On Defining Denotational Semantics for Attribute Grammars

MASAYUKI TAKEDA* and TAKUYA KATAYAMA*

This paper presents a denotational semantics of attribute grammars and proposes a method of attribute evaluation based on it. The denotational semantics of an attribute grammar is defined by the least fixpoint semantics among attributes assigned to the nodes of a derivation tree and it is realized by a set of recursive functions which perform the required evaluations. The proposed method for attribute evaluation is directly based on this denotational semantics and handles a wider class of attribute grammars including the well-defined ones. It has the following characteristics: (1) the evaluation functions can be derived directly from the description of a given attribute grammar without resorting to augmented dependency graphs, (2) it is an *output-oriented evaluation*, that is, only attributes which are related to the required attributes are evaluated, (3) it can be easily implemented by a LISP-like programming language.

1. Introduction

The specification of the semantics of a programming language can be formulated in various ways [7, 15, 18]. Attribute grammars proposed by Knuth [13] have several advantages: (1) the semantic description of a language is structured according to the syntax, (2) the context sensitive restrictions can be described by semantic conditions between attributes, (3) the correctness of the description can be verified modularly (i.e., production-rule-wise), and (4) the description can be utilized for automatic compiler generation. This method has been used in various fields of computer science, such as programming language design, translation, program correctness, optimization, and question-answering systems.

Several authors have studied the problem of an attribute evaluation. Bochmann [1] proposed a method in which attribute evaluation is performed in a fixed number of left-to-right (depth-first) passes over a derivation tree, and Jazayeri [8] extended this method to a fixed number of alternating left-to-right and right-to-left passes. Lewis et al. [14] defined an attributed push-down machine, and presented the conditions of attribute grammars so that the syntax analysis and the evaluation of attributes can be performed deterministically. Kennedy and Warren [12] proposed a treewalk evaluator which traverses a derivation tree and performs attribute evaluations according to predefined plans formed by *augmented dependency graphs*. Saarinen [16] improved this method so that it performs the *output-oriented evaluation*, that is, it evaluates only attributes relevant to the attributes whose values we are going to obtain and that the size of the evaluator can be reduced by storing temporary (local) attributes in a stack. Katayama proposed an efficient evaluation algorithm in which the entire grammar is transformed into a set of mutually recursive procedures [10]. These evaluation methods, however, are applicable only to restricted classes of grammars.

Fang [5] proposed an evaluation algorithm which is applicable to general attribute grammars. He assigned a parallel process for each semantic function and expressed data dependency by synchronization primitives in a derivation tree, however this method is far from practical.

Chirica and Martin [2] have given an order-algebraic definition of attribute grammars within the framework of initial algebra semantics. Their approach is based on the *least fixpoint semantics* and the meaning of a derivation tree is considered to be the least fixpoint solution of equations about attributes assigned to its nodes.

The purpose of this paper is to define a denotational semantics of attribute grammars and to propose a method of attribute evaluation based on it. The difference between Chirica and Martin's method and ours is that we introduce a function associated with each pair of a nonterminal symbol and one of its synthesized attribute to evaluate only the necessary attributes using information found in the semantic analysis phase, whereas a least fixpoint semantic function about all attributes of a derivation tree is defined in theirs. Our evaluation method has the following characteristics: (1) the evaluation functions can be derived directly from the description of a given attribute grammar without resorting to augmented dependency graphs, (2) it is an output-oriented evaluation, that is, only attributes which are related to the required attributes are evaluated, (3) it can be easily implemented by a LISP-like programming language, (4) it is applicable to any *noncircular attribute grammars*, although it requires more computation overhead for environment management.

The rest of the paper is arranged as follows. Section 2 gives necessary definitions and notations for attribute

*Department of Computer Science, Tokyo Institute of Technology.

grammars in an algebraic formulation together with an example. In Section 3, we briefly review the least fixpoint semantics, define a denotational semantics of an attribute grammar, and propose an output-oriented evaluation method based on it. Section 4 presents an implementation of our method by the programming language LISP.

2. Algebraic Formulation of Attribute Grammars

In this section, we review and discuss attribute grammars in an algebraic formulation according to Chirica and Martin.

2.1 Attribute Grammar

An attribute grammar is a context free grammar augmented with semantic rules (attributes and semantic functions), and consists of the following four elements.

(1) *A context free grammar* $G=\langle V_N, V_T, P, S\rangle$. V_N is a finite set of nonterminal symbols, V_T is a finite set of terminal symbols, P is a finite subset of $V_N\times(V_N\cup V_T)^*$, (i.e., production rules), and S in V_N is the start symbol. We assume S occurs only once in the left part of a production and never in the right part of any productions without loss of generality.

(2) Two disjoint sets Inh[A] and Syn[A] of attributes. They are associated with each nonterminal symbol $A\in V_N$ of G and called *inherited* and *synthesized attributes* respectively. When a is an attribute of A, that is, $a\in$Inh[A]$\cup$Syn[A], $a.A$ is called an *attribute occurrence*.

(3) A Σ-*algebra* D. Let $\mathcal{S}$ be a set of symbols called *sorts* which represent the types of attributes, and let $D=\{D_s\}_{s\in\mathcal{S}}$ be a $\mathcal{S}$-indexed family of sets which are called *primitive semantic domains*. For a string $w=s_1s_2\cdots s_n\in\mathcal{S}^*$, we define D^w as follows:

if $w\neq\varepsilon$ then $D^w=S_{s_1}\times D_{s_2}\times\cdots\times D_{s_n}$, and $D^\varepsilon=\{\varepsilon\}$.

If Σ is a family $\{\Sigma_{w,s}\}_{\langle w,s\rangle\in\mathcal{S}^*\times\mathcal{S}}$ of sets which is called $\mathcal{S}$-*sorted operator domain*, then a Σ-algebra D consists of a family $D=\{D_s\}_{s\in\mathcal{S}}$ together with a function f_D: $D^w\to D_s$ for each $f\in\Sigma_{w,s}$, $w\in\mathcal{S}^*$, $s\in\mathcal{S}$, which is called *primitive semantic function*. We associate with each nonterminal symbol $A\in V_N$ two strings $\bar{A}$ and $\underline{A}$ of sorts which represent the types of inherited and synthesized attributes respectively. The set $D^{\bar{A}}$ and $D^{\underline{A}}$ are called *inherited* and *synthesized attribute domains* of A respectively. It is assumed that $\bar{S}=\varepsilon$ and $\underline{S}\neq\varepsilon$.

(4) *Semantic functions* G_p and F_{pk}'s associated with each production $p\in P$ of G. For each production $p=A\to b_0B_1b_1\cdots B_rb_r$ in P, where $b_0, b_1, \cdots, b_r\in V_T^*$, $A, B_1, \cdots, B_r\in V_N$, and $r\geq 0$, there is a function G_p: $D^{\bar{A}\underline{A}\underline{B}_1\cdots\underline{B}_r}\to D^{\underline{A}}$ iff $\underline{A}\neq\varepsilon$, and for each k, $1\leq k\leq r$, there is a function F_{pk}: $D^{\bar{A}\underline{A}\underline{B}_1\cdots\underline{B}_r}\to D^{\bar{B}_k}$ iff $\bar{B}_k\neq\varepsilon$. These functions are composed of primitive semantic functions f_D of D and extended to have variables as their arguments in the sence of derived operations. Note that G_p (F_{pk}) can be viewed as $\underline{A}$-tuple ($\bar{B}_k$-tuple) of functions which define the synthesized (inherited) attribute occurrences in the left (right) part of the production. For a terminal production $A\to b_0$, there are no semantic functions of the form F_{pk}.

If the underlying primitive semantic functions f_D are *continuous* on flat CPO's (Complete Partially Ordered Sets),* then the semantic functions G_p and F_{pk}'s are also continuous and we call such an attribute grammar continuous. In the following, we deal with the continuous attribute grammars.

2.2 Example

Fig. 1 presents the Knuth's binary number attribute grammar which gives a precise definition of binary notation for numbers [13]. Here three kinds of attributes v, s, and l are used as follows:

1) synthesized attribute v stands for the 'value' of a binary number,
2) inherited attribute s for the 'scale' of each digit, and
3) synthesized attribute l for the 'length' of digits be-

$V_N=\{B, L, N\}$, $V_T=\{0, 1, .\}$,
Inh[B]$=\{s\}$, Syn[B]$=\{v\}$,
Inh[L]$=\{s\}$, Syn[L]$=\{v, l\}$,
Inh[N]$=\phi$, Syn[N]$=\{v\}$

production	semantic rules
1. $B\to 0$	$v.B=0$,
2. $B\to 1$	$v.B=2\uparrow s.B$,
3. $L\to B$	$v.L=v.B$, $s.B=s.L$, $l.L=1$
4. $L_1\to L_2B$	$v.L_1=v.L_2+v.B$, $l.L_1=l.L_2+1$, $s.L_2=s.L_1+1$, $s.B=s.L_1$,
5. $N\to L$	$v.N=v.L$, $s.L=0$,
6. $N\to L_1.L_2$	$v.N=v.L_1+v.L_2$, $s.L_2=-l.L_2$, $s.L_1=0$

Fig. 1 Binary number attribute grammar.

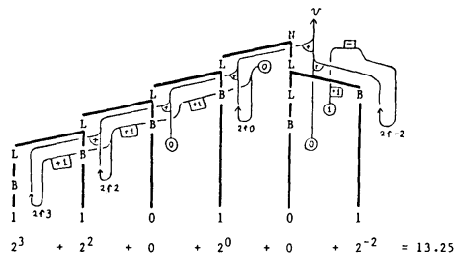


Fig. 2 An example of evaluation for binary number attribute grammar.

*Let D and R be CPO's. A function f: $D\to R$ is continuous if for all nonempty chains $X\subseteq D$, the set $f(X)$ has a lub in R and $f(\text{lub}_D X)=\text{lub}_R f(X)$. The notion of continuity on flat CPO's is related to the well-definedness of the values.

low a radix point.

The meaning of a given binary number can be expressed as the value of v at the root node N of its derivation tree. Fig. 2 shows the derivation tree of the binary notation 1101.01 and its attribute evaluation.

2.3 Circularities

Several classes of attribute grammars have been proposed in relation to circularity. Knuth [13] called an attribute grammar *'well-defined'* if all of its attributes can be defined at all nodes in any conceivable derivation tree, and gave an algorithm to determine whether a given attribute grammar is well defined or not. Bochmann [1] suggested the distinction between local and global circularities in the dependency relations of attributes. Considering potential dependency chains through subtrees, Kennedy and Warren [12] defined the *absolutely noncircular* class in which no augmented dependency graph contains a directed cycle and showed that the evaluation of attributes can be efficiently performed by a table driven *treewalk evaluator* for this class. Katayama [10] suggested that the evaluation of attributes in the absolutely noncircular class can be implemented efficient enough for practical use by translating the grammar into a program in a procedural language such as ALGOL, PASCAL, or PL/I. He also defined the *symbolwise noncircular* class in which no symbolwise dependency graph contains a directed cycle, and showed that the evaluation can be performed by a *simultaneous/parallel evaluator* which can evaluate the parallel attributes through a single procedure call. The symbolwise dependency graph is obtained from superposing IO graph and OI graph of each nonterminal symbol, where IO (OI) graph shows how synthesized (inherited) attributes are dependent on inherited (synthesized) attributes. These classes of attribute grammars have the following inclusion relation:

$$\text{symbolwise noncircular} \subset \text{absolutely noncircular} \subset \text{well-defined}.$$

Definition 1. Noncircular Attribute Grammars

An attribute grammar is *noncircular* if semantic rules are formulated in such a way that all synthesized attributes of the root node can be well-defined for any conceivable derivation tree.

This definition provides the weakest condition so as to evaluate the attributes of the root node and defines a wider class of attribute grammars than before, including the so called noncircular (i.e., well-defined) ones. For example, in the dependency relation of a conditional expression shown in Fig. 3(a), the attribute x has so far been considered to depend on not only the attribute b in 'then' clause but also c in 'else' clause together with a in 'if' condition. It is more precise to consider two dependency graphs (solid and dotted lines) as in Fig. 3(b), because the attribute x depends on either attribute b or c and not on both for any specific derivation tree. Such considerations are applicable to a structured type attribute. The above definition of noncircularity claims that there exist no directed cycles in any derivation tree if we adopt such a proper dependency.

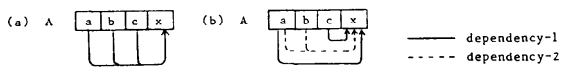


Fig. 3 Dependency relations of a conditional expression $x \leftarrow$ **if** a **then** b **else** c.

3. Denotational Semantics of Attribute Grammars

Although several methods for efficient attribute evaluation have been proposed as mentioned in Section 1 and 2.3, some of them requires the augmented dependency graphs and others are applicable only for severely restricted class of attribute grammars.

We now consider an evaluation method by going back to Knuth's original definition [13]. The principle is simple and stated as follows. A set of attributes directly necessary for an attribute evaluation is defined by its semantic function. Thus it is possible to evaluate the synthesized attributes at the root node of a derivation tree by traversing the dependency relations in the opposite direction. Saarinen [16] applied this concept to the treewalk evaluator [12] for the class of absolutely noncircular attribute grammars. Katayama also obtained an output-oriented evaluator for this class [10].

The objective of this paper is to show that an output-oriented evaluation procedure for any noncircular (see Definition 1) attribute grammars can be directly obtained from the description of a given attribute grammar without constructing the augmented dependency gaphs. What is necessary to realize our evaluator is (1) a semantic definition of attribute values by means of functions, that is, a *denotational semantics* of attribute grammars, and (2) a programming language with a mechanism for evaluating functions in an output-oriented way ('call by name' parameter passing mechanism).

In this section, we first define a denotational semantics of attribute grammars based on the least fixpoint semantics and then propose an output-oriented evaluation method together with a simple example.

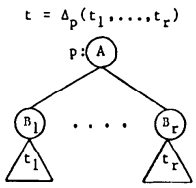
3.1 Least Fixpoint Semantics of Attribute Grammars

The semantics of a derivation tree is defined as a set of attribute values assigned to its nodes. These attribute values can be obtained as the least fixpoint solution of equations about the attributes. Here, we introduce the least fixpoint semantics presented by Chirica and Martin.

Definition 2. Set of Derivation Tree $T_{G,A}$

Let $T_{G,A}$ be a set of all A-rooted derivation trees for a context-free grammar G. It is defined formally as follows.

$$T_{G,A} = \{\Delta_p[t_1, \cdots, t_r] \mid p \in P,\ p = A \to b_0 B_1 b_1 \cdots B_r b_r,\ r > 0\} \cup \{\Delta_p \mid p \in P,\ p = A \to b_0\},$$



where $t_i \in T_{G,B_i}$, $1 \le i \le r$, and $\Delta = \{\Delta_p | p \in P\}$ is a set of symbols in one-to-one correspondence with P.

For a production $p = A \to b_0 B_1 \cdots B_r b_r$, let x_0 and y_0 be variables ranging over $D^{\underline{A}}$ and $D^{\underline{A}}$ respectively, and let x_i and y_i be variables over $D^{\underline{B_i}}$ and $D^{\underline{B_i}}$, $1 \le i \le r$. For any A-rooted derivation tree t in $T_{G,A}$, let w_t be the vector of variables x_i and y_i, recursively defined as follows:

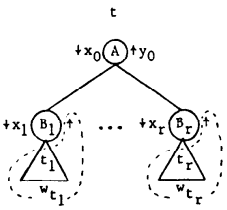
if $r=0$ (i.e., $t=\Delta_p$), then $w_t = y_0$;

if $r>0$ then $w_t = \langle y_0, x_1, \cdots, x_r, w_{t_1}, \cdots, w_{t_r} \rangle$.

Let $D^t = D^{\underline{A}} \times D^{\underline{B_1} \cdots \underline{B_r}} \times D^{t_1} \times \cdots \times D^{t_r}$, $r \ge 0$, then an element $\langle x, w \rangle \in D^{\underline{A}} \times D^t$ represents a set of values of all attributes in the derivation tree t.

An equation $w_t = H_t(x_0, w_t)$ associated with t is introduced to represent the relationship among the attribute values assigned to the nodes of t, and is recursively defined by

$$y_0 = G_p(x_0, y_0, x_1, y_1, \cdots, x_r, y_r)$$
$$x_k = F_{pk}(x_0, y_0, x_1, y_1, \cdots, x_r, y_r), \quad 1 \le k \le r$$
$$w_{t_k} = H_{t_k}(x_k, w_{t_k}), \quad 1 \le k \le r.$$

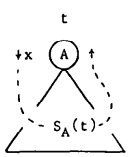


G_p and F_{pk}'s are the semantic functions introduced in Section 2.1. If $r=0$ then the above equation reduces to $y_0 = G_p(x_0, y_0)$.

The semantics of an attribute grammar is defined by the synthesized attributes at the root nodes of its derivation trees. Let pr_t: $D^t \to D^{\underline{A}}$, be the projection function extracting the y_0 component of

$$w_t = \langle y_0, x_1, \cdots, x_r, w_{t_1}, \cdots, w_{t_r} \rangle, \quad r \ge 0,$$

i.e., $pr_t(w_t) = y_0$. Then we can regard the meaning of the derivation tree t in $T_{G,A}$ as the *least fixpoint* (LFP) *semantic function* $S_A(t)$: $D^{\underline{A}} \to D^{\underline{A}}$ defined by



$$S_A(t) = \lambda x \in D^{\underline{A}}.\, pr_t(\mu w \in D^t.\, H_t(x, w)).$$

Note that DeBakker's μ-operator [4] is used as a convenient way of denoting the least fixpoint; namely, let f be a continuous function from D to D, then

$$\mu x.f(x) = fix_D(f), \text{ where } fix_D \in [[D \to D] \to D]$$
$$= \mathrm{lub}_{0 \le i} f^i(\perp_D)$$

where $\perp_D$ is the undefined value in D. If $\overline{A} = \varepsilon$ then $S_A(t)$ is the constant function

$$S_A(t) = pr_t(\mu w \in D^t.\, H_t(w)).$$

Remark. In any continuous attribute grammar, D^t is a finite height CPO for any derivation tree t. In particular, the height of D^t for $t \in T_{G,S}$ equals the number $|t|$ of attributes of all nodes in t. Therefore, the semantics of any tree t is computable by the following expression:

$$S_S(t) = pr_t(\mathrm{lub}_{1 \le i \le |t|} H_t^i(\perp)),$$

Where $\perp$ is the undefined value in D^t. The above expression shows that, for any derivation tree t in the non-circular attribute grammar (see Definition 1), at least one attribute on t can be defined with every application of H_t. Therefore, it is possible to evaluate the values of the attributes step by step although it is not efficient.

3.2 Denotational Semantics of Attribute Grammars in terms of Output-Oriented Evaluation Functions

The LFP semantic function $S_A(t)$ of Section 3.1 returns the values of all the synthesized attributes when given the values of the inherited attributes at the root of t. In this section, we introduce a function associated with each pair of a nonterminal symbol and one of its synthesized attribute, describe its construction algorithm, and show that its normal order (call by name) evaluation is output-oriented.

Notation. Let $at_{A,a}$: $D^{\underline{A}} \to D_s$ be the projection function extracting the value of the attribute a in Syn[A] from $y \in D^{\underline{A}}$, where s is the sort of a. For a derivation tree $t = \Delta_p[t_1, \cdots, t_r]$ and $r \ge 0$, let prod(t) be the production p at the root node of t, i.e., prod(t) $= p$.

Definition 3. Output-Oriented Evaluation Function $q_{A,a}$

Let $A \in V_N$, $a \in$ Syn[A], and s be the sort of a. For $t = \Delta_p[t_1, \cdots, t_r] \in T_{G,A}$, where $p = A \to b_0 B_1 b_1 \cdots B_r b_r$, $r \ge 0$, and $x \in D^{\underline{A}}$, we define a function $q_{A,a}$: $T_{G,A} \to [D^{\underline{A}} \to D_s]$ as follows.

$$q_{A,a}(t)(x) = at_{A,a}(\mu y \in D^{\underline{A}}.\, G_{\mathrm{prod}(t)}(x, y, z_1, Q_{B_1}(t_1)(z_1), \cdots, z_r, Q_{B_r}(t_r)(z_r))),$$

where

(1) $\langle z_1, \cdots, z_r \rangle = \mu x_1 \cdots x_r \in D^{\underline{B_1} \cdots \underline{B_r}}.\, F_{\mathrm{prod}(t)}(x, y, x_1, Q_{B_1}(t_1)(x_1), \cdots, x_r, Q_{B_r}(t_r)(x_r))$,

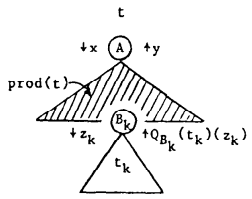
(2) $F_{\mathrm{prod}(t)} = \langle F_{\mathrm{prod}(t)1}, \cdots, F_{\mathrm{prod}(t)r} \rangle$, and

(3) if $\langle a_1, \cdots, a_n \rangle$ is a vector of attributes corresponding to sorts $\underline{A}$ for $a_i \in$ Syn[A], $1 \le i \le n$, then

$$Q_A(t)(x) = \langle q_{A,a_1}(t)(x), \cdots, q_{A,a_n}(t)(x) \rangle.$$

Obviously, if $r=0$ then

$$q_{A,a}(t)(x)=\mathrm{at}_{A,a}(\mu y\in D^{\underline{A}}.G_{\mathrm{prod}(t)}(x, y)).$$



$q_{A,a}(t)$ defined above returns the value of the synthesized attribute $a\in\mathrm{Syn}[A]$ associated with the root node of $t\in T_{G,A}$ when given its inherited attribute values. The following theorem shows the correctness of the attribute evaluation based on $q_{A,a}$, and guarantees the well-definedness for any noncircular attribute grammars.

Theorem 1. For $t\in T_{G,A}$, and $x\in D^{\underline{A}}$,

$$\mathrm{at}_{A,a}(S_A(t)(x))=q_{A,a}(t)(x).$$

Proof. By structural induction on the derivation tree t.

(i) Let $p=A\to b_0$ and $t=\Delta_p\in T_{G,A}$. If $x\in D^{\underline{A}}$ then

$$\begin{aligned}S_A(t)(x)&=\mathrm{pr}_t(\mu w\in D^t.H_t(x, w)) && \text{(Definition of } S_A)\\ &=\mathrm{pr}_t(\mu y\in D^{\underline{A}}.G_p(x, y)) && \text{(Definition of } D^t, H_t)\\ &=\mu y\in D^{\underline{A}}.G_p(x, y). && \text{(Definition of } \mathrm{pr}_t)\end{aligned}$$

Therefore

$$\mathrm{at}_{A,a}(S_A(\Delta_p)(x))=q_{A,a}(\Delta_p)(x). \quad \text{(Definition of } q_{A,a})$$

(ii) Let $p=A\to b_0B_1\cdots B_rb_r$, $r>0$, $t=\Delta_p[t_1,\cdots, t_r]\in T_{G,A}$, and $t_k\in T_{G,B_k}$, $1\le k\le r$. Let $x\in D^{\underline{A}}$ then, by Definition of S_A,

$$S_A(t)(x)=\mathrm{pr}_t(\mu w\in D^t.\ H_t(x, w)).$$

The expression $\mu w\in D^t.H_t(x, w)$ is, by Definition of H_t, the least fixpoint solution of the following equations.

$$y=G_p(x, y, x_1, y_1,\cdots, x_r, y_r) \tag{1}$$

$$x_k=F_{pk}(x, y, x_1, y_1,\cdots, x_r, y_r), \quad 1\le k\le r \tag{2}$$

$$w_{t_k}=H_{t_k}(x_k, w_{t_k}), \quad 1\le k\le r. \tag{3}$$

The above equations can be solved iteratively as follows. First, taking the projection (pr_{t_k}) of w_{t_k} yields the solution for y_k.

$$\begin{aligned}y_k&=\mathrm{pr}_{t_k}(\mu w.H_{t_k}(x_k, w))\\ &=S_{B_k}(t_k)(x_k)\\ &=\langle \mathrm{at}_{B_k,a_1}(S_{B_k}(t_k)(x_k)),\cdots, \mathrm{at}_{B_k,a_n}(S_{B_k}(t_k)(x_k))\rangle\\ &\qquad\qquad \text{(Definition of } \mathrm{at}_{B_k,a_i})\\ &=\langle q_{B_k,a_1}(t_k)(x_k),\cdots, q_{B_k,a_n}(t_k)(x_k)\rangle\\ &\qquad\qquad \text{(induction hypothesis)}\\ &=Q_{B_k}(t_k)(x_k) \qquad \text{(Definition of } Q),\end{aligned}$$

where $\langle a_1,\cdots, a_n\rangle$, $a_i\in\mathrm{Syn}[B_k]$, $1\le i\le n$, is a vector of the attribute symbols corresponding to sorts $\underline{B_k}$.

In order to solve the equation (2) for x_k, substituting $Q_{B_k}(t_k)(x_k)$ for y_k yields

$$\begin{aligned}&\langle z_1,\cdots, z_r\rangle\\ &=\mu x_1\cdots x_r.\ F_p(x, y, x_1, Q_{B_1}(t_1)(x_1),\cdots, x_r, Q_{B_r}(t_r)(x_r)).\end{aligned}$$

Substituting z_k and $Q_{B_k}(t_k)(z_k)$ for x_k and y_k respectively in (1) and solving (1) for y yields

$$v=\mu y.\ G_p(x, y, z_1, Q_{B_1}(t_1)(z_1),\cdots, z_r, Q_{B_r}(t_r)(z_r)).$$

The above solution is called an *iterative solution* and it is shown in [17] that the simultaneous and iterative solutions are equal. Therefore

$$\begin{aligned}&\mathrm{at}_{A,a}(S_A(t)(x))\\ &=\mathrm{at}_{A,a}(v)\\ &=\mathrm{at}_{A,a}(\mu y.\ G_p(x, y, z_1, Q_{B_1}(t_1)(z_1),\cdots, z_r, Q_{B_r}(t_r)(z_r))\\ &=q_{A,a}(t)(x) \qquad \text{(Definitin of } q_{A,a})\ \square\end{aligned}$$

Least fixpoint operations in $q_{A,a}$ can be reduced by a single μ-operator as follows.

$$\begin{aligned}&q_{A,a}(t)(x)\\ &=(\lambda y x_1\cdots x_r\in D^{\underline{A}\underline{B}_1\cdots\underline{B}_r}.\ G_{\mathrm{prod}(t),a}(x, y, x_1, Q_{B_1}(t_1)(x_1),\cdots,\\ &\qquad x_r, Q_{B_r}(t_r)(x_r)))\\ &\quad(\mu z_0z_1\cdots z_r\in D^{\underline{A}\underline{B}_1\cdots\underline{B}_r}.\ V_{\mathrm{prod}(t)}(x, z_0, z_1, Q_{B_1}(t_1)(z_1),\cdots,\\ &\qquad z_r, Q_{B_r}(t_r)(z_r))),\end{aligned}$$

where $G_{\mathrm{prod}(t),a}$ is the semantic function for the synthesized attribute $a\in\mathrm{Syn}[A]$ associated with a production $\mathrm{prod}(t)=A\to b_0B_1b_1\cdots B_rb_r$, and

$$V_{\mathrm{prod}(t)}=\langle G_{\mathrm{prod}(t)}, F_{\mathrm{prod}(t)1},\cdots, F_{\mathrm{prod}(t)r}\rangle.$$

The following algorithm presents a construction of such a function $q_{A,a}$ directly from the description of a given attribute grammar.

Construction Algorithm of $q_{A,a}$

The arguments of $q_{A,a}$ are a derivation tree $t\in T_{G,A}$ and inherited attributes $x\in D^{\underline{A}}$, so $q_{A,a}$ is rewritten in the following form.

$$\begin{aligned}q_{A,a}=\lambda t.\ \lambda x.\ [&\mathrm{prod}(t)=p1\to \mathrm{EXP}_{p1,a},\\ &\vdots\\ &\mathrm{prod}(t)=pk\to \mathrm{EXP}_{pk,a}],\end{aligned}$$

where $p1,\cdots, pk$ are the productions with left part nonterminal symbol A. $\mathrm{EXP}_{p,a}$ is the expression to compute the value of the attribute a based on the production p, and is composed of 1) semantic functions G_p and F_{pk}'s, 2) evaluation functions for the output attributes of right part nonterminal symbols in p, and 3) formal parameters t and x of $a_{A,a}$. In the following, we present this process with a simple attribute grammar shown in Fig. 4(a).

[step 1] Let $p=A\to b_0B_1\cdots B_rb_r$, $r\ge 0$, where $b_0,\cdots, b_r\in V_T^*$ and $A, B_1,\cdots, B_r\in V_N$. Construct the *dependency graph* D_p for the production p, which is defined by

$$D_p=(V_p, E_p)$$

rhere (1) the *node set* V_p is the set of all attribute occurrences of p and (2) an edge $\langle v_1, v_2\rangle$ is in the *edge set*

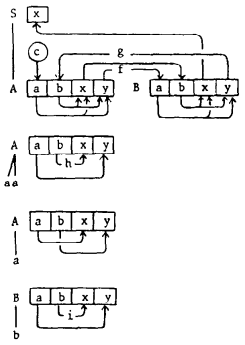
1. S→*AB*
   x.S=*x.B*
   a.A=*c*
   b.A=*g*(*y.B*)
   a.B=*f*(*y.A*)
   b.B=*x.A*

2. A→*aa*
   x.A=*h*(*b.A*)
   y.A=*a.A*

3. A→*a*
   x.A=*a.A*
   y.A=*b.A*

4. B→*b*
   x.B=i(*b.B*)
   y.B=*a.B*

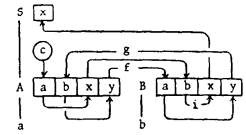
(a) Production and semantic rules.



(b) Dependency graphs.

$q_{S,x}=\lambda t.(\lambda u_1u_2u_3.q_{B,x}(t_2)(u_2,u_3))$
$(\mu v_1v_2v_3.\langle g(q_{B,y}(t_2)(v_2,v_3)),$
$f(q_{A,y}(t_1)(c,v_1)),$
$q_{A,x}(t_1)(c,v_1)\rangle),$
$q_{A,x}=\lambda t.\lambda ab.[\text{prod}(t)=2\to h(b),$
$\text{prod}(t)=3\to a],$
$q_{A,y}=\lambda t.\lambda ab.[\text{prod}(t)=2\to a,$
$\text{prod}(t)=3\to b],$
$q_{B,x}=\lambda t.\lambda ab.[\text{i}(b)],$
$q_{B,y}=\lambda t.\lambda ab.[a]$

(c) Evaluation functions.



(d) A derivation tree $t=\Delta_1[\Delta_3\Delta_4]$.

Fig. 4 An example of construction of output-oriented evaluation functions.

E_p if (a) v_1 is required to evaluate v_2 or (b) v_1 and v_2 are the inherited and synthesized attribute occurrences of the right part of p respectively. In this graph, the dependency relation of a conditional expression, for example, is considered as shown in Fig. 3(a). That is, the attribute x depends on a, b, and c.

D_p shows how attribute occurrences in p are interrelated when all attribute of B_k, $1\le k\le r$ are reduced to a single one. Fig. 4(b) illustrates the dependency graphs of the example grammar.

[step 2] To construct an expression $\text{EXP}_{p,a}$ where $p=A\to b_0B_1\cdots B_rb_r$ and $a\in\text{Syn}[A]$, traverse the directed edges in D_p starting from the node $a.A$ in the opposite direction. In this process, if an attribute occurrence is encountered which is either a synthesized one of A or an inherited one of B_k, $1\le k\le r$, then we call it the *defined attribute occurrence* of $a.A$.

[step 3] As the dependency graph D_p is constructed based on the semantic functions in p, at most only the values of attribute occurrences $u_1,\cdots,u_m$ which are the defined attribute occurrences of $a.A$ are necessary to evaluate the value of $a.A$ when given a derivation tree t and its inherited attribute x. Therefore, let $G'_{p,a}(t)$: $D^{\bar{A}\underline{A}\bar{B}_1\cdots\bar{B}_r}\to D_s$, $V'_p(t)$: $D^{\bar{A}\underline{A}\bar{B}_1\cdots\bar{B}_r}\to D^{\underline{A}\bar{B}_1\cdots\bar{B}_r}$ be the following functions

$$G'_{p,a}(t)(x,y,x_1,\cdots,x_r)$$
$$=G_{p,a}(x,y,x_1,Q_{B_1}(t_1)(x_1),\cdots,x_r,Q_{B_r}(t_r)(x_r)),$$

where

(1) $t\in T_{G,A}$, $x\in D^{\bar{A}}$, and s is the sort of a,

(2) $\langle y,x_1,\cdots,x_r\rangle$
$=\mu z_0z_1\cdots z_r\in D^{\underline{A}\bar{B}_1\cdots\bar{B}_r}.\ V'_p(t)(x,z_0,z_1,\cdots,z_r)$,

and

(3) $V'_p(t)(x,z_0,z_1,\cdots,z_r)$
$=V_p(x,z_0,z_1,Q_{B_1}(t_1)(z_1),\cdots,z_r,Q_{B_r}(t_r)(z_r))$,

then $G'_{p,a}(t)(x,y,x_1,\cdots,x_r)$ is rewritten as follows.

$$U_{p,a}(t)(x,v_1,\cdots,v_m)$$

where

$$\langle v_1,\cdots,v_m\rangle$$
$$=\mu v_1\cdots v_m\in D^{s_1\cdots s_m}.\ \langle V_{p,1}(t)(x,v_1,\cdots,v_m),\cdots,$$
$$V_{p,m}(t)(x,v_1,\cdots,v_m)\rangle,$$

and s_i, $1\le i\le m$ is the sort of u_i, i.e., the sort of the defined attribute occurrence of $a.A$.

The function $U_{p,a}(t)$: $D^{\bar{A}s_1\cdots s_m}\to D_s$ is obtained from $G'_{p,a}$ by assigning $\perp$'s to attribute occurrences of its arguments except for $x,u_1,\cdots,u_m$, and in the same way $V_{p,i}(t)$: $D^{\bar{A}s_1\cdots s_m}\to D_{s_i}$, $1\le i\le m$ is obtained from the semantic function which defines the value of u_i in V'_p. Note that the argument y or x_k, $1\le k\le r$ of $G'_{p,a}$ is a tuple of attribute occurrences and the above assignment of $\perp$'s to their occurrences does not essentially influence the evaluation of $a.A$. Thus $\text{EXP}_{p,a}$ is written in the following form.

$$\text{EXP}_{p,a}=(\lambda u_1\cdots u_m.\ U_{p,a}(t)(x,u_1,\cdots,u_m))$$
$$(\mu v_1\cdots v_m.\ \langle V_{p,1}(t)(x,v_1,\cdots,v_m),\cdots,$$
$$V_{p,m}(t)(x,v_1,\cdots,v_m)\rangle).$$

That is, we construct a tuple of defined attribute occurrences of $a.A$ and substitute its least fixpoint solution into $U_{p,a}(t)$.

In the attribute grammar shown in Fig. 4(a), since only production 1 is the one with left part nonterminal symbol S and S has no inherited attribute, an evaluation function $q_{S,x}$ is written as

$$q_{S,x}=\lambda t.\text{EXP}_{1,x}.$$

By traversing the dependency graph D_1 starting from the node $x.S$ we know that the defined attribute occurrences of $x.S$ are $a.A$, $b.A$, $a.B$, and $b.B$, so we introduce lambda variables u_0, u_1, u_2, u_3 for them into $\text{EXP}_{1,x}$ and we have

$$\text{EXP}_{1,x}=(\lambda u_0u_1u_2u_3.q_{B,x}(t_2)(u_2,u_3))$$
$$(\mu v_0v_1v_2v_3.\langle c,\ g(q_{B,y}(t_2)(v_2,v_3)),$$
$$f(q_{A,y}(t_1)(v_0,v_1)),$$
$$q_{A,x}(t_1)(v_0,v_1)\rangle).$$

Especially, if an element of the tuple is, for example, a constant value then it can be removed from the tuple and the above expression is rewritten as follows.

$$\text{EXP}_{1,x}=(\lambda u_1u_2u_3.q_{B,x}(t_2)(u_2,u_3))$$
$$(\mu v_1v_2v_3.\langle g(q_{B,y}(t_2)(v_2,v_3)),$$
$$f(q_{A,y}(t_1)(c,v_1)),$$
$$q_{A,x}(t_1)(c,v_1)\rangle).$$

Such a removable element is the one which does not form a directed cycle in D_p and is found in the traversing process of step 2.

The remaining process of this example grammar are exactly as above, and the complete definition is given in Fig. 4(c).

[step 4] In case of implementing a function $q_{A,a}$ by a programming language, an expression with μ-operator can be represented by a set of recursive functions as follows. When an expression $\mathrm{EXP}_{p,a}$ is given by

$$\mathrm{EXP}_{p,a}=(\lambda u_1\cdots u_m.U_{p,a}(t)(x, u_1,\cdots, u_m))\ (\mu v_1\cdots v_m.\langle V_{p,1}(t)(x, v_1,\cdots, v_m),\cdots, V_{p,m}(t)(x, v_1,\cdots, v_m)\rangle),$$

then it can be rewritten by a set of recursive functions as follows.

$$\mathrm{EXP}_{p,a}=U_{p,a}(t)(x, h_1(t)(x),\cdots, h_m(t)(x)),$$

$$h_i=\lambda t.\ \lambda x.\ V_{p,i}(t)(x, h_1(t)(x),\cdots, h_m(t)(x)),\quad 1\le i\le m.$$

Namely, a tuple $\langle h_1(t)(x),\cdots, h_m(t)(x)\rangle$ is the least fixpoint solution of the expression

$$\mu v_1\cdots v_m.\langle V_{p,1}(t)(x, v_1,\cdots, v_m),\cdots, V_{p,m}(t)(x, v_1,\cdots, v_m)\rangle.$$

The set of recursive functions for $q_{S,x}$ in the above example are given by

$$q_{S,x}=\lambda t.\ q_{B,x}(t_2)(h_2(t), h_3(t))$$
$$h_1=\lambda t.\ g(q_{B,y}(t_2)(h_2(t), h_3(t)))$$
$$h_2=\lambda t.\ f(q_{A,y}(t_1)(c, h_1(t)))$$
$$h_3=\lambda t.\ q_{A,x}(t_1)(c, h_1(t)).$$

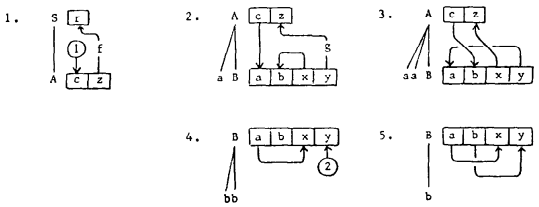
Remark. The attribute grammar in the above example is not-well-defined (see Section 2.3), because there is a directed cycle ($y.A\to a.B\to y.B\to b.A\to y.A$) in the derivation tree $t=\Delta_1[\Delta_3\Delta_4]$ shown in Fig. 4(d). In this paper, however, we regard the meaning of the derivation tree as the value of the synthesized attribute ($x.S$) at its root node, and so it is possible to define the semantics of this grammar as there are no cyclic dependencies containing $x.S$.

Output-Oriented Evaluation

An attribute $a\in\mathrm{Syn}[S]$ of start symbol S is computable according to the computability of $S_S(t)$, $t\in T_{G,S}$, which is shown in Section 3.1. In particular, if the attribute grammar under consideration is noncircular, then there exists a *normal form expression* for the value of the attribute a in lambda notation [18]. Thus, by Church-Rosser Theorem II, there exists a *normal order reduction* to obtain its value [3], and evaluation of our recursive functions with *'call by name' parameter passing mechanism* is *output-oriented.*

In the rest of this section, we compare our output-oriented evaluation with the method of Katayama [10] for the absolutely noncircular attribute grammars.

The attribute grammar shown in Fig. 5(a) is absolutely



(a) Production and semantic rules.

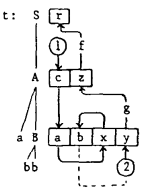
```
proc Az(c.A, t, var z.A);
  case prod(t) of
  2: begin
       a.B←c.A;
       call Bx(a.B, t1, x.B);
       b.B←x.B;
       call By(b.B, t1, y.B);
       z.A←g(y.B)
       end;
  3: begin
       b.B←c.A;
       call By(b.B, t1, y.B);
       a.B←y.B;
       call Bx(a.B, t1, x.B);
       z.A←x.B
       end
  end
end;

proc Sr(t, var r.S);
  c.A←1;
  call Az(c.A, t1, z.A);
  r.S←f(z.A)
end;

proc Bx(a.B, t, var x.B);
  x.B←a.B
end;

proc By(b.B, t, var y.B);
  case prod(t) of
  4: y.B←2;
  5: y.B←b.B
  end
end
```

(b) Translation to recursive procedures.



(c) A derivation tree $t=\Delta_1[\Delta_2[\Delta_4]]$.

$\mathrm{Sr}=\lambda t.[f(\mathrm{Az}(t_1)(1))]$,
$\mathrm{Az}=\lambda t.\lambda c.[\mathrm{prod}(t)=2\to g(\mathrm{By}(t_1)(\mathrm{c}, \mathrm{u}(t)(c))),$
$\quad \mathrm{prod}(t)=3\to \mathrm{Bx}(t_1)(\mathrm{v}(t)(c), c)]$,
$\mathrm{u}=\lambda t.\lambda c.[\mathrm{Bx}(t_1)(\mathrm{c}, \mathrm{u}(t)(c))]$,
$\mathrm{v}=\lambda t.\lambda c.[\mathrm{By}(t_1)(\mathrm{v}(t)(c), c)]$,
$\mathrm{Bx}=\lambda t.\lambda ab.[a]$,
$\mathrm{By}=\lambda t.\lambda ab.[\mathrm{prod}(t)=4\to 2,$
$\quad \mathrm{prod}(t)=5\to \mathrm{b}]$

(d) Translation to recursive functions.

Fig. 5 Two evaluation methods for an attribute grammar.

noncircular and it can be translated into the set of recursive procedures as Fig. 5(b) in the method of Katayama. In this method, the attribute grammar is completely compiled into procedures in the syntax analysis phase and the semantic analysis phase can concentrate only on the attribute evaluation. This carries out an efficient evaluation, however, it requires augmented dependency graphs and is applicable only to absolutely noncircular attribute grammars, although this class is amply wide for practical applications. The strategy of complete compilation may have another problem in connection with conditional expression. That is, there are cases where it executes unnecessary computations. Consider, for example, a derivation tree shown in Fig. 5(c). As it does not investigate whether

there is a dependency indicated by the dotted line in semantic analysis phase, it evaluates the attribute b, which is not necessary for this derivation tree.

On the other hand, the set of recursive functions $q_{A,a}$ for this attribute grammar is constructed as in Fig. 5(d), where the name of each function is represented by the pair of a nonterminal and one of its synthesized attribute symbols. The normal order (call by name) evaluation for this derivation tree is the following and it does not compute b.

$$\begin{aligned} S_r(\Delta_1[\Delta_2[\Delta_4]]) &= f(A_z(\Delta_2[\Delta_4])(1)) \\ &= f(g(B(\Delta_4)(1, u(\Delta_2[\Delta_4])(1)))) \\ &= f(g(2)). \end{aligned}$$

As known from this example, our method directly* interprets the description of a given attribute grammar and evaluates only the necessary attributes using information found in the semantic analysis phase, i.e., derivation trees, and semantic functions which may be conditional expressions.

*It means that the set of recursive functions can be constructed using the dependency graphs D_p obtained at the cost of $0(|G|)$.

4. Implementation of Evaluation Functions by LISP

In this section, we construct the evaluation functions for the binary number attribute grammar shown in Section 2.2, and implement them by the programming

$$\begin{aligned} Bv &= \lambda t.\lambda s.\mathrm{prod}(t)=1 \rightarrow 0, \\ &\quad \mathrm{prod}(t)=2 \rightarrow 2\uparrow s], \\ Lv &= \lambda t.\lambda s.[\mathrm{prod}(t)=3 \rightarrow Bv(t_1)(s), \\ &\quad \mathrm{prod}(t)=4 \rightarrow Lv(t_1)(s+1)+Bv(t_2)(s)], \\ Ll &= \lambda t.\lambda s.\mathrm{prod}(t)=3 \rightarrow 1, \\ &\quad \mathrm{prod}(t)=4 \rightarrow Ll(t_1)(s+1)+1], \\ Nv &= \lambda t.[\mathrm{prod}(t)=5 \rightarrow Lv(t_1)(0), \\ &\quad \mathrm{prod}(t)=6 \rightarrow Lv(t_1)(0)+Lv(t_2)(W(t))], \\ W &= \lambda t.[-Ll(t_2)(W(t))] \end{aligned}$$

(a) Recursive functions for binary number attribute grammar.

```
DEFLIST(((BV(LAMBDA (X Y)
        (COND((PROD X Y 1)(QUOTE 0))
              ((PROD X Y 2)(LIST(QUOTE EXPT)(QUOTE 2)
                                (SCALE X Y)))))))FEXPR)
DEFLIST(((LV(LAMBDA (X Y)
        (COND((PROD X Y 3)(BV(SUBTREE X Y 1)(SCALE X Y)))
              ((PROD X Y 4)(LIST(QUOTE PLUS)
                                (LV(SUBTREE X Y 1)(ADD1(SCALE X Y)))
                                (BV(SUBTREE X Y 2)(SCALE X Y)))))))FEXPR)
DEFLIST(((LL(LAMBDA (X Y)
        (COND((PROD X Y 3) 1)
              ((PROD X Y 4)(ADD1 (LL(SUBTREE X Y 1)
                                     (ADD1(SCALE X Y))))))
        )))FEXPR)
DEFLIST(((NV(LAMBDA (X Y)
        (COND((PROD X Y 5)(LV(SUBTREE X Y 1) 0))
              ((PROD X Y 6)(LIST(QUOTE PLUS)
                                (LV(SUBTREE X Y 1) 0)
                                (LV(SUBTREE X Y 2)
                                   (W(TREE X Y)))))))))FEXPR)
DEFLIST(((W(LAMBDA (X Y)
        (LIST(QUOTE MINUS)(LL(SUBTREE X Y 2)
                             (W(TREE X Y)))))))FEXPR)
DEFINE(((PROD(LAMBDA (X Y N)
          (EQUAL(CAR(TREE X Y)) N)))))
DEFINE(((SCALE(LAMBDA (X Y)
          (EVAL(CADR X) Y)))))
DEFINE(((TREE(LAMBDA (X Y)(EVAL (CAR X) Y)))))
DEFINE(((SUBTREE(LAMBDA (X Y N)
          (ST(TREE X Y) N)))))
DEFINE(((ST(LAMBDA (X N)
          (COND((ZEROP N)(CAR X))
                (T(ST(CDR X)(SUB1 N)))))))))
```

(b) Translation to LISP functions.

```
F EVAL
A  ((NV(QUOTE(6(4(4(4(3(2))(2))(1))(2))(4(3(1))(2))))NIL)
   =(PLUS(PLUS(PLUS(PLUS(EXPT 2 3)(EXPT 2 2))0)(EXPT 2 0))(PLUS 0
    (EXPT 2(MINUS 2))))
```

(c-1) Functional notation.

```
F EVAL
A  ((EVAL(NV(QUOTE(6(4(4(4(3(2))(2))(1))(2))(4(3(1))(2)))))NIL)NIL)
   =1.325000E+01
```

(c-2) Decimal notation.

(c) An example of evaluation for the derivation tree shown in Fig. 2.

Fig. 6 An implementation of evaluation functions for binary number attribute grammar.

language LISP.

Fig. 6(a) shows the set of recursive functions. There exists the least fixpoint representation in the expression $EXP_{6,v}$ of the function *Nv* and we replace it by the recursive function *W*.

These recursive functions can be directly implemented by LISP, as shown in Fig. 6(b). In this program, each LISP function corresponds to a recursive function, and has the FEXPR property so as to perform the normal order (call by name) evaluations. The LISP function with FEXPR property is interpreted as a function with two parameters: the first one is the list of given arguments and the second one is the association list containing the calling environment of its function. These functions return list expressions in functional notation. A derivation tree is represented by a list of production numbers. An example of evaluation for the derivation tree

$$t=\Delta_6[\Delta_4[\Delta_4[\Delta_3[\Delta_2]\Delta_2]\Delta_1]\Delta_2]\Delta_4[\Delta_3[\Delta_1]\Delta_2]]$$

of the binary number 1101.01 (see Fig. 2) is shown in Fig. 6(c). Fig. 6(c-1) represents the meaning of its binary number in terms of functional notation and the expression in the decimal notation is obtained by further evaluating it as shown in Fig. 6(c-2).

Remark. In general, the normal order (i.e., call by name) evaluation is slower than *applicative order*. The former evaluates operands as many times as necessary. The latter, on the other hand, evaluates them only once before they are substituted into the body of the operator. However, the normal order evaluation could be performed with less computation overhead by an *associative computation mechanism* which is realized, for example, in HLISP [6].

5. Conclusion

We have formalized attribute grammars by denotational semantics and proposed a method of attribute evaluation based on it. The denotational semantics of an attribute grammar is defined by the least fixpoint semantics among attributes assigned to the nodes of a derivation tree and it is realized by a set of recursive functions which perform the required evaluations. This semantic definition is more general than Knuth's original formulation, and is able to define the meaning of circular definitions by viewing them as recursive definitions [18].

The attribute evaluation method using the denotational semantics can handle a wider class of attribute grammars than before, including well-defined [13] ones. Our evaluation functions can evaluate only necessary attributes to determine the meaning of the derivation tree, and can be derived directly from the description of a given attribute grammar without resorting to augmented dependency graphs.

A verification procedure for atribute grammars has been proposed by Katayama and Hoshino [11]. This procedure, however, is not applicable to noncircular attribute grammars defined in this paper. We can apply our denotational semantics to the problem of proving the properties of attribute grammars and this will be the subject of a future paper.

Acknowledgements

The authors wish to thank Professor Hajime Enomoto for his valuable advice, and Naoki Yonezaki and Toshio Miyachi for careful reading of the original manuscript.

References

1. Bochmann, G. V. Semantic Evaluation from Left to Right, *CACM*, **19**, 2 (1976), 55–62.
2. Chirica, L. M. and Martin, D. F. An Order-Algebraic Definition of Knuthian Semantics, *Math. Syst. Th.*, 13 (1979), 1–27.
3. Curry, H. B. and Feys, R. Combinatory Logic, I, North-Holland, Amsterdam (1968).
4. DeBakker, J. W. Recursive Procedures, Mathematical Center, Amsterdam, Report No. MC-24 (1971).
5. Fang, I. FOLDS-A Declarative Formal Language Definition System, Rep. *STAN-CS*-329, Comp. Sci. Dept., Stanford U. (1972).
6. Goto, E. Monocopy and Associative Algorithms in an Extended Lisp, University of Tokyo, Japan (May 1974).
7. Hoar, C. A. R. and Lauer, P. E. Consistent and Complementary Formal Theories of the Semantics of Programming Languages, *Acta Informatica*, 3 (1974), 135–153.
8. Jazayer, M. On Attribute Grammars and the Semantic Specification of Programming Languages, Ph. D. Th., Comp. and Inf. Sci. Dept., Case Western Reserve U. (1974).
9. Jazayeri, M. and Ogden, W. F. and Rounds, W. C. The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars, *CACM*, **18**, 12 (1975), 697–706.
10. Katayama, T. Translation of Attribute Grammar into Procedures, *Tech. Rep.* CS-K8001, Dept. of Comp. Sci., Tokyo Inst. of Tech. (1980). Also submitted to TOPLAS.
11. Katayama, T. and Hoshino, Y. Verification of Attribute Grammars, *Proc. 8th ACM Symp.* on Principles of Programming Languages (1981).
12. Kennedy, K. and Warren, S. K. Automatic Generation of Efficient Evaluators for Attribute Grammars, *Conf. Rec. POPL* (1976), 32–49.
13. Knuth, D. E. Semantics of Context-Free Languages, *Math. Syst. Th.*, 2 (1968), 127–145.
14. Lewis, P. M. and Rosenkrantz, D. J. and Stearns, R. E. Attributed Translations, *J. Computer and System Science*, 9 (1974), 279–307.
15. Marcotty, M. and Ledgard, H. F. and Bochmann, G. V. A Sampler of Formal Definitions, *Computing Serveys*, **8**, 2 (1976), 191–276.
16. Saarinen, M. On Constructing Efficient Evaluators for Attribute Grammars, *Lecture Note in Computer Science*, 62, Springer-Verlag (1978), 382–396.
17. Scott, D. Data Type as Lattices, Unpublished Lecture Notes, Amsterdam (1972).
18. Stoy, J. E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press (1977).
19. Wilner, W. T. Declarative Semantic Definition, *Rep. STAN-CS*-233-71, Comp. Sci. Dept., Stanford U. (1971).

(Received March 3, 1981: revised September 17, 1981)