A finest partitioning algorithm for attribute grammars

Wuu Yang

Department of Computer and Information Science
National Chiao-Tung University
Hsin-Chu, Taiwan, R.O.C.
wuuyang@cis.nctu.edu.tw

Abstract

The attribute dependence graph of a syntax tree may be partitioned into disjoint regions. Attribute instances in different regions are independent of one other. The advantages of partitioning the attribute dependence graph include simplifying the attribute grammar conceptually and allowing the possibility of parallel evaluation. We present a static partitioning algorithm for attribute grammars. The algorithm builds the set of all feasible partitions for every production by analyzing the grammar. After the attributed syntax tree is constructed, one of the feasible partitions is chosen for each production instance in the syntax tree. Gluing together the selected partitions for individual production instances results in a partition of the attribute dependence graph of the syntax tree. No further merging or partitioning is needed at evaluation time. In addition to static partitioning, the algorithm always produces the finest partition of every attribute dependence graph.

1. Introduction

Since their introduction in 1968 [8], attribute grammars have attracted much research interest. Attribute grammars [10] are a concise and powerful formalism for specifying computations on context-free languages.

A main research focus on attribute grammars is designing algorithms for evaluating the attribute instances in a syntax tree. Traditionally, researchers attempt to design *sequential* evaluation algorithms. Due to the rapid progress in multi-processing hardware, it is interesting to design *parallel* evaluation algorithms. In many parallel evaluation methods, an attributed syntax tree or its attribute dependence graph is partitioned into blocks; the blocks are assigned to different processes for evaluation.

Partitioning may be based on nonterminals. Certain nonterminals may be designated as dividers. The instances of the dividers divide a syntax tree into non-overlapping blocks. Partitioning may also be based on productions in a similar fashion. Because dependence relations among attribute instances are ignored in these partitioning methods, processes need to make frequent communication in order to exchange attribute values. Communication slows down evaluation seriously.

If the attribute instances evaluated by different processes are independent, communication between processes can be completely eliminated. Therefore, partitioning based on attribute dependence relation is also an attractive method. There are generally two approaches in dependence-based partitioning. In the dynamic approach, partitioning is performed by the attribute evaluator after the attribute dependence graph of a syntax tree is constructed. In the static approach, partitioning is carried out by a grammar analyzer before the attribute dependence graph is constructed. There are also hybrid methods in which some kind of preliminary partitioning is carried out by gathering useful information about the underlying attribute grammar. When attribute instances in a syntax tree are evaluated, the attribute evaluator performs further merging or partitioning based on the preliminary information to produce the actual partitioning. In the hybrid methods, the overhead of partitioning is split among the grammar analyzer and the evaluator.

In this paper, we present a static partitioning algorithm. A production may appear more than once in a syntax tree. The attribute dependence graphs of different instances of the same production may be partitioned differently due to different surrounding environments in the syntax tree. The algorithm first computes all possible partitions of the attribute occurrences of individual productions and the environments under which the partitions are applied. After the syntax tree is constructed, the attribute evaluator selects a partition for each production instance in the syntax tree. A partition of the attribute dependence graph of the syntax tree is naturally induced from the partitions of individual production instances in the tree. The attribute evaluator does not need to perform further merging or partitioning.

In addition to dividing the attribute dependence graphs into disjoint regions, dependence-based partitioning algorithms should divide the attribute dependence graphs into as many regions as possible in order to increase the effectiveness of the parallel evaluation. The partitioning algorithm presented in this paper produces the *finest* partitioning of the attribute dependence graphs. This means that the result of partitioning exposes the maximal parallelism for evaluation in an attributed syntax tree.

This paper is concerned only with the partitioning algorithm. Visit-oriented parallel evaluation plans may also be generated by extending the algorithms presented in [12], as follows: A set of plans is generated for each production in the attribute grammar. One of the plans is selected for each production instance q in the syntax tree. The selection is based on the parent and child production instances of q in the syntax tree and the partitions applied at these production instances. The combination of the partitioning algorithm and the visit-oriented parallel evaluation algorithms, thus, form a basis of parallel evaluation system. The parallel evaluation system is applicable to all non-circular attribute grammars.

In Klein's parallel ordered attribute grammars [7], the dependence graphs of individual productions are divided into segments. In the attribute dependence graph of a syntax tree, segments from distinct production instances that share common attribute instances are glued together at evaluation time. Segments are not independent; therefore, communication among the evaluation processes may be required. Klaiber and Gokhale [6] add parallelism to Katayama's algorithm [5] by re-structuring the syntax tree so that evaluation in a list production is translated into a loop. Parallelism is, therefore, extracted from the loop. Reps's scan grammar [11] carries a similar nature. However, parallelism is explicitly expressed in a data-parallel statement within an attribute grammar. In Boehm's algorithm [2], a syntax tree, rather than the attribute dependence graph, is divided into disjoint regions. Each region is assigned to a process. Attribute instances in a region may be evaluated according to statically generated plans or according to dynamically computed dependencies. Jourdan surveyed various parallel attribute evaluation methods on various hardware architectures [3]. One of the static methods is based on the visit sequence. The *fork* operations and corresponding *join* operations are added in the visit-sequence-based plans to evaluate independent attribute occurrences in a production concurrently. Kuiper and Swierstra [9] defined tree-based distributors and attribute-based distributors that divide the syntax tree and the attribute dependence graph, respectively, into disjoint, connected regions. By contrast, the partitioning algorithm presented in this paper is a dependence-based approach. Zaring [13] proposes several synchronous and asynchronous parallel evaluation algorithms for attribute grammars. The asynchronous methods add appropriate *spawn* and *lock* operations to the statically generated evaluation plans in order to coordinate the parallel evaluation processes. The synchronous methods make use of *fork* (which is called *synchset* in his thesis) and implicit *join* operations in the evaluation plans to evaluate mutually independent attribute instances. In contrast to the above algorithms, the algorithm presented in this paper always assigns inter-dependent attribute instances to the same process for evaluation.

Jourdan [3] and Zaring [13] also address various incremental parallel evaluation methods for attribute grammars, which is not considered in this paper.

The remainder of this paper is organized as follows. In Section 2, we introduce the notations used in this paper. Section 3 presents the partitioning algorithm. The selection algorithm is presented in Section 4. In Section 5, we discuss the correctness and the finest partitioning properties of the algorithm. Section 6 concludes this paper.

2. Notations

In this section, we define the notations used in this paper. Basically, we adopt Kastens's notations [4]. 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 $X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \dots X_k \alpha_k$, where $X_0, X_1, X_2, \dots, X_k$ are nonterminal symbols and $\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_k$ 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 are 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 an attribution equation defining each synthesized attribute occurrence of the left-hand-side symbol and each inherited attribute occurrence of the right-hand-side symbols. An example attribute grammar is shown in Figure 1(a).

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 q. Figure 1(b) shows the DP graphs for the example grammar in Figure 1(a).

Definition. The *attribute dependence graph* of an attributed syntax tree is made up of a finite set of nodes and a finite set of directed edges in which every attribute instance of every symbol in the syntax tree is represented by a node and every dependency between two attribute instances is represented by an edge.

(a) An attribute grammar

P0: $S \to X$	P1: $X \to 1\ Y$	P2: $X \to 2\ Y$	P3: $Y \to 3$	P4: $Y \to 4$
$S.j\ :=\ X.b$	$Y.e\ :=\ X.a$	$Y.e\ :=\ X.a$	$Y.f\ :=\ Y.e$	$Y.h\ :=\ Y.e$
$S.k\ :=\ X.d$	$X.b\ :=\ Y.f$	$X.b\ :=\ Y.f$	$Y.h\ :=\ Y.g$	$Y.f\ :=\ Y.g$
$X.a\ :=\ 0$	$Y.g\ :=\ X.c$	$Y.g\ :=\ X.c$		
$X.c\ :=\ 1$	$X.d\ :=\ Y.h$	$X.d\ :=\ Y.h\ +\ Y.f$		

(b) The DP graphs

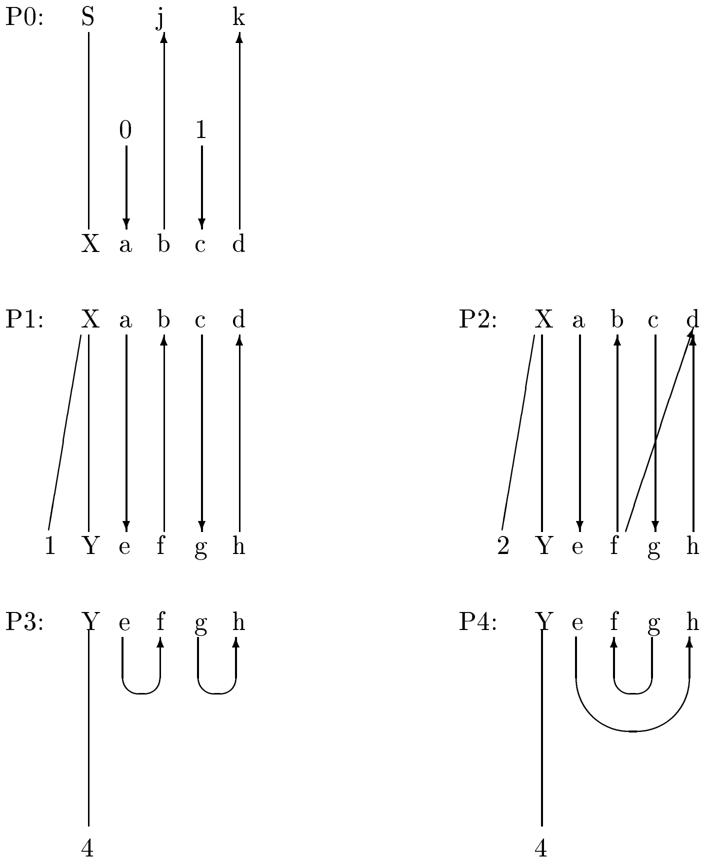

Figure 1: An attribute grammar and its DP.

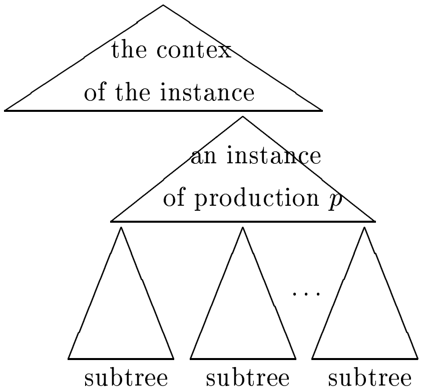

Figure 2: The context and the subtrees of an instance of a production.

Two syntax trees based on the grammar in Figure 1 and the corresponding attribute dependence graphs are shown in Figure 7.

3. The *PartitionAG* algorithm

An attribute dependence graph may be partitioned into disjoint regions so that there is no dependence relation among attribute instances in different regions. There are, in general, two approaches to partitioning an attribute dependence graph. We may partition the attribute dependence graph of a syntax tree *after* the attribute dependence graph has been built up. This is a *dynamic* approach. In a dynamic approach, the partitioning must be done on a per tree basis, which slows down the evaluation process. Alternatively, we may partition the dependence graphs of individual productions. When the attribute dependence graph of a syntax tree is built by gluing together the dependence graphs of individual productions, a partition of the attribute dependence graph is naturally induced from the partitions of the dependence graphs of individual productions. This is a *static* approach. The advantage of a static approach is that partitioning is performed on the attribute grammars, not on the attribute dependence graphs. The evaluator does not need to spend time on partitioning when evaluating the attribute instances in a syntax tree. In this section, we propose a static partitioning algorithm. A cooperating evaluation algorithm will be presented in the next section.

There are many criteria to judge a partition of an attribute dependence graph. From the point of view of parallel evaluation, we would prefer that attribute instances from different regions are independent and the partition is the *finest* one.

Definition. A *feasible partition* of an attribute dependence graph is a partition of the nodes of the attribute dependence graph that satisfies the following condition: Let $X.a$ and $Y.b$ be two attribute instances in the attribute dependence graph. If there is a dependence relation $X.a \rightarrow Y.b$ in the dependence graph, the two nodes $X.a$ and $Y.b$ are in the same partition block.

For instance, the partition $\{< a, b, e, f, j >, < c, d, g, h, k >\}$ is a feasible partition of the attribute instances in Figure 7(a) (the notation $< a, b, e, f, j >$ means that $< a, b, e, f, j >$ is a block of the partition). So is the partition $\{< a, b, e, f, j, c, d, g, h, k >\}$. Note that some partitions are not feasible. For instance, the partition $\{< a, b, e, f >, < c, d, g, h, j, k >\}$ is not a feasible one for the attribute dependence graph in Figure 7(a). There is a straightforward algorithm to partition a graph: the finest feasible partition of an attribute dependence graph is the reflexive, symmetric, transitive closure of the graph, which is an equivalence relation or, equivalently, a partition.

An attribute dependence graph is partitioned into independent regions by a feasible partition. Attribute in different regions can be evaluated by different processes on different processors concurrently. There is no need for synchronizing the evaluation processes.

Definition. Let π be a feasible partition of the attribute dependence graph of an attribute syntax tree T. Let X be an instance of a nonterminal in T (that is, an internal node in T). The projection of π onto X's attribute instances is a *feasible partition* of X's attributes.

A nonterminal X may appear in T more than once. Though different instances of X have the same attribute instances, their attribute instances may be partitioned in different ways. All these partitions of X's attributes are feasible partitions. For instance, in Figure 7(a), the set of attributes of nonterminal X is partitioned as $\{< a, b >, < c, d >\}$ whereas it is partitioned as $\{< a, b, c, d >\}$ in Figure 7(b). Both are feasible partitions. Note that some partitions are not feasible. For instance, consider the grammar in Figure 1. The partition $\{< a >, < b, c, d >\}$ of X's attributes is not feasible because it is not a partition of the attribute instances of any instance of X in any syntax tree.

Because a nonterminal can carry only a finite number of attributes, the number of feasible partitions of a nonterminal's attributes is also finite. The $PartitionAG$ algorithm in this section computes all the feasible partitions of a nonterminal's attributes. The evaluation algorithm in the next section attempts to find a feasible partition of the attribute dependence graph of a syntax tree by selecting a feasible partition for each production instance in the syntax tree and gluing together these feasible partitions.

Definition. Let π be a feasible partition of the attribute dependence graph of an attribute syntax tree T. Let q be an instance of a production in T. The projection of π onto q's attribute instances is a *feasible partition* of q's attribute occurrences.

Similarly, a production q may appear in T more than once. Though different instances of q have the same attribute instances, their attribute instances may be partitioned in different ways. All these partitions of q's attribute occurrences are feasible partitions. For instance, in Figure 7(a), the set of attribute instances of production $P0$ is partitioned as $\{< a, b, j >, < c, d, k >\}$ whereas it is partitioned as $\{< a, b, c, d, j, k >\}$ in Figure 7(b).

The $PartitionAG$ algorithm also computes all the feasible partitions of the attribute occurrences in every production. In computing the feasible partitions, we need to make use of certain refinements of the feasible partitions, which is defined as follows. Let X be a nonterminal node in a syntax tree T. Let T_X be the subtree of T rooted at X. Let G be the attribute dependence graph of T. The attribute dependence graph G_X corresponding to T_X is a sub-graph of G that contains a node for each attribute instance of symbols in T_X and all the edges of G of which both end points are attribute instances of symbols in T_X. Note that G_X does not contain the upward transitive dependencies (to be discussed later in this section) among X's attribute instances.

Definition. Let X be a nonterminal node in a syntax tree T. Let T_X be the subtree of T rooted at X. Let G_X be the attribute dependence graph corresponding to T_X. The partition based on the projection of G_X onto X's attributes is called a *plausible partition* of X's attributes.

Obviously, a plausible partition of X's attributes is a refinement of a feasible partition of X's attributes. For instance, from Figure 7(b), we know that $\{< e, h >, < f, g >\}$ is a plausible partition, which is a refinement of the feasible partition $\{< e, f, g, h >\}$. We may define the plausible partitions of attribute occurrences of a production in a similar way.

Definition. Let X be a nonterminal node in a syntax tree T. Let q be the production applied at X in T. Let T_X be the subtree of T rooted at X. Let G_X be the attribute dependence graph corresponding to T_X. The partition based on the projection of G_X onto the attribute occurrences of q is called a *plausible partition* of the attribute occurrences of q.

Similarly, a plausible partition of the attribute occurrences of a production is a refinement of a feasible partition of the attribute occurrences of the production.

Algorithm: *PartitionAG*
/* $\Xi(X)$, for each nonterminal X, is the set of all plausible */
/* partitions of X's attributes due to the subtrees rooted at X. */
/* $\Theta(X)$, for each nonterminal X, is the set of all feasible */
/* partitions of X's attributes due to X's position in a syntax tree. */
for each nonterminal X **do**
    $\Xi(X) := \emptyset$; $\Theta(X) := \emptyset$;
end for
for each production q **do**
    $\pi_q :=$ the finest partition of attribute occurrences of q based solely on $DP(q)$
end for
repeat /* bottom-up pass */
    $changed := false$
    for each production $q : X_0 \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$ **do**
        for each $\sigma_1 \in \Xi(X_1)$, $\sigma_2 \in \Xi(X_2)$, ..., $\sigma_k \in \Xi(X_k)$ **do**
            $\pi := merge(\ldots merge(merge(\pi_q, \sigma_1), \sigma_2), \ldots, \sigma_k)$
            $\sigma := project(\pi, \{X_0\text{'s attributes }\})$
            $\Sigma(q|\sigma_1, \sigma_2, \ldots, \sigma_k) := \sigma$
            if σ does not belong to $\Xi(X_0)$ **then**
                $changed := true$
                $\Xi(X_0) := \Xi(X_0) \cup \{\sigma\}$
            end if
        end for
    end for
until $changed = false$
/* top-down pass */
$\Theta(S) := \Xi(S)$, where S is the start symbol of the grammar
repeat
    $changed := false$
    for each production $q : X_0 \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$ **do**
        for each $\sigma_0 \in \Theta(X_0)$, $\sigma_1 \in \Xi(X_1)$, $\sigma_2 \in \Xi(X_2)$, ..., $\sigma_k \in \Xi(X_k)$ **do**
            if $\Pi(q|\sigma_0, \sigma_1, \ldots, \sigma_k)$ has not been defined yet **then**
                $\pi := merge(\ldots merge(merge(\pi_q, \sigma_0), \sigma_1), \ldots \sigma_k)$
                $\Pi(q|\sigma_0, \sigma_1, \ldots, \sigma_k) := \pi$
                for $i := 1$ **to** k **do**
                    $\sigma := project(\pi, \{X_i\text{'s attributes }\})$
                    $\Phi(q, \pi, i) := \sigma$
                    if σ does not belong to $\Theta(X_i)$ **then**
                        $changed := true$
                        $\Theta(X_i) := \Theta(X_i) \cup \{\sigma\}$
                    end if
                end for
            end if
        end for
    end for
until $changed = false$

Figure 3: The *PartitionAG* algorithm.

```
function project(π, V) return a new partition
/* π is a partition. V is a set of attribute occurrences. */
/* The result is the finest partition of V that is consistent with π. */
for each block β of π do
   for each element t of β do
      if t does not belong to V then β := β − {t} end if
   end for
   if β = ∅ then π := π − {β} end if
end for
return π
end function project

function merge(π, σ) return a new partition
/* π and σ are partitions of the attributes of a production or a nonterminal. */
/* merge finds the finest partition that is consistent with both π and σ. */
for each block γ of σ do
   β := ∅
   for every element t of γ do
      if t does not belong to β then
         δ = the block in π that contains element t
         π := π − {δ}
         β := β ∪ δ
      end if
   end for
   π := π ∪ {β}
end for
return π
end function merge
```

Figure 4: The *project* and *merge* functions.

In the rest of this paper, we will use the Greek letter σ to denote a partition of a nonterminal's attributes and the Greek letter π to denote a partition of the attribute occurrences in a production.

A syntax tree is made up of many instances of productions. An instance of a production in a syntax tree is surrounded by the subtrees rooted at the nonterminals on the right-hand side of the production and the *context* to which the instance of the production is adjoined. Figure 2 shows a syntax tree that contains an instance of production p. The three subtrees and the context of the instance of production p are also annotated in the figure.

In order to partition the dependence graph of a production p, three kinds of dependencies must be taken into consideration: (1) the dependencies due to attribution equations in production p; (2) the dependencies due to the context of an instance of p in a syntax tree; and (3) the dependencies due to the subtrees rooted at the nonterminals on the right-hand side of production p. The first kind of dependence, which is called the *base dependence*, occurs among attribute occurrences in p. The second kind of dependence, which is called the *upward transitive dependence*, occurs among attribute occurrences of the left-hand-side nonterminal. The third kind of dependence, which is called the *downward transitive dependence*, occurs among attribute occurrences of individual nonterminals on the right-hand side of p. Different instances of production p in a syntax tree carry the same base dependencies but possibly different upward and downward transitive dependencies. For this reason, a production may have more than one feasible partition on its attribute occurrences. A key observation of the *PartitionAG* algorithm is that the upward and downward transitive dependencies of an instance of a production are independent. (This is due to the fact that the context and the subtrees of a production instance do not overlap.) The two kinds of transitive dependencies may be computed separately.

Definition. The *base partition* π_q, for each production q, is the partition of attribute occurrences in q based solely on the dependences induced by the attribution equations in production q.

The base partition π_q may be computed from the dependence graph $DP(q)$ of production q. Note that the base partition π_q must be a refinement of every feasible and every plausible partition of q's attribute instances.

The *PartitionAG* algorithm in Figure 3 computes all the feasible partitions of the attribute occurrences of the productions. The algorithm consists of two passes. Each pass is a *repeat* loop that examines the productions repeatedly. In the first pass, *PartitionAG* computes all the plausible partitions of the attributes of individual nonterminals. In the second pass, the algorithm finds all feasible partitions of the attribute occurrences in individual productions.

In the first pass, the algorithm considers both the base dependencies and the downward transitive dependencies. For each production $q : X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$, the *PartitionAG* algorithm determines a plausible partition of X_0's attributes by merging the base partition of q and a plausible partition of X_i's attributes, where $i = 1, 2, \ldots, k$, and then projecting the resultant partition onto X_0's attributes. The feasible partitions computed during the first pass are stored as the $\Sigma(q|\sigma_1, \sigma_2, \ldots, \sigma_k)$ function, which maps a combination of plausible partitions of X_i's attributes, where $i = 1, 2, \ldots, k$, to a plausible partition of X_0's attributes.

All the plausible partitions of the attributes of nonterminals are also collected in the Ξ function. When the *repeat* loop of the first pass completes, $\Xi(S)$, where S is the start symbol of the grammar, is a set of plausible partitions of the attributes of S.

Because we assume that the start symbol S of the grammar does not appear on the right-hand side of any production, S is always located on the root of a syntax tree. Note that there is no upward transitive dependence among the attributes of the start symbol. By the definition of plausible and feasible partitions, a plausible partition of the attributes of S is also a feasible partition of the attribute of S. Therefore, $\Xi(S)$ is also the set of feasible partitions of the attributes of S. The *PartitionAG* algorithm uses the Θ function to collect feasible partitions of attributes of nonterminals. Hence, we have $\Theta(S) = \Xi(S)$.

In the second pass, the algorithm considers all three kinds of dependencies. For each production $q : X_0 \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$, the *PartitionAG* algorithm determines a feasible partition of the attribute occurrences in production q by merging the base partition of q, a feasible partition of X_0's attributes (called σ_0), and a plausible partition of X_i's attributes (called σ_i), where $i = 1, 2, \ldots, k$. The result of merging, called π, is a feasible partition of the attribute occurrences of production q. The partition π is stored as $\Pi(q|\sigma_0, \sigma_1, \ldots, \sigma_k)$. The Π function will be used in the evaluation of attributes.

In addition to constructing the Π function in the second pass, the *PartitionAG* algorithm also finds feasible partitions of attributes of nonterminals. Note that the projection of π onto the attributes of X_i, for $i = 1, 2, \ldots, k$, is a feasible partition of the attributes of X_i. This feasible partition is stored as $\Phi(q, \pi, i)$. When a new feasible partition of the attributes of a nonterminal, say X, is found, it is used to compute new feasible partitions of the attribute occurrences of productions whose left-hand sides are the nonterminal X. The *repeat* loop of the second pass completes when no more new feasible partitions of attributes of nonterminals can be found.

There is a slight ambiguity in the above discussion. In the Σ and Π functions, k denotes the number of nonterminals on the right-hand side of a production. Obviously, k is probably different for different productions. This ambiguity can be remedied by adding an extra level of subscripts. We determined to leave the ambiguity in the algorithm in order to simplify the presentation of the

the Σ function	the Π function	the Φ function	the Ξ and Θ tables
$\Sigma(P0\|\sigma_3) = \sigma_6$	$\Pi(P0\|\sigma_6, \sigma_3) = \pi_1$	$\Phi(P0, \pi_1, 1) = \sigma_3$	$\Xi(S) = \{\sigma_6, \sigma_7\}$
$\Sigma(P0\|\sigma_4) = \sigma_6$	$\Pi(P0\|\sigma_6, \sigma_4) = \pi_2$	$\Phi(P0, \pi_2, 1) = \sigma_4$	$\Xi(X) = \{\sigma_3, \sigma_4, \sigma_5\}$
$\Sigma(P0\|\sigma_5) = \sigma_7$	$\Pi(P0\|\sigma_6, \sigma_5) = \pi_3$	$\Phi(P0, \pi_3, 1) = \sigma_5$	$\Xi(Y) = \{\sigma_1, \sigma_2\}$
	$\Pi(P0\|\sigma_7, \sigma_3) = \pi_3$		$\Theta(S) = \{\sigma_6, \sigma_7\}$
	$\Pi(P0\|\sigma_7, \sigma_4) = \pi_3$		$\Theta(X) = \{\sigma_3, \sigma_4, \sigma_5\}$
	$\Pi(P0\|\sigma_7, \sigma_5) = \pi_3$		$\Theta(Y) = \{\sigma_1, \sigma_2, \sigma_8\}$
$\Sigma(P1\|\sigma_1) = \sigma_3$	$\Pi(P1\|\sigma_3, \sigma_1) = \pi_4$	$\Phi(P1, \pi_4, 1) = \sigma_1$	
$\Sigma(P1\|\sigma_2) = \sigma_4$	$\Pi(P1\|\sigma_3, \sigma_2) = \pi_5$	$\Phi(P1, \pi_5, 1) = \sigma_8$	
	$\Pi(P1\|\sigma_4, \sigma_1) = \pi_5$	$\Phi(P1, \pi_6, 1) = \sigma_2$	
	$\Pi(P1\|\sigma_4, \sigma_2) = \pi_6$		
	$\Pi(P1\|\sigma_5, \sigma_1) = \pi_5$		
	$\Pi(P1\|\sigma_5, \sigma_2) = \pi_5$		
$\Sigma(P2\|\sigma_1) = \sigma_5$	$\Pi(P2\|\sigma_3, \sigma_1) = \pi_5$	$\Phi(P2, \pi_5, 1) = \sigma_8$	
$\Sigma(P2\|\sigma_2) = \sigma_5$	$\Pi(P2\|\sigma_3, \sigma_2) = \pi_5$		
	$\Pi(P2\|\sigma_4, \sigma_1) = \pi_5$		
	$\Pi(P2\|\sigma_4, \sigma_2) = \pi_5$		
	$\Pi(P2\|\sigma_5, \sigma_1) = \pi_5$		
	$\Pi(P2\|\sigma_5, \sigma_2) = \pi_5$		
$\Sigma(P3) = \sigma_1$	$\Pi(P3\|\sigma_1) = \pi_7$		
	$\Pi(P3\|\sigma_2) = \pi_8$		
	$\Pi(P3\|\sigma_8) = \pi_8$		
$\Sigma(P4) = \sigma_2$	$\Pi(P4\|\sigma_1) = \pi_8$		
	$\Pi(P4\|\sigma_2) = \pi_9$		
	$\Pi(P4\|\sigma_8) = \pi_8$		

Figure 5: The Π and Φ functions for the example in Figure 1.

PartitionAG algorithm.

The *PartitionAG* algorithm may be applied at the same time when evaluation plans are generated.

Example. We will use the example in Figure 1 to illustrate the *PartitionAG* algorithm. There are three nonterminals S, X, and Y in the grammar. Initially, $\Xi(S) = \Xi(X) = \Xi(Y) = \Theta(S) = \Theta(X) = \Theta(Y) = \emptyset$. The base partitions of the productions are as follows: $\pi_{P0} = \{< a >, < b, j >, < c >, < d, k >\}$, $\pi_{P1} = \{< a, e >, < b, f >, < c, g >, < d, h >\}$, $\pi_{P2} = \{< a, e >, < b, d, f, h >, < c, g >\}$, $\pi_{P3} = \{< e, f >, < g, h >\}$, and $\pi_{P4} = \{< e, h >, < f, g >\}$.

In the first iteration of the first *repeat* loop, the algorithm considers only productions $P3$ and $P4$ because the right-hand-sides of the two productions do not contain any nonterminal symbols. Let $\sigma_1 = \pi_{P3}$ and $\sigma_2 = \pi_{P4}$. At the end of the first iteration of the first *repeat* loop, we have $\Sigma(P3) = \sigma_1$, $\Sigma(P4) = \sigma_2$, and $\Xi(Y) = \{\sigma_1, \sigma_2\}$. Note that there are two partitions of Y's attributes, that is, σ_1 and σ_2.

During the second iteration of the first *repeat* loop, the algorithm considers the four productions: $P1$, $P2$, $P3$, and $P4$. Let $\sigma_3 = \{< a, b >, < c, d >\}$, $\sigma_4 = \{< a, d >, < b, c >\}$, and $\sigma_5 = \{< a, b, c, d >\}$. At the end of the second iteration of the first *repeat* loop, we have $\Sigma(P1|\sigma_1) = \sigma_3$, $\Sigma(P1|\sigma_2) = \sigma_4$, $\Sigma(P2|\sigma_1) = \sigma_5$, $\Sigma(P2|\sigma_2) = \sigma_5$, and $\Xi(X) = \{\sigma_3, \sigma_4, \sigma_5\}$. Note that there are three partitions of X's attributes, that is, σ_3, σ_4, and σ_5. There is no change concerning productions $P3$ and $P4$ and the nonterminal Y.

During the third iteration of the first *repeat* loop, the algorithm considers all five productions. Let $\sigma_6 = \{< j >, < k >\}$, and $\sigma_7 = \{< j, k >\}$. At the end of the third iteration of the first *repeat* loop, we have $\Sigma(P0|\sigma_3) = \sigma_6$, $\Sigma(P0|\sigma_4) = \sigma_6$, $\Sigma(P0|\sigma_5) = \sigma_7$, and $\Xi(S) = \{\sigma_6, \sigma_7\}$. There is no change concerning productions $P1$, $P2$, $P3$, and $P4$ and the nonterminals X and Y.

After three iterations, no more change will occur and hence the first *repeat* loop terminates. At the beginning of the second *repeat* loop, $\Theta(S) = \Xi(S)$.

In the first iteration of the second *repeat* loop, the algorithm first considers production $P0$. Let $\pi_1 = \{< a, b, j >, < c, d, k >\}$. Because $merge(merge(\pi_{P0}, \sigma_6), \sigma_3) = \pi_1$, $\Pi(P0|\sigma_6, \sigma_3) = \pi_1$. Because $project(\pi_1, X\text{'s attributes}) = \{< a, b >, < c, d >\} = \sigma_3$, $\Phi(P0, \pi_1, 1) = \sigma_3$. Similarly, we may compute the Π and Φ functions. The various functions are summarized in Figure 5. In Figure 5, $\pi_1 = \{< a, b, j >, < c, d, k >\}$, $\pi_2 = \{< a, d, k >, < b, c, j >\}$, $\pi_3 = \{< a, b, c, d, j, k >\}$, $\pi_4 = \{< a, b, e, f >, < c, d, g, h >\}$, $\pi_5 = \{< a, b, c, d, e, f, g, h >\}$, $\sigma_8 = \{< e, f, g, h >\}$, $\pi_6 = \{< a, d, e, h >, < b, c, f, g >\}$, $\pi_7 = \{< e, f >, < g, h >\} = \sigma_1$, and $\pi_8 = \{< e, f, g, h >\} = \sigma_8$. □

4. Selecting feasible partitions

The *PartitionAG* algorithm in the previous section computes all feasible partitions of the attribute occurrences of productions. After a syntax tree is built up by the parser, the *EvaluateAttributes* algorithm in Figure 6 chooses a feasible partition for every production instance in the syntax tree. The *EvaluateAttributes* algorithm consists of two passes. During the first pass, the algorithm traverses the syntax tree bottom up. A plausible partition of the attribute instances of every nonterminal node X is chosen. The choice is based on the plausible partitions of the attribute instances of the nonterminal children of X. The necessary information for selecting plausible partitions is encoded in the Σ function, which is constructed by the *PartitionAG* algorithm. At the end of the first pass, a plausible partition of the attribute instances of the root of the syntax tree has been chosen.

Because the context of the root node is a (trivial) empty graph, no additional constraints will be imposed by the context of the root. Therefore, the plausible partition of the attribute instances of the root is also a feasible partition.

During the second pass, which is a top-down traversal of the syntax tree, the *EvaluateAttributes* algorithm makes use of the Π function to select a feasible partition for each production instance in the syntax tree. For each production instance, the selection is based on the feasible partition of the attribute instances of the left-hand-side nonterminal and the plausible partitions of the attribute instances of the right-hand-side nonterminals. After a feasible partition for a production instance is selected, the *EvaluateAttributes* algorithm makes use of the Φ function to select a feasible partition of the attribute instances for every nonterminal on the right-hand side of the production. The selected feasible partition of a nonterminal's attributes will be used to determine a feasible partition of the child production.

After the second pass completes, a feasible partition has been selected for every production instance in the syntax tree. Gluing together these feasible partitions will result in a feasible partition of the whole syntax tree. Note that no merging or partitioning is needed during evaluation.

After the attribute dependence graph of the syntax tree is partitioned into independent regions, the attribute instances in different regions may be evaluated concurrently, possibly by independent evaluators.

Example. Figure 7 shows two syntax trees based on the attribute grammar in Figure 1. The first syntax tree is made up of three productions: $P0$, $P1$, and $P3$. In the bottom-up pass of the *EvaluateAttributes* algorithm, a plausible partition is chosen for each nonterminal node. The partition chosen for the Y node is $\Sigma(P3)$, which is σ_1; the partition chosen for the X node is $\Sigma(P1|\sigma_1)$, which is σ_3; and the partition chosen for the S node is $\Sigma(P0|\sigma_3)$, which is σ_6. During the top-down pass, the *EvaluateAttributes* algorithm chooses a partition for each production instance in the syntax tree.

Algorithm: *EvaluateAttributes*
/* T is an unevaluated syntax tree. */
BottomUpChoose(root of T)
$\sigma :=$ the partition chosen for the root of T
TopDownChoose(root of T, σ)
Use any traditional visit-oriented evaluator for each partitioning block of T.

procedure *BottomUPChoose*(n)
/* n is a nonterminal node in the syntax tree T. */
/* The *BottomUpChoose* procedure chooses a partition for each production instance in T. */
$q :=$ the production applied at node n
Let $m_1, m_2, \ldots, m_k$ be the nonterminal child nodes of n in T.
for each nonterminal child m_i of n **do**
    BottomUpChoose(m_i)
end for
Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be the partitions chosen for nodes $m_1, m_2, \ldots, m_k$, respectively.
choose the partition $\Sigma(q|\sigma_1, \sigma_2, \ldots, \sigma_k)$ for node n.
end *BottomUpChoose*

procedure *TopDownChoose*(n, σ)
$q :=$ the production applied at node n
Let $m_1, m_2, \ldots, m_k$ be the nonterminal child nodes of n in T.
Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be the partitions chosen for nodes $m_1, m_2, \ldots, m_k$, respectively.
$\pi := \Pi(q|\sigma, \sigma_1, \sigma_2, \ldots, \sigma_k)$
choose the partition π for production q.
for each nonterminal child m_i of n **do**
    TopDownChoose($m_i, \Phi(q, \pi, i)$)
end for
end *TopDownChoose*

Figure 6: The *EvaluateAttributes* algorithm.

The partition chosen for $P0$ is $\Pi(P0|\sigma_6, \sigma_3)$, which is π_1. Because $\Phi(P0, \pi_1, 1) = \sigma_3$, the partition chosen for $P1$ is $\Pi(P1|\sigma_3, \sigma_1)$, which is π_4. Because $\Phi(P1, \pi_4, 1) = \sigma_1$, the partition chosen for $P3$ is $\Pi(P3|\sigma_1)$, which is π_7; The attribute instances in the syntax tree is thus partitioned into two independent regions by π_1, π_4, and π_7: $\{< j, a, b, e, f >, < k, c, d, g, h >\}$.

The second syntax tree is made up of three productions: $P0$, $P2$, and $P4$. In the bottom-up pass of the *EvaluateAttributes* algorithm, a partition is chosen for each nonterminal node. The partition chosen for the Y node is $\Sigma(P4)$, which is σ_2; the partition chosen for the X node is $\Sigma(P2|\sigma_2)$, which is σ_5; and the partition chosen for the S node is $\Sigma(P0|\sigma_5)$, which is σ_7. During the top-down pass, the *EvaluateAttributes* algorithm chooses a partition for each production instance in the syntax tree. The partition chosen for $P0$ is $\Pi(P0|\sigma_7, \sigma_5)$, which is π_3. Because $\Phi(P0, \pi_3, 1) = \sigma_5$, the partition chosen for $P1$ is $\Pi(P1|\sigma_5, \sigma_2)$, which is π_5. Because $\Phi(P1, \pi_5, 1) = \sigma_8$, the partition chosen for $P3$ is $\Pi(P3|\sigma_8)$, which is π_8; The attribute instances in the syntax tree is thus partitioned into a single region by π_3, π_5, and π_8: $\{< j, k, a, b, c, d, e, f, g, h >\}$. □

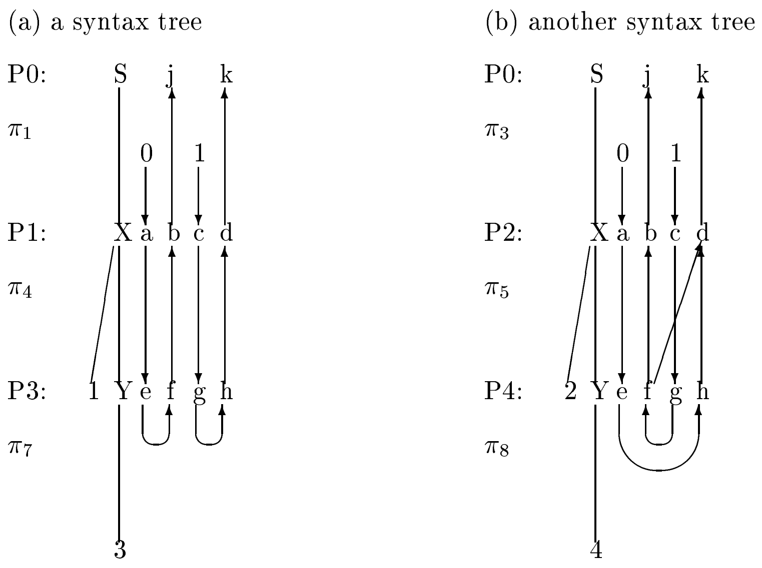

Figure 7: Two syntax trees.

5. The finest partition property

There may be more than one feasible partition on the attribute dependence graph of a syntax tree. These feasible partitions may be organized as a lattice by the refinement relation. In this section, we show that the partition found by the *PartitionAG* and *EvaluateAttributes* algorithms is the *finest* one, that is, it is the refinement of all feasible partitions. We will need a new definition.

Definition. Let G be the attribute dependence graph of a syntax tree and a and b be two attribute instances (that is, two nodes) in G. We say that a and b are *related* in G, denoted by $a \simeq b$, if and only if one of the following four conditions holds: (1) a and b are the same attribute instance; (2) a is dependent on b in G; (3) b is dependent on a in G; or (4) there is an attribute instance c in G such that c is related to both a and b.

In short, the $\simeq$ relation is the reflexive, symmetric, and transitive closure of the *dependence* relation. The following theorem is an immediate corollary to the above definition.

Theorem. Let G be the attribute dependence graph of a syntax tree. A partition π on G is a feasible partition if and only if related attribute instances are in the same partition block in π.

We need to prove the following theorem.

Theorem. Let G be the attribute dependence graph of the syntax tree T. Let a and b be two attribute instances (nodes) in G. Let π be the partition of G computed by the *PartitionAG* and *EvaluateAttributes* algorithms. $a \simeq b$ if and only if a and b belong to the same partition block in π.

Proof. First suppose that $a \simeq b$. By the definition of the $\simeq$ relation, there is a sequence of attribute instances $m_1, m_2, \ldots, m_j$, where $m_1 = a$ and $m_j = b$, of attribute instances in G such that there is a dependence relation between every pair of adjacent attribute instances in the sequence. Consider an arbitrary pair m_i and m_{i+1}, where $i = 1, 2, \ldots, j-1$, of attribute instances in the sequence. Let q be the production instance in which the dependence relation between m_i and m_{i+1} occurs. Because there is a dependence relation between m_i and m_{i+1}, m_i and m_{i+1} must be in the same block in the base partition of production q. Since the base partition of the production q is always a refinement of every partition recorded in the $\Pi(q|\ldots)$ function, m_i and m_{i+1} must be in the same block in the partition π, which is computed by the *PartitionAG* and *EvaluateAttributes* algorithms.

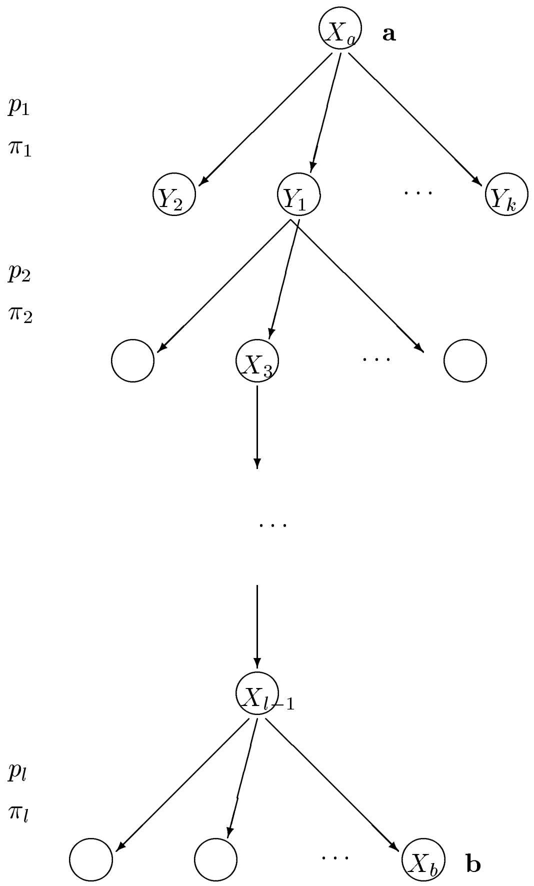

Figure 8: The X_a node is an ancestor of X_b in T.

Because every pair of adjacent attribute instances in $m_1, m_2, \ldots, m_j$ belong to the same partition block in π, a and b must belong to the same partition block in π.

Next, suppose that a and b belong to the same block in π. We need to show that $a \simeq b$. Let X_a be the nonterminal instance in T to which a belongs. Similarly, let X_b be the nonterminal instance in T to which b belongs. Because T is a tree, there is a unique path between X_a and X_b in T. There are four cases to consider: (1) X_a is an ancestor of X_b in T; (2) X_b is an ancestor of X_a in T; (3) X_a and X_b are the same nonterminal instance; or (4) X_a and X_b are distinct nonterminal instances in T and neither is an ancestor of the other. We will prove the theorem for case (1) above; the other three cases can be proved similarly.

We may label the production instances from X_a to X_b as $p_1, p_2, \ldots, p_l$, which are shown in Figure 8. Let π_i be the partition chosen for the production instance p_i, for $i = 1, 2, \ldots, l$, by the *EvalauteAttributes* algorithm. The partition π of G is the combination of $\pi_1, \pi_2, \ldots, \pi_l$, and partitions for other production instances in T.

Let $Y_1, Y_2, \ldots, Y_k$ be the nonterminal children of X_a in T. Assume without loss of generality that Y_2 is on the path from X_a to X_b. We show that $c \simeq a$ in G for every attribute instance c of Y_2 that is in the same partition block as a in the partition π_1.

Note that the partition π_1 is selected by the *EvaluateAttributes* algorithm as $\Pi(p_1 | \sigma_0, \sigma_1, \ldots, \sigma_k)$, where σ_0 is the feasible partitions chosen for the node X_a and $\sigma_1, \sigma_2, \ldots, \sigma_k$ are the plausible partitions chosen for the nodes $Y_1, Y_2, \ldots, Y_k$, respectively. π_1 is actually the merging of the base partition of production p_1 and $\sigma_0, \sigma_1, \ldots, \sigma_k$. Without a detailed proof, we claim that there must be an attribute instance d of X_a and an attribute instance e of Y_2 such that $a \simeq d$ (due to σ_0), $e \simeq c$ (due to σ_2), and there is a dependence relation between d and e in production p_1 (due to the base partition of production p_1).

We also need to show that there is at least one attribute instance c of Y_2 that is in the same partition block as a in the partition π_1. This claim is quite obvious because the two partitions π_1 and π_2 are glued together by at least one common attribute instance of Y_2.

At this point, we have shown that $c \simeq a$ in G for every attribute instance c of Y_2 that is in the same partition block as a in the partition π_1. By the same argument, we can show that $d \simeq c$ in G for every attribute instance d of X_3 that is in the same partition block as c in the partition π_2 (where X_3 is a child of Y_2 and is on the path from Y_2 to X_b). Repeating the same argument, we can show that $a \simeq b$ and a and b are in the same block in the partition π. This completes the proof. □

6. Conclusion

We have proposed an algorithm for partitioning the attribute dependence graph of a syntax tree. It is a static algorithm in that partitioning is done on the grammar, rather than on individual syntax trees. It produces the *finest* partition for every syntax tree. The partition algorithm, when combined with visit-oriented evaluation algorithm in [12] is applicable to all non-circular attribute grammars.

Bibliography

[1] Bochmann, G.V., "Semantic evaluation from left to right," *Comm. ACM,* Vol. 19, No. 2, pp. 55-62, February 1976.

[2] Boehm, Hans-Juergen, W. Zwaenepoel, "Parallel attribute grammar evaluation," *Proceedings of the 7th International Conference on Distributed Computing Systems*, IEEE, pp. 347-354, September 1987.

[3] Jourdan, M., "A survey of parallel attribute evaluation methods," *Proceedings of the International Summer School SAGA, (Prague, Czechoslovakia, June 1991), Lecture Notes in Computer Science*, Vol. 545, pp. 234-255, 1991.

[4] Kastens, U., "Ordered attribute grammars," *Acta Informatica*, Vol. 13, pp. 229-256, 1980.

[5] Katayama, T., "Translation of attribute grammars into procedures," *ACM Trans. Programming Languages and Systems*, Vol. 6, No. 3, pp. 345-369, July 1984.

[6] Klaiber, Alexander and Maya Gokhale, "Parallel evaluation of attribute grammars," *IEEE Trans. Parallel and Distributed Systems*, Vol. 3, No. 2, pp. 206-220, March 1992.

[7] Klein, E., "Parallel ordered attribute grammars," *Proceedings of the 1992 International Conference on Computer Languages*, pp.106-16, 1992.

[8] Knuth, D.E., "Semantics of context-free languages," *Mathematical System Theory*, Vol. 2, No. 2, pp. 127-145, June 1968. Correction. *ibid.* Vol. 5, No. 1, pp. 95-96, March 1971.

[9] Kuiper, M.F. and S.D. Swierstra, "Parallel attribute evaluation: Structure of evaluators and detection of parallelism," *Proceedings of the International Summer School SAGA, (Prague, Czechoslovakia, June 1991), Lecture Notes in Computer Science*, Vol. 545, pp. 61-75, 1991.

[10] Paakki, J., "Attribute grammar paradigms–A high-level methodology in language implementation," *ACM Computing Surveys*, Vol. 27, No. 2, pp. 196-255, June 1995.

[11] Reps, T., "Scan Grammars: Parallel Attribute Evaluation Via Data-Parallelism," TR-1120, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI, November 1992.

[12] Yang, W., "A classification of non-circular attribute grammars," *submitted for publication*, Computer and Information Science Dept., National Chiao-Tung Univ., Hsinchu, Taiwan, 1997.

[13] Zaring, A.K., "Parallel evaluation in attribute grammar-based systems," Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, NY, August 1990.