# Systematic Debugging Method for Attribute Grammar Description

Masataka Sassa and Takuya Ookubo *

Department of Mathematical and Computing Sciences

Tokyo Institute of Technology

O-okayama, Meguro-ku, Tokyo 152, Japan

Email: sassa@is.titech.ac.jp

Tel: +81–3–5734–3228

Fax: +81–3–5734–2754

October 24, 1995

*Currently with Sharp Co., Nara Pref.

**Abstract**

Attribute grammars are commonly used in such areas as formalization of compilers. However, the method of debugging software described in attribute grammars is not yet established. In this paper, we propose a debugging scheme of attribute grammars using an algorithmic debugging technique, which was originally proposed for debugging logic programming languages. We show that algorithmic debugging is applicable to attribute grammars by introducing a virtual function, and then we propose an extension of algorithmic debugging which takes advantage of the characteristic features of attribute grammars. Further, we show that this technique is applicable to almost all classes of attribute grammars in the correspondence of their attribute evaluators. Although this paper mainly presents the theoretical framework, the applicability of the above technique was confirmed by implementing a prototype debugger.

**Key words**

Attribute grammar, Algorithmic debugging, Debugger, Programming environment

# 1 Introduction

Since the attribute grammar was initially proposed by Knuth in 1968 [1], it has been commonly used in various areas such as the formalization of compilers. On the other hand, however, it seems that researches for the development support environment, especially debuggers, for attribute grammar have received little attention until now, which are indispensable for the development of software by means of attribute grammar description.

Because the attribute grammar is based on a paradigm different from usual program languages, the debugging methods traditionally used cannot be applied to it as such. In many attribute grammar applications, their descriptions are translated into the evaluator written in a procedural language. Therefore, in traditional debugging, a debugger for the procedural language is applied to this evaluator, and from that result the errors in original attribute grammar are inferred. (There is already a debugging method devised for a functional language which was modelled after attribute grammars [14].) However, with such a method, users should consider the attribute grammar as a procedural language and the advantages of the attribute grammar such as conciseness would be lost. So in this research, we examined the debugging method for attribute grammars, and developed a debugging scheme which can handle the attribute grammar directly.

The most important feature of the debugging method presented here is the introduction of the technique called 'algorithmic debugging' [13]. This technique was proposed as a method for debugging logic programming languages; it presents to the programmer the local computational results of predicates during program execution, and through examination of whether these results agree with the programmer's intention, it systematically narrows the search space in which bugs can exist. In this research, we show that this technique can be applied to attribute grammars, and at the same time we expand this technique in order to realize more efficient debugging by utilizing the characteristic features of attribute grammar. Also, we show that this method can be applied to almost all classes of attribute grammars, making correspondence with their attribute evaluators. Although this paper mainly deals with the theoretical framework, we confirmed the above

application by implementing a prototype debugger.

Through introduction of the proposed technique, the following advantages can be obtained:

1. It can effectively locate bugs in the description of attribute grammar by means of minimum examination of attributes.

2. Since attributes to be examined by the user are managed by the debugger, he or she is not puzzled by the complex dependencies of attributes.

A preliminary version of this paper first appeared in Japanese [11].

# 2 Attribute Grammar

*Attribute grammar* is a formalism which can describe the syntax and semantics of a language in an integrated fashion. In this section, we present a simple example and an outline of attribute grammars. For details, refer to [1].

## 2.1 Example of Attribute Grammar

An attribute grammar is, in short, a context-free grammar with information called *attribute* added. Figure 1 gives an example of an attribute grammar which computes a value based on its binary fraction representation (this example is a slightly modified version of that presented in [8]).

Lines which include '$\rightarrow$' like (1) of this figure and lines beginning with '|' like (3) are productions of the context-free grammar, and the parts enclosed by { } following these lines are the definitions of attributes (*semantic rules*) corresponding to these productions. For example, (2) of this figure indicates that attribute `pos` of grammar symbol `L` has the value `1`.

Defining the value of the attribute after parsing the sentence according to the grammar is called *evaluating* an attribute, and the procedure to evaluate attributes is called *attribute evaluator*. When a string ".011" is evaluated according to the above grammar,

```
F →  . L                                                    (1)
     { L.pos = 1;                                           (2)
       F.val = L.val }
L0 → B L1
     { L1.pos = L0.pos + 1;
       B.pos = L0.pos + 1;                                  (bug)
       L0.val = B.val + L1.val }
   | B                                                      (3)
     { B.pos = L0.pos; L0.val = B.val }
B →  1
     { B.val = 2^-B.pos }
   | 0
     { B.val = 0 }
```

Figure 1: Example of Attribute Grammar

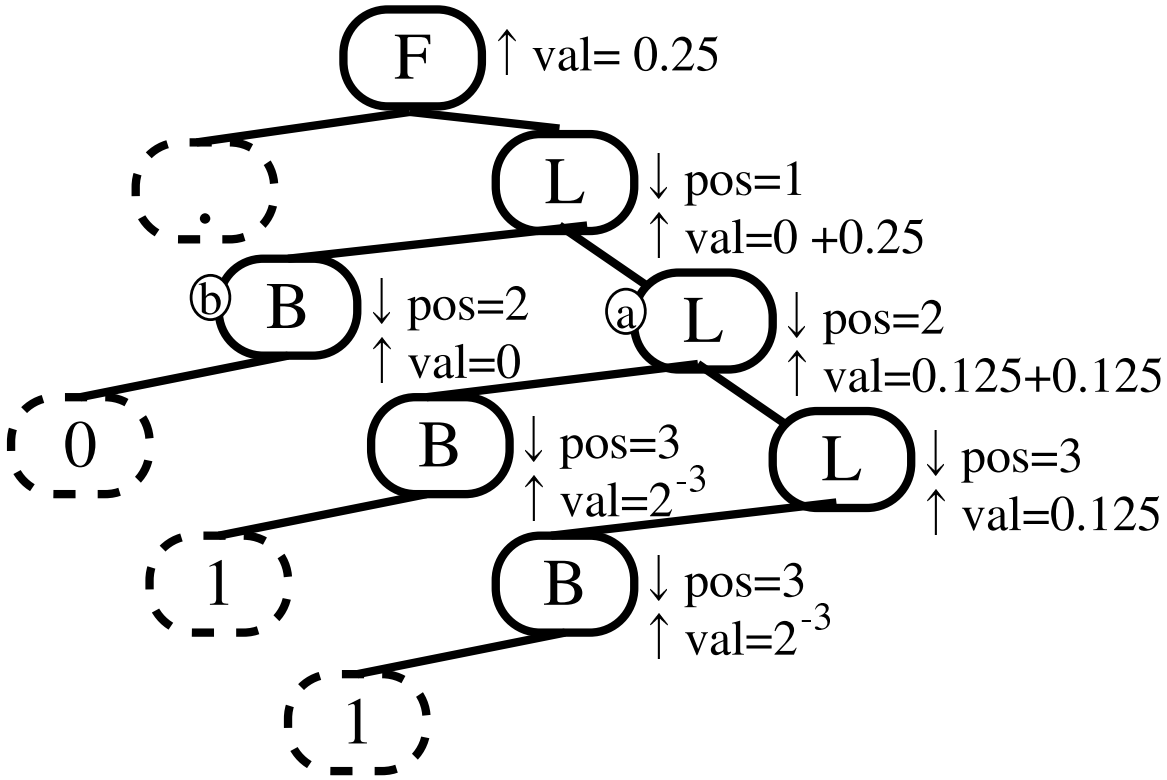


Figure 2: Attributed Parse Tree

an *attributed parse tree*, as shown in Fig. 2, is made. In this case, attribute `F.val` which belongs to the root is solved for. Note that, since there is a bug in the attribute grammar shown in Fig. 1, this value is incorrect (this problem will be discussed in detail in section 3).

## 2.2 Formal Definition of Attribute Grammar

In this section, definitions and terminologies used in attribute grammars are briefly explained.

Generally, an attribute grammar is defined by a triple $\langle G_u, A, R \rangle$.

$G_u = \langle N, T, P, Z \rangle$ is a context-free grammar called *underlying context-free grammar*, which represents the syntactic part of the attribute grammar.

$A$ is a set of *attributes*. Attributes are information added to grammar symbols of $G_u$. Attributes are classified into two kinds, according to the way their values are defined. Attributes whose values are defined from nodes corresponding to their parent or siblings of this grammar symbol on the parse tree, are called *inherited attributes*. Attributes whose values are defined from nodes corresponding to their children are called *synthesized attributes*. In Fig. 2, attributes prefixed by '$\downarrow$' are inherited attributes and those prefixed by '$\uparrow$' are synthesized attributes. Each grammar symbol $X$ of $G_u$ has two associated disjoint sets, set $I(X)$ of inherited attributes of $X$ and set $S(X)$ of synthesized attributes. (Normally, we assume $I(X) = \phi$ for terminals $X$.) In this case, $A$ is defined as:

$$\begin{aligned} A(X) &= S(X) \cup I(X) \\ A &= \bigcup_{X \in N \cup T} A(X) \end{aligned}$$

An attribute $a$ of symbol $X$ is represented as $X.a$.

$R$ is a set of *semantic rules*. For each production $p$ of $G_u$

$$p : X_0 \rightarrow X_1 \cdots X_{n_p}$$

is associated with a set $R(p)$ of semantic rules which defines all and only the attributes

$$a \in S(X_0) \cup I(X_1) \cup \cdots \cup I(X_{n_p})$$

Then $R$ is defined as follows:

$$R = \bigcup_{p \in P} R(p)$$

Note that for any attribute, the rule defining its value is unique.

Each semantic rule associated with $p$ is written as follows:

$$X_k.a = f(X_{i_1}.a_1, \ldots, X_{i_m}.a_m)$$

Here, $0 \leq k, i_1, \ldots, i_m \leq n_p$ and $f$ is a function defining the value of the attribute. Here, we say that $X_k.a$ *depends* on $X_{i_1}.a_1, \ldots, X_{i_m}.a_m$ in $p$.

# 3 Application of Algorithmic Debugging to Attribute Grammar

## 3.1 Algorithmic Debugging

In logic programming languages, a program is composed of parts called 'predicates'. Algorithmic debugging [13] is a method of narrowing the search space in which bugs can exist, by querying the programmer whether the behavior of predicates at run time is correct or not. This method is known to be applicable to procedural or functional languages if one ignores the influences of side effects.[1]

Here, we outline the algorithmic debugging according to reference [13] with some re-formulation.

Figure 3 shows a program of insertion sort written in Prolog. Predicate 'isort' sorts the list given in the first parameter in ascending order and returns the result to the second parameter. Predicate 'insert' inserts the element of the first parameter to the list of the second parameter and returns the result to the third parameter.[2] However, since this program contains one bug, it does not behave correctly.

[1] Although influences of side effects are discussed in [5], for example, these are omitted here because they are irrelevant to the attribute grammar.

[2] Although concepts such as parameter or return value do not originally exist in Prolog, these terminologies are used here for simplicity.

```
isort([X|Xs],Ys)
    :- isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).
insert(X,[Y|Ys],[Y|Zs])
    :- Y>X, insert(X,Ys,Zs).                                    (bug)
insert(X,[Y|Ys],[X,Y|Ys]) :- X=<Y.
insert(X,[],[X]).
```

Figure 3: An Erroneous Program

```
  isort([3],[3])     : ok? y
  isort([1,3],[3,1]) : ok? n
  insert(1,[3],[3,1]): ok? n
  insert(1,[],[1])   : ok? y
Error diagnosed:
 insert(X,[Y|Ys],[Y|Zs])
     :- Y>X, insert(X,Ys,Zs).
```

Figure 4: An Example of Algorithmic Debugging

Next, an example of finding a bug in this program using algorithmic debugging is shown. Let us assume that, when the user gave parameter [2,1,3] to this program, he or she obtained an erroneous value [2,3,1] instead of the expected value [1,2,3]. In this case, the debugger performs queries as shown in Fig. 4 to the user based on the execution history of this erroneous computation, and finds the bug in the program.

Underline indicates user input. Here, through four queries, the debugger inferred the clause with the bug. In this way, the user can find bugs by merely responding to the queries of whether the behavior of predicates is correct or not, without looking at the inner part of each predicate.

Now, the mechanism of the debugger is explained. First, the debugger makes a tree

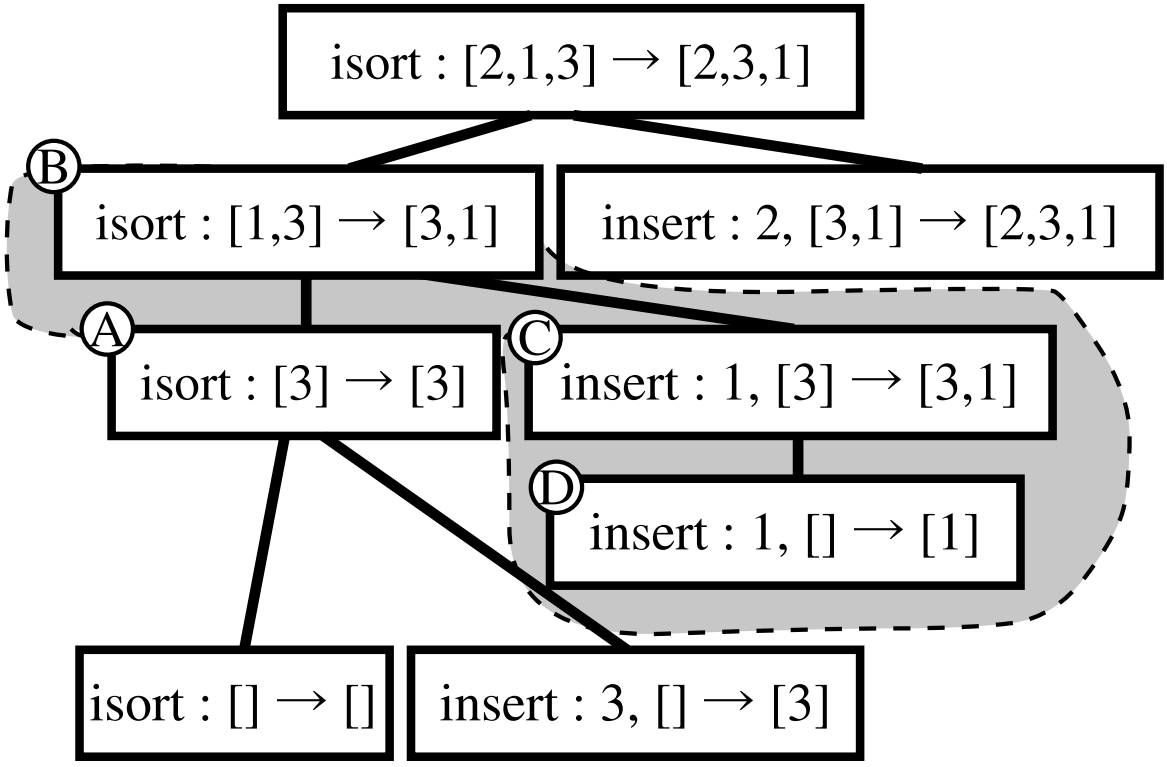


Figure 5: Computation Tree

called *computation tree* from the erroneous computation. This is a kind of execution history in which calls of other predicates during the computation of 'isort' for parameter [2,1,3] is recorded, together with the parameters and return value of each call. Figure 5 shows the computation tree in this example.

Next, the debugger selects one node of the computation tree and queries the user as to whether the behavior of the predicate of that node is correct or not. In the above example, the debugger first looks at Ⓐ in Fig. 5. Here, it finds from the query to the user that "it is a correct behavior that 'isort' returned value [3] given the parameter [3]". From this, it can conclude that there is at least one bug in the part of the computation other than the subtree rooted at Ⓐ of this computation tree.

Next, the debugger turns its attention to Ⓑ in Fig. 5. According to the user's response, this behavior of 'isort' is incorrect. So, it finds that there is at least one bug somewhere in the computation corresponding to the subtree rooted at Ⓑ of the computation tree.

From these two queries, the debugger finds that there is a bug in the computation

represented by the shaded part of Fig. 5. In this way, by querying the user, the search space containing the bug can be rapidly narrowed.

In this case, the debugger knows that Ⓒ is incorrect and Ⓓ is correct from the responses of the user. Based on these results, it can be said that "although 'insert(1,[3],[3,1])' is incorrect, all behavior of predicates called within this computation process is correct". This indicates that there is a bug somewhere in the definition of 'insert'. Then, the debugger outputs this part as the line including bugs. More precisely, it identifies a single clause including a bug based on the execution history. (Hereafter 'bugs' may be 'a single bug'.)

Note that in this debugging method, the nodes of the computation tree, which are subject to query by the debugger, can be selected arbitrarily from the part that could include bugs. The selection methods of nodes are discussed in references such as [6], and it is known that if the number of nodes of the computation tree is $n$, the bug can be found in $O(\log\ n)$ queries.

## 3.2 Formal Definition of Algorithmic Debugging

Here, we show a formal definition of the algorithmic debugging. This is a slightly modified and simplified version of that in reference [13].

Hereafter, the components of a program, such as predicates presented in the previous section, functions of usual languages or procedures which return a value, are simply called 'functions'. We deal only with computations which terminate after giving output. Side effects are not considered here.

**Definition 1** [Behavior] When a function $f$ returns an output $y$ for an input $x$, the triple $\langle f, x, y\rangle$ representing this computation is called *behavior* of this function. Note that $x$ and $y$ are assumed to be vectors since we also consider the case in which the function handles multiple input and output. ■

**Definition 2** [Top level trace] For a behavior $\langle f, x, y\rangle$, consider function calls directly made by $f$ in the process of computing $f$. We make a sequence $\langle f_1, x_1, y_1\rangle, \ldots, \langle f_n, x_n, y_n\rangle$

by arranging the behaviors of all those calls in the order of their occurrence. This sequence is called the *top level trace* for behavior $\langle f, x, y \rangle$ and is represented as $trace(\langle f, x, y \rangle)$. ■

**Definition 3** [Computation tree] An ordered tree $Ct$ having behaviors of functions as its nodes and having the following features is called *computation tree* for the behavior $\langle f, x, y \rangle$, and this tree is represented as $ctree(\langle f, x, y \rangle)$.

1. The root is $\langle f, x, y \rangle$
2. Each node of $Ct$ has the elements of the top level trace of its node as the first child, the second child, ... in this order.

A computation tree having a finite number of nodes is called '*complete* computation tree'.

■

An 'incorrect behavior' of a function is not an incorrect behavior of the computer, but rather represents a difference between "behavior of the program intended by the programmer" and "behavior of the program as written". Then, we consider a way of representing this "intention of the programmer" as follows.

**Definition 4** [Interpretation] *Interpretation* $M$ is a set of behaviors, which represents all correct behaviors of functions (i.e. output $y$ corresponding to input $x$) that the programmer intends. When a behavior $\langle f, x, y \rangle$ is included (or not included) in $M$, we say that $\langle f, x, y \rangle$ is *correct* (or *incorrect*). Furthermore, for any $x$ in the domain of function $f$, if $\langle f, x, y = f(x) \rangle$ is always included in $M$, we say that $f$ is *correct*; otherwise it is *incorrect*. ■

For function $f$, suppose that although all subordinate functions called by $f$ behave correctly, $f$ itself behaves incorrectly. This indicates that there is a bug(s) in the definition of $f$.

**Definition 5** [Bug] For behavior $b = \langle f, x, y \rangle$, if $b \notin M$ and if for all $b' \in trace(b)$ $b' \in M$ holds, we say that $b$ *includes bugs*. In this case, we also say that function $f$ *includes bugs*. (Note that 'bugs' may be 'a single bug' throughout this paper.) ■

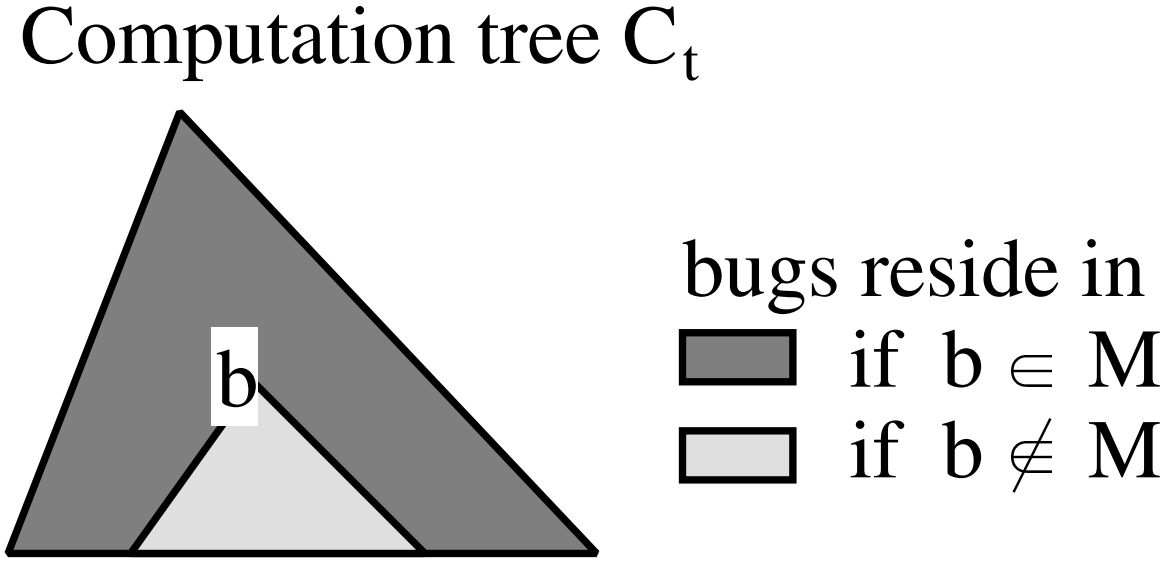


Figure 6: Principle of Algorithmic Debugging

The case where a behavior $\langle f, x, y \rangle$ is incorrect means that $f$ itself includes bugs or that there are incorrect behaviors in the top level trace of $\langle f, x, y \rangle$.

**Theorem 1** [13] For a node $b = \langle f, x, y \rangle$ in a complete computation tree $Ct$, if $b \notin M$, then there is a behavior(s) having bugs somewhere in the subtree rooted at $b$ of $Ct$. ■

**Theorem 2** If a computation tree $Ct$ having an incorrect behavior $\langle f, x, y \rangle$ as its root is complete, and if a correct behavior $b = \langle f', x', y' \rangle$ is included in $Ct$, then the tree which is made by excluding $ctree(b)$ from $Ct$ has behavior(s) including bugs as its node. (For a proof, refer to [13].) ■

Figure 6 illustrates Theorems 1 and 2.

From the above theorems, when an incorrect computation is given, at least one function including bugs can always be found by the procedure shown in Fig. 7.

This method of locating bugs is called 'algorithmic debugging'.

### 3.3 Application to Attribute Grammar

In attribute grammars, there does not exist one component corresponding to 'function' introduced in the beginning of section 3.2. However, if we find a component in attribute

```
procedure Algorithmic-Debugging();
begin
  T := make_ctree(); { make the computation tree }
  while there are two or more nodes of T do begin
    t := arbitrary node of T other than the root;
    if t shows the correct behavior then
      delete subtree below t from T
    else
      T := subtree below t
  end;
  display the definition of function for the root of T as a bug
end.
```

Figure 7: Procedure of Algorithmic Debugging

grammars which can be regarded as a 'function', it is possible to apply algorithmic debugging to attribute grammars.

We can consider several possible components in attribute grammars which "behave like a function". What is suited to algorithmic debugging is a 'function' (i) whose calls make a nested structure, and (ii) whose input and output are easy to grasp for the user. Therefore, we focus our attention on the following property.

**Theorem 3** Assume that an attribute grammar $AG$ was evaluated and that an attributed parse tree $T$ and all attribute values of $T$ were obtained. Also, assume that dependencies among attributes are all known when these attribute values were evaluated. Hereafter, we suppose that when we refer to a parse tree of $T$, it includes 'intrinsic attributes' associated with leaf nodes ('intrinsic attributes' are attributes of terminals, and are defined from tokens).

Here, consider a synthesized attribute $N.s$ of a node $N$ of $T$. As shown in Fig. 8, this value is uniquely defined by:

1. Sub parse tree $tree_N$ rooted at $N$

2. All attributes, among inherited attributes $N.I$ of $N$, on which $N.s$ directly or indirectly depends in $tree_N$. They are denoted by $N.I_{N.s}$.

■

*Proof (Sketch).* We can assume the Bochmann normal form [1] without loosing generality. Then, attributes on which $N.s$ directly depends are a subset of inherited attributes of $N$ and a subset of synthesized attributes of children of $N$. Thus, attributes on which $N.s$ directly or indirectly depends are obtained by making a closure of these dependencies. Here, they are only $tree_N$ and $N.I_{N.s}$. □

From this theorem, we can consider an abstract 'function' which defines the value of any synthesized attribute $N.s$:

$$N.s = F_{N.s}(N.I_{N.s}, tree_N)$$

This function $F_{N.s}$ is called *Synth function* for $N.s$.

Based on this Synth function, we define 'behavior', 'top level trace', 'computation tree', 'interpretation' and 'bug' for attribute grammars. In this way, we can realize algorithmic debugging for attribute grammars.

A *behavior* can be defined using parameters and return value of the Synth function, as $\langle F_{N.s}, \{N.I_{N.s}, tree_N\}, N.s\rangle$, similarly to section 3.2. (Precisely, $\{\cdots\}$ is a tuple rather than a set, but this notation is used for readability.)

As for 'top level trace', since Synth function is an abstract one and does not call other Synth functions during computation, the definition presented in section 3.2 cannot be used. Therefore, we define the top level trace for behavior $\langle F_{N.s}, \{N.I_{N.s}, tree_N\}, N.s\rangle$ as follows, using the dependencies among attributes:

**Definition 6** [Top Level Trace of Synth Function] Consider attributes which are synthesized attributes of children nodes $N_1, \ldots, N_n$ of node $N$ in the attributed parse tree $T$, and on which $N.s$ directly or indirectly depends in the sub parse tree rooted at $N$. By

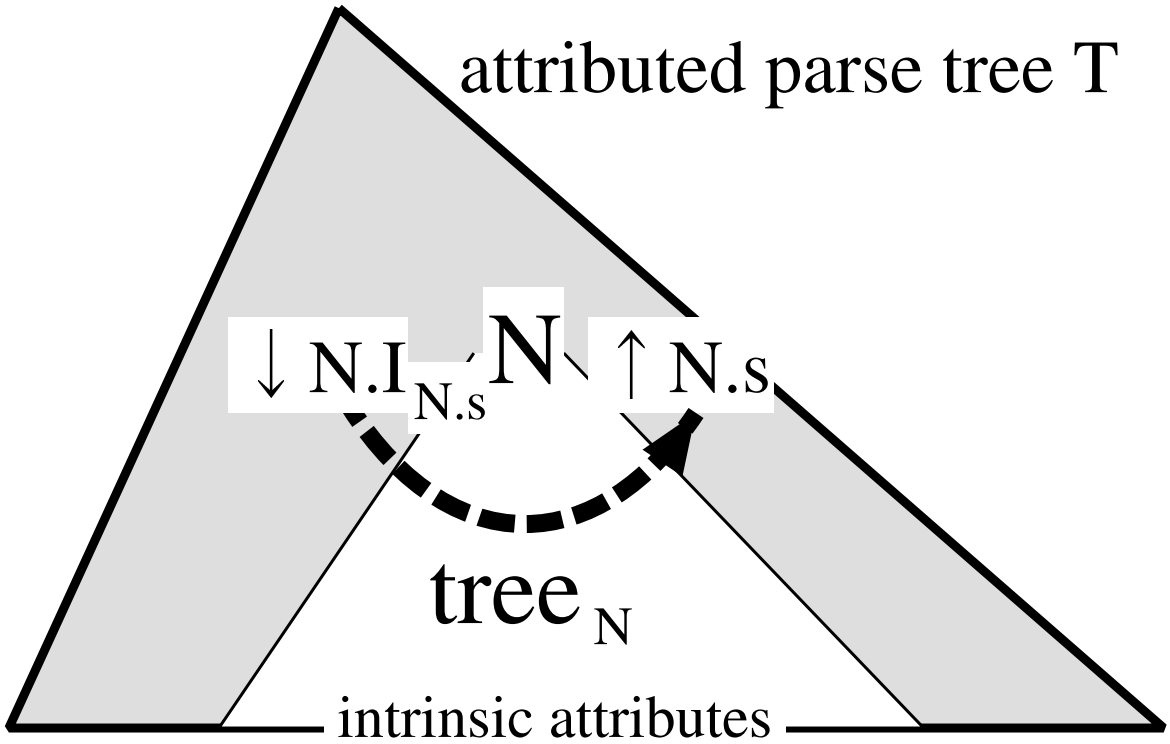


Figure 8: Principle of Synth Function

topologically sorting the whole such synthesized attributes according to the dependency among them, we get a sequence $N_{i_1}.s_{j_1}, \ldots, N_{i_m}.s_{j_m}$. Then, we let the *top level trace* of $\langle F_{N.s}, \{N.I_{N.s}, tree_N\}, N.s\rangle$ be the sequence of behaviors (of Synth functions) $\langle F_{N_{i_1}.s_{j_1}}, \{N_{i_1}.I_{N_{i_1}.s_{j_1}}, tree_{N_{i_1}}\}, N_{i_1}.s_{j_1}\rangle, \ldots, \langle F_{N_{i_m}.s_{j_m}}, \{N_{i_m}.I_{N_{i_m}.s_{j_m}}, tree_{N_{i_m}}\}, N_{i_m}.s_{j_m}\rangle$. This top level trace is denoted by $trace(\langle F_{N.s}, \{N.I_{N.s}, tree_N\}, N.s\rangle)$ (Fig. 9). ■

The *computation tree* is defined using the above top level trace similarly to section 3.2. The *interpretation* $M$ is now a set of behaviors of Synth functions. A *bug* is defined similarly to section 3.2 by taking the Synth function as a 'function'. Then, the following holds.

**Theorem 4** In the computation tree defined above, assume that there is a bug in node $\langle F_{N.s}, \{N.I_{N.s}, tree_N\}, N.s\rangle$. Then, among semantic rules associated with production $p$ applied to node $N$ of the parse tree, there is a bug in either the definition of $N.s$ or the definition of attributes on which $N.s$ directly or indirectly depends in $tree_N$. ■

*Proof.* From the definition of top level trace, the assumption means the following:

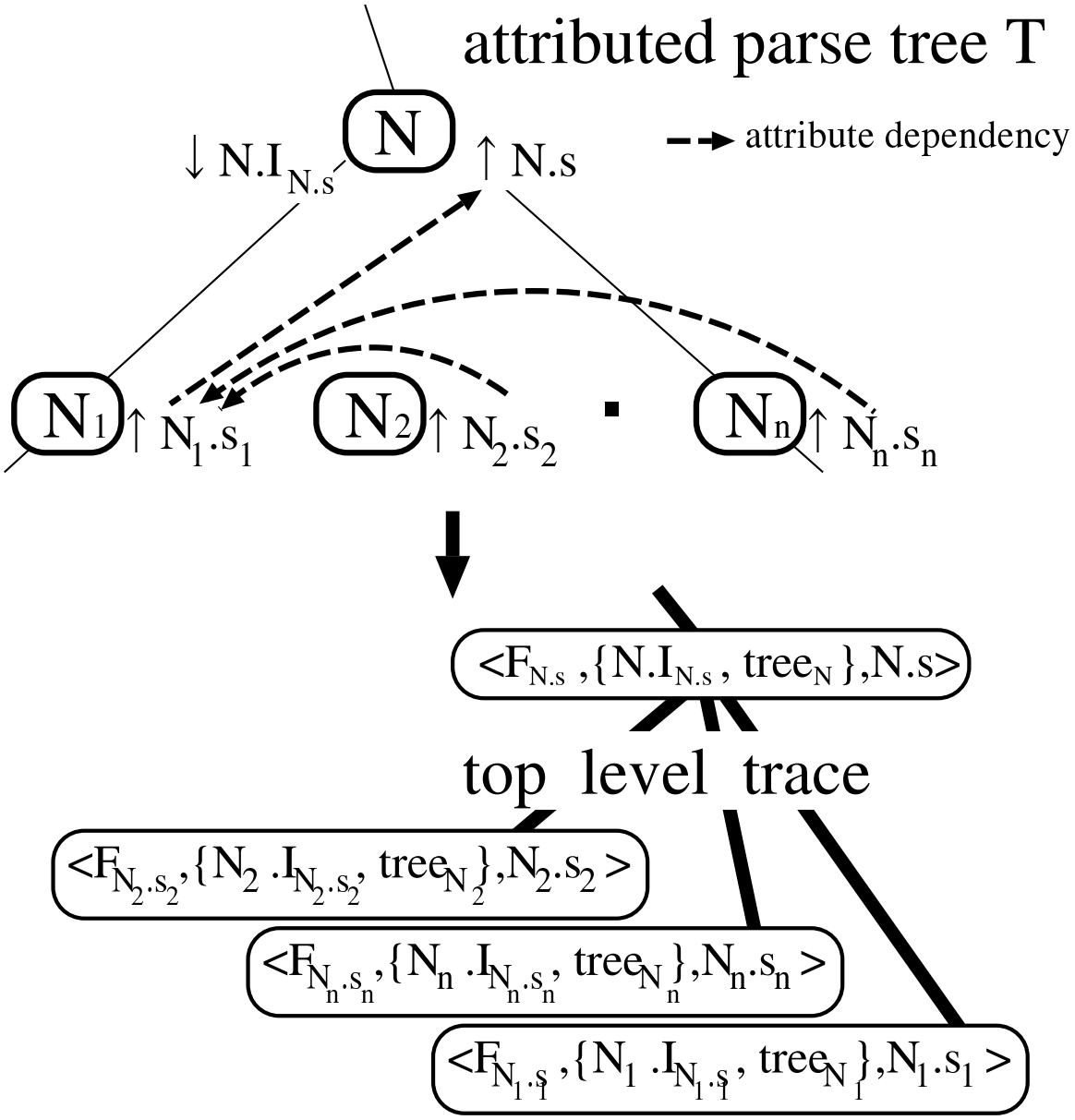


Figure 9: Example of Generation of Top Level Trace

1. The value of $N.s$ is incorrect with respect to $tree_N$ and the set of inherited attributes $N.I_{N.s}$ given to $N$, and

2. The values of synthesized attributes $N_{i_1}.s_{j_1}, \ldots, N_{i_m}.s_{j_m}$ of children nodes of node $N$ of the parse tree are all correct, with respect to $tree_N$ and to inherited attributes $N_{i_1}.I_{N_{i_1}.s_{j_1}}, \ldots, N_{i_m}.I_{N_{i_m}.s_{j_m}}$ of children nodes of $N$.

This indicates that either (1) the semantic rule is incorrect, which defines the value of $N.s$ from values of $N_{i_1}.s_{j_1}, \ldots, N_{i_m}.s_{j_m}$ or from values of $N.I_{N.s}$, or (2) the semantic rules are incorrect, which define values of $N_{i_k}.I_{N_{i_k}.s_{j_k}}$ associated with $N_{i_k}$, a child of $N$. These semantic rules are all associated with production $p$ applied to node $N$ of the parse tree.

Case (1) is trivial. In case (2), since $N_{i_k}.I_{N_{i_k}.s_{j_k}}$ is "the set of attributes on which $N.s$ directly or indirectly depends in $tree_N$" considering the definition of the top level trace, the conclusion can be derived. □

From the above, we can say similarly to section 3.2 that, when a synthesized attribute of the root of $T$ is incorrect, bugs can be located by applying algorithmic debugging.

However, it should be noted that this method is solely based on 'behavior', i.e. "whether a Synth function returns a correct output or not for a given input". Cases such as when the input itself (e.g. value of inherited attribute) is incorrect, are discussed in section 4.

As an example, let us see the attribute grammar of Fig. 1 which computes the value of a binary fraction. From the attributed parse tree of Fig. 2 for which an incorrect value is obtained, the computation tree shown in Fig. 10 is constructed.

Note that point $a$ in Fig. 10 corresponds to ⓐ in Fig. 2. It indicates that an erroneous computation "when the binary number '11' appears after the second decimal place of the binary fraction, the value it represents is 0.25" is performed. (What is actually displayed to the user is the attributed parse tree like the one in Fig. 2.)

If the number after the second decimal place is '11', the value should be 0.375. Therefore, $a \notin M$, and from this, we know that there is a bug in the subtree rooted at $a$ of the computation tree. Then, if for example we examine $b$ next, we find that $b \in M$ in a

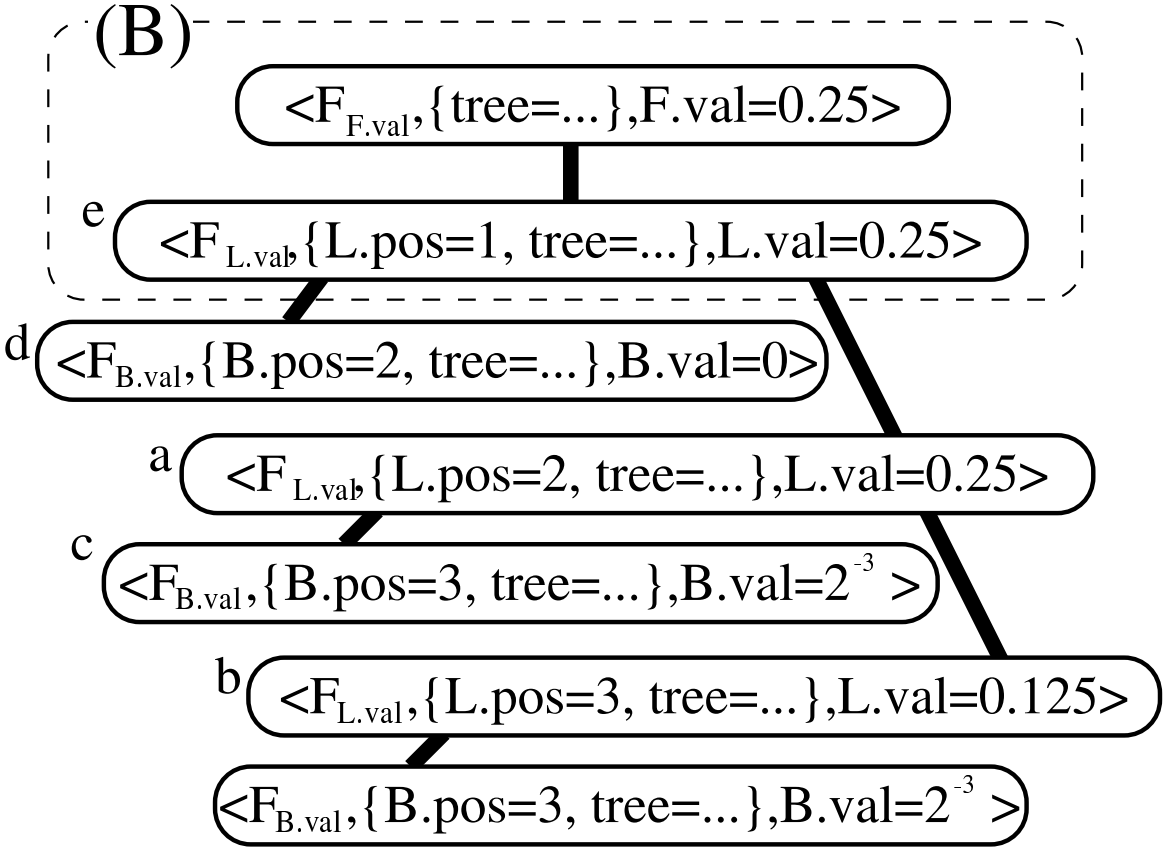


Figure 10: Computation Tree

similar fashion. When we examine $c$, we find that $c \in M$. From this, we know that a bug exists in the definition of behavior which $a$ represents, i.e. in the part on which `L.val` depends in the semantic rules associated with the production '`L` $\rightarrow$ `B L`'. In the current case, the whole part enclosed by {} can include a bug in the following:

```
L0 → B L1
    { L1.pos = L0.pos + 1;
      B.pos = L0.pos + 1;                                    (bug)
      L0.val = B.val + L1.val }
```

Theoretically, we can only say that there is a bug somewhere in the above 3 lines of semantic rules, but practically, the semantic rule indicating '(bug)' can be identified as erroneous by displaying the value of each attribute appearing in these semantic rules and by allowing it to be checked by the user.

Note that, since this method is only based on the assumption for attribute grammars that "all attribute values can be defined", it can be applied to almost all classes of attribute grammars, e.g. each class below noncircular attribute grammars (see section 5).

# 4 Extension of Algorithmic Debugging

## 4.1 Incorrect Input/Output Value

Input (excluding the parse tree) and output of the Synth function introduced in section 3.3 correspond to the inherited and synthesized attributes of an attribute grammar, respectively. The programmer of an attribute grammar might be able to determine, sometimes very easily, whether these attribute values are correct or not, if the nodes of the parse tree to which these attributes belong are identified.

For example, in Fig. 10, while $B.pos$ should represent the number of places from the decimal point, $B.pos = 2$ is given in $d$ for the node representing the first decimal place

of the fraction. ($d$ corresponds to ⓑ in Fig. 2.) By paying attention to this node, the programmer might find that "$B.pos$ is invalid" before thinking that "$\langle F_{B.val}, \{B.pos = 2, tree = \cdots\}, B.val = 0\rangle$ is a correct behavior".

Therefore, we devised a method of efficient debugging, incorporating this kind of information into algorithmic debugging.

### Extended Algorithm 1

We say that an attribute value is *valid* (or *invalid*) when we can show that this attribute value is correct (or incorrect) with respect to the attribute grammar and the entire parse tree.

Consider the case in which the programmer can say that $x$ or $y$ is valid or invalid for a node $b = \langle f, x, y\rangle$ of computation tree $Ct$.

We here divide the nodes included in computation tree $Ct$ into the following sets, with respect to a node $ct$:

***Child*** Entire nodes of subtree of $Ct$ having $ct$ as the root;

***Before*** The set consisting of the following elements;

1. Nodes corresponding to the parent or ancestor of $ct$;
2. Among children of elements of 1, the set of nodes $ct'$ which is positioned in order before those included in 1 (or $ct$), and the set of entire nodes of the subtree having $ct'$ as the root.

***After*** The set of entire nodes which are neither included in *Before* nor *Child*, among nodes of $Ct$.

**Theorem 5** For a node $ct = \langle f, x, y\rangle$ of computation tree $Ct$ of an incorrect computation, the following 1 to 3 hold (Fig. 11):

1. If $x$ is invalid, there is a behavior in *Before* which includes a bug;
2. If $y$ is invalid, there is a behavior in *Before* or *Child* which includes a bug;

3. If $y$ is valid, there is a behavior in *After* or *Before* which includes a bug.

■

*Proof (Sketch).*

1. The fact that $x$ is invalid means that either: (i) the computation was incorrect in obtaining some value of attributes on which $x$ directly or indirectly depends, or (ii) the computation was incorrect in obtaining the value of $x$. In the former case, there is a bug in *Before*. In the latter case, there is a bug in the parent of $ct$. In either case, there is a bug in *Before*.

2. The fact that $y$ is invalid means that either $ct \notin M$ or $x$ is invalid. In the former case, there is a bug in the subtree rooted at $ct$, i.e. in *Child*. In the latter case, there is a bug in *Before* from 1.

3. In general, if $x$ is valid and $y$ is valid, then $ct \in M$. Therefore, since $y$ is valid, either $ct \in M$ holds or $x$ is invalid. In the former case, we can say from Theorem 2 that there is a bug in the part of $Ct$ excluding *Child*, i.e. in *Before* or *After*. For the latter case, there is a bug in *Before*, from 1. □

In the example in Fig. 10 mentioned before, we can say that there is a bug in (B) of this figure from the information that $B.pos$ of $d$ is invalid. By examining $e$, we know that $e \notin M$. And from the fact that there are no children of $e$ in (B), we can say that there is a bug in $e$. Therefore, it is found that there is a bug in the semantic rules associated with '`L` → `B L`' on which `L.val` depends. This is equivalent to the result in section 3.3.

Note that the method described here is an extension of the algorithmic debugging itself, and it can be applied to debugging of other languages.[3] However, this method is especially useful in the application to attribute grammars, as mentioned at the beginning of this section.

[3]For details, refer to appendix of reference [10].

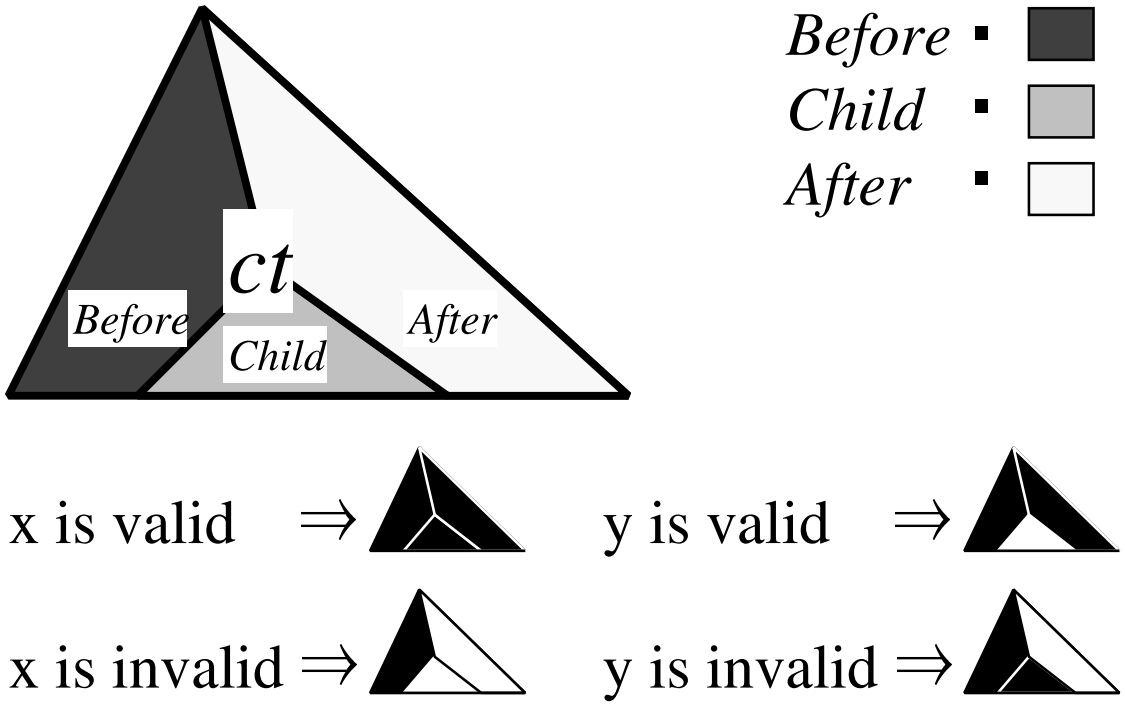


Figure 11: Extended Algorithmic Debugging

## 4.2 Difficult Query

To follow the procedure of algorithmic debugging, the user must by all means find a response to any query. However, this query is not always easy to respond.

Algorithmic debugging was originally devised for logic programming languages and this query is, by nature, for 'predicates' defined by the user. However, in attribute grammars, it is an abstract 'Synth function' which is subject to query. Since this is not defined by the user, it is entirely possible that the user cannot respond to the query.

There are several ways to address this problem in the framework of algorithmic debugging (cf. section 7). Here, however, we present a method to extend the algorithm more positively to allow a 'don't know' response.

### Extended Algorithm 2

**Theorem 6** For a node $ct$ on computation tree $Ct$, if either of the following holds:

1. Either the parent node or the ancestor node of $ct$ is correct, or
2. Any node other than the above is incorrect

then, we can exclude $ct$ from the search space for bug(s). ■

*Proof (Sketch).* In case 1, from Theorem 2, and in case 2, from Theorem 1, we can say that at least one bug exists in the part other than $ct$. □

From Theorem 6, even if the user were to return a 'don't know' response for a node $ct$, in many cases the debugger could exclude $ct$ from the search space by making queries to other nodes.

Inversely, if all of the following hold for $ct$ in the process of debugging:

1. Its parent is incorrect
2. Its siblings are all correct
3. Its children are all correct

then, *ct* cannot be excluded from the search space. However, even in this case, *ct* should be either correct or incorrect. From Definition 5, we can say that if *ct* is incorrect, a bug exists in the definition of the semantic rule corresponding to *ct*, and if *ct* is correct, a bug exists in the definition of the semantic rule corresponding to its parent. Therefore, if we present the union of these definitions as possible errors, the debugging process can be said to be completed.

As shown above, we can allow a 'don't know' response to queries of algorithmic debugging. Although this does not improve the power of the debugging method, it would be very useful in actual debugging for practical reasons. Note that this extension can also be applied to the debugging of other programming languages, but it will be especially useful in attribute grammars.

# 5 Relationship between the Proposed Debugging Method and Attribute Grammar Classes

So far, we considered that the Synth function is an abstract one and it does not really exist. However, we can say that in certain classes of attribute grammars Synth functions can be directly constructed, and in other classes, the computation tree based on Synth functions can be easily obtained from existing attribute evaluators.

## 5.1 Simple Multi-Visit Attribute Grammar Class and its Subclasses

Let $X$ and $Y$ be attribute grammar (AG) classes. In this paper, we define that $X$-AG is a *subclass* of $Y$-AG, when the attribute evaluation algorithm of $X$-AG is a special case of the attribute evaluation algorithm of $Y$-AG [3].

Engelfriet et al. showed that the evaluators for the simple multi-visit attribute grammar and its subclasses are a special form of a certain attribute evaluator [3]. Therefore, we show that for these classes, the computation tree can be made based on this evaluator.

```
procedure sV-evaluate(N_0: node);
{ Let  p: X_0 → w_0 X_1 w_1 ··· X_n w_n
    be the production applied at N_0 }
begin
1   count(N_0) := count(N_0) + 1;
2   let j = count(N_0);
3   compute all inherited attributes of N_0 in A_j(X_0);
4   for i := 1 to m_{p,j} do { visit children nodes }
5       sV-evaluate(X_k) where k = v_{p,j}(i);
6   compute all synthesized attributes of N_0 in A_j(X_0)
end;
begin
  set count(N) to 0 for every node N of the parse tree;
  while count(root) < k(root) do
    sV-evaluate(root)
end
```

(Note) $X_i$ is a nonterminal symbol and $w_i$ is a string of terminal symbols. $A_j(X_0)$ is the set of attributes evaluated at the j-th visit to the node $N_0$ labeled $X_0$. $v_{p,j}(i)$ $(1 \le i \le m_{p,j})$ is a sequence called 'visit-sequence'. It indicates that when visiting $N_0$ for the j-th time, the children of $N_0$ are visited in the order of the visit-sequence $v_{p,j}$, where $p$ is the production applied at $N_0$.

Figure 12: Attribute Evaluator for Simple Multi-Visit Class

**function** sV-evaluate($N_0$: node): computation tree;

...

4 **for** i := 1 **to** $m_{p,j}$ **do** { visit children nodes }

5 Tree(i) := sV-evaluate($X_k$) **where** k = $v_{p,j}$(i);

6 compute all synthesized attributes of $N_0$ in $A_j(X_0)$;

7 **return** computational tree

where the parent is

$\langle\ f.S_j(X_0),\ \{I_j(X_0), N_0\},\ S_j(X_0)\ \rangle$

and children are Tree(1) $\cdots$ Tree($m_j$)

...

(Note) $I_j(X_0)$ and $S_j(X_0)$ represent inherited and synthesized attributes of $A_j(X_0)$, respectively, which are evaluated in this visit to node $N_0$.

Figure 13: Modified Evaluator to obtain the Computational Tree

First, the evaluator for the simple multi-visit attribute grammar class is shown in Fig. 12.

By rewriting part of Fig. 12 to match Fig. 13, a function is obtained which returns the computation tree of Synth functions as its value.

According to references [2] [3], (hereafter, 'simple' is omitted), the subclasses of multi-visit (=$\ell$-ordered) AG include multi-sweep AG, multi-alternating pass AG, multi-pass AG, Ordered AG, one-visit AG = one-sweep AG, and one-pass AG = L-attributed AG. From the above, it is shown that the algorithmic debugging can be easily applied to attribute grammars of these classes.

## 5.2 Noncircular and Absolutely Noncircular Attribute Grammar Classes

In case of absolutely noncircular attribute grammar class, the "procedure to compute the value of synthesized attribute" in the evaluator by Katayama [7], which is based on the

augmented dependency graph, can be directly considered to be a Synth function. This 'procedure' actually calls other 'procedures' in the process of computing the synthesized attribute value. The called 'procedures' here compute synthesized attributes of nodes, which are children (in the parse tree) of the node corresponding to the synthesized attribute which is computed by the calling 'procedure'. The calls of these 'procedures' are performed in order by topologically sorting all attributes of the augmented dependency graph of this node. The computation tree can be also created in one-to-one correspondence to these calls. Namely, the computation tree can be made by letting the 'behavior' node correspond to the 'procedure' and by placing the nodes of its children in the order of the called 'procedures'.

On the other hand, in the case of the noncircular attribute grammar class, the augmented dependency graph cannot be directly created from the attribute grammar. However, after the step where the attributed parse tree is determined, we can make a graph equivalent to the augmented dependency graph for attributes of an arbitrary node in the parse tree and attributes of its children nodes. (This can be made by taking the closure of attribute dependency in each subtree of children of this node. Since this attribute grammar is noncircular, there is no cycle in this graph equivalent to the augmented dependency graph.) In this way, the Synth functions and the computation tree can be made similarly to the case of the absolutely noncircular attribute grammar.

From the above, it can be said that the algorithmic debugging can be easily applied to these two classes of attribute grammar.[4]

[4] The systematic debugging method proposed in this paper is closely related to the form of attribute evaluators. Therefore, we thought it insufficient to show only its applicability to the noncircular attribute grammar class. So, in section 5 we showed the applicability to each evaluator of typical attribute grammar classes.

# 6 Implementation of a Prototype Debugger

We implemented a prototype debugger called Aki, in order to confirm the applicability of the algorithm presented so far. It uses an experimental system called Jun [12] as a basis. Jun can handle the absolutely noncircular attribute grammar class. (In fact, it can also handle part of a class which allows cycles of attribute dependencies with certain conditions [4], but the discussion is omitted since it is irrelevant to the subject of this paper.) Aki can perform debugging for attribute grammars which Jun handles. Through this implementation, it is considered that the applicability of the proposed debugging method is confirmed for typical classes of attribute grammars given in section 5.[5]

Aki realizes the following three basic functions:

1. displaying the attributed parse tree
2. referencing definition of attributes from the attributed parse tree
3. algorithmic debugging

Using these basic functions, Aki can debug the attribute grammar directly. An example of its execution is shown in Fig. 14.

Jun is written in Common Lisp. The implementation of Aki was performed by revising Jun so that it outputs the computation tree as mentioned in section 5.2, and by adding the body of the debugger and the GUI part using Garnet [9], a GUI-builder for Common Lisp. Aki is organized by considering the linkage with the editor.

In the future, our goal is to improve the operability of its GUI and to make an integrated development environment for attribute grammars.

[5] What was actually confirmed is the applicability to the absolutely noncircular attribute grammar class. However, if we investigate reference [3], we can say that the attribute grammar classes presented in section 5.1 are subclasses of the absolutely noncircular attribute grammar class. Therefore, our implementation can be applied to these classes. On the other hand, if we make an attribute evaluator as presented in section 5.2, it can be applied to the noncircular attribute grammar class.

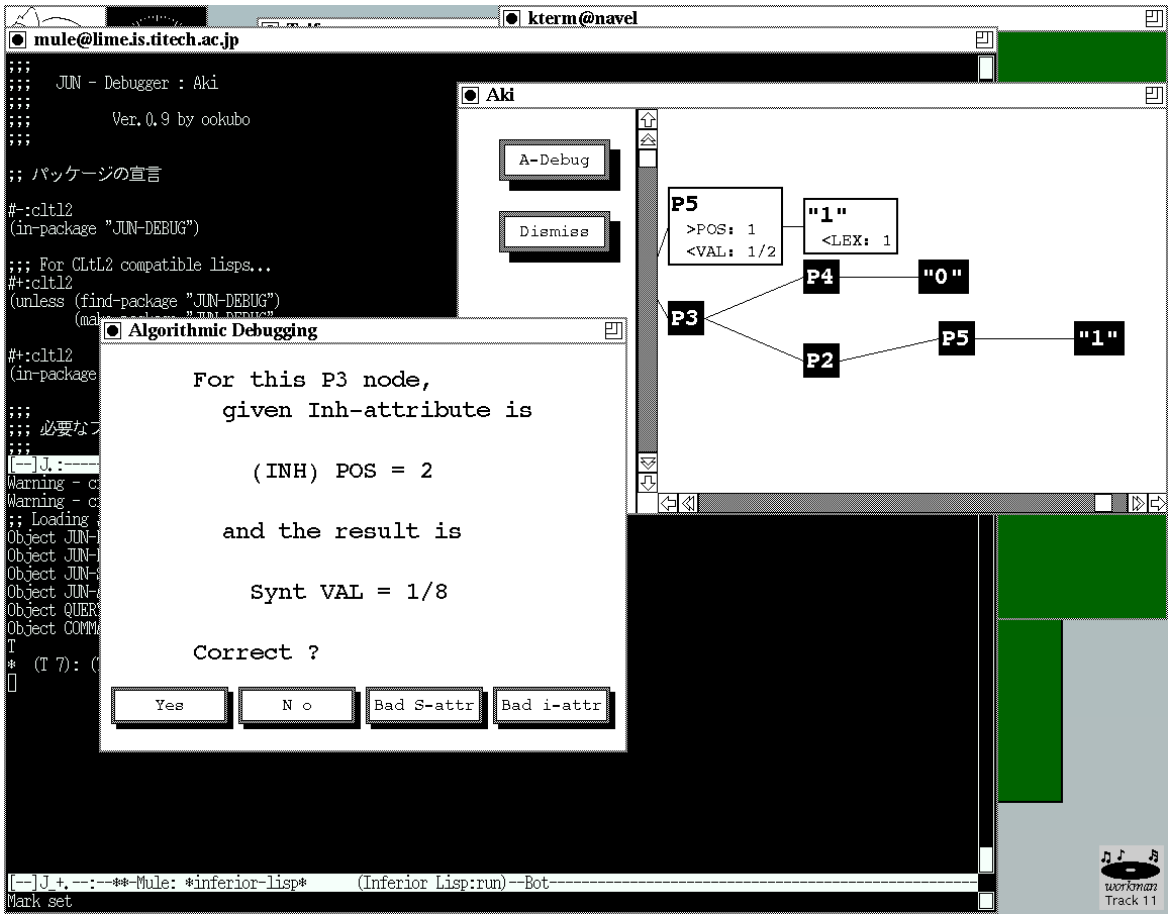


Figure 14: Example of Execution of Aki

# 7 Discussion

The primary objective of the method presented in this paper was to develop a debugging method where the user is not required to know the dependency among attributes. However, since there are many difficult aspects in the debugging of attribute grammars, we discuss in this section how these difficulties are minimized by the proposed method through various examples of debugging.

### Relation among Attribute Values

In many applications of attribute grammars, the only value the user needs to obtain is that of some attribute. If this value is erroneous, in order to debug it, the user has to check whether the relation among attribute values at the adjacent nodes of the parse tree matches the user's intention. At that time, it is difficult to know from the grammar description which attribute values should be checked, and even if the user can determine which one, the relation itself is often complex and difficult to understand.

In our method, since management of the dependency of attributes and selection of attributes to be checked are carried out by the debugger, the user does not need to perform these tasks. Moreover, in our method, the debugger can often show the relation of attribute values in a way that is understandable to the user.

For example, when the attribute is a long character string like the object code of a compiler, checking the relation among attribute values is difficult for the user. In the proposed method, this corresponds to the case where the input and output of the Synth function are both long character strings. Normally, the object code of a node of the parse tree is made by concatenating the object codes of the child nodes of the node in question and by inserting some instructions between them. In such a case, using our method, the debugger knows the input and output of the Synth function. Therefore, the debugger can identify the part of the output object code which corresponds to the input object codes. Then it can facilitate debugging by highlighting the corresponding part on the display, for example, by using different colors.

### Semantic Rules

After identifying which relation of attribute values is erroneous, the user needs to check the corresponding semantic rules or the subordinate functions used in these semantic rules. Since, in our method, the debugger can directly identify the set of erroneous semantic rules, the burden on the user is greatly reduced.

Once the semantic rules are identified, the user will check the semantic rules. Checking the top level semantic rules is straightforward, but from our experience, errors in the subordinate function occur often. In this case, the user will read and correct the description of the subordinate function, or he can actually execute the attribute evaluator and perform traditional debugging such as tracing the subordinate function.

### Long Sentence

As the character string given to the attribute grammar becomes longer, debugging becomes progressively more difficult, and this problem is not alleviated by the proposed method. This occurs, for example, when the user performs debugging of a compiler described in an attribute grammar by letting the compiler read a long source program. Even by our method, the debugger will show a big sub parse tree and a big symbol table to the user, and the user is forced to perform the arduous work of checking a long object code.

In such a case, the technique of narrowing the search space of bugs as presented in section 4 could be applied. However, it is advisable that the user shortens the source program as much as possible in advance so that it still causes the same bugs, as is performed by the traditional debugging method, and then apply the proposed method.

### Attribute Grammar Class

When the attribute grammar described by the user exceeds the range of the attribute grammar class that the system supposes, the user must modify the grammar description to meet the required class, but this process is not supported by the proposed method.

As stated above, our method can reduce the user's burden in debugging typical errors of attribute grammars, but there are still difficulties which are not supported by our method. Since the current implementation of Aki is a prototype debugger, we are planning to make a practical debugger which can solve the above problems for various attribute grammar descriptions.

# 8 Concluding Remarks

In this paper, a systematic debugging method for attribute grammars is presented based on the concept of algorithmic debugging, and its extension was shown taking advantage of features of attribute grammars. Then, after showing that this method can be applied to virtually all classes of attribute grammars in the correspondence of their attribute evaluators, we confirmed the method by implementing a prototype debugger which can directly handle an attribute grammar.

To date, no attempts have been made to design a systematic debugging method for attribute grammars. Therefore, developers of attribute grammar applications have been obliged to rely upon their experience and intuition. The proposed technique based on algorithmic debugging allows attribute grammar to be handled more easily, and we hope it can serve as a basis of development of actual applications using attribute grammars.

For future research, extension of the application range of the proposed debugger is needed. Since, in algorithmic debugging, the debugging proceeds on the assumption that output of the root of the computation tree is incorrect, in this paper we dealt with only the case where the attribute values of the root of the attributed parse tree are incorrect. Although this is usually sufficient, there are instances where attribute values other than those of the root node should be obtained, for example, in an attribute grammar for getting the data flow information for optimization. To cope with these cases, it will be necessary to consider an extension where the debugging starts from an inner error point of the computation tree.

In reference [13], the cases where the program aborts or does not terminate (i.e. enters

an infinite loop) are also considered. In attribute grammars, these cases correspond to error termination or to the cycle of attribute dependency (excluding well-defined ones [4]).

## Acknowledgments

The authors would like to thank Akira Sasaki who made improvements to Jun. Thanks are also due to Takashi Imaizumi, Ken Wakita, Akihito Mori and members of the Programming Language Laboratory for their useful comments and discussions.

A part of this research was supported by a Grant-in-Aid for Scientific Research from the Ministry of Education, Science, Sports and Culture.

# References


[1] Deransart, P., Jourdan, M., and Lorho, B.: *Attribute Grammars.* Lect. Notes Comput. Sci., Vol. 323, Springer-Verlag, 1988.

[2] Engelfriet, J. and Filè, G.: Simple multi–visit attribute grammars. *J. Comput. Syst. Sci.*, Vol. 24, pp. 283–314 (1982).

[3] Engelfriet, J. and Filè, G.: Passes, sweeps, and visits in attribute grammars. *J. ACM*, Vol. 36, No. 4, pp. 841–869 (1989).

[4] Farrow, R.: Automatic generation of fixed–point–finding evaluator for circular, but well–defined, attribute grammars. *Proc. ACM SIGPLAN '86 Symp. on Compiler Construction*, pp. 85–98, 1986.

[5] Fritzson, P., Shahmehri, N., Kamkar, M., and Gyimothy, T.: Generalized algorithmic debugging and testing. *ACM Lett. Prog. Lang. Syst.*, Vol. 1, No. 4, pp. 303–322 (1992).

[6] Hirunkitti, V. and Hogger, C. J.: A generalized query minimisation for program debugging. In Fritzson, P. (ed.) *Automated and Algorithmic Debugging, First International Workshop, AADEBUG '93*, Lect. Notes Comput. Sci., Vol. 749, pp. 153–170, Springer-Verlag, 1993.

[7] Katayama, T.: Translation of attribute grammars into procedures. *ACM Trans. Prog. Lang. Syst.*, Vol. 6, No. 3, pp. 345–369 (1984).

[8] Knuth, D. E.: Semantics of context-free languages. *Math. Syst. Th.*, Vol. 2, No. 2, pp. 127–145 (1968). correction: *ibid.* Vol. 5, No. 1, pp. 95–96 (1971).

[9] Myers, B. A., Guise, D. A., Dannenberg, R. B., Zanden, B. V., Kosbie, D. S., Pervin, E., Mickish, A., and Marchal, P.: Garnet : Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, Vol. 23, No. 11, pp. 71–85 (1990).

[10] Ookubo, T.: A debugger for attribute grammars (in Japanese). Master Thesis, Dept. of Information Science, Tokyo Institute of Technology, 1995.

[11] Ookubo, T., Sasaki, A., Wakita, K. and Sassa, M.: A debugger for attribute grammars (in Japanese). Preprints WG for Symbol Manipulation, 95-SYM-78, Inf. Proc. Soc. Japan, pp. 1–8 (1995).

[12] Sassa, M.: Rie and Jun: Towards the generation of all compiler phases. *3rd Int. Workshop on Compiler Compilers*, Lect. Notes Comput. Sci., Vol. 477, pp. 56–70, Springer-Verlag, 1991.

[13] Shapiro, E. Y.: *Algorithmic Program Debugging*. The MIT Press, 1982.

[14] Shinoda, Y. and Katayama, T.: Attribute grammar based programming and its environment. *Proc. 21st Hawaii Int. Conf. on System Sciences*, pp. 612–620, IEEE Computer Society Press, 1988.