. 2 takes $O(h^3(l+1)^3)$ time.

There will be $\mid P \mid$ iterations of the *for* loop in each iteration of the *repeat* loop. For each symbol X, there are at most $h(h-1)/2$ dependency edges in $Down(X)$ and it takes at most $\mid N \mid$ iterations of the *repeat* loop to propagate a dependence edge to all related symbols. Hence, there will be at most $\mid N \mid h(h-1)/2$ iterations of the *repeat* loop. The total amount of time needed by the $ComputeDownGraph$ algorithm is $O(\mid P \mid\mid N \mid h^5 l^3)$.

The amount of space required is $O(\mid P \mid h^2(l+1)^2)$ for the DP graphs and $O(\mid N \mid h^2)$ for the $Down$ graphs.

The $Down(X)$ graph is a safe approximation to the downward transitive dependency graph of X. Spurious downward transitive dependency edges in $Down(X)$ may occur in two ways. First, certain dependencies in $Down(X)$ may not occur *simultaneously* in any instance of X in any syntax tree. For instance, consider $Down(Y)$ in Fig. 1c. The two dependencies $i2 \rightarrow s2$ and $i3 \rightarrow s3$ will never occur in the same instance of Y in any syntax tree, though they may occur in different instances of Y in some syntax trees. Second and more seriously, certain dependencies may never occur in any syntax tree at all. For instance, in the example in Fig. 3, the $Down(S)$ graph contains a spurious transitive dependency edge $i0 \rightarrow s0$, which will never occur in any syntax tree. The edge $i0 \rightarrow s0$ is introduced into $Down(S)$ due to the assumption that the two transitive dependency edges $i2 \rightarrow s2$ and $i3 \rightarrow s3$ may occur *simultaneously* in an instance of production $P1$. This scenario will not happen because only one of the two edges can occur in any instance of $P1$.

Some spurious transitive dependency edges may be gradually removed by looking ahead more and more generations of descendants. This finite look-ahead behavior results in the $NC(m)$ class of grammars, where m is a nonnegative integer. The rest of the spurious transitive dependency edges may be eliminated by looking ahead as many generations of descendants as there are in a particular syntax tree. This infinite look-ahead behavior, in turn, results in the $NC(\infty)$ class of grammars. In the next section, we will show how to look ahead one generation of descendants in the computation of the downward transitive dependencies. In Section 8, we will discuss the $NC(\infty)$ class.

```
Algorithm: ComputeDCG
/* Initially, DCG(X, q) = a graph with nodes only, for every symbol X */
/* and for every production q with X on its left-hand side. */
for each symbol X and every production q with X on its left-hand side do
   DCG(X, q) := a graph whose nodes are attributes of symbol X and which has no edges
end for
repeat
   changed := false
   for each production p : X0 --> α0 X1 α1 X2 α2 ... Xk αk do
      for each combination of productions (p1, p2, ..., pk), where pi's has Xi's on its left-hand sides, respectively do
         /* Augment DP(p) with DCG(X1, p1), DCG(X2, p2), ..., DCG(Xk, pk). */
         G := DP(p) ∪ DCG(X1, p1) ∪ DCG(X2, p2) ∪ ... ∪ DCG(Xk, pk)
         /* Compute NewDCG(X0, p). */
         NewDCG(X0, p) := project(G, { attributes of X0 })
            /* Check if NewDCG(X0, p) contains new dependencies. */
            if NewDCG(X0, p) ⊈ DCG(X0, p) then
               changed := true
               DCG(X0, p) := DCG(X0, p) ∪ NewDCG(X0, p)
            end if
      end for
   end for
until changed = false
```

Fig. 4. The *ComputeDCG* algorithm.

4 THE DOWNWARD CHARACTERISTIC GRAPHS

As discussed in the previous section, the $Down$ graph of a nonterminal is a conservative estimation to the actual downward transitive dependencies that may occur in all derived structures of the nonterminal. It is possible to compute more accurate estimations by considering the individual productions separately.

Definition. *Let X be the left-hand-side nonterminal of production q. The* downward characteristic graph of X in the subtrees derived via production q, *denoted by $DCG_X(q)$, is a graph in which the nodes are X's attributes and the edges, say $X.a \rightarrow X.b$, denote a (transitive) dependency of $X.b$ on $X.a$ in some subtree derived from X via production q.*

Equivalently, $DCG_X(q)$ may be defined as follows: Let q be the production $X \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$. Let p_i be a production whose left-hand-side nonterminal is X_i, for all $i = 1, 2, \ldots, k$.

$$DCG_X(q) = \cup \{project(ADP(q \mid p_1, p_2, \ldots, p_k), \{X's\ attributes\}) \mid p_1, p_2, \ldots, p_k$$

are productions whose left-hand-side nonterminals are $X_1, X_2, \ldots X_k$, respectively, where X is the left-hand-side nonterminal of production q.

$$ADP(q \mid p_1, p_2, \ldots, p_k) = DP(q) \cup DCG_{X1}(p_1) \cup DCG_{X2}(p_2) \cup \ldots \cup DCG_{Xk}(p_k).$$

$ADP(q \mid p_1, p_2, \ldots, p_k)$ is called the *augmented dependency graph* of production q with the subtrees derived from $p_1, p_2, \ldots, p_k$. Furthermore, we define the set of all possible augmented dependency graphs of q as follows:

$$SADP(q) = \{ADP(q \mid p_1, p_2, \ldots, p_k) \mid p_1, p_2, \ldots, p_k$$

are productions whose left-hand-side nonterminals are $X_1, X_2, \ldots X_k$, respectively.

The $ComputeDCG$ algorithm for computing the $DCG_X(q)$ graphs, shown in Fig. 4, is obtained by modifying the $ComputeDownGraph$ algorithm in Fig. 2.

Example. Fig. 5 shows three DCG graphs for the example in Fig. 1. The other DCG graphs contain no edges and, hence, are not shown here. Note that $Down(Y)$ in Fig. 1c is divided into $DCG_Y(P2)$ and $DCG_Y(P3)$ in Fig. 5. The grammar in Fig. 1 is not an $ANCAG$ because *IDP-ANCAG(P1)*, shown in Fig. 1d, contains circular dependencies. Fig. 5 also shows $ADP(P1 \mid P4, P2, P5)$ and $ADP(P1 \mid P4, P3, P5)$, both of which are acyclic.

Comparing the definitions, we can verify the following observation:

Observation.

$$\bigcup_{all\ q\ whose\ left\text{-}hand\ side\ is\ X} DCG_X(q) \subseteq Down(X).$$

Note the $\subseteq$ relation in the above observation. It is possible that some dependence edges in $Down(X)$ do not appear in any $DCG_X(q)$, for any production q.

It is obvious that $DCG_X(q)$ is a more accurate representation of the downward transitive dependencies than $Down(X)$ in the sense that $DCG_X(q)$ contains fewer spurious dependency edges. Next we define a new class of noncircular attribute grammars based on this more accurate representation.

Definition. *An attribute grammar G is an $NC(1)$ grammar if and only if, for all productions q in G, every graph in $SADP(q)$ is acyclic.*

The time complexity for computing the DCG graphs (as well as the ADP graphs) is analyzed as follows: The size of a $DP(p)$ graph, for each production p, is $O(h^2(l+1)^2)$. There are $\mid P \mid$ such graphs. The size of $DCG_X(p)$ is $O(h^2)$. There

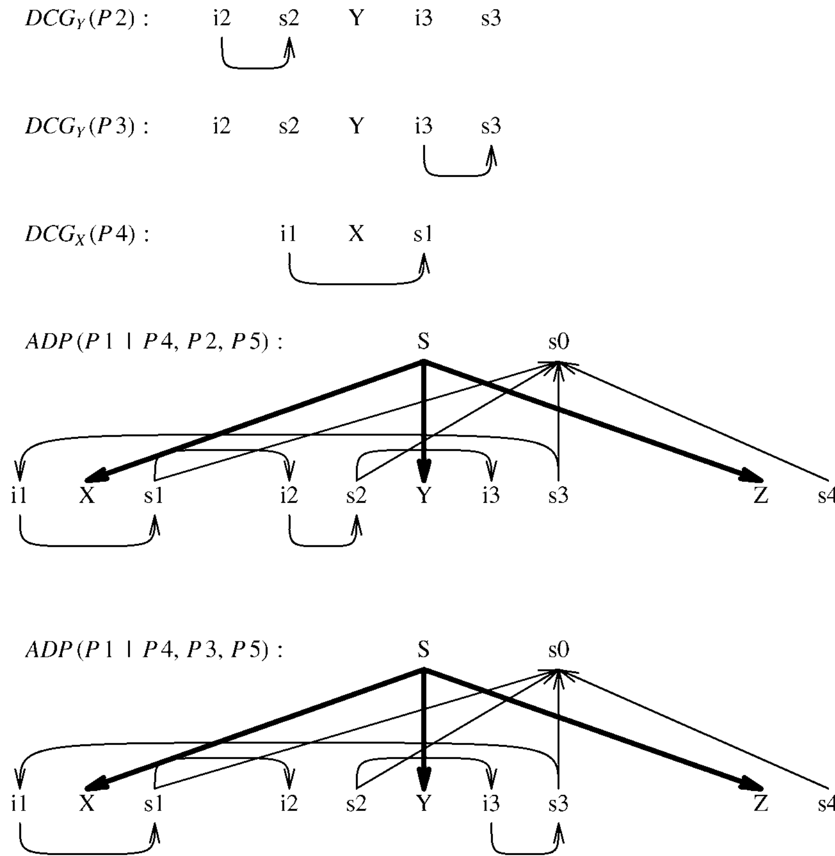


Fig. 5. The DCG graphs and the ADP graphs for the example in Fig. 1.

are $O(| P || N |)$ such graphs. The union operations take $O(l)$ time. A projection operation takes $O(h^3(l+1)^3)$ time. A comparison operation takes $O(h^2)$ time. Thus, an iteration of the inner *for* loop in the $ComputeDCG$ algorithm takes $O(h^3(l+1)^3)$ time. There are at most $| P |^{(l+1)}$ iterations of the inner *for* loop in each iteration of the *repeat* loop. Each iteration adds at least one edge to a $DCG_X(p)$ graph. Therefore, there are $O(| P || N | h^2)$ iterations of the *repeat* loop. The total amount of time required by the $ComputeDCG$ algorithm is $O(| P |^{(l+2)}| N | h^5 l^3)$.

The amount of space required is $O(| P | h^2(l+1)^2)$ for the DP graphs and $O(| P || N | h^2)$ for the DCG graphs.

To determine whether $G \in NC(1)$, it is necessary to check every graph in $SADP(q)$, for every production q. For each production q, the number of graphs in $SADP(q)$ may be $O(| P |^l)$, where l, as defined in Section 3, is the maximum number of (nonterminal) symbols on the right-hand side of a production. The transitive closure of $ADP(q \mid p_1, p_2, \ldots, p_k)$ is already computed by the $ComputeDCG$ algorithm. It suffices to check only the diagonal elements in the Boolean matrix for each $ADP(q \mid p_1, p_2, \ldots, p_k)$, which takes $O(h(l+1))$ time. Therefore, it is possible to determine $G \in NC(1)$ in at most $O(| P |^{l+1}\, h(l+1))$ time.

The fundamental property of the $NC(1)$ class is that all $NC(1)$ grammars are well-defined attribute grammars.

Theorem. *Every $NC(1)$ grammar is a well-defined attribute grammar (that is, every attributed syntax tree derived from an $NC(1)$ grammar contains no circular dependencies).*

Proof. Suppose that the theorem is wrong. There must be a syntax tree T derived from an $NC(1)$ grammar with circular dependencies. The circular dependencies in T must include one or more instances of productions. Let q be the production instance involved in the circular dependencies that is located *nearest* to the root of T. Let $p_1, p_2, \ldots, p_k$ be the productions applied to the right-hand-side nonterminals of the production instance q in T. Since the syntax tree contains circular dependencies, $ADP(q \mid p_1, p_2, \ldots, p_k)$ must also contain corresponding circular dependencies because $ADP(q \mid p_1, p_2, \ldots, p_k)$ is a *safe* approximation to the actual transitive dependencies in the production instance q in T. This contradicts the definition of $NC(1)$ □

It is interesting to compare the $NC(1)$ class with other classes of attribute grammars. According to the characterization of the absolutely noncircular attribute grammars ($ANCAG$) in [18], an attribute grammar is an $ANCAG$ if and only if *IDP-ANCAG(q)*, for every production q, is acyclic. $NC(1)$ differs from $ANCAG$ in that the DCG graphs are used instead of the $Down$ graphs. Because the DCG graphs are subgraphs of the corresponding $Down$ graphs, $NC(1)$ is strictly larger than $ANCAG$.

Theorem. *$\bigcup SADP(q) \subseteq$ IDP-ANCAG(q), for every production q.*

Theorem. *Every $ANCAG$ grammar is an $NC(1)$ grammar, but* not vice versa.

Proof. Note that $DCG_X(p)$ is a subgraph of $Down(X)$ for all nonterminals X and all appropriate productions p. If $ADP(q \mid p_1, p_2, \ldots, p_k)$ contains a cycle, so too does *IDP-ANCAG(q)*. Thus, every $ANCAG$ grammar is an $NC(1)$ grammar. That the $NC(1)$ class is not equivalent to the $ANCAG$ class is witnessed by the example in Fig. 1. □

5 Evaluation Orders for the $NC(1)$ Class

Similar to the ordered attribute grammars [16], $NC(1)$ grammars can be evaluated with evaluation plans that are computed purely from the grammar (not from any particular syntax trees). The only difference is that, in OAG, there is an evaluation plan for each production. In contrast, in $NC(1)$, there is a set of evaluation plans for each production. During evaluation, one plan of the set is chosen based on the contexts and derived structures of instances of the production.

To find evaluation plans for a production is essentially to find the evaluation orders of attribute occurrences of the production [16], [17]. The evaluation orders must be consistent with the dependencies among the attribute occurrences. There are three kinds of dependencies among the attribute occurrences in a production: the direct dependencies due to the attribution equations in the production, the upward transitive dependencies among attributes occurrences of the left-hand-side nonterminal of the production, and the downward transitive dependencies among attribute occurrences of the right-hand-side nonterminals. The evaluation orders are computed from $ADP(q \mid p_1, p_2, \ldots, p_k)$, which contains both the direct dependencies and approximations to the downward transitive dependencies. The upward transitive dependencies are implicitly used to choose an evaluation order among the set of evaluation orders associated with a production during attribute evaluation (see Section 6).

Suppose that an attribute grammar G belongs to the $NC(1)$ class. According to the definition of $NC(1)$, every $ADP(q \mid p_1, p_2, \ldots, p_k)$ in G is acyclic. Any topological order derived from $ADP(q \mid p_1, p_2, \ldots, p_k)$ could be a feasible evaluation order for attribute occurrences in production q. However, the context of an individual instance of production q in a syntax tree may further enforce constraints on the evaluation order of attribute occurrences of the left-hand-side nonterminal of q.

We will use the Greek letter ω to denote a constraint on the evaluation order of attributes of a symbol and ψ to denote an evaluation order of attribute occurrences of a production in what follows. A constraint ω is actually a total order among the attributes of a symbol and ψ is a total order among the attribute occurrences of a production. Examples of ω and ψ are given at the end of this section.

Fig. 6 is an algorithm for computing the evaluation plans for productions. The algorithm follows a work-list scheme. The elements of the work list are tuples of the form (q, ω), where 1) q is a production and 2) ω is a total evaluation order of the attributes of the left-hand-side nonterminal of production q.

Let S be the start symbol of the attribute grammar. First, we need to determine an evaluation order for S's attributes. Because the attribute grammar is assumed to be in normal form, there will be no outgoing edges from S's attribute instances in any syntax tree. Hence, any arbitrarily chosen evaluation order of the (synthesized) attributes of the start symbol will never cause circular evaluation order in a syntax tree. Let μ be an arbitrarily chosen evaluation order of S's attributes. Initially, the work list WL contains a tuple (q, μ), for each production q whose left-hand-side nonterminal is S.

The elements of the work list are picked up one by one. When an element (q, ω) is picked up, let the production q be denoted by $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$. The algorithm computes a total evaluation order ψ of the attribute occurrences in $ADP(q \mid p_1, p_2, \ldots, p_k)$, for each $ADP(q \mid p_1, p_2, \ldots, p_k) \in SADP(q)$ under the constraint that ψ must be compatible with ω, that is, the projection of ψ onto the attribute occurrences of the left-hand-side nonterminal of production q is identical to ω. That we can always find such an evaluation order ψ is supported by the theorem discussed later in this section.

The Π function in the $ComputePlan$ algorithm is used to avoid repeated processing of a tuple. In Fig. 6, $\Pi(q, \omega)$ is *undefined* initially for every q and ω. $\Pi(q, \omega)$ becomes *defined* when the pair (q, ω) is added to the work list WL.

The $ComputePlan$ algorithm also builds a selection function Γ and a projection function Θ when computing the evaluation orders. (The two functions are represented as two arrays in Fig. 6.) The Γ function will be used to select a plan from the set of plans associated with a production during the evaluation of attribute instances in a syntax tree. $\Gamma[q, \omega, p_1, p_2, \ldots, p_k]$ is the evaluation order of attribute occurrences in production q when the attribute occurrences of the left-hand-side nonterminal of q must be evaluated in the ω order (this is a constraint) and the productions applied at the right-hand-side nonterminals are $p_1, p_2, \ldots, p_k$, respectively. The projection function $\Theta[q, \omega, i, p_1, p_2, \ldots, p_k]$ is the projection of $\Gamma[q, \omega, p_1, p_2, \ldots, p_k]$ onto the attribute occurrences of the ith nonterminal on the right-hand side of production q. A projection of a total order of some nodes onto a subset of the nodes is to restrict the total order to the subset of the nodes. Examples of projection are given at the end of this section.

Note that $\Gamma[q, \omega, p_1, p_2, \ldots, p_k]$ is a total order derived from a partial order in the $ComputeOrder$ function. Obviously, there could be more than one total order compatible with a given partial order. The $ComputeOrder$ function chooses one of the compatible total orders arbitrarily.

There is a small notational misuse in $\Gamma[q, \omega, p_1, p_2, \ldots, p_k]$ and $\Theta[q, \omega, i, p_1, p_2, \ldots, p_k]$. Note that the symbol k denotes the number of nonterminal symbols on the right-hand side of production q. Because different productions may have different numbers of nonterminals on the right-hand sides, the values of k vary from production to production. This misuse could have been remedied with one more level of indexing notations. For the sake of simplicity, we omit that level of indexing and bear in mind that k means different constants for different productions.

For each nonterminal X_i on the right-hand side of production q, let ω_i be the projection of ψ onto X_i's attributes. If $\Pi(p_i, \omega_i)$ is not already defined, the tuple (p_i, ω_i) is added to

```
Algorithm: ComputePlan
WL := ∅
Π := a function that is undefined in every place
μ := an arbitrarily chosen evaluation order of the start symbol's attributes
for each production q whose left-hand-side non-terminal is the start symbol S do
   Π(q, μ) := defined
   WL := WL ∪ { (q, μ) }
end for
repeat
   (q, ω) := an element of WL
   WL := WL − { (q, ω) }
   Let production q be the production X0 ⟶ α0X1α1X2α2 . . . Xk αk .
   for each ADP(q | p1, p2, . . . , pk) ∈ SADP(q) do
      ψ := ComputeOrder(ADP(q | p1, p2, . . . , pk), ω)
      Γ[q, ω, p1, p2, . . . , pk] := ψ
      for each non-terminal Xi appearing on the right-hand-side of production q do
         ωi := project(ψ, { Xi's attribute occurrences })
         Θ[q, ω, i, p1, p2, . . . , pk] := ωi
         if Π(pi, ωi) = undefined then
            Π(pi, ωi) := defined
            WL := WL ∪ { (pi, ωi) }
         end if
      end for
   end for
until WL = ∅

function ComputeOrder(G, ω) return a new total order
/* G is an ADP graph of a production. ω is a total order */
/* of the attributes of the left-hand-side non-terminal in the production. */
Let ω denote the total order a1 ⟶ a2 ⟶ . . . ⟶ am .
for i from 1 to m − 1 do
   G := G ∪ { the edge ai ⟶ ai+1 }
end for
ψ := any total order compatible with the partial order represented by the graph G.
return ψ
end function ComputeOrder

function project(ψ, N) return a new total order
/* ψ is a total order of certain attribute occurrences in a production. */
/* N is a subset of the attribute occurrences appearing in ψ. */
Remove, from ψ, all the attribute occurrences not in N
return the resulting total order
end function project
```

Fig. 6. The *ComputePlan* algorithm.

the work list. The *ComputePlan* algorithm examines tuples in the work list one by one, possibly adding new tuples to the list until the list becomes empty.

Definition. *Let ψ be a (total or partial) order of certain attributes. Let ω be a (total or partial) order of a subset of attributes in ψ. We say that ψ* is compatible with *ω if every edge $a \to b$ in ω is an edge in ψ.*

The following lemma is similar to Lemma 1 of [21].

Lemma. *Let q be the production $X_0 \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \dots X_k \alpha_k$. In $ComputeOrder(ADP(q \mid p_1, p_2, \dots, p_k), \omega)$ in Fig. 6, if $ADP(q \mid p_1, p_2, \dots, p_k)$ contains no cycles and ω is compatible with $DCG_{X0}(q)$, then 1) the introduction of the edges $\{a_i \to a_{i+1} \mid i = 1, \dots, m-1\}$ in the ComputeOrder function in Fig. 6 will not introduce cycles into $ADP(q \mid p_1, p_2, \dots, p_k)$ and 2) the resulting total order ψ is compatible with $DCG_{Xi}(p_i)$, for each right-hand-side nonterminal X_i of production q.*

Proof. Note that, in $ComputeOrder(ADP(q \mid p_1, p_2, \dots, p_k), \omega)$ in Fig. 6, the total order ω is represented by the set of edges $\{a_i \to a_{i+1} \mid i = 1, \dots, m-1\}$, where $\{a_i \mid i = 1, \dots, m\}$ are X_0's attribute occurrences. If the introduction of the edges $\{a_i \to a_{i+1} \mid i = 1, \dots, m-1\}$ does introduce a cycle into $ADP(q \mid p_1, p_2, \dots, p_k)$, ω is incompatible with the projection of $ADP(q \mid p_1, p_2, \dots, p_k)$ onto X_0's attribute occurrences, which is a subgraph of $DCG_{X0}(q)$. Hence, ω must be incompatible with $DCG_{X0}(q)$, which contradicts an assumption in the lemma. This proves the first assertion in the lemma.

The total order ψ computed by the *ComputeOrder* function is compatible with $ADP(q \mid p_1, p_2, \dots, p_k)$. Note that $DCG_{Xi}(p_i)$, for each right-hand-side nonterminal X_i,

is a subgraph of $ADP(q \mid p_1, p_2, \ldots, p_k)$. Hence, ψ is compatible with $DCG_{Xi}(p_i)$, for each right-hand-side nonterminal X_i. □

Theorem. *The ComputePlan algorithm in Fig. 6 correctly computes a set of evaluation orders for every production.*

Proof. It suffices to show that 1) every $ADP(q \mid p_1, p_2, \ldots, p_k)$ is acyclic and 2) in every call to

$$ComputeOrder(ADP(q \mid p_1, p_2, \ldots, p_k), \omega)$$

in Fig. 6, ω is compatible with $DCG_{X0}(q)$, where X_0 is the left-hand-side nonterminal of production q.

That every $ADP(q \mid p_1, p_2, \ldots, p_k)$ is acyclic is a direct consequence of the definition of $NC(1)$.

We will use an informal argument for the second assertion. We will prove the following invariant of the algorithm: If $(q, \omega) \in WL$, then ω is compatible with $DCG_{X0}(q)$. The $ComputePlan$ algorithm is an iterative method. The first ω considered is the arbitrarily chosen order μ of the attributes of the start symbol S. Since there are no dependencies among S's attribute occurrences, μ is trivially compatible with $DCG_S(q)$. The ω order considered in the later iterations of the $ComputePlan$ algorithm is the projection of ψ onto the attribute occurrences of the right-hand-side nonterminal X of production p, where p is the parent production of production q in syntax trees (note that X is a nonterminal on the right-hand side of production p and is the left-hand-side nonterminal of production q) and ψ is an evaluation order of attribute occurrences in production p. By the above lemma, this projection of ψ onto X's attributes is compatible with $DCG_X(q)$.

Due to 1) and 2) above, every evaluation order ψ computed by $ComputeOrder$ is a topological order compatible with the partial order $ADP(q \mid p_1, p_2, \ldots, p_k) \cup \omega$ and, hence, is a feasible evaluation for production q when the attribute occurrences of the left-hand-side nonterminal of q are evaluated in the ω order and $p_1, p_2, \ldots$, and p_k are the productions applied at the right-hand-side nonterminals of q.

Because the $ComputePlan$ algorithm is an exhaustive method, we claim that all productions q and all necessary orders ω are examined by the algorithm. The detailed proof of this claim should be obvious and is omitted for the sake of brevity. □

Example. For the example in Fig. 1, we need to compute the following seven ADP graphs: $ADP(P1 \mid P4, P2, P5)$, $ADP(P1 \mid P4, P3, P5)$, $ADP(P2)$, (which is the same as $DP(P2)$), $ADP(P3)$ (which is the same as $DP(P3)$), $ADP(P4)$ (which is the same as $DP(P4)$), $ADP(P5 \mid P2)$, and $ADP(P5 \mid P3)$. The start symbol has only one attribute $s0$. Choose $\mu = < S.s0 >$. For $ADP(P1 \mid P4, P2, P5)$ and μ, we may compute the evaluation order

$$\psi_1 = < Y.s3, X.i1, X.s1, Y.i2, Y.s2, Y.i3, Z.s4, S.s0 > .$$

For $ADP(P1 \mid P4, P3, P5)$ and μ, we may compute the evaluation order.

$$\psi_2 = < Y.s2, Y.i3, Y.s3, X.i1, X.s1, Y.i2, Z.s4, S.s0 > .$$

The projections of ψ_1 and ψ_2 onto X's attributes are $\omega_1 = < X.i1, X.s1 >$. For $ADP(P4)$ and ω_1, we may compute the evaluation order $\psi_3 = < X.i1, X.s1 >$.

The projections of ψ_1 and ψ_2 onto Y's attributes are

$$\omega_2 = < Y.s3, Y.i2, Y.s2, Y.i3 >$$

and

$$\omega_3 = < Y.s2, Y.i3, Y.s3, Y.i2 >,$$

respectively. For $ADP(P2)$ and ω_2, we may compute the evaluation order

$$\psi_4 = < Y.s3, Y.i2, Y.s2, Y.i3 > .$$

For $ADP(P3)$ and ω_3, we may compute the evaluation order $\psi_5 = < Y.s2,\ Y.i3,\ Y.s3,\ Y.i2 >$.

The projections of ψ_1 and ψ_2 onto Z's attribute is $\omega_4 = < Z.s4 >$. For $ADP(P5 \mid P2)$ and ω_4, we may compute the evaluation order

$$\psi_6 = < Y.s3, Y.i2, Y.s2, Y.i3, Z.s4 > .$$

For $ADP(P5 \mid P3)$ and ω_4, we may compute the evaluation order $\psi_7 = < Y.s2, Y.i3, Y.s3, Y.i2, Z.s4 >$.

The projections of ψ_6 and ψ_7 onto Y's attributes are ω_2 and ω_3, respectively. Since ω_2 and ω_3 are already processed, the work list becomes empty and, hence, the $ComputePlan$ algorithm terminates.

There are two evaluation orders ψ_1 and ψ_2 for production $P1$ and two evaluation orders ψ_6 and ψ_7 for production $P5$. There is only one evaluation order for each of the remaining productions, that is, ψ_4 for $P2$, ψ_5 for $P3$, and ψ_3 for $P4$.

There are other possible sets of evaluation orders depending on the choices of the topological sorting of a partial order in the $ComputeOrder$ function in Fig. 6. For instance, instead of the two evaluation orders ψ_6 and ψ_7 for production $P5$, we may use the following evaluation order $\psi_8 = < Y.i2,\ Y.s2,\ Y.i3,\ Y.s3,\ Z.s4 >$ for production $P5$. The projection of ψ_8 onto Y's attributes is $\omega_5 = < Y.i2,\ Y.s2, Y.i3, Y.s3 >$. For $ADP(P2)$ and ω_5 and for $ADP(P3)$ and ω_5, we may compute the evaluation order $\psi_9 = < Y.i2,\ Y.s2, Y.i3, Y.s3 >$. In this set of evaluation orders, the number of evaluation orders for $P5$ is reduced by one; however, the numbers of evaluation orders for $P2$ and $P3$ are increased by one each.

It is possible to reduce the number of evaluation plans by choosing an appropriate total order that is compatible with the partial order represented by the graph G in the $ComputeOrder$ function in Fig. 6. Techniques for similar issues are discussed in [13], [14].

To analyze the time complexity of the $ComputePlan$ algorithm, first note that the maximum number of potential evaluation orders of a symbol's attributes is $h!$, where h, as defined in Section 3, is the maximum number of attributes per symbol. Thus, the number of iterations of the outer *repeat* loop is at most $\mid P \mid h!$ since there are at most $\mid P \mid h!$ distinct tuples. For a fixed production q, the number of potential $ADP(q \mid p_1, p_2, \ldots, p_k)$ graphs is at most $\mid P \mid^l$, where l, as defined in Section 3, is the maximum number of nonterminal symbols on the right-hand side of a production. Thus, in each iteration of the outer *repeat* loop, there will be at most $\mid P \mid^l$ iterations of the middle *for* loop. Furthermore, in each iteration of the middle *for* loop, there will be at most l iterations of the

```
Algorithm: EvaluateAttributes
algorithm eval(T)
/* T is an unevaluated syntax tree. */
/* Choose a plan for each non-terminal node in T. */
/* μ is the evaluation order of the attributes of the start symbol chosen in ComputePlan (Figure 6). */
traverse(root of T, μ)
Use any traditional visit-oriented evaluator to evaluate attribute instances in T.

procedure traverse(n, ω)
/* n is a non-terminal node in the syntax tree. */
/* ω is the evaluation order of attributes of the left-hand-side non-terminal located at node n. */
q := the production applied at node n
Let m1, m2, ..., mk be the non-terminal child nodes of n in T.
Let p1, p2, ..., pk be the productions applied at m1, m2, ..., mk, respectively.
plan[n] := Γ[q, ω, p1, p2, ..., pk]
for each non-terminal child mi of n do
    traverse(mi, Θ[q, ω, i, p1, p2, ..., pk])
end for
end procedure traverse
```

Fig. 7. The *EvaluateAttribute* algorithm for the $NC(1)$ class.

inner *for* loop. The total number of iterations of the inner *for* loop is, thus, at most $\mid P \mid h! \mid P \mid^l l$. The *project* function simply removes unnecessary attributes in a linked list, which takes $O(h)$ time. Each iteration of the inner *for* loop takes $O(h)$ time. The *ComputeOrder* function needs to calculate a transitive closure in order to find a total order. Computing transitive closure takes $O(h^3(l+1)^3)$ time. The *ComputePlan* algorithm takes $O(\mid P \mid^{l+1} h!hl)$ for the inner *for* loop plus $O(\mid P \mid^{l+1} h!h^3(l+1)^3)$ for the cumulative *ComputeOrder* operations. Thus, the total time is $O(\mid P \mid^{l+1} h!h^3(l+1)^3)$. Note that no duplicate tuples (q, ω) will be inserted into the work list.

We may also analyze the space requirement as follows: There are $\mid P \mid^l$ *ADP* graphs. Each $ADP(q \mid p_1, p_2, \ldots, p_k)$ graph contains $O(h(l+1))$ attribute occurrences and can induce $O((h(l+1))!)$ evaluation orders. Each evaluation order needs $O(h(l+1))$ space. Thus, the total amount of space for plans is $O(\mid P \mid^l (h(l+1))!h(l+1))$. The space for storing the constraints ω is $O(\mid N \mid h!)$. The space for the Γ table is $O(\mid P \mid^{(l+1)} \mid N \mid h!)$ and the space for the Θ table is $O(\mid P \mid^{(l+1)} \mid N \mid h!l)$.

Every evaluation order for a production corresponds to an evaluation plan [16], [17]. An evaluation order for a production is a total order of all the attribute occurrences in the production. In this total order, we simply replace each consecutive block of the inherited attributes of the left-hand-side nonterminal of the production with a *visit-parent* operation, and replace each consecutive block of the synthesized attributes of the symbols on the right-hand side of the production with a *visit-child* operation, and replace the remaining attribute occurrences in the total order with corresponding *compute* operations. Thus, a plan is a sequence of *visit-parent*, *visit-child*, and *compute* operations. In this paper, the two terms *evaluation order for a production* and *evaluation plan for a production* are used as synonyms for each other. The evaluation plans are used by the attribute evaluator, which is discussed in the next section.

6 THE EVALUATOR FOR $NC(1)$

An input sentence is transformed into an attributed syntax tree based on the attribute grammar, in which all attribute instances are not evaluated yet. These attribute instances are evaluated according to the plans for the productions. There is a set of evaluation plans for each production. For each nonterminal node n in the syntax tree, a plan is chosen from the set of plans for the production that is applied at n based on the productions applied at the parent and child nodes of n. Once a plan is chosen for a nonterminal node in the syntax tree, it will be used throughout the evaluation process. Hence, a top-down traversal over the unevaluated syntax tree will select appropriate plans for the nonterminal nodes in the tree. Further evaluation may proceed as traditional visit-oriented evaluators [16], [17].

Fig. 7 is the evaluation algorithm for $NC(1)$. The evaluation algorithm first calls the *traverse* procedure to select a plan for each nonterminal node. The plan for the root node is $\Gamma[q, \mu, p_1, p_2, \ldots, p_k]$, where q is the topmost production instance (that is, the production instance applied at the root node) of the syntax tree, μ is the arbitrarily chosen order in Fig. 6, and $p_1, p_2, \ldots, p_k$ are the production instances applied at the nonterminal child nodes of the root. $\Gamma[q, \mu, p_1, p_2, \ldots, p_k]$ actually is, the evaluation order of all attribute occurrences in production q. This evaluation order corresponds to an evaluation plan. $\Theta[q, \omega, i, p_1, p_2, \ldots, p_k]$ is the projection of $\Gamma[q, \omega, p_1, p_2, \ldots, p_k]$ onto X_i's attribute occurrences in q. After the plan for the nonterminal node n is chosen, the *traverse* procedure selects a plan for each nonterminal child X_i of n by a recursive call to itself.

Though the *EvaluateAttributes* algorithm is presented in two passes, a top-down traversal for selecting plans and another for evaluation, the two passes can be fused together. It is for the sake of simplicity that the algorithm is presented in two passes.

The *traverse* procedure takes time proportional to the number of nodes in a syntax tree. A traditional, visit-oriented evaluator usually takes a comparable amount of time under the assumption that evaluating each attribute instance takes

a constant amount of time. The whole *EvaluateAttributes* algorithm takes time *O*(size of the syntax tree).

Example. Consider the syntax trees corresponding to the two derivations $S \rightarrow XYZ \rightarrow^* mmm$ and $S \rightarrow XYZ \rightarrow^* mnm$ using the grammar in Fig. 1. The evaluation order for the production instance $S \rightarrow XYZ$ in the first tree is ψ_1, whereas that for the second tree is ψ_2 (ψ_1 and ψ_2 are computed in the example in the previous section).

Correctness of the *EvaluateAttributes* algorithm is established by the Γ and Θ functions. For each production instance q in a syntax tree, let ω be the evaluation order of the attribute instances of the left-hand-side nonterminal of q and let $p_1, p_2, \ldots, p_k$ be the child production instances of q. Based on ω and $p_1, p_2, \ldots, p_k$, the Γ function selects an appropriate plan for the production instance q. Furthermore, the Θ function recalls the projection of the plan for q onto the attribute instances of each nonterminal on the right-hand side of q. Repeating the selection and projection operations, a plan is selected for every production instance in the syntax tree.

7 The NC Hierarchy

The $NC(1)$ class and the $ANCAG$ class are generalizations of Kastens's OAG class in that the DS graphs (dependence graphs for a symbol's attributes) used in Kastens's OAG algorithm is divided into the $Down$ graphs (used in this paper) and corresponding Up graphs (not defined nor used in this paper). The $Down$ graphs are further refined into the DCG graphs. The $Down$ graphs are used in the $ANCAG$ algorithm whereas the DCG graphs are used in the $NC(1)$ algorithm. The $Down$ graphs are more accurate estimations of downward transitive dependencies than the DS graphs; the DCG graphs are still more accurate. Hence, OAG is a subclass of $ANCAG$, which, in turn, is a subclass of $NC(1)$.

Compare the two graphs, which are used in computing plans for productions in $NC(1)$ and $ANCAG$, respectively:

$$ADP(q \mid p_1, p_2, \ldots, p_k) = DP(q) \cup DCG_{X1}(p_1) \cup DCG_{X2}(p_2) \cup \ldots \cup DCG_{Xk}(p_k)$$

$$IDP\text{-}ANCAG(q) = DP(q) \cup Down(X_1) \cup Down(X_2) \cup \ldots \cup Down(X_k).$$

The $Down$ graphs are replaced with the corresponding DCG graphs in the above definitions. The ADP graphs are more accurate than the *IDP-ANCAG* graphs in that the ADP graphs *look ahead* one generation of production instances (that is, the production instances $p_1, p_2, \ldots, p_k$ applied at the child nodes) due to the use of the DCG graphs. From this point of view, $NC(1)$ can be further generalized by looking ahead more generations of production instances.

From this look-ahead behavior, we may define a new family of attribute-grammar classes. The $NC(m)$ class (*noncircular attribute grammars with m-generation look-ahead*) consists of those attribute grammars for which a set of static evaluation plans for each production may be found by looking ahead at most m generations of production instances. The name NC signifies that all such grammars are noncircular. In particular, $NC(0)$ is the same as $ANCAG$; $NC(\infty)$ is the class of the well-defined attribute grammars. We may show that, for any finite m, $NC(m)$ is a proper subclass of $NC(m+1)$.

In order to define the $NC(m)$ class precisely, we need to extend the definitions of the downward characteristic graphs.

Definition. *Let X be a nonterminal and m be a nonnegative integer. An* $< X, m >$-phrase tree, *denoted by $\tau_{<X,m>}$, is a (potentially incomplete) derivation tree in which 1) the root is the nonterminal X, 2) the height of the tree (that is, the length of a longest path from the root to a leaf) is at most m and 3) the length of the path from the root to every nonterminal leaf is exactly m.*

Definition. *Let m be a nonnegative integer. Given a tree T, the* restriction of T to the topmost m levels, *denoted by $T_{<m>}$, is a subgraph of T obtained by removing from T all nodes below the mth level (by our convention, the root of a tree resides on level 0 and level $k+1$ is below level k). Thus, $T_{<1>}$ consists of the root and its children, which is essentially the production applied at the root.*

Definition. *Let m be a nonnegative integer. Let τ be an $< X, m >$-phrase tree. The* m-generation downward characteristic graph of nonterminal X with respect to τ, *denoted by $DCG_X(\tau)$, is a graph in which the nodes are X's attributes and the edges, say $X.a \rightarrow X.b$, denote a (transitive) dependency of $X.b$ on $X.a$ in some subtree T derived from X such that $T_{<m>}$ is identical to τ.*

$DCG_X(q)$ defined in Section 4 is a 1-generation downward characteristic graph of the nonterminal X with respect to q. $Down(X)$ is for the 0-generation case.

Equivalently, $DCG_X(\tau)$ may be defined as follows. Let m be a nonnegative integer. Let τ be an $< X, m >$-phrase tree. Let q be the production applied at the root of τ. We may write q as $X \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$. Let τ_i be an $< X_i, m >$-phrase tree, for all $i = 1, 2, \ldots, k$.

$$DCG_X(\tau) = \bigcup \{project(ADP_{<m>}(q \mid \tau_1, \tau_2, \ldots, \tau_k), \{X'\text{s attributes}\}) \mid (q \cup \tau_1 \cup \tau_2 \cup \ldots \cup \tau_k)_{<m>} = \tau\}$$

$$ADP_{<m>}(q \mid \tau_1, \tau_2, \ldots, \tau_k) = DP(q) \cup DCG_{X1}(\tau_1) \cup DCG_{X2}(\tau_2) \cup \ldots \cup DCG_{Xk}(\tau_k).$$

The notation $q \cup \tau_1 \cup \tau_2 \cup \ldots \cup \tau_k$ denotes a tree obtained by grafting the trees τ_is onto the corresponding X_is in the tree representing the production q. $ADP_{<m>}(q \mid \tau_1, \tau_2, \ldots, \tau_k)$ is called the *m-generation augmented dependency graph* of production q with the phrase trees $\tau_1, \tau_2, \ldots, \tau_k$. Furthermore, we define the set of all possible augmented dependency graphs of q as follows:

$$SADP_{<m>}(q) = \{ADP_{<m>}(q \mid \tau_1, \tau_2, \ldots, \tau_k) \mid \tau_i$$

is an $< X_i, m >$-phrase tree, for all $i = 1, 2, \ldots, k$ }.

Definition. *An attribute grammar G is an $NC(m)$ grammar if and only if, for all productions q in G, every graph in $SADP_{<m>}(q)$ is acyclic.*

The algorithms for generating plans and for evaluating attributes for the $NC(m)$ class are extensions of the algorithms in Figs. 6 and 7.

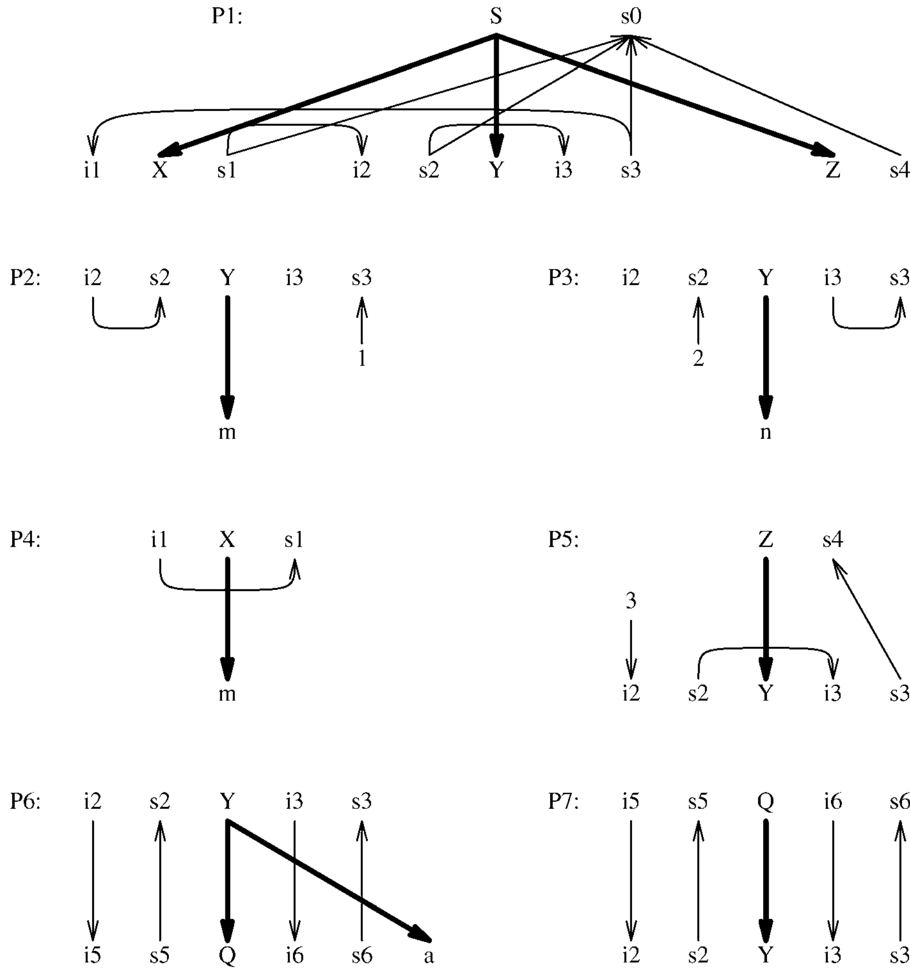


Fig. 8. The DP graphs of an attribute grammar. This grammar belongs to the $NC(\infty)$ class.

It is interesting to see how an evaluator can look ahead a potentially *infinite* number of generations of production instances for the $NC(\infty)$ class. In the next section, we will discuss the $NC(\infty)$ class.

7.1 Transforming $NC(m)$ Grammars to $NC(0)$ Grammars

The union of a production and appropriate phrase trees at the right-hand-side nonterminals, $q \cup \tau_1 \cup \tau_2 \cup \ldots \cup \tau_k$, that is used in the definition of the DCG graph can be viewed as an extended form of a production. From this point of view, an attribute grammar in the $NC(m)$ class is equivalent to an attribute grammar in the $NC(0)$ class. In this section, we show a method to transform an $NC(m)$ grammar, for any finite m, into an equivalent $NC(0)$ grammar.

Let tree T be the union $q \cup \tau_1 \cup \tau_2 \cup \ldots \cup \tau_k$. We first remove all the internal nodes of T, except the root, and make all the leaf nodes the children of the root (of course, the left-to-right order of the leaf nodes must be maintained). The resulting flat tree, thus, becomes a new production. The dependency relations among attribute occurrences of the new production are the transitive dependency relations among attributes of the root and the leaf nodes in T. The collection of all new productions forms a new grammar. Obviously, the new grammar is equivalent to the original grammar. Furthermore, the new grammar belongs to the $NC(0)$ class.

Though every $NC(m)$ grammar can be transformed into an equivalent $NC(0)$ grammar, the number of productions increases dramatically. The number of productions in the equivalent $NC(0)$ grammar could be $O(|P||N|^{lm})$, where $|P|$ is the number of productions, $|N|$ is the number of nonterminals, and l is the maximal length of a production in the $NC(m)$ grammar. The number of attribute occurrences per production increases similarly.

8 The $NC(\infty)$ Class

There are attribute grammars that cannot be evaluated with $NC(m)$, for any finite m. The example in Fig. 8, which is the same as the one in Fig. 1 with the addition of productions $P6$ and $P7$, does not belong to $NC(m)$, for any finite m. Fig. 9 is the DCG graphs for this example. The grammar is not $NC(m)$ for any finite m because the two nonterminals Y and Q are mutually recursive. A syntax tree may contain the derivations $Y \to Q \to Y$ for any number of repetitions. On the other hand, an $NC(m)$ evaluator is allowed to look ahead at most m generations down the syntax tree. Therefore, we observe that the grammar is not circular; it belongs to the class $NC(\infty)$ and can be evaluated with an evaluator similar to that for $NC(1)$.

The evaluator for the $NC(\infty)$ grammars needs to look ahead a *potentially* (not *actually*) infinite number of

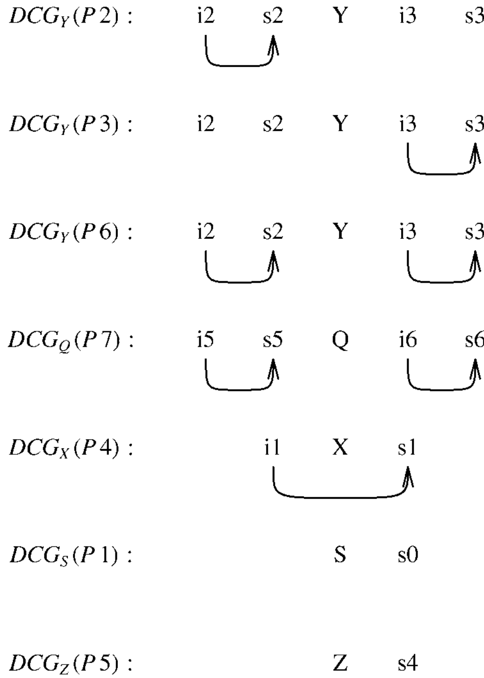


Fig. 9. The DCG graphs for the example in Fig. 8.

```
Algorithm: InfiniteLookAhead
/* Δ(X), for each symbol X, is the set of all */
/* possible dependency graphs of X's attributes. */
/* Initially, Δ(X) = ∅, for all symbols X. */
for each non-terminal X do
   Δ(X) := ∅
end for
repeat
   changed := false
   for each production q : X0 ⟶ α0 X1 α1 X2 α2 ... Xk αk do
      for each δ1 ∈ Δ(X1), δ2 ∈ Δ(X2), ..., δk ∈ Δ(Xk) do
         G := DP(q) ∪ δ1 ∪ δ2 ∪ ... ∪ δk
         δ := project(G, {attributes of X0})
         Λ[q | δ1, δ2, ..., δk ] := δ
         if δ ∉ Δ(X0) then
            changed := true
            Δ(X0) := Δ(X0) ∪ { δ }
         end if
      end for
   end for
until changed = false
```

Fig. 10. The $InfiniteLookAhead$ algorithm.

generations of descendants for the purpose of evaluation. In terms of a fixed syntax tree, this means that the evaluator needs to look ahead the syntax tree down to the leaves before it can choose an evaluation plan for a nonterminal node. To perform this infinite-look-ahead behavior, we find that it suffices to add a bottom-up traversal to the evaluator that records appropriate information (that is, $\Lambda[q \mid \delta_1, \delta_2, \ldots, \delta_k\ ]$ in the $mark$ procedure in Fig. 11, which is the actual downward transitive dependencies of each node's attribute instances) of the syntax tree in the nodes. Following this bottom-up traversal, a top-down traversal, similar to the $traverse$ procedure in Fig. 7, chooses plans for the nonterminal nodes. Finally, a traditional visit-oriented evaluator will perform the actual evaluation.

Each nonterminal X is associated with a set $\Delta(X)$ of possible dependency graphs among its attributes. Since every nonterminal has only a finite number of attributes, the set $\Delta(X)$ is finite. An element δ of $\Delta(X)$ denotes the actual downward transitive dependencies among X's attributes in a subtree whose root is the symbol X. δ will play the role of DCG in the plan generation for $NC(\infty)$. The $InfiniteLookAhead$ algorithm in Fig. 10 computes all possible downward transitive dependencies of the attributes of all nonterminals in a grammar. Initially, the set $\Delta(X)$ for every nonterminal X is an empty set. By examining the productions repeatedly, new members of $\Delta(X)$ for each nonterminal X may be found. The algorithm terminates when all possible downward transitive dependencies of all nonterminals are found. The $InfiniteLookAhead$ algorithm also constructs the $\Lambda[q \mid \delta_1, \delta_2, \ldots, \delta_k\ ]$ function (which is represented as an array). Let q be the production $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$. Let $\delta_i \in \Delta(X_i)$, for $i = 1, 2, \ldots, k$. $\Lambda[q \mid \delta_1, \delta_2, \ldots, \delta_k\ ]$ denotes the dependency relation (as a graph) among X_0's attributes if the production applied at X_0 is production q and the downward transitive dependency relations among X_i's attributes is δ_i for $i = 1, 2, \ldots, k$.

The $InfiniteLookAhead$ algorithm turns out to be essentially the same as Knuth's corrected algorithm [19] with one addition: It also computes $\Lambda[q \mid \delta_1, \delta_2, \ldots, \delta_k\ ]$, which is used in the evaluator for $NC(\infty)$. As far as we know, there are no static $NC(\infty)$ evaluators in the published literature.

To analyze the time complexity of the $InfiniteLookAhead$ algorithm, note that each symbol X has at most h attributes. Thus, $\mid \Delta(X) \mid = O(h!)$. Since there are $\mid N \mid$ nonterminals, the sum of all $\mid \Delta(X) \mid$ is $O(\mid N \mid h!)$. This implies that the *repeat* loop executes $O(\mid N \mid h!)$ iterations because at least one new δ must be added in each iteration. The outer *for* loop executes $\mid P \mid$ times in each iteration of the *repeat* loop. The inner *for* loop executes $O((h!)^l)$ times in each iteration of the outer *for* loop. The union operation $DP(q) \cup \delta_1 \cup \delta_2 \cup \ldots \cup \delta_k$ takes $O(l)$ time since $k \leq l$. The $project$ operation takes $O(h^3(l+1)^3)$ time since a transitive closure must be performed. Assuming all other operations take one unit time, the total time needed is $O(\mid N \mid\mid P \mid (h!)^{l+1} l^3 h^3)$.

To compute plans for the $NC(\infty)$ grammars, we need one definition.

Definition. *Let q be the production*

$$X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k.$$

Let $\delta_i \in \Delta(X_i)$, *for* $i = 1, 2, \ldots, k$. *Define the* look-ahead dependency graph of production q with respect to $\delta_1, \delta_2, \ldots, \delta_k$ *as follows:*

$$LDP(q \mid \delta_1, \delta_2, \ldots, \delta_k) = DP(q) \cup \delta_1 \cup \delta_2 \cup \ldots \cup \delta_k.$$

Similarly, we define the set of all possible look-ahead dependency graphs of q as follows:

$$SLDP(q) = \{LDP(q \mid \delta_1, \delta_2, \ldots, \delta_k) \mid \ \delta_i \in \Delta(X_i),$$

$$for\ i = 1, 2, \ldots, k\}.$$

```
Algorithm: EvaluateAttributes
algorithm eval(T)
/* T is an unevaluated syntax tree. */
/* Choose a plan for each non-terminal node in T. */
/* μ is the evaluation order of the attributes of the start symbol chosen in ComputePlan (Figure 6). */
mark(root of T)
traverse(root of T, μ)
Use any traditional visit-oriented evaluator to evaluate attribute instances in T.

procedure traverse(n, ω)
/* n is a non-terminal node in the syntax tree. */
/* ω is the evaluation order of attributes of the left-hand-side non-terminal located at node n. */
q := the production applied at node n
Let m1, m2, . . . , mk be the non-terminal child nodes of n in T.
Let δ1, δ2, . . . , δk be the marks associated with m1, m2, . . . , mk, respectively.
plan[n] := Γ[q, ω, δ1, δ2, . . . , δk]
for each non-terminal child mi of n do
    traverse(mi, Θ[q, ω, i, δ1, δ2, . . . , δk])
end for
end procedure traverse

procedure mark(n)
/* n is a non-terminal node in the syntax tree. */
q := the production applied at node n
Let m1, m2, . . . , mk be the non-terminal child nodes of n in T.
for each non-terminal child mi of n do
    mark(mi)
end for
Let δ1, δ2, . . . , δk be the marks associated with m1, m2, . . . , mk, respectively.
mark the node n with Λ[q | δ1, δ2, . . . , δk]
end procedure mark
```

Fig. 11. The $EvaluateAtrribute$ algorithm for the $NC(\infty)$ class.

To compute the plans, we use the $ComputePlan$ algorithm in Fig. 6 with five simple modifications:

1. $ADP(q \mid p_1, p_2, \ldots, p_k)$ is replaced with
$$LDP(q \mid \delta_1, \delta_2, \ldots, \delta_k);$$
2. $SADP(q)$ is replaced with $SLDP(q)$,
3. $\Gamma[q, \omega, p_1, p_2, \ldots, p_k]$ is replaced with
$$\Gamma[q, \omega, \delta_1, \delta_2, \ldots, \delta_k],$$
4. $\Theta[q, \omega, i, p_1, p_2, \ldots, p_k]$ is replaced with
$$\Theta[q, \omega, i, \delta_1, \delta_2, \ldots, \delta_k],$$
and
5. the work list WL is a set of constraints ω rather than a set of pairs (q, ω).

The $ComputePlan$ algorithm for the $NC(\infty)$ class also adopts a work-list approach. Initially, an arbitrary constraint μ is chosen for the attributes of the start symbol of the grammar. An evaluation order is computed for each $LDP(q \mid \delta_1, \delta_2, \ldots, \delta_k) \cup \omega$, where q is a production, $LDP(q \mid \delta_1, \delta_2, \ldots, \delta_k)$ is a look-ahead dependency graph of production q, and ω is a constraint on the evaluation order of the attributes of the left-hand-side nonterminal of production q. Such an evaluation order also imposes a constraint on the evaluation order of the attributes of each nonterminal on the right-hand side of q. Thus, new evaluation orders are generated from existing constraints and new constraints are generated from existing evaluation orders. This process terminates when no new evaluation orders are generated.

The analysis of the time complexity of the modified $ComputePlan$ algorithm is similar to that of the original $ComputePlan$ algorithm discussed in Section 3.

Example. The result of applying the $InfiniteLookAhead$ algorithm to the example in Fig. 8 is as follows: (A pair of square brackets $[\ldots]$ represents a dependence graph, which is also given a name such as δ_3.)

$\Delta(S) = \{\delta_1 = [s0]\}$.
$\Delta(Y) = \{\delta_2 = [i2 \rightarrow s2\ i3\ s3], \delta_3 = [i2\ s2\ i3 \rightarrow s3]\}$.
$\Delta(X) = \{\delta_4 = [i1 \rightarrow s1]\}$.
$\Delta(Z) = \{\delta_5 = [s4]\}$.
$\Delta(Q) = \{\delta_6 = [i5 \rightarrow s5\ i6\ s6], \delta_7 = [i5\ s5\ i6 \rightarrow s6]\}$.

The evaluation orders for the productions computed by the $ComputePlan$ algorithm are as follows: (A pair of angle brackets $< \ldots >$ represents a total order.) $\mu = < s >$.

$$\Gamma[P1, \mu, \delta_4, \delta_2, \delta_5] = < s3, i1, s1, i2, s2, i3, s4, s0 > .$$

Let $\omega_1 = < i1, s1 >$,
$$\omega_2 = < s3, i2, s2, i3 >,$$
and $\omega_3 = < s4 >$.

$$\Gamma[P1, \mu, \delta_4, \delta_3, \delta_5] \\ =< s2, i3, s3, i1, s1, i2, s4, s0 > .$$

Let $\omega_4 =< i1, s1 >$,

$$\omega_5 =< s2, i3, s3, i2 >,$$

and

$$\omega_6 =< s4 > .$$
$$\Gamma[P2, \omega_2] =< s3, i2, s2, i3 > .$$
$$\Gamma[P2, \omega_5] =< i2, s2, i3, s3, i2 >,$$

which contains a cycle.

$$\Gamma[P3, \omega_2] =< s3, i2, s2, i3, s3 >,$$

which contains a cycle.

$$\Gamma[P3, \omega_5] =< s2, i3, s3, i2 > .$$
$$\Gamma[P4, \omega_1] =< i1, s1 > .$$
$$\Gamma[P4, \omega_4] =< i1, s1 > .$$
$$\Gamma[P5, \omega_3, \delta_2] = \Gamma[P5, \omega_6, \delta_2] =< s3, i2, s2, i3, s4 > .$$
$$\Gamma[P5, \omega_6, \delta_3] = \Gamma[P5, \omega_6, \delta_3] =< s2, i3, s3, i2, s4 > .$$
$$\Gamma[P6, \omega_2, \delta_6] =< s6, s3, i2, i5, s5, s2, i3, i6 > .$$

Let

$$\omega_7 =< s6, i5, s5, i6 > .$$
$$\Gamma[P6, \omega_2, \delta_7] =< i3, i6, s6, s3, i2, i5, s5, s2, i3 >,$$

which contains a cycle.

$$\Gamma[P6, \omega_5, \delta_6] =< i1, i5, s5, i2, i3, i6, s6, s3, i1 >,$$

which contains a cycle.

$$\Gamma[P6, \omega_5, \delta_7] =< s5, s2, i3, i6, s6, s3, i2, i5 > .$$

Let

$$\omega_8 =< s5, i6, s6, i5 > .$$
$$\Gamma[P7, \omega_7, \delta_2] =< s3, s6, i5, i2, s2, s5, i6, i3 > .$$
$$\Gamma[P7, \omega_7, \delta_3] =< i5, i2, s2, s5, i6, i3, s3, s6, i5 >,$$

which contains a cycle.

$$\Gamma[P7, \omega_8, \delta_2] =< i5, i2, s2, s5, i6, i3, s3, s6, i5 >,$$

which contains a cycle.

$$\Gamma[P7, \omega_8, \delta_3] =< s2, s5, i6, i3, s3, s6, i5, i2 > .$$

Note that, in the *ComputePlan* algorithm for the $NC(1)$ class, the work list is a set of pairs (q, ω), which means that the constraint ω is applicable to production q. When the *ComputePlan* algorithm is generalized to $NC(m)$, the work list becomes a set of pairs (τ, ω), which means that the constraint ω is applicable to the phrase tree τ. When the *ComputePlan* algorithm is further generalized to $NC(\infty)$, the work list should be a set of pairs (T, ω), where T is a subtree with an appropriate root symbol, say X. However, there could be an infinite number of subtrees whose roots are X. The *ComputePlan* algorithm would not terminate. Thus, the work list for the $NC(\infty)$ case is a set of constraints ω. And we will assume that ω is applicable to all subtrees with root X.

Due to this modification, the *ComputePlan* algorithm may compute some spurious plans, that is, plans that will never be used in any evaluation. For instance, in the above example, a plan $\Gamma[P2, \omega_5]$ will never be used because the constraint ω_5 is generated by projecting the plan $\Gamma[P1, \mu, \delta_4, \delta_3, \delta_5]$ to Y's attributes. The plan $\Gamma[P1, \mu, \delta_4, \delta_3, \delta_5]$ is generated under the assumption that production $P3$ will be applied at the nonterminal Y. Thus, $P2$ will not be applied at Y. Hence, $\Gamma[P2, \omega_5]$ is spurious.

Sometimes, circular dependencies may occur in a spurious plan, for example, $\Gamma[P2, \omega_5]$ and $\Gamma[P3, \omega_2]$. Such spurious circular dependencies may be safely discarded.

Definition. *An attribute grammar belongs to the $NC(\infty)$ class if and only if, for every production q, every graph in the set $SLDP(q)$ is acyclic.*

We can easily prove the following theorem.

Theorem. *The $NC(\infty)$ class is exactly the same as the class of the well-defined attribute grammars.*

Proof. Suppose that the attribute grammar G is circular. There must exist a syntax tree T with circular dependencies. Due to the tree structure, the circular dependencies must be confined in a subtree of T. Let q be the production instance in T that is part of the circular dependencies and that is closest to the root of T. We may write q as $X_0 \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \dots X_k \alpha_k$. Let T_i be the subtree rooted at X_i, for $i = 1, 2, \dots, k$. Let δ_i be the transitive dependence graph of the attribute instances of X_i in the subtree T_i. Obviously, $\delta_i \in \Delta(X_i)$, for $i = 1, 2, \dots, k$, due to the exhaustive nature of the *InfiniteLookAhead* algorithm in Fig. 10. Thus, $DP(q) \cup \delta_1 \cup \delta_2 \cup \dots \cup \delta_k$ must be a member of $SLDP(q)$. However, $DP(q) \cup \delta_1 \cup \delta_2 \cup \dots \cup \delta_k$ contains circular dependencies. Thus, the grammar G does not belong to the $NC(\infty)$ class.

On the other hand, assume that G belongs to the $NC(\infty)$ class. Let T be any syntax tree derived from G and q be any production instance in T. We may write q as $X_0 \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \dots X_k \alpha_k$.

Let T_i be the subtree rooted at X_i, for $i = 1, 2, \dots, k$. Let δ_i be the transitive dependence graph of the attribute instances of X_i in the subtree T_i. Obviously, $\delta_i \in \Delta(X_i)$, for $i = 1, 2, \dots, k$, due to the exhaustive nature of the *InfiniteLookAhead* algorithm in Fig. 10. Thus, $DP(q) \cup \delta_1 \cup \delta_2 \cup \dots \cup \delta_k$ must be a member of $SLDP(q)$ and, hence, it must be acyclic. This implies that there are no circular dependencies in every syntax tree derived from G. Hence, G is a well-defined attribute grammar. □

The evaluator for $NC(\infty)$, which is a general, static, visit-oriented evaluator for all well-defined attribute grammars, is shown in Fig. 11. There are three steps in the evaluator: First, the *mark* procedure traverses the syntax tree from bottom up. It marks each nonterminal node in the syntax tree with the downward transitive dependencies derived from the subtree rooted at the node. Note that it is not necessary to compute the downward transitive dependencies in the *mark* procedure because all the necessary information is already encoded in the Λ function, which is built in the *InfiniteLookAhead* algorithm.

After all the nonterminal nodes are marked, the $traverse$ procedure traverses the syntax tree from the top down and chooses a plan for each nonterminal node. The $traverse$ procedure in Fig. 11 is identical to that in Fig. 7, except that the downward transitive dependency graphs of the nonterminals, δs, are used in the selection of plans. Finally, when a plan is chosen for each nonterminal node, an ordinary visit-oriented evaluator actually evaluates the attribute instances in the syntax tree. The extra cost of the $NC(\infty)$ evaluator is the bottom-up traversal (the $mark$ procedure), which takes time linear in the size of the syntax tree.

9 Conclusion and Related Work

We have identified a new classification, the NC hierarchy, of the well-defined attribute grammars. The classification is based on the look-ahead behaviors of the evaluators. Based on the generators and the evaluators for the NC classes, we have confirmed a result of Riis and Skyum, which says that all well-defined attribute grammars allows a (static) pure multivisit evaluator.

The evaluators for the NC classes are extensions of the visit-oriented evaluators. A well-known result of Deransart [4] showed that a well-defined attribute grammar can be transformed into an equivalent *l-ordered* AG (but with exponential increase in grammar size). In contrast, the approach taken in this paper does not attempt to transform the grammar. Rather, a set of plans is generated for every production. To transform a grammar and to add a selection of plans serve the same purpose: for evaluating well-defined attribute grammars. Engelfriet and Filè [6] also propose a method to transform $ANCAG$ to *l-ordered* grammars.

The $Down$ graph introduced in Section 3 has been used under different names in the literature [18], [8], [13], [9]. Variations of the $Down$ graphs have been used in the incremental updates to attributed trees [23] and in the computation of transitive dependencies in the linkage grammars [10].

There are many other classes of attribute grammars, such as *doubly noncircular* grammars [6], *m alternating-pass* grammars [11], *n-left-to-right-pass* grammars [2], *L-AG* [20], and *S-AG* [20]. All these classes of grammars are subsets of $NC(0)$ and can be evaluated efficiently.

Acknowledgments

The author wishes to thank Tom Reps for his helpful discussion on the material in this paper. The author also wishes to thank the three anonymous referees for their valuable detailed comments. This work was supported in part by National Science Council, Taiwan, R.O.C. under grants NSC 86-2213-E-009-021 and NSC 86-2213-E-009-079.

References

[1] H. Alblas, "Attribute Evaluation Methods," *Proc. Int'l Summer School (SAGA, '91),* vol. 545, pp. 48-113, June 1991.
[2] G.V. Bochmann, "Semantic Evaluation from Left to Right," *Comm. ACM,* vol. 19, no. 2, pp. 55-62, Feb. 1976.
[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithm.* Cambridge, Mass.: MIT Press, 1990.
[4] P. Deransart, "Validation des Grammaries D'attributs," PhD thesis, Univ. de Bordeaux I, Oct. 1984.
[5] P. Deransart, M. Jourdan, and B. Lorho, "Attribute Grammars: Definitions, Systems, and Bibliography," *Lecture Notes in Computer Science,* vol. 323, 1988.
[6] J. Engelfriet and G. Filè, "Simple Multivisit Attribute Grammars," *J. Computer and System Sciences,* vol. 24, pp. 283-314, 1982.
[7] J. Engelfriet, "Attribute Grammars: Attribute Evaluation Methods," *Methods and Tools for Compiler Construction,* B. Lorho, ed., pp. 103-138, Cambridge, England: Cambridge Univ. Press, 1984.
[8] G. Filè, "Classical and Incremental Evaluators for Attribute Grammars," *Computer Algebra and its Application to Physics, (CAAP '86),* vol. 214, 1986.
[9] P. Franchi and B. Courcelle, "Attribute Grammars and Primitive Recursive Schemes (I and II)," *Theoretical Computer Science,* vol. 17, 1982.
[10] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Trans. Programming Languages and Systems,* vol. 12, no. 1, pp. 26-60, Jan. 1990.
[11] M. Jazayeri and K.G. Walter, "Alternating Semantic Evaluation," *Proc. ACM Ann. Conf.* 1975.
[12] M. Jazayeri, W.F. Ogden, and W.C. Rounds, "The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars," *Comm. ACM,* vol. 18, no. 12, pp. 697-706, Dec. 1975.
[13] M. Jourdan, D. Parigot, and C. Juliè, "Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System," *Proc. SIGPLAN 90 Conf. Programming Language Design and Implementation,* vol. 25, no. 6, pp. 209-222, June 1990.
[14] M. Jourdan and D. Parigot, "Techniques for Improving Grammar Flow Analysis," *Proc. Third European Symp. Programming,* vol. 432, pp. 240-255, May 1990.
[15] U. Kastens, "Ordered Attribute Grammars," *Bericht,* No. 7/78, Univ. Karlsruhe, 1978.
[16] U. Kastens, "Ordered Attribute Grammars," *Acta Informatica,* vol. 13, pp. 229-256, 1980.
[17] U. Kastens, "Implementation of Visit-Oriented Attribute Evaluators," *Proc. Int'l Summer School SAGA,* vol. 545, pp. 114-139, June 1991.
[18] K. Kennedy and S. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars," *Conf. Record Third ACM Symp. Principles of Programming Languages,* pp. 32-49, Jan. 1976.
[19] D.E. Knuth, "Semantics of Context-Free Languages," *Math. System Theory,* vol. 5, no. 1, pp. 95-96, Mar. 1971.
[20] P.M. Lewis, D.J. Rosenkrantz, and R.E. Stearns, "Attributed Translations," *J. Computer and System Science,* vol. 9, pp. 279-307, 1974.
[21] H.R. Nielson, "Computation Sequences: A Way to Characterize Classes of Attribute Grammars," *Acta Informatica,* vol. 19, pp. 255-268, 1983.
[22] J. Paakki, "Attribute Grammar Paradigms–A High-Level Methodology in Language Implementation," *ACM Computing Surveys,* vol. 27, no. 2, pp. 196-255, June 1995.
[23] T. Reps, T. Teitelbaum, and A. Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Trans. Programming Languages and Systems,* vol. 5, no. 3, pp. 449-477, July 1983.
[24] H. Riis and S. Skyum, "*K*-Visit Attribute Grammars," *Math. Systems Theory,* vol. 15, pp. 17-28, 1981.
[25] W.M. Waite, "A Complete Specification of a Simple Compiler," Technical Report CU-CS-638-93, Computer Science Dept., Univ. of Colorado at Boulder, Jan. 1993.
[26] W. Yang and Y.-T. Liu, "SSCC: A Software Tool Based on Extended Ordered Attribute Grammars," *Proc. National Science Council (Republic of China), Part A: Physical Science and Engineering,* vol. 23, no. 1, pp. 85-99, 1998.

Wuu Yang received the BS degree in computer science from National Taiwan University in 1982 and the MS and PhD degrees in computer science from University of Wisconsin at Madison in 1987 and 1990, respectively. Currently, he is an associate professor at the National Chiao-Tung University, Taiwan, Republic of China. His research interests include programming languages and compilers, attribute grammars, and parallel and distributed computing. He is also very interested in the study of human languages and human intelligence.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dilb.