記　号　処　理 78－1
（1995． 3． 24）

# 属性文法に対するデバッガ

大久保 琢也　佐々木 晃　脇田 建　佐々 政孝

東京工業大学 情報理工学研究科

属性文法は，コンパイラの定式化の分野などで広く用いられている．しかし，属性文法で記述したソフトウェアのデバッグ法は，いまだに確立していない．本研究は，論理型言語のデバッグ法として提案されたアルゴリズミック・デバッギングの手法による，属性文法デバッガを提案するものである．本稿では，属性文法に仮想的な関数を導入することにより，アルゴリズミック・デバッギングが適用可能であることを示し，さらに属性文法の特長を生かしたアルゴリズミック・デバッギングの拡張を提案する．また，本手法がほぼ全てのクラスの属性文法に適用できることを，属性評価器との対応で述べ，実装したデバッガのプロトタイプを用いて，以上の点を確認した結果を示す．

# A debugger for attribute grammar description

OOKUBO Takuya, SASAKI Akira, WAKITA Ken and SASSA Masataka

Graduate School of Information Science and Engineering,
Tokyo Institute of Technology

**The attribute grammar is a popular formal basis for compiler construction. However, there is no established method of debugging software written in attribute grammars. We modified the algorithmic debugging, which is a bug locating method developed for logic programming, and adapted it for attribute grammars. The paper proves that it is possible to apply the algorithmic debugging method to attribute grammars by using a virtual function. The paper proposes two extensions of the debugging method suited to more efficient debugging of attribute grammars. In addition, we show that our technique can be applied to almost all classes of attribute grammars by giving a correspondence with attribute evaluators. Based on the presented algorithmic debugging method, we implemented a prototype debugger.**

## 1 はじめに

属性文法 [1] は，1968 年に Knuth により提唱されて以来，コンパイラの定式化をはじめとするさまざま分野において，広く用いられてきた．しかしその一方で，属性文法記述によるソフトウェア開発に不可欠なデバッガの研究には，これまであまり力点がおかれてこなかったようである．

属性文法は，多くのプログラム言語などとは異なるパラダイムに基づいているため，一般に用いられているデバッグ法をそのまま適用することはできない．多くの属性文法アプリケーションでは，属性文法記述は手続き型言語で書かれた評価器に変換されるので，デバッグを行なう際にはこの評価器に対して手続き型言語のデバッガを適用し，その結果からもとの属性文法の誤りを推測するという方法が用いられている．しかし，このような方法では，ユーザが属性文法を手続き型言語として捉えることになり，属性文法の簡潔性などの利点が損なわれてしまう．そこで本研究では，属性文法を直接扱うことが可能なデバッガの開発を行なった．

ここで提案するデバッグ法の最大の特長は，アルゴリズミック・デバッギング [6] の手法の導入である．本研究では，この手法が属性文法に対して適用できることを示すとともに，属性文法の特長を生かしてさらに効率の良いデバッグが可能となるように，この手法を拡張した．また，本手法がほぼすべてのクラスの属性文法に対して適用できることを，属性評価器との対応で示した．

なお，本稿では紙面の制約から形式的定義や証明等をかなり省略した．詳しくは文献 [4] を見られたい．

## 2 属性文法

属性文法は，言語の構文と意味を一体化して記述することが可能な定式化手法である．本節では簡単な属性文法の例を示し，属性文法の概説を行なう．詳細については，文献 [1] を当たられたい．

属性文法とは，簡単にいえば，文脈自由文法に**属性**という情報を付加したものである．図 1に例として，2 進小数表現をもとにその値を計算する属性文法を示す．図の (1) のように → を含んだ行と，(4) のように | で始まる行が文脈自由文法の生成規則であり，それに続く {} に囲まれた部分が，その生成規則に対

```
F  →  . L                                     (1)
      { L.pos = 1;                            (2)
        F.val = L.val }                       (3)
L0 → B L1
      { L1.pos = L0.pos + 1;
        B.pos = L0.pos + 1;               (バグ)
        L0.val = B.val + L1.val }
    | B                                       (4)
      { B.pos = L0.pos; L0.val = B.val }
B  →  1
      { B.val = 2^-B.pos }
    | 0
      { B.val = 0 }
```

図 1 属性文法の例

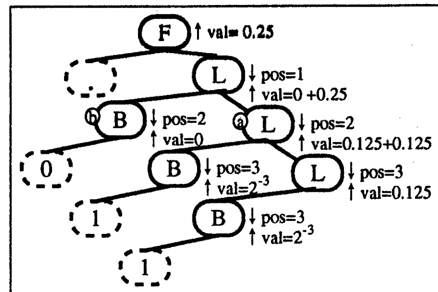


図 2 属性付構文解析木

応する属性の定義 (意味規則) である．例えば図の (2) は，文法記号 L の属性 pos が，値 1 を持つことを示している．

意味規則には，図の (2) の L.pos の定義のように，生成規則の右辺の文法記号の属性を定義するものと，(3) の F.val の定義のように，左辺の文法記号の属性を定義するものがある．前者により定義される属性を**継承属性**，後者により定義される属性を**合成属性**という．(3) の F.val は，L.val の値を基にして値が計算される．このような関係が成り立つとき，F.val は L.val に依存するという．

文法に従って文字列を構文解析した後，属性の値を定めることを，属性を評価するといい，属性を評価するソフトウェアを，**属性評価器** と呼ぶ．上の文法に従って ".011" という文字列を評価すると，図 2のような「属性付構文解析木」ができる．この場合は根の部分に属する属性 F.val が求める値となっている．なお，図 1の属性文法にはバグがあるため，この値は誤った値となっている．

## 3 属性文法に対するアルゴリズミック・デバッギングの適用

### 3.1 アルゴリズミック・デバッギング

論理型言語では，プログラムを「述語」と呼ばれる部品により構成する．アルゴリズミック・デバッギング [6] とは，実行時に起きた述語の振舞いが正しいものであるかどうかをプログラマに問い合わせながら，バグの存在し得る範囲を狭めてゆくという方法である．この方法は，手続き型言語や関数型言語にも応用できることが知られている．

ここで，文献 [6] に倣って，アルゴリズミック・デバッギングの説明をすることにする．図3に示すの

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

図 3 誤ったプログラム

は，Prolog で書かれた挿入法ソートのプログラムである．述語 isort は第１引数に与えられた list を昇順に整列して第２引数に返す．述語 insert は第１引数の要素を第２引数の list へ挿入し，第３引数に返す[1]．ただしこのプログラムは１箇所バグを含んでいるため，正常には動作しない．

ここで，アルゴリズミック・デバッギングを使ってこのプログラムのバグを見つける例を示す．今，ユーザがこのプログラムに [2,1,3] という引数を与えて，本来期待されている [1,2,3] という値でなく，[2,3,1] という誤った値を得たとしよう．この時デバッガは，この誤った計算の実行履歴を元に，ユーザに図4のような問い合わせをして，プログラム中のバグを発見する．下線部はユーザの入力を示す．ここでは４回の問い合わせから，デバッガがバグのあるクローズを推定した．このように，ユーザは個々の述語の内部に立ち入ることなく，それらの動作が正しいかどうかという質問に答えるだけで，バグを発見できる．

では，デバッガが内部でどのような動作を行なったかについて説明する．デバッガはまず，誤った計算の実行結果から，**計算木**と呼ばれる木を作る．これは，isort が [2,1,3] という引数に対する計算を行なう過程で，他の述語を呼び出した様子を，その時の引数と返り値とともに記録した，一種の実行履歴である．図5に，この時の計算木を示す．

[1]本来 Prolog には引数や返り値といった概念は存在しないが，ここでは説明の便宜上それらの用語を用いる

```
  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).
```

図 4 アルゴリズミック・デバッギングの例

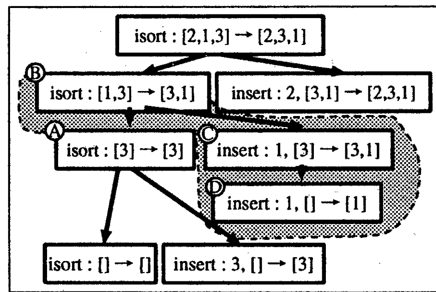


図 5 計算木

次にデバッガは，計算木のノードを１つ選び，ユーザに対しその部分の述語の動作が正しいものかどうかを問い合わせる．上の例では，まず図5のⒶに着目している．ここではユーザへの問い合わせから，「isort が [3] という引数に対し，値 [3] を返したのは正しい動作である」ことが分かった．このことから計算木の Ⓐ 以下が表す計算の過程以外の部分に，少なくとも１つバグがあることが分かる．

デバッガは次に図5のⒷに注目している．ユーザによればこの時の isort の動作は誤りであるので，計算木の Ⓑ 以下が表す計算の過程のどこかに，少なくとも１つバグがあることが分かる．

ここまでの問い合わせから，図5の網かけの領域が表す計算の過程にバグがあることが分かる．このように，ユーザへの問い合わせにより，バグを含む可能性のある領域をどんどん狭めることができる．

ユーザの答から Ⓒ は誤りで，Ⓓ は正しいことが分かる．これらの結果から「insert(1,[3],[3,1]) は

誤りだが，この過程で呼び出された述語の動作はすべて正しい」ことがいえる．このことは insert の定義のどこかにバグがあるということに他ならない．そこで，デバッガはその部分を，バグを含む行として出力したのである (正確にいうと，履歴を基にバグを含むクローズを特定している).

### 3.2 アルゴリズミック・デバッギングの形式的定義

ここで，アルゴリズミック・デバッギングの形式的定義をする．これは，文献 [6] にあるものを少し簡略化したものである．

前節の述語や，一般の言語の関数，値を返す手続きなどの，プログラムを構成する部品を，以下単に「関数」と表す．なお，ここでは副作用は考えない．

**定義 1** [挙動] 関数 $f$ が入力 $x$ に対し，出力 $y$ を返したとき，これを表す3つ組 $\langle f,x,y\rangle$ を関数の挙動という．

**定義 2** [トップレベル・トレース] ある挙動 $\langle f,x,y\rangle$ に対し，その計算の過程で起こった $f$ によるすべての直接の関数呼び出しについて，その時の挙動を呼び出しが起こった順に並べた列 $\langle f_1,x_1,y_1\rangle$ ，$\ldots$，$\langle f_n,x_n,y_n\rangle$ を考える．このときこの列を，挙動 $\langle f,x,y\rangle$ についてのトップレベル・トレースと呼び，$trace(\ \langle f,x,y\rangle\ )$ と表す．

**定義 3** [計算木] 関数の挙動をノードとする次のような順序木 $Ct$ を，挙動 $\langle f,x,y\rangle$ についての計算木と呼び，$ctree(\ \langle f,x,y\rangle\ )$ と表す．

1. 根は，$\langle f,x,y\rangle$
2. $Ct$ の全てのノードは，そのノードのトップレベル・トレースの要素を順に，第 1 子，第 2 子，... として持つ

なお，ノードの数が有限個であるような計算木を「完全な計算木」という．

**定義 4** [託宣] 託宣 $M$ は挙動の集合であって，プログラマが意図している，関数のすべての正しい振舞い (入力 $x$ に対する出力 $y$) を表している．挙動 $\langle f,x,y\rangle$ が $M$ に含まれる (含まれない) とき，$\langle f,x,y\rangle$ は正しい (誤っている) という．また，関数 $f$ の定義域中の任意の $x$ について，$\langle f,x,y=f(x)\rangle$ が常に $M$ に含まれる (含まれない) とき，$f$ は正しい (誤っている) という．

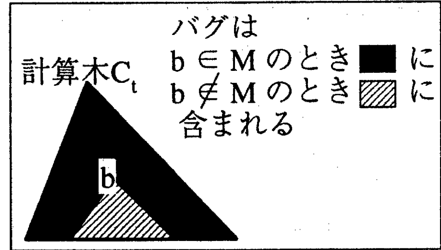


図 6 アルゴリズミック・デバッギングの原理

**定義 5** [バグ] 挙動 $b=\langle f,x,y\rangle$ について，$b\notin M$ かつ，$\forall b'\in trace(b); b'\in M$ がいえたとき，$b$ はバグを含むという．また，この時関数 $f$ もバグを含むという．

挙動 $\langle f,x,y\rangle$ が誤っているということは，$f$ がバグを含むか，$\langle f,x,y\rangle$ のトップレベル・トレースの中に，誤った挙動が存在するということである．

**定理 1** 完全な計算木 $Ct$ のあるノード $b=\langle f,x,y\rangle$ について，$b\notin M$ であれば，$Ct$ の $b$ 以下の部分木のどこかに，バグを含む挙動がある．

**定理 2** 誤った挙動 $\langle f,x,y\rangle$ を根とする計算木 $Ct$ が完全で，その中に正しい挙動 $b=\langle f',x',y'\rangle$ を含むとき，$Ct$ から $ctree(b)$ を除いてできる木 $\overline{Ct'}$ はバグを含む挙動をノードとして持つ．(証明は [6] 参照)

図 6は，定理 1，2を図に示したものである．

上の定理から，誤った計算が与えられたとき，図 7のような手順により，バグを含む関数を少なくとも一つ必ず発見できる．

```
Procedure Algorithmic-Debugging();
begin
  T := make_ctree(); { 計算木を作る }
  while T のノードが2つ以上ある do begin
    t := T の，根以外の任意のノード;
    if t が正しい挙動 then
      T から t 以下の部分木を削除
    else
      T := t 以下の部分木
  end;
  T の根に関する関数の定義を，バグとして表示
end.
```

図 7 アルゴリズミック・デバッギングの手順

このようにしてバグを見つける方法を，アルゴリズミック・デバッギングと呼ぶ．

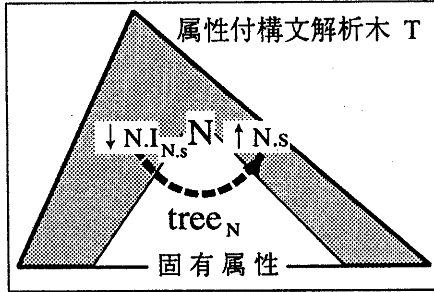


図 8 Synth 関数の原理

### 3.3 属性文法に対する適用

属性文法には，一般のプログラム言語における関数に相当するものが存在しない．しかし，属性文法の中の関数的な動作をする部分に着目し，その部分を関数とみなせば，アルゴリズミック・デバッギングを適用することが可能となる．

属性文法 AG が評価され，属性付構文解析木 $T$ と，$T$ のすべての属性値が求まったとする．また，この属性値が評価された時の，属性間の依存関係はすべて分かっているものとする．なお，以下 $T$ の構文解析木というときには，葉に付属する固有属性を含むものとする

このとき $T$ の，あるノード $N$ の合成属性 $N.s$ について考える．図8に示すように，この値は

1. $N$ 以下の部分構文解析木 $tree_N$
2. $N$ の継承属性 $N.I$ のうち，$tree_N$ において $N.s$ が直接間接に依存しているもの全体 $N.I_{N.s}$

により一意に定まる (証明は文献 [4])．

そこで，この値を定義する抽象的な関数

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

を考えることができる．この関数 $F_{N.s}$ を $N.s$ についての **Synth 関数**と呼ぶ．

この Synth 関数について，挙動，トップレベル・トレース，計算木，託宣，および，バグについて考えてみよう．

**挙動**は，Synth 関数の引数と値を使って，3.2節同様 $\langle F_{N.s}, \{N.I_{N.s}, tree_N\}, N.s\rangle$ として定義できる．

トップレベル・トレースについては，Synth 関数は抽象的なものであって，計算の途中で他の Synth 関数を呼び出すわけではないので，3.2節の定義を用いることはできない．そこで挙動 $\langle F_{N.s}$, $\{N.I_{N.s}$, $tree_N\}$, $N.s\rangle$ についてのトップレベル・トレース

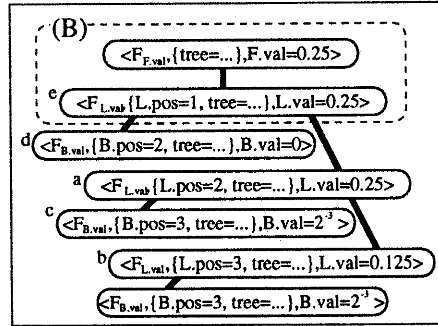


図 9 計算木

を，属性間の依存関係を用いて次のように定義する．

**定義 6** [Synth 関数のトップレベル・トレース] 属性付構文解析木 $T$ におけるノード $N$ の子ノード $N_1$, ..., $N_n$ の合成属性のうち，$N$ 以下の部分構文解析木において $N.s$ が直接間接に依存するもの全体を，それらの合成属性どうしの依存関係に従いトポロジカル・ソートして，列 $N_{i_1}.s_{j_1}$, ..., $N_{i_m}.s_{j_m}$ を作る．このとき Synth 関数の挙動の列 $\langle F_{N_{i_1}.s_{j_1}}$, $\{N_{i_1}.I_{N_{i_1}.s_{j_1}}$, $tree_{N_{i_1}}\}$, $N_{i_1}.s_{j_1}\rangle$, ..., $\langle F_{N_{i_m}.s_{j_m}}$, $\{N_{i_m}.I_{N_{i_m}.s_{j_m}}$, $tree_{N_{i_m}}\}$, $N_{i_m}.s_{j_m}\rangle$ を $\langle F_{N.s}$, $\{N.I_{N.s}$, $tree_N\}$, $N.s\rangle$ のトップレベル・トレース $\mathrm{trace}(\langle F_{N.s}, \{N.I_{N.s}, tree_N\}, N.s\rangle)$ とする．

**計算木**は，上のトップレベル・トレースを用いて3.2節同様に定義する．また，託宣 $M$ は Synth 関数の挙動の集合とし，バグの定義は，関数として Synth 関数をとることで，3.2節同様に定義する．すると，次がいえる．

**定理 3** 上のように定義した計算木において，ノード $\langle F_{N.s}, \{N.I_{N.s}, tree_N\}, N.s\rangle$ にバグがあることがいえたとする．このとき，解析木のノード $N$ に適用された生成規則 $p$ に付随する意味規則のうち，$N.s$ の定義か，$tree_N$ において $N.s$ が直接間接に依存する属性の定義のいずれかに，バグがある．

以上から，$T$ の根の合成属性が誤っていた時，3.2節同様にアルゴリズミック・デバッギングを適用すれば，バグが発見できることが分かる．

例として，図 1の 2 進小数の値を計算する属性文法について考える．誤った値が求まった図 2の属性付構文解析木から，図 9 のような計算木が得られる．

図 9の a に着目したとする．a は図 2のⓐ に対応し，「2 進小数の，小数点第 2 位以下に"11" という

数がある時，これが表す値は 0.25 である」という計算が行なわれたことを示している．

小数点第2位以下が"11" ならば，値は 0.375 である．よって，$a \notin M$ であり，そのことから計算木の a より下の部分にバグがあることが分かる．そこで，例えば次に b を調べると，同様に今度は $b \in M$ なので，続いて c を調べることにする．同じく $c \in M$ であることから，バグは a が表す挙動の定義，すなわち生成規則 L →B L に付随する意味規則の中で，L.val が依存する部分にあることが分かる．この場合は以下の {} の中全体がバグを含み得る．

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

## 4 アルゴリズミック・デバッギングの拡張

### 4.1 誤った入出力値

3.3節で述べた Synth 関数の入力 (解析木を除く) および出力は，それぞれ属性文法の継承属性，および合成属性に対応している．属性文法のプログラマは，それぞれの属性が解析木のどのノードに属するものかが分かれば，(時にごく容易に) その値が正しいものか否かを判定できることがあるはずである．

図9 でいえば，*B.pos* は小数点からの桁数を表すはずなのに，d (これは図2のⓑに相当する) では小数第1位を示すノードに対して $B.pos = 2$ が与えられている．このノードに着目した場合，プログラマは「$\langle F_{B.val}, \{B.pos = 2, tree = \cdots\}, B.val = 0\rangle$ は正しい挙動である」と思う前に，「*B.pos* が不当である」と気づくかも知れない．

そこで，著者はこのような情報をアルゴリズム・デバッギングに取り入れ，効率良くデバッグを進める方法を考案した．

**拡張アルゴリズム 1**

属性値が正当 (不当) とは，属性文法と構文解析木全体に照らしてその属性値が正しい (誤っている) と示せることをいう．

計算木 $Ct$ のノード $b = \langle f,x,y\rangle$ について，$x$ または $y$ が，正当である，もしくは不当であるといえた場合について考える．

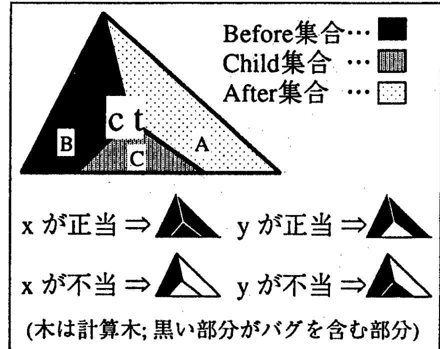


図 10 拡張アルゴリズミック・デバッギング

ここで，計算木 $Ct$ に含まれるノードを，任意のノード $ct$ を基準にして，次のように集合に分割することを考える．

**Child** $ct$ を根とする $Ct$ の部分木のノード全体 C

**Before** 次の要素よりなる集合 B

1. $ct$ の親もしくは祖先に当たるノード
2. 1 の要素の子どものうち，1 に含まれるもの (もしくは $ct$) より順序が前であるもの $ct'$ と，$ct'$ を根とする部分木のノード全体

**After** $Ct$ のノードのうち，B にも C にも含まれないノード全体 A

**定理 4** 誤った計算の計算木 $Ct$ のノード $ct = \langle f,x,y\rangle$ について，以下の 1 ～ 3 が成立する (図 10参照)．

1. $x$ が不当ならば，B の中にバグを含む挙動が存在する
2. $y$ が不当ならば，B か C の中にバグを含む挙動が存在する
3. $y$ が正当ならば，A か B の中にバグを含む挙動が存在する

先ほどの図9の例では，d の *B.pos* が不当であるという情報から，図の (B) にバグが存在することが分かる．調べると $e \notin M$ であることが分かり，(B) 中には e の子どもが存在しないことから，e にバグがあることがいえる．従って，L →B L に付随する意味規則の中の L.val が依存する部分にバグが含まれることが分かる．これは3.3節の結果と同等である．

### 4.2 難解な問い合わせ

アルゴリズミック・デバッギングの手順を先に進めるためには，ユーザはどんな問い合わせに対してもど

うにかして答を見つける必要がある．しかし，この問い合わせは，必ずしもユーザに答え易いものになるとは限らない．

そこで，ユーザに対し「分からない」という答を認めるように，アルゴリズムを拡張する．

**拡張アルゴリズム 2**

**定理 5** 計算木 $Ct$ 上のあるノード $ct$ について，

1. $ct$ の親および先祖のノードの，いずれかが正しい
2. それ以外のノードの，いずれかが誤っている

のいずれかが言えた時，$ct$ をバグ探索の対象から外すことができる．

この定理5から，ユーザがあるノード $ct$ に対し「分からない」という答を返したとしても，他のノードに対して問い合わせを行なえば，多くの場合 $ct$ を探索の対象から外すことができる．

逆に，デバッグの過程で，$ct$ について，

1. 親が誤っている
2. 兄弟はすべて正しい
3. 子どもはすべて正しい

のすべてがいえた時は，$ct$ を探索対象から外すことはできない．しかし，この場合にしても，$ct$ は正しいか，誤っているかのいずれかしかない，定義5から，$ct$ が誤っていれば，バグは $ct$ が示す関数の定義にあり，$ct$ が正しければ，バグはその親が示す関数の定義にある．そこで，これらの関数の定義の和集合を答として提示することにすれば，デバッグは完結したといえるであろう．

以上のように，アルゴリズミック・デバッギングの問い合わせには「分からない」という答を導入することができる．これはデバッグ法の能力そのものを向上させるものではないが，実際のデバッグには大いに役立つものといえるだろう．なお，この拡張も他の言語のデバッグに応用することができる．

## 5 本デバッグ法と属性文法クラスの関係

これまで Synth 関数は抽象的なもので，実際に存在する関数ではないとしてきた．しかし，あるクラスの属性文法では Synth 関数を直接に作成でき，また別のクラスでは，Synth 関数に基づく計算木を，既存の属性評価器から容易に求めることができる．

```
  procedure DEVAL(N0 : node);
  { N0 に適用されたプロダクションを
    p: X0 → w0 X1 w1… Xn wn とする }
  begin
1   count(N0) := count(N0) + 1; { 初期値は 0 }
2   j := count(N0);
3   Aj(X0) のすべての継承属性の値を定める;
4   for i := 1 to mj do { 子のノードを訪問 }
5       DEVAL(XVj(i));
6   Aj(X0) のすべての合成属性の値を定める
  end;
begin
  while count(root) < k(root) do
    DEVAL(root)
end
```

※ $X_i$ は非終端記号，$w_i$ は終端記号列，$A_j(X)$ は，X に対応するノードの j 回目の訪問で評価する属性の集合である．また，Vj(i) ($1 \leq i \leq m_j$) は，生成規則 p に対してあらかじめ与えられた「訪問列」と呼ばれる列であり，このノードを j 回目に訪れたときに，どの順に子を訪問すればよいかを表す．

図 11 単純多重訪問クラスに対する属性評価器

### 5.1 単純多重訪問クラスとその部分クラスの場合

単純多重訪問属性文法とその部分クラスの評価器は，ある属性評価器の特殊な形になることが Engelfriet らにより示されている [2]．そこで，これらのクラスについては，この評価器を基に計算木を作ることができることを示す．まず，単純多重訪問クラスに対する評価器は，図 11のようなものである [2].

図 11の一部を，図 12のように書き換えることにより，値として Synth 関数の計算木を返す関数が得られる．

文献 [2] によれば (以下「単純」を省略する)，多重訪問 (= $\ell$-ordered) 属性文法の部分クラスには，多重スイープ，多重交互パス，多重パス，さらに Ordered，単一訪問 = 単一スイープ，単一パス = L 属性がある．上により，このようなクラスの属性文法には，容易にアルゴリズミック・デバッギングを適用できることが分かる．

### 5.2 非循環および絶対非循環属性文法クラスの場合

絶対非循環属性文法クラスの場合は，拡張依存グラフに基づく片山の評価器 [3] の「合成属性の値を計算する手続き」が 3.2節の「関数」とみなせる．

```
function DEVAL(N0 : node) : 計算木;
   ...
4  for i := 1 to mj do { 子のノードを訪問 }
5       Tree(i) := DEVAL(Xvj(i));
6  Aj(X0) のすべての合成属性の値を定める;
7  return
    親が ⟨ f.Sj(X0),{Ij(X0),N0},Sj(X0) ⟩
    子どもが Tree(1) ··· Tree(mj) である計算
木
```

※ $I_j(X_0)$ と $S_j(X_0)$ は $X_0$ に付随する属性のうち，今回の訪問で評価した継承属性と合成属性を表す．

図 12 計算木を求めるための書き換え

非循環属性文法クラスの場合は，直接属性文法から拡張依存グラフを作成することはできない．しかし，属性付解析木が定まった段階で，その中の任意のノードの属性とその子どものノードの属性について，拡張依存グラフと同様のものを作成できる．これにより，絶対非循環属性文法の場合と同様に Synth 関数と計算木を作ることができる．

以上により，この 2 つの属性文法クラスにも，容易にアルゴリズミック・デバッギングが適用できることがいえる (詳しくは文献 [4] 参照).

## 6 プロトタイプ・デバッガ Aki の実装

著者は，これまで述べてきたアルゴリズムの有効性を確認するため， Aki というデバッガのプロトタイプを実装した．基にしたのは， Jun[5] という実験系である． Jun は，絶対非循環属性文法に，属性の依存関係のサイクルを許すようにしたクラスの文法が扱える． Aki は， Jun が扱うすべての属性文法に対しデバッグを行なうことができる．これにより，ほぼすべてのクラスの属性文法について，本稿で示したデバッグ法の有効性が確認されたと思われる．実行時の様子を図 13に示す．

## 7 おわりに

本稿では，属性文法に対する系統的デバッグ法を，アルゴリズミック・デバッギングの考えに基づいて示し，属性文法の特徴を生かしてその拡張を行なった．また，本手法がほぼすべての属性文法クラスに適用できることを，属性評価器との対応で示した．

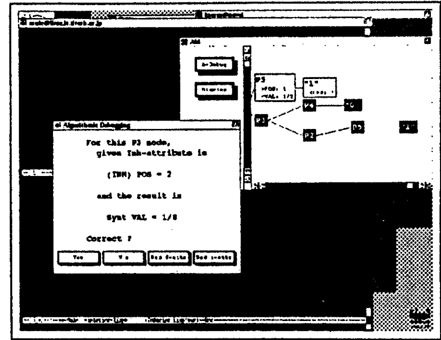


図 13 Aki の実行例

属性文法に対する系統的デバッグ法の研究は従来行なわれていなかったものである．そのため属性文法アプリケーションの開発者は，これまで経験と勘に基にした開発を余儀なくされていた．本稿でアルゴリズミック・デバッギングに基づく手法を提案したことは，今後属性文法を用いた本格的なアプリケーションを開発する上での基礎となる，意義のある研究であると考える．

デバッガ Aki はまだ作成されたばかりなので，実用規模の属性文法のデバッグに使われた実績がない．今後使用実績を積み上げる中で，問題点の洗いだしをする必要がある．今後さらなる改良を加え，最終的には統合属性文法開発環境の完成を目ざしたい．

## 参考文献

[1] Deransart, P., Jourdan, M., and Lorho, B.: *Attribute Grammars*, Lec. Notes in Comp. Sci., No. 323, Springer-Verlag, 1988.

[2] Engelfriet, J. and Filè, G.: Passes, Sweeps, and Visits in Attribute Grammars, *Journal of the ACM*, Vol. 36,No. 4(1989), pp. 841–869.

[3] Katayama, T.: Translation of Attribute Grammars into Procedures, *ACM Transactions on Programming Languages and Systems*, Vol. 6,No. 3(1984), pp. 345–369.

[4] 大久保琢也: 属性文法に対するデバッガ, 修士論文, 東京工業大学大学院理工学研究科, 1995.

[5] 佐々木晃, 徳田雄洋, 脇田建, 佐々政孝: 循環属性文法に基づく生成系 Jun について, 第 *50* 回情報処理学会全国大会講演論文集 *(5)*, 1995.

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