Translation of Attribute Grammars into Procedures

TAKUYA KATAYAMA
Tokyo Institute of Technology

An efficient method for evaluating attribute grammars by translating them into sets of procedures is presented. The basic idea behind the method is to consider nonterminal symbols of the grammar as functions that map their inherited attributes to their synthesized attributes. Associated with the nonterminal symbols are procedures that realize the functions. The attribute grammar is translated into a program consisting of these procedures. The essential point about this method is that attribute grammars are completely compiled into procedures, in contrast with evaluation algorithms that work interpretively in a table-driven manner. No information is stored in the nodes of derivation trees.

Although this evaluation method is applicable principally to absolutely noncircular attribute grammars in which the dependency relation among attribute occurrences of every production rule does not contain cycles, it is shown how the method is extended to the general noncircular attribute grammars. Problems of evaluating a set of attributes simultaneously and of recursive descent evaluation are also discussed.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.4 [**Programming Languages**]: Processors—*translator writing systems and compiler generators*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Attribute grammar, attribute evaluator

1. INTRODUCTION

An algorithm to evaluate an attribute grammar is presented. As is well known, the attribute grammar of Knuth [14, 15] is a very convenient tool for specifying the semantics of programming languages, especially for automating compiler construction. Several compiler generator systems have been built based on it [5, 10, 20]. The use of attribute grammars is not limited to compiler construction; other applications include text editing and program optimization [2, 4]. Recent work suggests that attribute grammars can be used for hierarchical and functional programming [11].

We must have efficient methods for attribute evaluation for these applications to have practical importance. Semantic analysis algorithms based on attribute

Author's address: Department of Computer Science, Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro, Tokyo, Japan 152.

grammars, however, are currently not efficient enough compared with ad hoc algorithms used in the usual handwritten compilers. The lack of good algorithms for attribute evaluation has restricted their usage to experimental compilers. Although many efforts have been made to obtain efficient evaluators [3, 6, 7, 8, 12, 13, 18, 19], we have not yet succeeded in getting an evaluator applicable to production quality compilers.

Here we propose an efficient and natural algorithm for attribute evaluation in which the administrative tasks of keeping track of attributes both already and not yet evaluated are put into the preprocessing or syntax analysis phase. The evaluation phase concentrates on the evaluation of attributes. The algorithm accepts absolutely noncircular attribute grammars, although extensions to general noncircular attribute grammars are possible.

For the absolutely noncircular grammars, Kennedy and Warren proposed a tree walk algorithm for attribute evaluation [12], and Saarinen improved it in the output-oriented form [21]. Their tree walk evaluator is a recursive procedure that visits nodes of derivation trees and evaluates their attributes. Each node of a derivation tree is equipped with two data fields, one for the state that shows attributes already evaluated and the other for the attribute values. When the evaluator visits a node with a set of the inherited attributes currently available, called an input set, it finds an appropriate sequence of actions on the basis of the input set and the state of the node. The actions are either computations of attribute values by means of semantic functions or visits to descendant nodes. After the actions have been executed, the state of the node is updated. In general, nodes may be visited several times with different input sets and node states. The evaluator is table driven.

In our algorithm, we consider nonterminal symbols to be functions that map their inherited attributes to their synthesized attributes. We associate procedures to realize these functions with the nonterminal symbols. The entire attribute grammar is then transformed into a set of mutually recursive procedures. Our algorithm is output oriented in the sense of Saarinen's.

A difference between the tree walk evaluators and ours is that we do not attach any information to the nodes of derivation trees. We thoroughly analyze data dependency among attributes and completely compile the original grammar into procedures, so we do not require the states of nodes to indicate which attributes have been evaluated and which are left unevaluated. Another difference is storage allocation for attributes. In the tree walk evaluator of Kennedy and Warren, the attributes are stored in the nodes of derivation trees. Saarinen's evaluator stores some of them in the nodes and the others in a stack. We, however, store all of them in the stack of the activation records of procedure calls.

When applied to an attribute grammar whose attribute evaluation process can be performed in a single scan from left to right [3], our algorithm can generate an evaluator that can be combined with the top-down parsers to result in recursive-descent compilers if the underlying context-free grammars are LL(k).

In the following, we first state how absolutely noncircular attribute grammars are translated into procedures and then extend the method to general noncircular grammars. Next, we describe our storage allocation strategy and finally consider recursive-descent evaluation.

2. DEFINITIONS AND NOTATIONS

An *attribute grammar* G is a context-free grammar $G = (V_N, V_T, P, S)$ augmented by semantic rules. In the following we use the same symbol G to denote both the entire attribute grammar and the underlying context-free grammar. We assume without loss of generality that the initial symbol S never appears in the right side of any rule in P.

With each symbol $X \in V_N \cup V_T$ is associated a set of *attributes* that is denoted by $A[X]$. $A[X]$ is a disjoint union of the set $I[X]$ of *inherited attributes* and the set $S[X]$ of *synthesized attributes*. We assume that $I[X] = \emptyset$ if $X = S$ and $S[X] = \emptyset$ if $X \in V_T$.

If p is a rule

$$p\colon X_0 \rightarrow X_1 X_2 \cdots X_{np}$$

and a is an attribute of X_k, that is, $a \in A[X_k](k = 0, 1, \ldots, np)$, we say that p has an *attribute occurrence* $a.k$. It is called a *synthesized occurrence* if $a \in S[X_k]$ and an *inherited occurrence* if $a \in I[X_k]$.

A *semantic function* $f_{p,v}$ is associated with every synthesized occurrence $v = a.k$ for $k = 0$ and inherited occurrence $v = a.k$ for $k = 1, \ldots, np$, and is defined in terms of other attribute occurrences of p. We denote the set of these attribute occurrences by $D_{p,v}$. It is called a *dependency set* of $f_{p,v}$. If $D_{p,v} = \{v_1, \ldots, v_n\}$, then $f_{p,v}$ is a mapping,

$$\text{domain } (v_1) \times \cdots \times \text{domain}(v_n) \rightarrow \text{domain}(v).$$

Let p be a production rule $X_0 \rightarrow X_1 X_2 \cdots X_{np}$. A *dependency graph* DG_p *for the production rule* p, which gives dependency relationships among attribute occurrences of p, is defined by

$$\mathrm{DG}_p = (DV_p, DE_p)$$

where the node set DV_p is the set of all attribute occurrences of p and the edge set DE_p is the set of dependency pairs for p. Formally

$$DV_p = \{a.k \mid k = 0, \ldots, np \text{ and } a \in A[X_k]\}$$

$$DE_p = \{(v_1, v_2) \mid v_1 \in D_{p,v_2}\}.$$

When a derivation tree T is given, a *dependency graph* DG_T *for the derivation tree* T is defined to represent dependencies among attributes of nodes in T. DG_T is obtained by pasting together DG_p's according to the syntactic structure of T. Let T be a derivation tree, $p\colon X_0 \rightarrow X_1 \cdots X_{np}$, the production rule applied at the root of T and T_k, the kth subtree of T. DG_T is recursively constructed from $\mathrm{DG}_p, \mathrm{DG}_{T_1}, \ldots, \mathrm{DG}_{T_{np}}$ by identifying the nodes for attributes of X_k in DG_p with the corresponding nodes for the attributes of the root of T_k in DG_{T_k}, $1 \le k \le np$. Figure 1 gives an example of DG_T for the attribute grammar given in Example 1 (see Section 3, p. 352).

An attribute grammar is *noncircular* if DG_T does not contain cycles for any T.

Let T be a derivation tree with root note $X \in V_N$. DG_T determines an *IO graph* $\mathrm{IO}[X, T]$ of X with respect to T. It gives an input-output relationship among

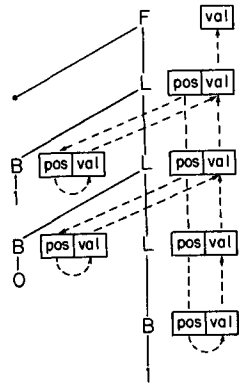


Fig. 1. An example of DG_T.

attributes of X, which is realized by the derivation tree T. That is,

$$IO[X, T] = (A[X], E_{IO}[T])$$

where an edge (a, s) is in $E_{IO}[T] \subset A[X] \times S[X]$ iff there is in DG_T a path from v_a to v_s, where v_a and v_s are nodes for attributes a and s of the root of T, and the attribute a is required to evaluate the synthesized attribute s.

For general attribute grammars, there may be finitely many IO graphs for $X \in V_N$. We denote the set of these IO graphs by $IO(X)$, that is,

$$IO(X) = \{IO[X, T] \mid T \text{ is a derivation tree}\}.$$

When $IO(X) = \{IO_1, IO_2, \ldots, IO_N\}$ and $IO_k = (A[X], E_k)$, superimposing IO_k results in a *superimposed IO graph*

$$IO[X] = (A[X], E_{IO}), \quad E_{IO} = \cup_{k=1}^{N} E_k$$

which represents possible input–output relationships among attributes of X. An algorithm to obtain $IO(X)$ and $IO[X]$ is given in the literature [12, 15].

For a synthesized attribute s of a nonterminal symbol X, its *input set* $in[s, X]$ is defined to be a set of attributes that may be required to evaluate s, that is,

$$in[s, X] = \{a \mid (a, s) \text{ is an edge of } IO[X]\}.$$

For a production rule p: $X_0 \rightarrow X_1 X_2 \cdots X_{np}$ its *augmented dependency graph* is defined by

$$DG_p^* = (DV_p^*, DE_p^*)$$

where

$$DV_p^* = DV_p$$

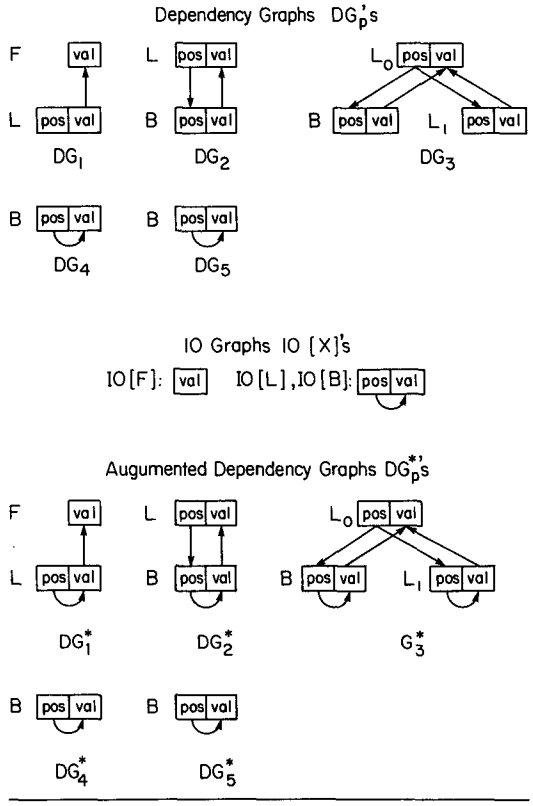
$$DE_p^* = DE_p \cup \{(a.k, s.k) \mid k = 1, \ldots, np \text{ and } (a, s) \in IO[X_k]\}.$$

DG_p^* represents a dependency relation among attribute occurrences in p, which is realized partly by semantic functions and partly by derivation trees.

An attribute grammar is *absolutely noncircular* iff DG_p^* does not contain cycles for any production rule p.

An example of $IO[X]$ and DG_p^* is given in Table I.

Table I. Dependency Graphs, IO Graphs, and Augmented Dependency Graphs for G_1.



3. TRANSLATION OF AN ABSOLUTELY NONCIRCULAR ATTRIBUTE GRAMMAR INTO A SET OF PROCEDURES

Let X be a nonterminal symbol of an absolutely noncircular attribute grammar $G = (V_N, V_T, P, S)$ and let s be a synthesized attribute of X. We associate with each pair (X, s) a procedure

$$R_{X,s}(v_1, \ldots, v_m, T; v)$$

where $v_1, \ldots, v_m$ are parameters corresponding to the attributes in $I = \text{in}[s, X]$, T is a derivation tree, and v is a parameter corresponding to s. Parameters to the left (right) of ";" are input (output) parameters. This procedure is intended to evaluate the synthesized attribute s when supplied with the values of attributes in I and a derivation tree T. When given the initial derivation tree T_0 and a synthesized attribute s_0 of the initial symbol S, we begin evaluation of s_0 by executing the procedure call statement

$$\textbf{call } R_{S,s_0}(T_0; v_0)$$

where v_0 is a variable corresponding to s_0. (We assume without loss of generality that the purpose of attribute evaluation is to know the value of the only attribute s_0 of S, though a more general case is touched upon in the later discussion.)

In the following we describe how to construct the procedure $R_{X,s}(v_1, \ldots, v_m, T; v)$. First we have to introduce variable symbols for attribute occurrences. However, for the sake of convenience, the same symbols are used for attribute occurrences and variables that correspond to them. In what follows, $X_0 = X$, $v = s.0$, and $\{v_1, \ldots, v_m\} = \{a.0 \mid a \in \text{in}[s, X_0]\}$.

The procedure $R_{X,s}$ is formed in the following way. It first examines what production rule is applied at the root of the derivation tree T and then selects a sequence of statements to calculate values of attribute occurrences of the production rule. We assume that nodes in derivation trees are labeled by the names of production rules applied there. The procedure has the form

procedure $R_{X,s}(v_1, \ldots, v_m, T; v)$
  case production(T) **of**
    p_1: $H_{p_1,s}(v_1, \ldots, v_m, T; v)$
    p_2: $H_{p_2,s}(v_1, \ldots, v_m, T; v)$
    ⋮
  end
end

where (1) "production(T)" is a function that returns the name of the production rule applied at the root of T, (2) $p_1, p_2, \ldots$ are production rules with left-side symbol X, and (3) $H_{p,s}(v_1, \ldots, v_m, T; v)$ is a sequence of statements for evaluating s when the production rule at the root of T is p.

Considering that absolute noncircularity allows attribute occurrences of p to be evaluated consistently in a fixed order, whatever derivation trees follow right-side nonterminal symbols of p, construction of $H_{p,s}$ proceeds as follows. First, attribute occurrences of p on which $s.0$ is dependent, directly or indirectly, in the augmented dependency graph DG_p^* are listed in topological order. Associated with each attribute occurrence is a statement for computing its value. It is an assignment statement with a semantic function as its right side when the attribute occurrence is defined in p; otherwise it is a procedure call statement. The sequence of these statements is $H_{p,s}$.

The construction of $H_{p,s}$ is

(1) For the production rule p: $X_0 \rightarrow X_1X_2 \cdots X_{np}$, make an augmented dependency graph

$$\text{DG}_p^* = (DV_p^*, DE_p^*).$$

(2) From DG_p^* remove nodes and edges that are not located on any path leading to $s.0$. Denote the resulting graph by

$$\text{DG}_p^*[s] = (V, E).$$

(3) To each attribute occurrence $x \in V_0 = V - \{i.0 \mid i \in I[X_0]\}$, assign a statement st[$x$] for evaluating x as follows.

Case 1. If $x = i.k$ for some $i \in I[X_k]$ and $k = 1, \ldots, np$ or $x = s.0 (= v)$ for the attribute $s \in S[X_0]$ that $R_{X,s}$ is to evaluate, then st[x] is the assignment statement

$$x \leftarrow f_{p,x}(z_1, \ldots, z_r)$$

where $f_{p,x}$ is the semantic function for the attribute occurrence x and $D_{p,x} = \{z_1, \ldots, z_r\}$ is the dependency set for the semantic function $f_{p,x}$.

Case 2. If $x = t.k$ for some $t \in S[X_k]$ and $k = 1, \ldots, np$, then st[x] is the procedure call statement

$$\textbf{call } R_{X_k,t}(w_1, \ldots, w_h, T[k]; x)$$

where $\{w_1, \ldots, w_h\} = \{a.k \mid a \in \text{in}[t, X_k]\}$ and $T[k]$ is the kth subtree of T.

(4) Let $x_1, x_2, \ldots, x_M$ be elements in V_0 that are listed according to the topological ordering determined by E; that is, if $(x_a, x_b) \in E$, then $a < b$. Then the sequence $H_{p,s}$ of statements becomes as follows:

st[x_1];

st[x_2];

⋮

st[x_M]

Note that statements in $H_{p,s}$ satisfy the single-assignment rule. It is easy to see that the ordering $x_1, \ldots, x_M$ ensures that values of attribute occurrences are determined consistently if the attribute grammar is absolutely noncircular.

So far we have only considered construction of a procedure for a particular $X \in V_N$ and $s \in S[X_0]$. Now we state how the entire attribute grammar G is translated into the corresponding program prog[G].

We start from the start symbol S and the synthesized attribute s_0 of S. We first construct the initial procedure R_{S,s_0} by the algorithm we have stated. The body of R_{S,s_0} may contain calls of other procedures $R_{X,s}$, and they are constructed in the same way. Repeat this process until no new procedures appear.

Let $R_{S,s_0}, R_{X_1,s_1}, \ldots, R_{X_N,s_N}$ be procedures thus obtained; then the entire program prog[G] for evaluating s_0 becomes (declarations for variables and types are omitted)

```
program
  procedure R_{S,s_0}
  .
  .
  .
  end;
  procedure R_{X_1,s_1}
  .
  .
  .
  end;
  .
  .
  .
  procedure R_{X_N,s_N}
  .
  .
  .
  end;
  input_derivation_tree(T_0);
  call R_{S,s_0}(T_0; v_0);
  output_attribute(v_0)
end
```

This program consists of declarations for the procedures R followed by the statements to input a derivation tree, activate the initial procedure, and output the value of s_0.

We now give two examples.

Example 1. The following attribute grammar $G_1 = (V_N, V_T, P, F)$ transforms the fractional part of binary notation into the corresponding number, where attribute occurrence $a.k$ is denoted by $X_k.a$.

Nonterminals	*Terminals*
$V_N = \{F, L, B\}$	$V_T = \{0, 1, .\}$
Attributes	
$I[F] = \emptyset$	$S[F] = \{\text{val}\}$
$I[L] = \{\text{pos}\}$	$S[L] = \{\text{val}\}$
$I[B] = \{\text{pos}\}$	$S[B] = \{\text{val}\}$

Productions	*Semantics*
1: $F \rightarrow .L$	$F.\text{val} = L.\text{val}$; $L.\text{pos} = 1$
2: $L \rightarrow B$	$L.\text{val} = B.\text{val}$; $B.\text{pos} = L.\text{pos}$
3: $L_0 \rightarrow BL_1$	$L_0.\text{val} = B.\text{val} + L_1.\text{val}$; $L_1.\text{pos} = L_0.\text{pos} + 1$; $B.\text{pos} = L_0.\text{pos}$
4: $B \rightarrow 0$	$B.\text{val} = 0$
5: $B \rightarrow 1$	$B.\text{val} = 2 \uparrow (-B.\text{pos})$

In what follows we illustrate how this grammar G_1 is transformed into the corresponding program.

First we construct dependency graphs DG_p, IO graphs IO[X], and augmented dependency graphs. These are given in Table I.

Since all the augmented dependency graphs are acyclic, the attribute grammar is absolutely noncircular. The procedure RL for the nonterminal symbol L and its attribute val, for example, is of the following form.

```
procedure RL(L.pos, T; L.val)
  case production(T) of
    p2: H_{p2,val}(L.pos, T; L.val)
    p3: H_{p3,val}(L.pos, T; L.val)
  end
end
```

Because the nodes of DG_3^* (except L_0.pos) can be topologically ordered as

$$B.\text{pos},\ B.\text{val},\ L_1.\text{pos},\ L_1\text{val},\ L_0.\text{val}$$

$H_{p3,\text{val}}$ is a sequence of statements

$B.\text{pos} \leftarrow L_0.\text{pos};$

call $RB(B.\text{pos}, T[1]; B.\text{val});$

$L_1.\text{pos} \leftarrow L_0.\text{pos} + 1;$

call $RL(L_1.\text{pos}, T[2]; L_1.\text{val});$

$L_0.\text{val} \leftarrow B.\text{val} + L_1.\text{val}$

The complete form of the procedure RL is contained in the next program which is the desired one for the attribute grammar G_1. (We dropped the suffix "0" from L_0.)

```
program
  procedure RF(T; F.val)
    L.pos ← 1;
    call RL(L.pos, T[2]; L.val);
    F.val ← L.val
  end;
  procedure RL(L.pos, T; L.val)
    case production(T) of
      p2: B.pos ← L.pos;
          call RB(B.pos, T[1]; B.val);
          L.val ← B.val
      p3: B.pos ← L.pos;
          call RB(B.pos, T[1]; B.val);
          L1.pos ← L.pos + 1;
          call RL(L1.pos, T[2]; L1.val);
          L.val ← B.val + L1.val
    end
  end;
  procedure RB(B.pos, T; B.val)
    case production(T) of
      p4: B. val ← 0
      p5: B.val ← 2 ↑ (−B.pos)
    end
  end;
  input_derivation_tree(T0);
  call RF(T0; F.val);
  output_attribute(F.val)
end
```

Example 2. In Example 1, a single procedure is associated with each nonterminal symbol. Now we give an example of attribute grammars that require multiple procedures for single nonterminal symbols. The grammar $G_2 = (V_N, V_T, P, S)$ computes $4 * n$ when a^{n+1} is given as an input string, by going down and up derivation trees twice. Multiple procedures are essential and cannot be reduced to a single procedure even by resorting to the simultaneous evaluation algorithm presented in the next section.

Nonterminal	*Terminals*
$V_N = \{S, A\}$	$V_T = \{a\}$

Attributes	
$I[S] = \varnothing$	$S[S] = \{k\}$
$I[A] = \{f, h\}$	$S[A] = \{g, k\}$

Productions	*Semantics*
1: $S \rightarrow A$	$S.k = A.k$; $A.h = A.g$; $A.f = 0$
2: $A_0 \rightarrow aA_1$	$A_1.f = A_0.f + 1$; $A_0.g = A_1.g + 1$; $A_1.h = A_0.h + 1$; $A_0.k = A_1.k + 1$
3: $A \rightarrow a$	$A.g = A.f$; $A.k = A.h$

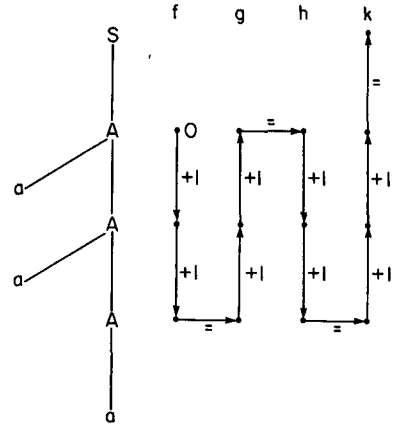


Fig. 2. Dependencies among attribute occurrences in a derivation tree of G_2.

Figure 2 shows how attributes are evaluated. The arrows represent dependencies among occurrences of attributes in the derivation tree. It is easy to see that the value of every k cannot be determined until all the occurrences of f, g, and h have been evaluated. This means that the synthesized attributes k and g cannot be evaluated simultaneously in a single pass.

G_2 is absolutely noncircular and IO graphs are

IO[S] : $\boxed{k}$

IO[A] : $\boxed{f}\boxed{g}\boxed{h}\boxed{k}$

Prog[G_2] is given below. Procedures RAg and RAk are obtained from the nonterminal symbol A.

```
program
  procedure RS(T; S.k)
    A.f ← 0;
    call RAg(A.f, T[1]; A.g);
    A.h ← A.g;
    call RAk(A.h, T[1]; A.k);
    S.k ← A.k
  end;
  procedure RAg(A0.f, T; A0.g)
    case production(T) of
      p2: A1.f ← A0.f + 1;
          call RAg(A1.f, T[1]; A1.g);
          A0.g ← A1.g + 1
      p3: A0.g ← A0.f
    end
  end;
  procedure RAk(A0.h, T; A0.k)
    case production(T) of
      p2: A1.h ← A0.h + 1;
          call RAk(A1.h, T[1]; A1.k);
          A0.k ← A1.k + 1
      p3: A0.k ← A0.h
    end
  end;
```

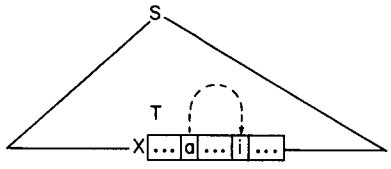


Figure 3

```
  input_derivation_tree(T0);
  call RS(T0; S.k);
  output_attribute(S.k)
end
```

4. SIMULTANEOUS EVALUATION

In the previous section we have associated one procedure with each synthesized attribute. However, data dependency sometimes allows several attributes to be evaluated simultaneously. It is desirable that our algorithm be modified to take advantage of this fact and to produce procedures in which these attributes are evaluated in a single procedure call, because this reduces overhead due to procedure activations and increases the chances of parallel execution.

4.1 Simultaneous Evaluability

We begin this modification by introducing an OI graph, the dual concept of the IO graph, which specifies how the values of inherited attributes are affected by other attributes.

Let T be a derivation tree containing $X \in V_N$ as one of its leaf nodes. An *OI graph* $\mathrm{OI}[X, T]$ of X with respect to T is given by

$$\mathrm{OI}[X, T] = (A[X], E_{\mathrm{OI}}[T]), \qquad E_{\mathrm{OI}}[T] \subset A[X] \times I[X]$$

where $(a, i) \in E_{\mathrm{OI}}[T]$ iff there is in DG_T a path from v_a to v_i, where v_a and v_i are nodes for attribute a and i of the leaf node X (Figure 3).

A *superimposed OI graph* $\mathrm{OI}[X]$ is defined in a way similar to $\mathrm{IO}[X]$. That is, if $\mathrm{OI}_1, \ldots, \mathrm{OI}_N$ are possible OI graphs of X and $\mathrm{OI}_k = (A[X], E_k)$, then

$$\mathrm{OI}[X] = (A[X], E_{\mathrm{OI}}), \qquad E_{\mathrm{OI}} = \bigcup_{k=1}^{N} E_k.$$

We further define a *dependency graph* $\mathrm{DG}[X]$ *of the nonterminal symbol* X as the union of IO graph and OI graph; that is,

$$\mathrm{DG}[X] = (A[X], E_{\mathrm{IO}} \cup E_{\mathrm{OI}}).$$

For an absolutely noncircular G a set $O \subseteq S[X]$ of synthesized attributes is *simultaneously evaluable* iff no $s_1, s_2 \in O$ are connected in $\mathrm{DG}[X]$. We extend the function $\mathrm{in}[s, X]$ to allow such O as its first argument.

$$\mathrm{in}[O, X] = \bigcup_{s \in O} \mathrm{in}[s, X].$$

Example 3

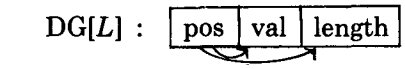
(1) DG[X] for the attribute grammar G_2 are given below.

Since k and g are connected in DG[A], the set $\{g, k\}$ of attributes of A is not simultaneously evaluable.

(2) Let G_1' be a grammar obtained from G_1 by adding the following semantic rules.

Production	*Added semantics*
1: $F \rightarrow .L$	F.length = L.length
2: $L \rightarrow B$	L.length = L.pos
3: $L_0 \rightarrow BL_1$	L_0.length = L_1.length

Of course, the synthesized attribute "length" is added to $S[F]$ and $S[L]$. It represents the length of the binary notation. The dependency graph for L is as follows.



As "val" and "length" are not connected, they are simultaneously evaluable.

4.2 Procedure Construction

Now we modify our algorithm. In essence it consists of assigning a single procedure

$$R_{X,O}(v_1, \ldots, v_m, T; u_1, \ldots, u_n)$$

to each set O that is simultaneously evaluable instead of assigning n procedures, where $u_1, \ldots, u_n$ are parameters corresponding to the synthesized attributes in O and $v_1, \ldots, v_m$ are those for attributes in in[O, X]. Construction of $R_{X,O}$ parallels that of $R_{X,s}$, except at a few points. As in the case of $R_{X,s}$, the procedure $R_{X,O}$ has the following form.

procedure $R_{X,O}(v_1, \ldots, v_m, T; u_1, \ldots, u_n)$
  case production(T) **of**
    p_1: $H_{p_1,O}(v_1, \ldots, v_m, T; u_1, \ldots, u_n)$
    p_2: $H_{p_2,O}(v_1, \ldots, v_m, T; u_1, \ldots, u_n)$
    .
    .
    .
  end
end

For a production rule p: $X_0 \rightarrow X_1X_2 \cdots X_{np}$ and $O \subset S[X_0]$, which is simultaneously evaluable, construction of statement sequence $H_{p,O}$ proceeds in the following steps.

The construction of $H_{p,O}$ is

(1) Make DG_p^*.

(2) Make $DG_p^*[O] = (V, E)$ by removing from DG_p^* nodes and edges which are not located on any path leading to $s.0$ for $s \in O$.

(3) For each $k = 1, \ldots, np$ decompose the set

$$S^*[X_k] = S[X_k] \cap \{t \mid t.k \in V\}$$

into a set of mutually disjoint subsets

$$O_{k1}, O_{k2}, \ldots, O_{kr}$$

such that they satisfy the following two conditions.

(a) Each O_{kj} is simultaneously evaluable.
(b) There are no cyclic dependencies among the set

$$\mathbf{O} = \{O_{k1}, O_{k2}, \ldots, O_{kr}\}.$$

That is, if we define a relation $<$ on $\mathbf{O}$ by

$O_{ki} < O_{kj}$ iff some $t \in O_{ki}$ and $s \in O_{kj}$ are connected in $\mathrm{DG}[X_k]$,

then no O_{ki} satisfies

$$O_{ki} \mathbin{<\!\!<} O_{ki}$$

where $<\!\!<$ is the transitive closure of $<$.

When the decomposition is not unique, we should choose a maximal decomposition, that is, one where the number r becomes minimum, to obtain high efficiency in evaluation. A method of obtaining a maximal decomposition is given in the following.

$$O_{k1} = \mathrm{bottom}(S^*[X_k])$$
$$O_{k2} = \mathrm{bottom}(S^*[X_k] - O_{k1})$$
$$O_{k3} = \mathrm{bottom}(S^*[X_k] - O_{k1} - O_{k2})$$
$$\vdots$$

where $\mathrm{bottom}(A)$ is a set of elements dependent on no elements of A in $\mathrm{DG}[X_k]$. That is,

$$\mathrm{bottom}(A) = \{a \mid a \in A \text{ and there is no edge leading to } a \text{ in } \mathrm{DG}[X]\}.$$

(4) Let $\mathrm{DG}'_p[O] = (V', E')$ be a graph obtained from $\mathrm{DG}^*_p[O]$ by grouping elements of each O_{kj} into a single new node $v_{kj} \notin V$ (Figure 4). Formally,

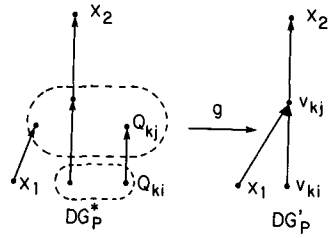
$$V' = \{g[v] \mid v \in V\}$$
$$E' = \{(g[u], g[v]) \mid (u, v) \in E\}$$

where g is a function defined by

$$g[v] = v_{kj} \quad \text{if } v = s.k \text{ for some } s, k, \text{ and } j \text{ such that } s \in O_{kj}$$
$$= v \quad \text{otherwise.}$$

(5) To each element x in $V_0 = V' - \{i.0 \mid i \in I[X_0]\}$ assign a statement $\mathrm{st}[x]$ as follows.

Figure 4



Case 1. If $x = i.k$ for some $i \in I[X_k]$ and $k = 1, \ldots, n$ or $x = s.0$ for $s \in S[X_0]$, then st$[x]$ is the assignment statement

$$x \leftarrow f_{p,x}(z_1, \ldots, z_r)$$

where $D_{p,x} = \{z_1, \ldots, z_r\}$ is the dependency set for $f_{p,x}$.

Case 2. If $x = v_{kj}$, then st$[x]$ is the procedure call statement

$$\textbf{call } R_{X_k,O_{kj}}(w_1, \ldots, w_h, T[k]; x_1, \ldots, x_c)$$

where (1) $\{w_1, \ldots, w_h\} = \{i.k \mid i \in \text{in}[O_{kj}, X_k]\}$ and (2) $\{x_1, \ldots, x_c\} = \{t.k \mid t \in O_{kj}\}$.

(6) Same as (4) for $H_{p,s}$.

The translation of the entire attribute grammar G into the corresponding program prog$[G]$ is similar to the one given in the previous section. Let O be a set of synthesized attributes of the initial symbol S. We start by constructing the procedure $R_{S,O}$ and then proceed to procedures that are called in it. Repeating this process until all the necessary procedures are obtained and adding statements to the input derivation tree, activate the initial procedure and output the values of attributes in O to give the desired program.

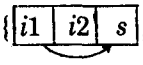
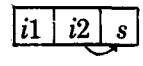
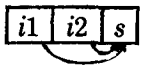
Example 4. Consider the grammar G_1' introduced in Example 3. The translation algorithm in Section 3 generates five procedures $R_{F,\text{val}}$, $R_{F,\text{length}}$, $R_{L,\text{val}}$, $R_{L,\text{length}}$, and $R_{B,\text{val}}$. On the other hand, if we use the simultaneous evaluation algorithm, three procedures $R_{F,\{\text{val,length}\}}$, $R_{L,\{\text{val,length}\}}$, and $R_{B,\text{val}}$ are constructed. $R_{L,\{\text{val,length}\}}$, for example, is given below, where it is denoted by RL.

```
Procedure RL(L.pos, T; L.val, L.length)
  case production(T) of
    p2: B.pos ← L.pos;
        call RB(B.pos, T[1]; B.val)
        L.val ← B.val;
        L.length ← L.pos
    p3: B.pos ← L.pos;
        call RB(B.pos, T[1]; B.val)
        L1.pos ← L.pos + 1;
        call RL(L1.pos, T[2]; L1.val, L1.length)
        L.val ← B.val + L1.val
        L.length ← L1.length
  end
end
```

5. EXTENSION TO GENERAL NONCIRCULAR GRAMMARS

The algorithm just stated is based on the assumption that augmented dependency graphs do not contain cycles; that is, the grammar is absolutely noncircular. The algorithm cannot be directly applied to general noncircular grammars. It has another problem, stated below, in evaluating absolutely noncircular grammars; we do not consider this a serious drawback in the usual situation.

In our approach we constructed procedures for nonterminal symbols on the basis of the superposed IO graphs. Although this made it possible to completely compile attribute grammars into procedures, there may occur cases for which we have to supply extra input parameters that are not used in the particular calls of the procedures. Consider, for example, a nonterminal symbol X with $I[X] = \{i1, i2\}$, $S[X] = \{s\}$ and

$$\mathrm{IO}(X) = \{$$  $$,$$  $$\}, \quad \mathrm{IO}[X]:$$  $$.$$

In our approach we associate a procedure $R_{X,s}(i1^*, i2^*, T; s^*)$ to X and s, where a^* denotes parameters for the attribute a, and values of $i1$ and $i2$ must be prepared in calling $R_{X,s}$, although only one of them is actually required. Of course this is remedied by the call-by-name parameter mechanism with a heavy implementation overhead. A slight additional effort, however, enables our algorithm to be adapted to this situation.

5.1 Conversion to Simple Attribute Grammars

First we define a restricted class of attribute grammars. A noncircular attribute grammar G is *simple* iff there is only one IO graph for any nonterminal symbol X, that is, $\mathrm{IO}[X, T]$ does not depend on the derivation tree T. It is obvious that G is absolutely noncircular if it is simple. By definition, procedures obtained from a simple attribute grammar do not suffer from the preparation of extra parameters discussed above.

We can show that *any noncircular grammar G is convertible to an equivalent simple one.* This is based on the following observation.

Suppose we are given a derivation tree T of a noncircular attribute grammar G. We attach an IO graph $d = \mathrm{IO}[X, T_X]$ to each nonterminal node X in T, where T_X is the subtree of T with X as its root. Denote the resulting tree by T'. By definition, every distinct nonterminal symbol $[X, d]$ in this augmented tree T' has the unique IO graph d, and T' is considered a derivation tree in some simple attribute grammar G'.

Formally, the simple attribute grammar G', which is equivalent to the noncircular G, is obtained in the following way.

Syntactically, $G' = (V'_N, V'_T, P', S')$ is given by

$$V'_N = \{[X, d] \mid X \in V_N, d \in \mathrm{IO}(X)\}$$

$$V'_T = V_T$$

$$P' = \{[X_0, d_0] \to [X_1, d_1] \cdots [X_{np}, d_{np}] \mid X_0 \to X_1 \cdots X_{np} \in P, d_k \in \mathrm{IO}(X_k) \text{ for } 1 \le k \le np, \text{ and } d_0 = \mathrm{DG}_p[d_1, \ldots, d_{np}] \mid \{a.0 \mid a \in A[X_0]\}\}$$

$$S' = [S, d_S]$$

where $\mathrm{DG}_p[d_1, \ldots, d_{np}]$ is a graph obtained from DG_p, the dependency graph for the production p, by adding edges $(i.k, s.k)$ for $1 \le k \le np$ and $(i, s) \in d_k$. When a graph $D = (V, E)$ with a node set V and an edge set E is given, the notation $D \mid V_0$, where $V_0 \subset V$, denotes a graph $D_0 = (V_0, E_0)$ where $E_0 = \{(v_1, v_2) \mid v_1, v_2$ are in V_0 and there is a path from v_1 to v_2 in $D\}$. We assume without loss of generality that the IO graph for S is unique.

As for the semantics, G' has the same structure as G. That is, (1) $[X, d]$ has the same set of attributes as X

$$A[[X, d]] = A[X]$$

and (2) if p is $X_0 \to X_1 \cdots X_{np}$ and p' is the corresponding production in G', that is, p': $[X_0, d_0] \to [X_1, d_1] \cdots [X_{np}, d_{np}]$, then

$$f_{p',v} = f_{p,v}$$

It is easy to show that *G and G' are syntactically and semantically equivalent.* That is, the following theorem holds.

THEOREM

(1) *For any derivation tree T in G there exists a derivation tree T' in G' such that, if the root of T is X and $IO[X, T] = d$, then the root of T' is $[X, d]$, and, conversely, for any T' there exists T such that, if the root of T' is $[X, d]$, then the root of T is X and* $\mathrm{IO}[X, T] = d$.

(2) *For such T and T' attribute values of corresponding nodes are identical.*

PROOF

(1) The second part is proved in the same way as the first part, so we only prove the first part. The proof is performed by induction on derivation trees.

It is obviously true for T with $|T| = 1$, where $|T|$ denotes the height of T. Assume it holds for any T with $|T| \le h$. Let T be a derivation tree such that $|T| = h + 1$ and the production rule applied at its root is p: $X_0 \to X_1 \cdots X_{np}$. If we denote its subtree with root X_k by T_k $(1 \le k \le np)$, then, by assumption, there exists in G' a derivation tree T_k whose root is $[X_k, d_k]$, where $d_k = \mathrm{IO}[X_k, T_k]$. Therefore construction of P' assures the existence of T' whose root is $[X_0, d_0]$ such that $d_0 = \mathrm{DG}_p[d_1, \ldots, d_{np}] \mid [a.0 \mid a \in A[X_0]\}$. It is easy to see $d_0 = \mathrm{IO}[X_0, T]$ and we have proved our claim. □

(2) The proof is obvious by the choice of $A[[X, d]]$ and $f_{p',v}$. □

Generally, the complexity of transforming G into G' is intrinsically exponential in the size of G [7], because the transformation requires the enumeration of all the IO graphs $\mathrm{IO}[X, T]$, and the number of possible IO graphs is exponential in the size of G. However, this does not mean that it is impossible to do the transformation in practice, since the usual attribute grammars used for the description of programming languages will not differ greatly from absolutely

noncircular ones, and, in addition, the transformation may well be performed off-line, that is, only once prior to generating evaluators.

5.2 Attribute Evaluation

Now, when a derivation tree T in G is given, its attribute evaluation under G can be replaced by the following two steps.

(1) Transform T into T'.
(2) Evaluate T' under G'.

Step (2) is, of course, to feed T' to a program prog[G'], which is obtained from G' by our algorithm. Step (1) can be performed with a small additional cost in a syntax analysis phase. When given a nonterminal node X_0 in T and a production rule p: $X_0 \rightarrow X_1 \cdots X_{np}$, which is applied there, IO graph $d_0 = \mathrm{IO}[X_0, T_{X_0}]$ is determined uniquely from $d_k = \mathrm{IO}[X_k, T_{X_k}]$ for $1 \leq k \leq np$ by

$$d_0 = \mathrm{DG}_p[d_1, \ldots, d_{np}] \| \{a.0 \mid a \in A[X_0]\};$$

so, a single bottom-up scan of T is enough to decide the IO[X, T_X]'s for all the X's in T and to transform T into T'.

The above process for evaluating a general noncircular attribute grammar G may be reformulated to produce a single program for attribute evaluation. We first augment G with semantic rules to calculate IO graphs of nonterminal nodes in derivation trees. The augmented grammar G^* is translated into a program prog*[G^*] whose structure is similar to prog[G^*] except at one point.

Suppose we attempt to evaluate attributes of a nonterminal node X in a derivation tree T and there are several production rules with left-side symbol X in G^*. In prog[G^*] we made the selection of a sequence $H_{p,s}$ of statements on the basis of what production rule is applied at X in T. In prog*[G^*], on the other hand, a pair of production rules p applied at X and the IO graph d of X is used to select the proper sequence of statements. That is, the following procedure is associated with X and $s \in S[X]$, where $d(T)$ is the IO graph of the root node of T.

procedure $R_{X,s}(v_1, \ldots, v_m, T; v)$
  case (production(T), $d(T)$) **of**
    ⋮
  (p, d): $H_{p,t}(w_1, \ldots, w_h, T; x)$
    ⋮
  end

Needless to say, the semantic rules of G^* are made so that the evaluation of the d's precedes that of other attributes existing in the original grammar G, and the evaluated values of the d's must be attached to the nodes of the derivation trees.

Extension to the case of the simultaneous evaluation is similar. We attach to each nonterminal symbol of G a pair from its IO graph and OI graph. This process can be accomplished by a top-down traversal of derivation trees followed by a bottom-up one, that is, by applying a recursive procedure to them. This may be also done in the syntax analysis phase.

Example 4. Consider the following attribute grammar $G_3 = (V_N, V_T, P, S)$.

Nonterminals	*Terminals*
$V_N = \{S, A, B\}$	$V_T = \{\alpha 1, \alpha 2\}$

Attributes

$I[S] = \emptyset$	$S[S] = \{r\}$
$I[A] = \{a1, a2\}$	$S[A] = \{a3, a4\}$
$I[B] = \{b1, b2\}$	$S[B] = \{b3, b4\}$

Production	*Semantics*
1: $S \rightarrow A$	$A.a1 = 1$; $A.a2 = 1$; $S.r = A.a3 + A.a4$
2: $A \rightarrow B$	$B.b1 = A.a1$; $B.b2 = A.a2$;
	$A.a3 = B.b3$; $A.a4 = B.b4$;
3: $B \rightarrow \alpha 1$	$B.b3 = B.b1$; $B.b4 = B.b2$
4: $B \rightarrow \alpha 2$	$B.b3 = B.b2$; $B.b4 = B.b1$

This grammar is not simple because we have two IO graphs for A and B. So we split A into $A1$ and $A2$, and B into $B1$ and $B2$.

$$\mathrm{IO}(A) = \{d_{A1} : \boxed{a1}\boxed{a2}\boxed{a3}\boxed{a4},\ d_{A2} : \boxed{a1}\boxed{a2}\boxed{a3}\boxed{a4}\}$$

$$\mathrm{IO}(B) = \{d_{B1} : \boxed{b1}\boxed{b2}\boxed{b3}\boxed{b4},\ d_{B2} : \boxed{b1}\boxed{b2}\boxed{b3}\boxed{b4}\}$$

Production rules and semantic rules of the simple grammar G_3' are given below, where we write, for example, $A1$ for $[A, d_{A1}]$.

Production	*Semantics*
1: $S' \rightarrow A1$	$A1.a1 = 1$; $A1.a2 = 1$; $S'.r = A1.a3 + A1.a4$
2: $S' \rightarrow A2$	$A2.a1 = 1$; $A2.a2 = 1$; $S'.r = A2.a3 + A2.a4$
3: $A1 \rightarrow B1$	$B1.b1 = A1.a1$; $B1.b2 = A1.a2$;
	$A1.a3 = B1.b3$; $A1.a4 = B1.b4$
4: $A2 \rightarrow B2$	$B2.b1 = A2.a1$; $B2.b2 = A2.a2$;
	$A2.a3 = B2.b3$; $A2.a4 = B2.b4$
5: $B1 \rightarrow \alpha 1$	$B1.b3 = B1.b1$; $B1.b4 = B1.b2$
6: $B2 \rightarrow \alpha 2$	$B2.b3 = B2.b2$; $B2.b4 = B2.b1$

Correspondence between derivation trees in G and G' is given in Figure 5.

6. STORAGE ALLOCATION FOR ATTRIBUTES

Efficiency of an attribute evaluator depends largely on the policy of where to store the values of attribute instances. The tree walk evaluator of Kennedy and Warren stores them in the nodes of derivation trees. This policy is natural and has the advantage that the computed attribute values can be referenced several times without doing recomputations. This advantage can turn out to be a disadvantage, because the memory space allocated to attribute instances cannot be freed until the entire process of attribute evaluation is completed. To solve

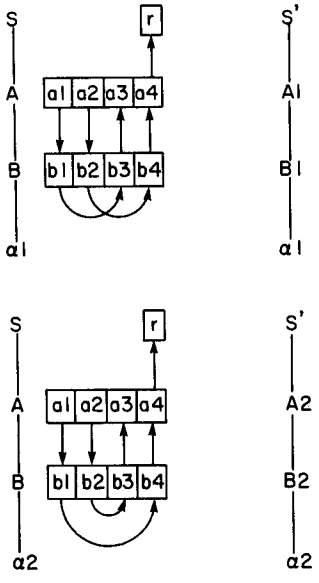


Fig. 5. Correspondence between derivation trees in G and G'.

this problem, Saarinen classified the attribute instances into significant and temporary ones. An attribute instance is defined to be significant if its lifetime reaches beyond one visit call to a descendant node; otherwise it is defined to be temporary. He proposed a tree walk evaluator in which the significant instances were stored on the derivation trees and the temporary ones in a stack. That strategy makes it possible to reuse the memory space allocated to temporary instances.

In our evaluator, we place the value of every attribute instance on the stack of the activation records of procedure calls, and no values are stored in derivation trees. Since we do not require the evaluation states to be stored in their nodes, no information is attached to the nodes. This is an advantage over the tree walk evaluators, especially in the case of constructing evaluators for the class of grammars that do not require derivation trees for their attribute evaluation [17].

Though every attribute instance is stored in a stack in our evaluator, we do not have to recompute it, even in the case in which its life span reaches beyond one visit to a descendant. This is accomplished by allocating the entire attribute space for all descendants at once before the first visit to them and freeing it after the last visit. It is easy to see that the maximum length of the stack is proportional to the height of the derivation tree.

One of the merits of this strategy is that it can be realized very naturally by the standard procedure mechanism of programming languages, such as Pascal. We do not have to manipulate the stack explicitly in this case because it is represented by the stack of activation records of procedure calls. This is also preferable from the standpoint of efficiency, since many machines offer hardware support for procedure mechanisms.

Up to now, we have mentioned neither the parameter-passing mechanism nor the declaration of variables for our procedures. This was to clarify the control structure of the evaluator. Now we make these things explicit. We assume that our procedures are written in a programming language where (1) every parameter is passed by reference and (2) local variables of procedures are created when the procedures are called and destroyed when they return. We use the symbol **var** to denote variable declaration. Type declaration is omitted, as it is irrelevant here.

6.1 Procedure Construction

The construction of the procedures is similar to the constructions given in Sections 3 and 4. We show briefly how to construct the procedure $R_{X,s}$ for the nonterminal symbol X and its synthesized attribute s. As in the case of Section 3, we first prepare formal parameters $v_1, \ldots, v_m$, T, and v, which correspond to attributes in in$[s, X]$, a derivation tree, and s, respectively. These are reference parameters. We assume a data type definition for the derivation tree T such that $T[k]$, which expresses the kth subtree of T, can be treated as a variable.

For each production rule $p = p_1, p_2, \ldots$, with left side symbol X, find a list of p's attribute occurrences $x_1, x_2, \ldots$, which correspond to attributes of the right-side nonterminal symbols of p and are necessary to evaluate $s.0$. Then construct the sequence $H_{p,s}$ of statements for evaluating s as in a manner given earlier. The procedure $R_{X,s}$ has, at this time, inner procedures $Q_{p,s}$ for each $p = p_1, p_2, \ldots$, and $Q_{p,s}$ consists of the declaration of variables $x_1, x_2, \ldots$, and the body $H_{p,s}$. The body of $R_{X,s}$ is a case statement for selecting an appropriate call of $Q_{p,s}$ on the basis of the production rule applied. Thus we have the following.

```
Procedure R_{X,s}(v_1, ..., v_m, T; v)
    :
  procedure Q_{p,s}
    var x_1, x_2, ...,
    H_{p,s}
  end
    :
  case production(T) of
    :
    p: call Q_{p,s}
    :
 end
end
```

It is easy to see that this procedure realizes our storage allocation strategy. That is, when the procedure $R_{X,s}$ is called at a node with label X to evaluate s, it determines the production rule applied there and allocates space for attribute occurrences $x_1, x_2, \ldots$, of descendant nodes. Note that the space for X has been provided by the caller of $R_{X,s}$ and passed to $R_{X,s}$ via a reference parameter-passing mechanism. Of course, the space for $x_1, x_2, \ldots$ becomes free after $R_{X,s}$ has returned.

6.2 A Simple Optimization

It has been observed by compiler writers that many attribute definitions serve only to copy attribute values without changing them. Omission of these attribute transfers greatly increases the efficiency of evaluators. It is easy to modify our evaluator to exploit this fact.

Let p be a production rule

$$X_0 \rightarrow X_1 X_2 \cdots X_{np}$$

and s a synthesized attribute of X_0. When x is an attribute occurrence of p, which is necessary to evaluate $s.0$ and whose value is defined by a mere transfer of the value of an attribute occurrence y, the straightforward implementation of the procedure $R_{X,s}$ requires the assignment statement

$$x \leftarrow y$$

to be included in $H_{p,s}$. We can, however, remove it from $H_{p,s}$ unless both x and y correspond to attributes of X_0. There are two cases:

(1) $x = i.k$ for $k \neq 0$, $i \in I[X_k]$.
(2) $x = s.0$ and $y = a.k$ for $k \neq 0$, $a \in A[X_k]$.

In case (1), we replace $H_{p,s}$ by $H'_{p,s}$ and remove x from the list of variable declaration of $Q_{p,s}$. $H'_{p,s}$ is obtained by first removing the assignment $x \leftarrow y$ from $H_{p,s}$ and then substituting y for x, that is,

$$H'_{p,s} = [H_{p,s} - \{x \leftarrow y\}]_{x \leftarrow y}.$$

Similarly, in case (2), $H_{p,s}$ is replaced by

$$H'_{p,s} = [H_{p,s} - \{x \leftarrow y\}]_{y \leftarrow x}$$

and y is removed from the variable declaration. It is easy to see that the resulting procedure $R'_{X,s}$ is equivalent to the original $R_{X,s}$.

Example 5. We show in the following the complete form of the program for the attribute grammar G_1 given in Example 1, for which we have already shown a program that does not consider storage allocation. Note that we removed assignment statements for the mere transfer of attribute values.

```
program
  var T, F.val
  procedure RF(T; F.val);
    var L.pos
    L.pos ← 1;
    call RL(L.pos, T[2]; F.val);
  end;
  procedure RL(L.pos, T; L.val)
    procedure Qp2
      call RB(L.pos, T[1]; L.val)
    end;
    procedure Qp3
      var B.val, L1.pos, L1.val;
      call RB(L.pos, T[1]; B.val);
      L1.pos ← L.pos + 1;
```

```
      call RL(L1.pos, T[2]; L1.val);
      L.val ← B.val + L1.val
    end;
    case production(T) of
      p2: call Qp2
      p3: call Qp3
    end
  end;
  procedure RB(B.pos, T; B.val)
    case production(T) of
      p4: B.val ← 0
      p5: B.val ← 2 ↑ (−B.pos)
    end
  end;
  input_derivation_tree(T);
  call RF(T; F.val);
  output_attribute(F.val)
end
```

7. RECURSIVE DESCENT COMPILATION

In this section we show that our evaluation method can be used to generate recursive-descent compilers mechanically from absolutely noncircular attribute grammars if their attribute evaluation proceeds from left to right [3] and their underlying context-free grammar is LL(k).

Suppose an attribute grammar G admits left-to-right evaluation. That is, for any production rule p: $X_0 \rightarrow X_1X_2 \cdots X_{np}$ we assume the following holds:

(1) $D_{p,s.0} \cap S_0 = \varnothing$ for all $s \in S[X_0]$,
(2) $D_{p,i.k} \cap \{S_0 \cup \bigcup_{h=k}^{np} (I_h \cup S_h)\} = \varnothing$ for all $k = 1, \ldots, np$ and $i \in I[X_k]$,

where I_k and S_k are the sets of inherited and synthesized attribute occurrences of X_k, respectively, and $D_{p,v}$ is the dependency set for the semantic function $f_{p,v}$. Bochmann [3] showed that this condition implies that the attributes in any derivation tree can be evaluated in a single pass from left to right. As a result, already evaluated nodes are not necessary if the results of the evaluations are saved somewhere. We also do not need attributes of nodes that are located to the right of the node whose attributes we are going to evaluate. Therefore, if the underlying context-free grammar is LL(k), that is, top-down parsing can proceed from left to right by looking k symbols ahead, construction of derivation trees is not necessary since the node to be evaluated can be known from the lookahead symbols. The idea of evaluating attributes without constructing derivation trees is found in several papers [8, 19]. Lewis et al. [17] combined the left-to-right evaluation and LL(k) to obtain an "attributed pushdown machine." If, in addition, the grammar is absolutely noncircular, a slightly modified version of our algorithm can generate what are called recursive-descent compilers. In the following, we show an example of constructing a recursive-descent evaluator.

The first thing we have to do is to make the constructed procedures not explicitly contain the derivation tree T. Instead, we set the input string as global data that are accessible from all procedures and insert read statements in appropriate places to ensure that correct substrings, that is, correct subtrees, are visible from the procedures. We assume the existence of a function "lookahead"

which returns the lookahead symbols and replaces the function "production." We also have to change the case statements to reflect this replacement so that they make selections based on these lookahead symbols.

Example 6. Let us reexamine the grammar G_1 considered in Example 1. G_1 is LL(2), left-to-right evaluable and absolutely noncircular, so we can apply the above method. The generated program for the recursive-descent evaluation is shown below, where lookahead[1] and lookahead[2] are the first and the second symbols of the current input string, respectively, and "read" is a statement to delete its first symbol. Note that the generated procedures do not contain a parameter for the derivation tree.

```
program
  var F.val;
  procedure RF( ; F.val)
    var L.pos
    L.pos ← 1;
    read;
    call RL(L.pos; F.val)
  end;
  procedure RL(L.pos; L.val);
    procedure Qp2
      call RB(L.pos; L.val);
    end;
    procedure Qp3
      var B.val, L1.pos, L1.val;
      call RB(L.pos; B.val);
      L1.pos ← L.pos + 1;
      read;
      call RL(L1.pos; L1.val);
      L.val ← B.val + L1.val
    end;
    case lookahead[2] of
         ' ': Qp2
      '0', '1': Qp3
    end
  end;
  procedure RB(B.pos; B.val)
    case lookahead[1] of
      '0': B.val ← 0
      '1': B.val ← 2 ↑(−B.pos)
    end
  end;
  input_string;
  call RF( ; F.val);
  output_attribute(F.val)
end
```

8. DISCUSSION

We have presented an efficient method for evaluating attribute grammars by translating them into sets of procedures. The basic idea behind the method is to consider nonterminal symbols of the grammar as functions that map their inherited attributes to their synthesized attributes and to associate with the grammar procedures that realize the functions.

The essential point about our method is the complete compilation of attribute grammars into procedures, in contrast to the tree walk evaluators, which work interpretively in a table-driven manner [12, 13]. It is one of the characteristics of our evaluator that we do not attach any information to the nodes of derivation trees, not even the values of attribute instances. Our strategy for allocating storage for the attribute instances is to store their values in the stack of the activation records of procedure calls. This is realized implicitly by the standard procedure mechanism of the programming languages.

An attractive feature of our method is that well-developed techniques for program optimizations can be applied to the generated procedures. For example, the next program for recursive-descent evaluation of binary numbers is obtained from the one constructed in Example 6 by applying replacement of a nonrecursive procedure call by its body.

```
program
  var F.val, L.pos;
  procedure RL(L.pos; L.val)
    procedure Qp3
        var B.val, L1.pos, L1.val;
        case lookahead[1] of
              '0': B.val ← 0
              '1': B.val ← 2 ↑ (−L.pos)
        end;
        L1.pos ← L.pos + 1;
        read;
        call RL(L1.pos; L1.val);
        L.val ← B.val + L1.val
    end;
    case lookahead[2] of
        ' ': case lookahead[1] of
                  '0': L.val ← 0
                  '1': L.val ← 2 ↑ (−L.pos)
             end
    '0', '1': Qp3
    end
  end;
  input_string;
  L.pos ← 1;
  read;
  call RL(L.pos; F.val);
  output_attribute(F.val)
end
```

ACKNOWLEDGMENTS

The author wishes to thank Masayuki Takeda for a critical reading of the manuscript. He pointed out the superiority of associating procedures with nonterminal symbols over associating them with production rules, which was the author's original formulation. The author also thanks referees for their suggestions for improving the paper.

REFERENCES

1. ASBROCK, B. Attribut-Implementierung und -Optimierung für Attribute Grammatiken. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, West Germany, July 1979.

2. BABICH, W.A., AND JAZAYERI, M. The method of attributes for data flow analysis, parts I and II. *Acta Inf. 10* (1978), 245–272.
3. BOCHMANN, G.V. Semantic evaluation from left to right. *Commun. ACM 19*, 2 (Feb. 1976), 55–61.
4. DEMERS, A., REPS, T., AND TEITLEBAUM, T. Incremental evaluation for attribute grammars with applications to syntax-derected editors. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 26–28, 1981). ACM, New York, pp. 105–116.
5. GANZINGER, H., RIPKEN, K., AND WILHELM, R. MUG1—An incremental compiler-compiler. In *Proceedings of the ACM Annual Conference* (Houston, Tex., Oct. 20–22). ACM, New York, 1976, pp. 415–418.
6. JAZAYERI, M. On attribute grammars and the semantic specification of programming languages. Ph.D. dissertation, Computer and Information Science Dept., Case Western Reserve Univ., Cleveland, Ohio, Oct. 1974.
7. JAZAYERI, M., OGDEN, W.F., AND ROUNDS, W.C. The intrinsically exponential complexity of the circularity problem for attribute grammar. *Commun. ACM 18*, 12 (Dec. 1975), 697–706.
8. JAZAYERI, M., AND POZEFSKY, D. Algorithms for efficient evaluation of multi-pass attribute grammars without a parse tree. Tech. Rep. TR77-001, Department of Computer Science, Univ. of North Carolina, Chapel Hill, N.C., May 1974.
9. KASTENS, U. Ordered attribute grammars. *Acta Inf. 13* (1980), 229–256.
10. KASTENS, U., AND ZIMMERMANN, E. GAG-A generator based on attributed grammar. Bericht Nr. 14/80, Institute für Informatik II, Universität Karlsruhe, Karlsruhe, West Germany.
11. KATAYAMA, T. HFP: A hierarchical and functional programming based on attribute grammar. In *Conference Record of the 5th International Conference on Software Engineering* (San Diego, Calif., Mar. 1981), pp. 343–352.
12. KENNEDY K., AND WARREN, S.K. Automatic generation of efficient evaluators for attribute grammars. In *Conference Record of the 3rd ACM Symposium on Principles of Programming Languages* (Atlanta, Ga., Jan. 1976), ACM, New York, pp. 32–49.
13. KENNEDY, K., AND RAMANATHAN, J. A deterministic attribute grammar evaluator based on dynamic sequences. *ACM Trans. Prog. Lang. Syst. 1*, 1 (July 1979), 142–160.
14. KNUTH, D.E. *Examples of Formal Semantics.* Lecture Notes in Mathematics, vol. 188. Springer-Verlag, New York, 1971.
15. KNUTH, D.E. Semantics of context-free languages. *Math. Syst. Theory J.2* (1968), 127–145.
16. KNUTH, D.E. Semantics of context-free languages: Correction. *Math. Syst. Theory J.5* (1971), 95.
17. LEWIS, P.M., ROSENKRANTZ, D.J., AND STEARNS, R.E. Attributed translations. *J. Comput. Syst. Sci. 9* (1974), 279–307.
18. MADSEN, O.L. On defining semantics by means of extended attribute grammars. Rep. DAIMI PB-109, Computer Science Department, Aarhus Univ., Aarhus, Denmark, Jan. 1980.
19. POZEFSKY, D. Building efficient pass-oriented attribute grammar evaluators. Tech. Rep. TR 79-006, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, N.C., 1979.
20. RAIHA, K.J., SAARINEN, M., SOISALON-SOININEN, E., AND TIENARI, M. The compiler writing system HLP. Rep. A-1978-2, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland, Mar. 1978.
21. SAARINEN, M. *On Constructing Efficient Evaluators for Attribute Grammars.* Lecture Notes in Computer Science, vol. 62. Springer-Verlag, New York, 1978.

Received July 1980; revised December 1981, March and September 1983; accepted October 1983