Vol. 35 No. 8　　情　報　処　理　　Aug. 1994

解　説

IPS

属性文法とその応用—VI

属性文法—現状と展望†

片山　卓也††

1. はじめに

本連載のこれまでの解説記事からお分かりのとおり，属性文法は主にコンパイラやエディタなどのような言語処理のための形式体系として D. E. Knuth によって導入されたものである[19]．コンパイラの記述と生成に関する応用については，佐々政孝氏による，またエディタについては今泉貴史，篠田陽一氏による解説記事にあるように商用システムについては別として，実験的システムの構成には十分な性能のものが機械的に生成できる段階に達している．

一方，Knuth も彼の論文の中で述べているように，属性文法を一般的な宣言的プログラミング言語と考えることもできる．属性文法の原理の一つは，それが木構造の上の属性値の純関数的な計算モデルであることである．このような立場に立って，非手続き的プログラミング言語や仕様記述言語としての提案や，言語処理以外のより広い問題領域への拡張を目指した研究も行われている．これについては本連載の中では松田裕幸氏による解説がある．言語処理系への応用ほどは十分な研究成果や応用は得られていないが，今後の研究が楽しみな分野である．

このように属性文法は，今後のソフトウェア開発の基本要素となるであろう自動生成や形式的仕様記述の一つの成功例と考えることができるが，商用プロダクト，実用規模の問題への適用や，より先進的な問題領域への適用を考えると研究すべき課題が多い．本稿においては，属性文法の計算モデルとしての整理を行い，今後の研究と応用について展望する．

2. 計算モデルとしての属性文法

属性文法の原理は，上に述べたように木構造上での関数的な属性計算である．木構造は通常の計算過程で頻繁に現れる最も重要なデータ構造である．たとえば，

- プログラム言語や自然言語の処理に現れる構文木
- 複雑な作業の分解を表す階層的構造
- オブジェクトの性質に基づく分類木やクラスの階層構造
- プロセスや動作の動的分解過程

など非常に多くの計算対象が木構造で表現でき，これらに関する多くの重要な問題が木構造上での属性計算として定式化できる．ここでは，このような立場から計算モデルとしての属性文法について簡単に説明をする．

2.1 属性文法の基本的定式化

属性文法は，本質的には，木構造中のノードの属性値を決定するための関数的な計算モデルであり，次のように定式化できる．

$$\mathcal{G}=(\mathcal{T},\mathcal{A})$$

ここで，$\mathcal{G}$ は属性文法であり，$\mathcal{T}$ は木構造の記述であり，また，$\mathcal{A}$ は属性記述である．$\mathcal{T}$ は属性文法 $\mathcal{G}$ が計算対象とする木構造の集合を表し，$\mathcal{A}$ は属性およびその関数計算のための規則を表す．

2.1.1 木構造の記述

属性文法でその処理を記述する木構造は，ノードがノード記号でラベルづけされた木構造であり，$\mathcal{T}$ は $\mathcal{G}$ の対象とする全ての木構造の集合を規定する．これは通常文脈自由文法によって与えられる．すなわち，$\mathcal{T}$ は木構造に現れるノード記号の集合 N と可能な木構造の形を決める木構造構成規則の集合 P の対である．

† Attribute Grammars, A Perspective by Takuya KATAYAMA (School of Information Science, Japan Advance Institute of Science and Technology, Tokyo Institute of Tcchnogy).

†† 北陸先端科学技術大学院大学情報科学研究科，東京工業大学情報理工学研究科

$\mathcal{G}=(N, P)$

木構造構成規則集合 P は，

$p: X_0 \rightarrow X_1, \cdots, X_n$

の形の規則 p の集合である．p は，ノード記号 X_0 のノードはノード記号 $X_1, \cdots, X_n$ のノード群に展開可能なことを表している．ただし，$X_i \in N$ である．Pから構成される木構造は，これらの木構造構成規則を繰り返し適用する，すなわち，規則を高さ1の木構造と考えて，それらのコピーを貼り合わせることによって得られる．

2.1.2 属性計算の記述

さて，木構造が与えられたとき，その中のノードにどのような属性を割りつけ，それらの値を具体的にどのようにするかは，属性記述 $\mathcal{A}$ によって与えられる．

$\mathcal{A}=(Att, Dom, Func, Def)$

Att は $\mathcal{G}$ に現れる全ての属性の集合である．また，木構造中のノード記号 $X(\in N)$ の各ノードには同一の属性集合が割り当てられるが，これを $Att[X](\subseteq Att)$ で表す．属性集合 $Att[X]$ は通常，

$$Att[X]=Syn[X] \cup Inh[X],$$
$$Syn[X] \cap Inh[X]=\emptyset$$

のように互いに素な属性集合 $Syn[X]$ と $Inh[X]$ に分割される．属性 $a \in Syn[X]$ を X の合成属性とよび，また $a \in Inh[X]$ を相続属性とよぶ．一般に，ノード記号 X のノード（以後，X ノードとよぶ）を含む木 T が与えられたとき，この X ノード r の相続属性は r を根とする部分木 T_X 中の属性値を決定するための入力データとして使われ，その結果として X の合成属性の値が決定される．この意味で，合成属性は T_X で行われる計算の出力データを表す．また，X の相続属性の値は X の親ノードから与えられる．（**図-1**）

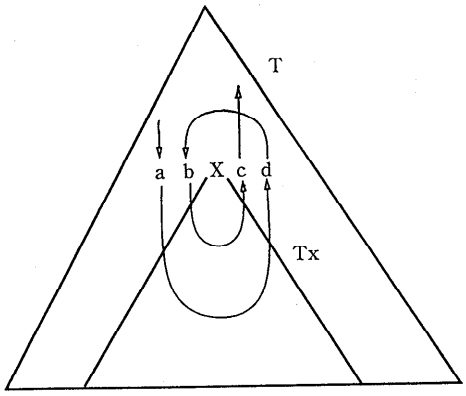


図-1 相続属性，合成属性 $(a, b \in Inh[X], c, d \in Syn[X])$

各属性にはそのとり得る値が決められているが，$Dom[a]$ は属性 a のとる値の集合を表す．$Func$ は属性計算に使われる基本関数の集合であり，Defは属性定義式の集合である．各木構造構成規則 $p \in P$

$p: X_0 \rightarrow X_1, \cdots, X_n$

には，p 中の X_i ノードの属性 $a \in A[X_i]$ が p 中の他のノードの属性 $a_1, a_2, \cdots, a_m$ $(a_k \in A[X_j],\ 0 \leq j \leq n)$ からいかに計算されるかを示す属性定義式

$$X_i.a=f(\cdots, X_j.a_k, \cdots)$$

が与えられている．ただし，$f \in Func.\ Def[p]$ は規則 p に付随したこのような属性定義式の集合を，また，Def は全ての属性定義式の集合 を表す．属性定義式は，通常の属性文法では，$i=0$ かつ $a \in Inh[X_i]$ であるか，$i \neq 0$ かつ $a \in Syn[X_i]$ に対してのみ与えられる．この制約は，木の中で属性値が一意に決定されることを保証するためのものである．また，記法 $X_i.a$ は，今考えている規則 p におけるノード記号 X_i の出現の属性 a を表す．これは p 中の複数のノード記号が同一の属性 a をもつ場合を考慮して，それらを区別するためのものである．なお，一般に，木構成規則 p においては，同一のノード名をもつノードが複数個現れることがあるが，属性定義式の中でそれらのノード記号の出現を区別するために，**図-2** の例のようにそれらには一連番号がふられているとする．

この例は，算術式に対するコードの生成のための記述の一部である．この記述では，親ノード $expression_0$ の属性 $symtable$ が子ノード $expres$-

ノード記号	$expression, term, +$
属性	$Inh[expression]=\{symtable\}$, $Syn[expression]=\{code\}$ $Inh[term]=\{symtable\}$, $Syn[term]=\{code\}$ $Inh[+]=Syn[+]=\emptyset$
木構成規則	$expression_0 \rightarrow expression_1 + term$
属性定義式	$expression_0.code=expression_1.code \| term.code \| [ADD]$ $expression_1.symtable=expression_0.symtable$ $term.symtable=expression_0.symtable$

図-2 属性文法による記述例

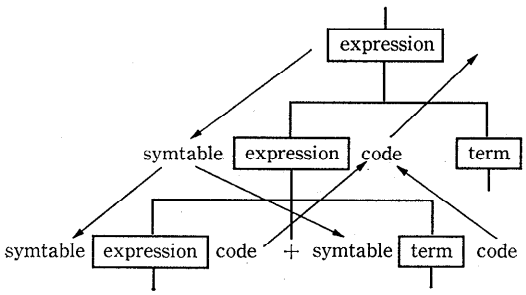


図-3 簡単なコンパイラにおける属性の依存関係

$sion_1$ と $term$ に渡され，(それらを根とする部分木で計算された) $expression_1.code$ と $term.code$ から親ノードの属性 $expression_0.code$ が合成されることを示している．また，'‖' は，$Func$ に属する関数であり，列の結合を表す．**図-3** は，木構造の中で属性がどのように関連しているかを示したものである．

2.2 属性文法の規定する計算＝構造指向関数計算

属性文法 $\mathcal{G}=(\mathcal{T},\mathcal{A})$ と $\mathcal{T}$ によって記述される木 T が与えられると，T の中のノードの属性値を決定するための関数的計算が規定される．

T を構成する各構成規則 p については，$\mathcal{A}$ によってそれに含まれる属性の値を計算するための属性定義式の集合 $Def[p]$ が与えられていた．これらの属性定義式は，X_0 の合成属性と $X_i(i\neq 0)$ の相続属性を他の属性から決定するもので，これによって p に含まれる属性生起の間の依存関係 D_p が定義されることになる．

$$D_p=\{(X_j.a_k, X_i.a)\mid X_i.a = f(\cdots, X_j.a_k, \cdots)\in Def[p]\}$$

この D_p は木構成規則 p にローカルなものであるが，これらから木全体に対するグローバルな属性依存関係を導くことができる．木構造は木構造構成規則を貼り合わせることにより構成されるが，いま，木 T の構成に用いられた構成規則を $p_1, p_2, \cdots, p_m$ とする．各 p_i に対する属性依存関係 D_{p_i} を，$p_1, p_2, \cdots, p_m$ から T が構成されたと同様に合成することにより，木 T の中での属性の依存関係 D_T を定めることができる．(D_T の構成に関する詳細については他の連載記事を参照されたい．)

もし，D_T に循環性がなければ，すなわち，$(v,v)\in D_T^*$ ('*' は閉包を表す) であるような v が存在しなければ，T の中のノードの属性の値を矛盾なく決定することができ，(決定可能な) 属性の値が全て定まった属性付き木 $A[T]$ が得られる．T が与えられたとき $A[T]$ を得る操作を属性評価という．また，

$$Eval_{\mathcal{G}}: T\rightarrow A[T]$$

である $Eval_{\mathcal{G}}$ を $\mathcal{G}$ に対する属性評価器という．ちなみに，属性文法 $\mathcal{G}$ は，$\mathcal{T}$ によって定義される全ての木 T に対して D_T に循環性がないとき，非循環な属性文法であるといわれる．非循環属性文法は，通常の方法で属性評価を行うことができる最も広いクラスの属性文法である．

次に属性文法の定める計算とは何であるかを考えてみよう．これは上でみたように属性文法は，木構造が与えられたとき，その上の属性値を決定するための計算モデルであるが，これは，次の二つのキーワードによって特徴づけることができる．

- 構造指向的計算
- 関数的計算

属性文法では，基礎となる木構造を前提にしており，この上での計算に限定していることが一般の計算とは異なるところである．これを利用して，計算内容も木構造構成規則と関連して記述されるようになっている．すなわち，属性文法における計算内容の詳細は，属性の計算がどのように行われるかを示す属性定義式によって表現されるが，これは木構造構成規則 $p\in P$ ごとに $Def[p]$ としてまとめて記述される．また，このような処理を通常のプログラミング言語で記述する場合に当然含まれる木構造自身の処理に関する記述，たとえば，部分木の取り出しなどは，属性文法の記述には含まれない．これにより，構造指向的計算を明快に記述することができるようになっている．

計算モデルとしての属性文法のもう一つの特徴は，それが関数的計算を採用していることである．木構造中の属性は他の属性から関数計算を通して決定され，これによって属性間の依存関係が明確になり，保守性や理解性の高い記述を可能にする．もちろん，関数的な計算を採用したことによって，基本的な属性文法では静的な計算しか記述できず，時間や状態といった動的なものを直接の計算対象とすることが困難になる．属性文法の

利点を保ってこの問題を解決するための試みについては後述する.

3. 構造指向計算モデルとしての属性文法

属性文法による計算では，一般に

木構造 T の構成$\Longrightarrow T$ 上の属性計算 $Eval[T]$

の形の計算が行われるが，この形に定式化できる問題は多い．伝統的に属性文法はコンパイラやエディタの記述に関して研究されてきたが，これは T として構文解析の結果として得られる構文木がとられた場合である．このような言語処理以外にも属性文法によって記述することが有効な問題が多い．ここでは，そのような問題のいくつかを概観してみる.

木構造は，計算過程の中で現れる最もポピュラーな構造であるが，これまでに属性文法による記述が試みられ，その有用性が確認されているものには次のものがある.

- 階層的構造データ処理の記述
- 作業・実行過程の分解の記述

3.1 階層的構造データ処理の記述

階層的構造データ処理の記述は，言語処理における構文木のように，処理すべきデータが深い構造をもっており，その構造に基づいた処理を行うことが必要な場合の記述である．このようなデータ処理の要求はコンパイラやエディタなどの言語処理のほかにもいろいろあり，ドキュメント処理，CAD，ファイルシステム，ソフトウェアデータベース，ビジネスフォームの処理などは属性文法による記述が適していると思われる．コンパイラやエディタなどの例は本連載にもあるのでここでは簡単なファイルシステムの記述を紹介しよう.

図-4 は，2分木状のバージョン木をもつファイルシステムを実際に記述したものの一部である．このシステムは RCS と同様の機能をもつバージョン管理システムと make と同様の機能の構成管理システムを合わせたようなシステムであり，Synthesizer Generator[8),21)] を用いて構築された．図-4 の記述は，そのうちのバージョン番号を決定する部分と，差分ファイルから各バージョンのファイルを構成するためのものである．このシステムの記述から，属性文法がファイルシステムのようなソフトウェアの記述に適していることが報告されている[10)].

階層的データ構造の処理の記述は，これまで最も実績のある属性文法の適用例であり，実用性のあるコンパイラやエディタの機械的な生成などが行われてきた．しかしながら，より広い応用を考える際にはいくつか考えなければならない問題がある．コンパイラなどのバッチ的処理を除くと多くの有用な応用がユーザとのインタラクション，既存システムとのインタフェース，処理対象のオブジェクトの永続性など，属性文法のもう一つの特徴である関数性と必ずしも親和性の良くない面を必要としていることである．一つの考え方は，Synthesizer Generator などでとられているように，組込みのユーザインタフェースによってこの問題を解決することであるが，もう一つの方法は，関数性とそうでない部分の境界を明確にし，その境界をある程度確立された標準的なインタフェースによって記述し，オープンなシステムとして実現することである.

3.2 作業・実行過程の分解の記述

作業や実行プロセスの分解の記述も属性文法による記述が適している例である．通常これらの作業や実行には，それに必要なデータや実行結果が付随するのが普通であり，それらはその作業や実行プロセスの属性として定式化される[16)]．次の例はこのような考え方にもとづいて，簡単なソフトウェア開発工程（JSP 法）の一部を記述したものである[17)]．一般に，ソフトウェアプロセスでは再実行や実行過程の柔軟な制御が要求される．属性文法に基づいたソフトウェアプロセスのための計算モデルが提案されている[24),11)]．(**図-5**)

ノード記号	*revision, log*
属性	*Inh*[*revision*]={*revbo, revno, prefiles*}, *Syn*[*revision*]={*deltafilelist*} *Inh*[*log*]=∅, *Syn*[*log*]={*deltafile*}
木構成規則	$revision_0 \rightarrow revision_1\ revision_2$
属性定義式	$revision_0.deltafilelist = revision_0.prefiles \| log.deltafile$ $revision_1.revbo = revision_0.revbo \| IntToString(revision_0.revno) \| [.]$ $revision_1.revno = 1$ $revision_1.prefiles = revision_0.deltafilelist$ $revision_2.revbo = revision_0.revbo$ $revision_2.revno = revision_0.revno + 1$ $revision_2.prefiles = revision_0.deltafilelist$

図-4 ファイルシステムの記述

ノード記号	*JSP, MakeProgTree, EnumOprs, MakeProg, DoProgInv*
属性	*Inh*[*JSP*]={*spec*}, *Syn*[*JSP*]={*prog*} *Inh*[*MakeProgTree*]={*spec*}, *Syn*[*MakeProgTree*]={*progTree*} *Inh*[*EnumOprs*]={*spec*}, *Syn*[*EnumOprs*]={*oprs*} *Inh*[*MakeProg*]={*oprs, progTree*}, *Syn*[*MakeProg*]={*prog*} *Inh*[*DoProgInv*]={*prog_in*}, *Syn*[*DoProgInv*]={*prog_out*}
木構成規則	*JSP→MakeProgTree EnumOprs MakeProg DoProgInv*
属性定義式	*JSP.prog=DoProgInv.prog_out* *MakeProgTree.spec=JSP.spec* *EnumOprs.spec=JSP.spec* *MakeProg.oprs=EnumOprs.oprs* *MakeProg.progTree=MakeProgTree.progTree* *DoProgInv.prog_in=MakeProg.prog*

図-5 JSP 法の記述とその説明

```
fibonacci(int.x|int.y)
    =if int.x==0 then return
                      where
                        int.y=1
                 else fibonacci(int.x−1|int.v)
                      where
                        int.y=int. x* int.v
```

図-6 フィボナッチ関数

また，次の例は属性文法を関数型言語として利用したものでフィボナッチ関数が記述されている[16]．fibonacci と return がノード記号であり，fibonacci は int.x と int.y の属性をつ．'|' の前の属性が相続属性であり，後のものが合成属性である．(**図-6**)

一般に，これらの実行や作業の記述では，それらの分解の過程が必ずしも始めから決まっているとは限らずに，実行によって決まるデータによって制御されるのが普通である．したがって，本節の始めに与えた図式「木構造 T の構成$\Longrightarrow$$T$ 上の属性計算 $Eval[T]$」をそのままの形では行うことができず，木の構成と属性計算を並行的に行い，属性計算の結果によって木の構成過程を制御する必要がある．上の fibonacci 関数の記述の例では，if の後に書かれる述語の評価によって木構造構成規則の適用を制御している．すなわち，int.x==0 の真偽によって，規則 fibonacci→return あるいは fibonacci→fibonacci が選ばれる．木の構成の後に属性評価に移る通常の属性文法では，木の上の複雑な属性依存関係を記述することが可能であるが，この種の属性文法で同様の属性の依存関係を実現するには，遅延評価の機構などが必要になる．

3.3 木構造の可能性と限界

属性文法との関連で木構造の典型的な二つの例をみてきたが，もちろんこれ以外にも木構造が有用な局面は数多くある．たとえば，オブジェクト指向プログラミングにおける性質の継承を表すクラスの構造などはその典型であろう．このような構造も属性文法を用いて記述することはもちろん可能である．このときには，クラス階層の構造を木構造構成規則として表し，クラスのもつ属性やメソッドをノードの属性とし，継承関係は属性定義式によって関数的に記述してやればよい．その用途は別にして，単なる一方向の継承のほかに多方向の性質の伝播を記述することが簡単にできる．

木構造の十分性については議論のあるところであろう．これは何も属性文法との関連に限ったことではないが，木構造では素直な記述が困難な問題も多い．上の例についていえば多種継承の記述には木構造では不十分であるし，また，作業・実行過程の分解については，並行プロセスの同期や join を表すには木構造では困難である．属性文法をより広い構造上に拡張することは，その応用領域を広げる意味で重要である[29),33)]．

4. 関数的計算モデルとしての属性文法

標準的な属性文法では，属性の値は他の属性の値から(純)関数を通して計算されるので，関数的計算モデルということができる．事実，前章の関数記述のように通常の関数を定義することももちろん可能であり，そのような関数型言語も考えられている[32)]．

属性文法が一般の関数的プログラミング言語と異なるところは，基本的には構文的側面についてである．通常の関数型言語が計算すべき値を表す「式」の構成という形でプログラムを記述するのに反し，属性文法では関数の呼出しを表すノード間の入出力データ間の関係として関数が記述される．単一代入による関数的プログラミングに対応すると考えても良い．図-2 で与えた簡単なコンパイラの一部の記述を *expression* の *code* を求める関数とみなして，Miranda ふうの関数として

記述したものを以下に示す．これを見ると両者の違いが明らかであろう．

type expression ::=*Exp expression*
　　+*term*|*Term term*
code(*Exp e*+*t*, *Symtab table*)
　=*code*(*e*, *table*)‖*code*(*t*, *table*)‖[*ADD*]

さて，関数的計算システムとしてみたときの属性文法は，定義すべき計算内容を

- 計算が行われる基底木構造
- 計算すべき属性
- 属性値の計算

という三つの独立した部分に分けて記述している．このような記述の分離は大規模なソフトウェアの構成や進化・保守では有利であると考えられる．ソフトウェアの仕様は，一般的には一度に与えられることは稀であり，その利用や評価を通して徐々に明確になっていくのが普通である．上のような記述の直交的な分離は，ソフトウェアの漸増的な構成法や進化に適していると考えられる．また，基底木構造の変更により，種々の計算対象を同一の枠組の中で記述することができることも直交的な記述の利点である．

5. 属性文法基本計算モデルの拡張

5.1 基本計算モデルの問題点

これまで述べてきたように属性文法基本計算モデルの特徴は，構造指向的計算と関数的計算である．また，その記述の基本は，(1)処理対象の木構造を表す木構成規則，(2)木構造中のノードに付けられる属性，(3)属性値の定義式，から成り立っており，おのおのの木構成規則ごとに独立に，閉じた形で行われる．すなわち，各規則 $p: X_0 \rightarrow X_1, \cdots, X_n$ ごとにその入力データ（X_0 の相続属性，X_i $(i \neq 0)$ の合成属性）と出力データ（X_0 の合成属性，X_i $(i \neq 0)$ の相続属性）とが明確に決まっており，それらのインタフェースを通してのみ各規則が関連する形になっている．

このような規則の集合としてプログラムを記述することにより，処理の概略を示す木構造，その上への属性の貼りつけ，属性値の定義という順番に詳細化を無理なく進めることができる．これにより，保守性や理解性が高いプログラムを得ることが可能である．その一方，計算モデルが基本的に純関数型であるために，記述できる対象が限られていること，および，たとえ記述ができる場合であっても各規則にわたる大域変数がないために，簡潔な記述が行いにくい，あるいは，能率の良い実行系が作りにくいなどの問題が生じる．純関数性の問題については次章で考えるとして，ここでは大域的な変数の問題について考えてみる．

5.2 大域変数

属性文法でコンパイラを記述する場合には，記号表や生成コードなどの大きなデータを構文木中を引き回さなければならないし，また，そのためにコピールールとよばれる，属性を単にコピーするための属性定義式を多数書かなければならない．標準的なコンパイラの記述では属性定義式の半数以上がコピールールであるという報告がある．

この問題を解決するためにいろいろな方策が取られてきた．木構造中の遠くのノードの属性を参照するための機構，コピールールの略記法，あるいはオブジェクト指向プログラミングにおける継承概念を採り入れたコピールールの省略記述[9]，さらには，大域変数とそれに対する書換え命令の導入など種々の方法が検討されてきた．遠いノードの属性の参照機構やコピールールの略記法などは限定された範囲内でそれなりの有効性は実証されてはいるが，これらの方法によってこの問題に対する十分な解決が得られているとは考えられない．また，場合によっては，これらの工夫によって属性文法本来の利点が失われることがありえる．たとえば，大域変数と書換え命令の導入は属性文法の宣言的性質を失わせ，その保守性や理解性を損なう原因になる．

今後ともこの問題に対して，新たな解決法が提案されると思われるが，ひとつの割切り方は基本計算モデルには手をつけずに，エディタやブラウザなどのプログラミング環境の工夫でプログラム構成上の問題点を解決し，属性評価アルゴリズムの改良によってコピールールの実現の効率化を図ることであろう．

5.3 基本計算モデルの拡張

属性文法の基本計算モデルでは，データの処理は「木構造 T の構成$\Longrightarrow$$T$ 上の属性計算 *Eval*[T]」として行われる．コンパイラなどのバッチ的処理ではこの図式で十分であるが，これに合わない処理も多く，属性文法のより広い応用を考え

るには，基本計算モデルの拡張を行う必要がある．ここでは，これらの拡張のいくつかについて述べる．

5.3.1 木構造の永続性

構造エディタの仕様記述とその自動生成は，コンパイラに次いで属性文法の応用が成功している例である．プログラムテキストの作成と同時にその意味的チェックを行うような構造エディタの記述では，その意味的チェックの内容が属性記述として行われる．プログラマによるテキストの編集作業によって，木構造が成長・変化し，それにともない属性計算を繰り返すことが行われる．これは，次の図式のように表すことができる．

$$T_0 \Longrightarrow Eval[T_0] \Longrightarrow T_1 \Longrightarrow Eval[T_1] \Longrightarrow \cdots$$

この過程では，$\Delta T_i = T_{i+1} - T_i$ に対応して，$\Delta E_i = Eval[T_{i+1}] - Eval[T_i]$ を求める漸増的属性計算技術 (incremental attribute evaluation, 木構造の変化 ΔT_i に応じて，新たに計算する必要のある属性値および変更のあった属性値のみを再計算することにより，木構造 T_{i+1} に対する属性評価 $Eval[T_{i+1}]$ を $Eval[T_i]$ から得る技術）と同時に，木構造 T を永続的なオブジェクトとして属性計算のサイクルの中に組み入れる必要がある．

5.3.2 属性計算の対象としての木構造：高階属性文法

Synthesizer Generator で導入された上の機構は，属性文法に新しい応用を開いたが，木構造の変更は（プログラマが編集作業を通して手作業で）属性文法の外側から与えなければならず，木構造の変更を計算の結果として行うことを困難にしている．属性計算の結果として木構造の変更を行わせる最も合理的な方法は，属性として木構造を許し，属性計算の結果を木構造に反映させることである．このような要求は，柔軟な編集環境の構成，ソフトウェア開発環境，多段階の翻訳過程の記述には本質的で，高階属性文法としていくつかの方法が提案されている[7),27),30)]．もちろん，新しい木構造の計算とともにその上の属性計算を行う必要があるが，この中で再び木構造が生成されるので，高階属性文法の属性評価は，一般的には基本属性文法のそれに比べてより困難な問題を含んでいる．高階属性文法のひとつである OOAG[7)] では，さらにそれまでにできあがった木構造を属性計算の中から変更することも許している．この機能を用いると，従来属性文法では記述されにくいとされてきたプログラム言語の動的セマンティックスの記述や，それに関連してインタプリタやデバッガの記述や機械的生成が行えることが期待される．

5.3.3 並行計算のための複数木構造モデル

これまでは，計算に現れる木構造はひとつであるという仮定のもとで考えてきた．通常の関数や逐次的な計算を記述するにはこれで十分であるが，通信プロトコルの記述[2)] など相互作用が本質的な並行的な計算の表現には新たな機構が必要である．属性文法のもつ高い理解性や解析性を生かしつつ，その適用領域を並行計算に広げることは，理論的に興味ある問題であると同時に実際的意義も大きい．

このような拡張の仕方としてはいくつか考えられるが，ひとつの方法は複数の木構造の間の相互作用を通して，並行計算のための計算モデルを構成することである[4)]．

二つの属性文法 $\mathcal{G}_1$ と $\mathcal{G}_2$ を考えよう．相互作用のひとつの考え方は，一方の文法，たとえば $\mathcal{G}_1$ における計算（木構造の構成を含む）によって $\mathcal{G}_2$ の木構造が変化・成長し，それぞれの木構造の上で属性計算が行われるというものである．おのおのの木構造が作られるメカニズムは，高階属性文法のように属性計算の中で考えることもできるし，あるいは，属性文法の外側で考えても構わない．次の例はこのような考えに従ってメッセージを介して動作する状態遷移システムを表現したものである．

この例は，人間と自動販売機の相互作用の一部を上の方法によって記述したものである．双対なノード $\overline{coin}$ と $coin$ の作用により，人間側の動作を表す木構造の成長によって自動販売機側の木構造が成長するメカニズムを仮定している．(図-7)

人間	$Start \rightarrow \overline{coin}\ M$ $M.paied = \overline{coin}.amount$ $M_0 \rightarrow \overline{coin}\ M_1$ $M_1.paied = M_0.paied + \overline{coin}.amount$
自動販売機	$Idle \rightarrow coin\ V$ $V.received = coin.amount$ $V_0 \rightarrow coin\ V_1$ $V_1.received = V_0.received + amount.coin$

図-7 属性文法による並行プロセスの記述

6. 属性評価

属性文法の記述を実際に「動かし」,必要な属性値を計算するためには，属性評価を行わなければならないことはもちろんである．属性文法 $\mathcal{G}$ からその評価器 $Eval_{\mathcal{G}}$ を構成する技術については，現在までに非常に多くの研究が行われてきた．これについての詳しい話しは本連載中の他の解説を参照されたいが，ここでは属性評価に関して今後研究すべきいくつかの話題について触れてみたい．

6.1 属性評価一般

基本的属性文法 $\mathcal{G}$ に対する評価器 $Eval_{\mathcal{G}}$ は，$\mathcal{G}$ が一般の非循環属性文法のときには，原理的には，木構造 T に対する属性依存関係 D_T を構成し，それをたどって必要な属性を計算することになる．文法 $\mathcal{G}$ に属性評価順序に関する制約を加えると，それに応じて属性評価器構成時に属性評価順序に関して事前の計算を行うことができ，その分だけ個々の T が与えられたとき（属性評価時）に行う計算量を減らすことができる．いろいろの制約条件（絶対非循環，順序つき，多重訪問・スイープ・パスなど）を満たす属性文法が定義され，それに対する評価器の研究が行われている[3]．また，実用的なコンパイラへの応用を考えると，構文解析と同時に属性評価を行うことが重要である．上昇型構文解析と並行して属性評価を行うことのできるLR属性文法は，実用的な観点からのみでなく，上昇型でありながら相続属性を許容しており理論的観点からも興味深い[13),22),28]．これらの概要については，木連載解説記事を読まれたい．

また，大規模問題に対する応用という点から，超並列マシン上での属性評価器の構成技術が今後重要になると思われる[1),18),20]．

一般の属性文法に対する評価器は，また，$\mathcal{G}$ を関数群に変換し，それを遅延評価することによってスマートに構成することができる．この方法によると循環性のある属性文法の評価が可能であり，属性文法の応用範囲を広げることができる[12),25]．これと関連して，最近，属性文法の意味をレコード計算によって与える方法が提案されているが，これによると高階属性文法も含めて評価器を非常に明解に与えることができる[31]．属性文法のより広い応用を考えるとき有効な手段になると思われる．また，循環性のある属性文法に対する上のアプローチは，関数方程式の最小不動点にその意味を求めるものであるが，属性依存関係を制約関係と考え，それにより循環性を解釈するという新しい見方が提案されている[33),34]．属性文法に新しい応用を開く可能性がある．

6.2 漸増的属性評価

木構造 T が与えられたとき，通常の属性評価器ではその上の全ての属性を計算する．もちろん，コンパイラなどの記述では，木構造の根ノードの合成属性としてオブジェクトコードが求められるように属性文法が作られるのが普通であり，最終的に必要な属性はただひとつということになる．しかし，これを得る過程において，属性評価器はそれが依存する属性を全て計算する必要があり，したがって,(無駄な属性が記述されていなければ）木構造上の属性を全て計算する．

エディタなどの記述では，この様子は変わってくる．プログラムテキストの編集作業と同時に，それに現れる変数のスコープや型の検査なども同時に行うことができる構造エディタが属性文法を用いて作られ，実用的に使われている．このエディタでは，編集作業によって木構造が漸増的(incremental)に構成あるいは変更され，そのたびごとに属性評価を行う必要が生じる．

このとき，バッチ的な属性評価をそのつど行うことは，原理的には可能であるが，計算時間の点で実際的ではない．漸増的属性評価器では，木構造の変化によって生じた分だけの属性の(再)評価を行い，計算時間の問題を解決している[21]．一方，これからのプログラム開発で重要になる分散開発では，ひとつの構文木を多人数が同時に編集できることが必要となるが，それに対応できる分散的漸増的属性評価アルゴリズムの研究が重要である[6),14]．

6.3 属性値の格納

基本属性文法に対する評価器に関して，重要な問題のひとつに属性値の格納問題がある．基本属性文法は純関数的計算モデルであり，「値」に関する計算のみを規定している．一方，属性評価器の構成ではこれらの値を実際の「場所」に格納する必要がある．小さな属性については大した問題ではないが，記号表のような大きな属性について

はその格納法の選択は評価器の実用性を大きく左右する重要な問題になる．これは，属性文法だけでなく純関数型言語一般に関する問題である．属性値の格納場所としては，木構造上，スタック，あるいは大域変数などが考えられる．このうちでも大域変数への割当ては，もしそれが可能なときには最も有効な方法である．属性値の格納問題はこれまでにいろいろ研究されてきたが[5),15),23),26)]，まだ十分な結果は得られていない．属性文法の今後の実用的応用のためにもより一層の研究を進めなければならない問題である．

7. おわりに

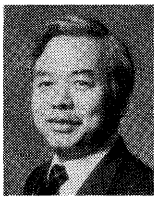
属性文法の計算モデルの整理を通して，今後の応用，特に新しい領域への応用の可能性について述べた．属性文法が Knuth によって提案されてから早くも 25 年が経過した．その間，主に言語処理系などへの応用を含めていろいろの研究がなされてきたが，研究プロダクトは別にして実用的ソフトウェアの構築への貢献は十分ではない．属性文法のもつ素性の良さを考えると，今こそさらに実用規模の問題へのチャレンジを積極的に行う時期であろうと思われる．最後に，本稿を書くに当たり，有益なコメントをいろいろいただいた菊池豊，今泉貴史，権藤克彦の諸氏に感謝いたします．

参考文献

1) Klaibar, A. and Gohale, M.: Parallel Evaluation of Attribute Grammars, *IEEE Transaction on Parallel and Distributed Systems*, 3: 206-219 (1992).
2) Chapman, N.P.: *Defining, Analysing and Implementing Communication Protocols Using Attribute Grammars*, volume 2, pp. 359-392 (1990).
3) Deransart, P., Jourdan, M. and Lorho, B.: *Attribute Grammars: Definitions, Systems, and Bibliography*, volume 323 of *Lecture Notes in Computer Science*, Springer-Verlag (1988).
4) Ding, S. and Katayama, T.: Specifying Reactive Systems with Attributed Finite State Machines, In *Proceedings of International Workshop on Software Specification and Design*, pp. 90-99, ACM (1993).
5) Farrow, R. and Yellin, D.: A Comparison of Storage Optimization in Automatically-Generated Attribute Evaluators, *Acta Informatica*, 23: 397-427 (1986).
6) Feng, A., Kikuno, T. and Torii, K.: *Incremental Attribute Evaluation for Multiple Subtree Replacements in Structure-oriented Environments*, volume 461 of *Lecture Notes in Computer Science*, pp. 192-206, Springer Verlag (1990).
7) Gondow, K., Imaizumi, T., Shinoda, Y. and Katayama, T.: Change Management and Consistency Maintenance in Software Development Environments Using Object Oriented Attribute Grammars, In *Object Technologies for Advanced Software* (*Proceedings of the First JSSST International Symposium*), volume 742 of *Lecture Notes in Computer Science*, pp. 77-94, Springer Verlag (1993).
8) GrammaTech, Inc.: One Hopkins Place, Ithaca, NY 14850, USA, *The Synthesizer Generator Reference Manual*, 1993. Release 4.1.
9) Hedin, G.: An Object-Oriented Notation for Attribute Grammars, In *Proceedings of the European Conference on Object-Oriented Programming* (*ECOOP '89*) (1989).
10) Imaizumi, T., Shinoda, Y. and Katayama, T.: Description and Implementation of File Management System Using Attribute Grammars, In *Proceedings of an International Conference organized by the IPSJ to Commemorate the 30th Anniversary*, pp. 143-150, Tokyo, JAPAN, Information Processiong Society of Japan (Oct. 1990).
11) Suzuki, M., Iwai, A. and Katayama, T.: A Formal Model of Re-execution in Software Process, In *Proceedings of the second International Conference of Software Processes*, pp. 84-99 (1993).
12) Johnsson, T.: *Attribute Grammars as a Functional Programming Paradigm*, volume 274 of *Lecture Notes in Computer Science*, pp. 154-173, Springer Verlag (1987).
13) Jones, N.D. and Madsen, C.M.: Attribute-Influenced LR Parsing, In *Lecture Notes in Computer Science 94*, pp. 393-407 (1980).
14) Kaiser, G.E. and Kaplan, S.M.: Parallel and Distributed Incremental Attribute Evaluation Algorithms for Multiuser Software Development Environments, *ACM Transactions on Software Engineering and Methodology*, 2(1): 47-92 (Jan. 1993).
15) Kastens, U.: Lifetime Analysis for Attributes, *Acta Informatica*, 24: 633-651 (1987).
16) Katayama, T.: HFP, A Hierarchical and Functional Programming, In *Proceedings of the 5th International Conference on Software Engineering*, pp. 343-353 (1981).
17) Katayama, T.: A Hierarchical and Functional Software Process Description and its Enaction, In *Proceedings of the 11th International Conference on Software Engineering*, pp. 343-

352 (1989).
18) Klein, E.: *Parallel Ordered Attribute Grammars*, Proc. of the International Conference on Computer Languages, pp. 106-116, IEEE (1992).
19) Knuth, D. E.: Semantics of Context-Free Languages, *Mathematical Systems Theory*, 2 (2): 127-145 (1968).
20) Kuipar, M.: *Parallel Attribute Evaluation: Structure of Evaluator and Detection of Parallelism*, volume 461, pp. 61-75 (1990).
21) Reps, T. W.: *Generating Language-Based Environments*, The MIT Press, Cambridge, Massachusetts 02142, London, England (1984).
22) Sassa, M., Ishizuka, H. and Nakata, I.: ECLR-Attributed Grammars: A Practical Class of LR-Attributed Grammars, *Information Processing Letters*, 24 (1): 31-41 (Jan. 1987).
23) Sonnenschern, M.: Global Storage Cells for Attributes in an Attribute Grammars, *Acta Informatica*, 22: 387-420 (1985).
24) Suzuki, M. and Katayama, T.: Meta-Operations in the Process Model HFSP for the Dynamics and Flexibility of Software Processes, In *Proceedings of the first International Conference of Software Processes*, pp. 202-217 (1991).
25) Takeda, M. and Katayama, T.: On Defining Denotational Semantics for Attribute Grammars, *Journal of Information Processing*, 5 (1): pp. 21-29 (1982).
26) Sasaki, H., Katayama, T.: Global Storage Allocation in Attribute Evaluation, *Proc. of 13th ACM Symposium on Principle of Programming Languages*, 22: 26-37 (1986).
27) Teitelbaum, T. and Chapman, R.: Higher-Order Attribute Grammars and Editing Environments, In *Proceedings of ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY (June 1990).
28) Tokuda, T. and Watanabe, Y.: An Attribute Evaluation of Context-Free Languages, Technical Report 93 TR-0036, Tokyo Institute of Technology (1993).
29) Tokuda, T. and Watanabe, Y.: An Attribute Grammar Modelling of Interactive Figures, In *Information Modelling and Knowledge Bases V*, pp. 212-226 (1994).
30) Vogt, H. H̃., Swierstra, S. D̃.. and Kuiper, M. F̃.: Higher Order Attribute Grammars, In *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (June 1989).
31) 権藤克彦：構造的オブジェクト指向計算モデルとその実行方式の研究，博士論文，東京工業大学，情報工学専攻 (3 1994).
32) 片山卓也，篠田陽一，菊池 豊，鈴木正人，今泉貴史，石原博史，高田 勝：自分自身のためのプログラム言語の作り方，Computer Today 別冊，サイエンス社 (Nov. 1988).
33) 菊池 豊：属性文法型計算モデルへの書換え系の応用，博士論文，東京工業大学 (1994).
34) 冨永和人，徳田雄洋：命令型制約文法とソフトウェアオブジェクト間の制約記述，In 1993 情報学シンポジウム講演論文集，pp. 147-156，日本学術会議他 (Jan. 1993).

(平成6年7月6日受付)



片山 卓也 (正会員)

1939年生．1962年東京工業大学理工学部電気工学科卒業．1964年同大学院理工学研究科修士課程修了．工学博士．同年日本IBM(株)入社．1967年東京工業大学工学部助手，1974年助教授，1985年教授．1991年北陸先端科学技術大学院大学情報研究科教授，現在に至る．ソフトウェアプロセス，ソフトウェアデータベース，ソフトウェア開発環境，ソフトウェア方法論，プログラミング言語，関数型プログラミング，属性文法などの研究を行っている．日本ソフトウェア科学会理事長．電子情報通信学会，ACM，IEEE 各会員．