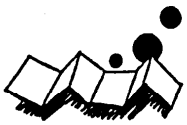
解　説



属　性　文　法†
——構造指向的かつ関数的計算モデル——

片　山　卓　也††

1. はじめに

属性文法が D. E. Knuth によって考え出されてから20年になるが[1]，この間，主にコンパイラの自動生成への応用という立場からさまざまな研究が行われてきた．当時はコンパイラの組織的構成法の研究が情報工学の大きな研究テーマの一つであったが，たとえば，やはり Knuth が，現在 Yacc などにおいて用いられている上昇型決定性構文解析の理論を発表したのはその数年前であった[2]．構文解析は文脈自由文法の構文記述への応用によって大きな進歩を遂げたが，コード生成のためにはそれだけでは十分ではなく，プログラミング言語の形式的意味記述が必要であった．Knuth はそれまでの研究成果，特に Irons などの仕事をもとにして[13]，基礎となる文脈自由文法の各非終端記号に意味記述に必要な種々の属性を付与し，その値を決定するための計算規則を各構文規則ごとに記述する方法を考案し，それを属性文法と名づけた．

以来，プログラミング言語の意味記述とそれからのコンパイラの自動生成を主な目的として，属性文法に関する理論および実際的面からの研究が続けられ今日に至っている．最近ではコンパイラ以外への応用もいろいろと試みられているが，20年も研究が続けられ，かつ新しい分野への応用が今なお開拓されつつあるのは，その素性の良さからくるものであろう．

コンパイラの自動生成については，研究室段階での使用に耐えるものは十分実用的に生成できるようになり，現在は商用コンパイラへの適用が試みられている段階といえる．解決すべき問題もいくつか残されてはいるが，属性文法による商用コンパイラの自動生成もそろそろ本格的に取り組む価値が十分にある時期にきているように思われる．ハードウェア技術の進歩により多様な CPU が比較的簡単にできるようになったが，それにともなって安価に，かつ，短期間にコンパイラを作成する技術が求められているし，また，一方ではスーパコンピュータのコンパイラのように高度な最適化機能がビジネスの大きな武器になっている．属性文法はこのような目的に対して非常に優れた道具となる可能性をもっている．

属性文法の研究がコンパイラ技術と関連して主に行われてきたことは事実であるが，これはその応用分野が単にコンパイラにだけに限られることを示すものではない．古くは，構造的パターンの記述に属性文法が用いられたことがあるし，最近では属性文法によって記述された CAD システムが現れたりしている[3]．また，コーネル大学で開発された構造エディタ生成システムでは，属性文法で記述された言語の仕様からそのための構造エディタを生成することができる[4]．さらには，ソフトウェア設計プロセスの記述やソフトウェアデータベースの記述に属性文法にもとづく体系を用いようとする試みもある[5),6]．

属性文法の基本的原理は，階層的構造上での属性値の関数的計算およびその構造指向的記述であり，非常に一般的な記述体系である．従来“文法”ということで言語的側面のみが強調されてきた感じがするが，構造的データの処理やソフトウェア工学を中心にして広い範囲の応用が期待できるものである．本稿ではこのような立場に立って，属性文法についての平易な解説を行ってみたい．

2. 属性文法の原理

2.1 簡単な例題による導入

2+3*5 のように数字と2項演算子+，*から構成される算術式の値を評価することを考えよう．これを行う一つの方法は，まず，この式を算術式に対する構文規則にしたがって構文解析し，その構造を明確にすることである．構文規則としては，例えば，次のものを用いることにする．

E→E+T

† Attribute Grammar—A Structure-oriented and Functional Computation Model—by Takuya KATAYAMA (Department of Computer Science, Tokyo Institute of Technology).

†† 東京工業大学情報工学科

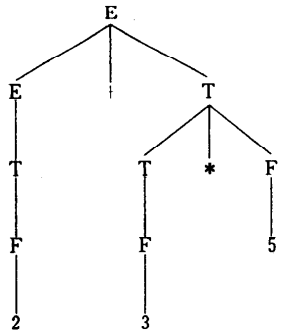


図-1　2+3*5 に対する構文木

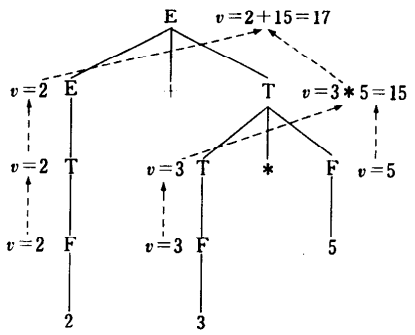


図-2　構文木中で属性 v が決定されてゆく様子

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow d$

規則 $F \rightarrow d$ においては，簡単のために，数字 0, 1, …, 9 を終端記号 d で代表させている．図-1 は 2+3*5 に対する構文木である．

構文木によって式の構造が表現されたら，次に，構文木中の各非終端記号に属性 v を付加する．v はその非終端記号を根とする部分木に対応する式の値を表しているが，その値は図-2 で示されるように構文木の葉から根に向って決定してゆくことができる．このプロセスは，各構文規則ごとに v がどのように決定されるかを示す規則——v の計算規則——を定め，これにしたがって v の値を構文木上で計算することにより実現することができる．たとえば，構文規則

$E_1 \rightarrow E_2 + T$

に対しては，次のような v の計算規則（意味規則という）を付加すればよい*．

$E_1.v = E_2.v + T.v$

この規則は，E_1 を根とする木構造の表す式の値 $E_1.v$ は，E_2 を根とする部分木に対する v の値 $E_2.v$ と，T を根とする部分木に対する値 $T.v$ の和となることを表現している．図-3 はこのようすを図示したものである．構文木全体に対してこの原理を適用すれば，与えられた式全体に対応する木構造の根 E の属性 v の値として式の値が求められることになる．図-4 はこの例題のための構文規則と v の計算のための規則をまとめたものである．

この例題は大変簡単なものであるし，この例にかぎっていえばもっと簡単な計算法があることはもちろんであるが，木構造上での属性値の計算という概念は理解していただけたと思う．これが属性文法の基本的原理である．

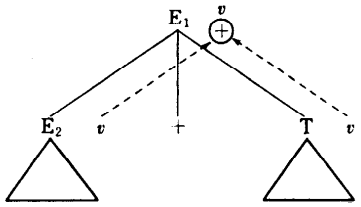


図-3　構文規則 E→E+T における属性 v の決定され方

構文規則	属性計算規則
$E_1 \rightarrow E_2 + T$	$E_1.v = E_2.v + T.v$
$E \rightarrow T$	$E.v = T.v$
$T_1 \rightarrow T_2 * F$	$T_1.v = T_2.v * F.v$
$T \rightarrow F$	$T.v = F.v$
$F \rightarrow d$	$F.v$ ＝数字 d の値

図-4　算術式に対する構文規則と属性計算規則

2.2　属性文法＝木構造上での属性値の計算システム

さて，話を一般論に戻そう．複雑なデータ構造やその処理の記述には木構造が現れることが多い．木構造は全体・部分関係，一般・特殊化関係，主作業・副作業関係など，複雑なものをより簡単なものに分解したり，あるいは，より一般的なものから特殊なものへの性質の継承などに現れるきわめて重要なデータ構造である．具体的には，言語の処理に現れる構文解析木，ファイルシステムのディレクトリ構造，CAD における部品の展開木，バージョン管理のバージョンの木，オブジェクト指向プログラミングにおけるクラス階層，などさまざまなものがある．また階層的設計プロセスにおけるプロセスの階層的分解構造や，手続きや関数の呼出し過程を表す計算の木などもこのような木構造の例である．

* E_1, E_2 はともに非終端記号 E を表しているが，v の計算規則のなかで E の異なる出現を区別する必要があるので添字をつけて区別してある．

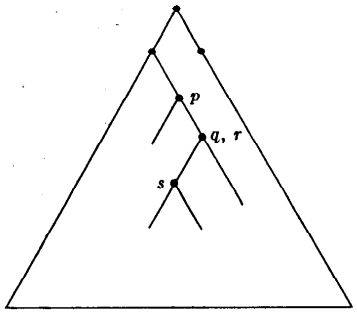


図-5 属性文法＝木構造上での属性値の計算システム体系

木構造の処理は，その構造の変更のほかに，前節でみたような木構造中の節点に付与される属性値の計算という形をとることも多い．たとえばCADのような例では部品関係を表す木構造のうえで各部品の重量や位置などの属性が問題となるであろうし，また，言語処理の例では構文解析木中の接点における意味属性が計算される．

属性文法はこのような木構造上での属性値の計算を記述するための一般的な記述体系である（図-5）．狭い意味では，言語処理の場合のように，記号列を構文解析して得られる構文木上での属性計算を記述するためのものと考えられることが多いが，その本質はあくまで木構造上での属性計算にある．

2.3 構造指向的かつ関数的記述

属性文法は，可能な木の形を記述するための構文規則（通常は文脈自由文法）と，木中の接点の属性値を定義するための意味規則から構成されている．

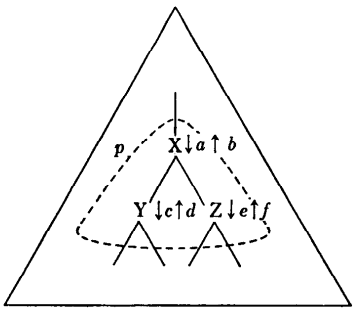
属性文法＝構文規則＋意味規則

木構造は構文規則を貼り合わせたものとして表現されるが，意味規則はこの木の中の接点（＝非終端記号）の意味（＝属性値）の計算を定義するためのものである．属性文法の特徴は，この意味規則が構文規則ごとに与えられている点であり，いわゆる，構文指向的記述，あるいは，構造指向的記述になっていることである（図-6）．意味記述の内容は，考えている構文規則に含まれる非終端記号の属性をその規則中のほかの属性を用いて定義する属性定義式の集合である．属性にはその構文規則内で陽に定義が与えられるものと，隣接する構文規則からその値が定まるものとがあるが，この区別については後で述べることにする．

属性文法のもう一つの特徴は，その意味記述のやり方が関数的であり，読解性，保守性に優れていることである．すなわち，属性定義式では，ある属性aがほかの属性$b_1, b_2, \cdots, b_n$を用いて定義されるが，この定義の仕方は，

$$a=f(b_1, b_2, \cdots b_n)$$

のように適当な関数fをとおして定義される．fが大域的変数などの“状態”あるいは“副作用”的要素を含まずに定義されていることは，記述の理解性や保守性を高めるうえできわめて重要である．これについては後でもう一度触れることにする．



構文規則 p: X→YZ
意味規則 $c=F(a)$
$e=G(a)$
$b=H(d, f)$

図-6 属性文法＝構文規則＋意味規則

2.4 相続属性と合成属性

図-6には構文規則 X→YZ に付随して三つの属性定義式が示されている．これらはXの属性b，Yの属性c，それにZの属性eを定義するためのものであり，他の属性a, d, fから関数F, G, Hをとおして定義されている．属性の前につけられている矢印↓，↑はその属性がそれぞれ相続属性，合成属性とよばれるものであることを示している．

木構造は構文規則を非終端記号を介して接続することにより得られるから，根と葉接点を除いては，同一の非終端記号は必ず二つの構文規則の接点として構文木に存在することになる．したがって，その非終端記号の属性はこの二つの構文規則のいずれに関連しても

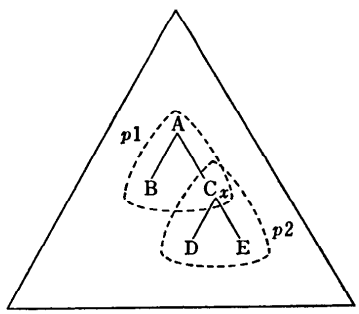


図-7 属性xの決定のされ方

その値が定義され，同一の属性に異なる二つの値が割り当てられる可能性が生じる．**図-7** はそのようすを示している．非終端記号 c の属性 x は規則 p_1 に関して定義することも可能であるし，また p_2 に関連して定義することも可能である．

これを防ぐためにすべての属性は相続属性か合成属性のいずれかに分類する．そして一つの構文規則に関しては，

（1） 左辺の非終端記号の合成属性
（2） 右辺の非終端記号の相続属性

非終端記号	P, D, E, T, F
終端記号	0, 1, ⋯, 9, $x, y, \cdots, =, +, *$
初期記号	P
属　性	P: $\uparrow v$, D: $\uparrow$tab, E, T, F: $\downarrow$tab$\uparrow v$

構文規則	意味規則
$P \rightarrow DE$	$P.v=E.v$, $E.tab=D.tab$
$D_1 \rightarrow D_2 n=d$	$D_1.tab=addtable(D_2.tab, n, d$ の値$)$
$D \rightarrow \varepsilon$	$D.tab=emptytable$
$E_1 \rightarrow E_2+T$	$E_1.v=E_2.v+T.v$ $E_2.tab=E_1.tab$, $T.tab=E_1.tab$
$E \rightarrow T$	$E.v=T.v$, $T.tab=E.tab$
$T_1 \rightarrow T_2*F$	$T_1.v=T_2.v*F.v$ $T_2.tab=T_1.tab$, $F.tab=T_1.tab$
$T \rightarrow F$	$T.v=F.v$, $F.tab=T.tab$
$F \rightarrow d$	$F.v=d$ の値
$F \rightarrow n$	$F.v=$findvalue $(F.tab, n)$

ただし，d は数字 0, 1, ⋯, 9 を，また，n は変数名 $x, y,$ ⋯ を表すものとする．

図-8 相続属性と合成属性をもつ属性文法の例

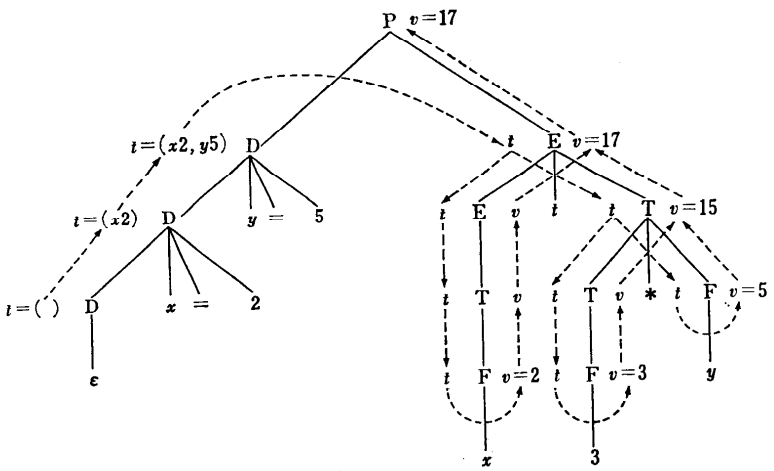


図-9 $x=2$ $y=5$ $x+3*y$ に対する属性つき構文木
ただし，属性 tab は記号 t によって表している．

のみを定義するようにする．これにより一つの属性が2重に定義されるという可能性を排除することが可能である．構文木中で，上の属性(1)，(2)のみがその構文規則内で定義され，他のものはその規則の外から与えられる．一般に，相続属性は構文木の中を上から下に，また，合成属性は下から上に伝播してゆく．

2.1 で考えた文法では，属性 v がすべての非終端記号に割り当てられていたが，これは合成属性であった．この例を少し拡張してみよう．今度は式に変数の使用を許すことにし，その代わり，その変数に対する値の割り当てはその前に行っておくことにする．すなわち，次の形のプログラムが与えられ，式の値を返すような属性文法を考えよう．

$x=2$　$y=5$　$x+3*y$

この文法は**図-8** のようになる．非終端記号 P はプログラムを，D は変数宣言部を，また E は式を表している．属性 tab が新たに導入されたが，これは変数とそれに割り当てられている値からなる表を表している．tab は D の合成属性であるが，E, T, F の相続属性である．上のプログラムに対する属性つき構文木が**図-9** に示されているが，属性 tab は D の部分で上向きに合成され，それが E, T, F に下向きに流され，変数に対する値の取り出しに利用されている．属性 tab に対しては，関数 emptytable, addtable および findvalue が適用されているが，それらの意味は字面より明らかであろう．

3. 属性文法の特徴

前章で述べたように属性文法の特徴は，その記述が構文指向的であり，かつ，関数的であることだが，そのほかには，記述がルール的であることをあげることができる．

3.1 構文指向的記述

記述が構文指向的であるということは，構文の記述と関連してそれに関する属性計算（属性定義式）の記述が行われることを指している．属性文法をプログラミングのための一般的なモデルと考える立場からすれば，構文の記述はデータ構造の記述ということになるから，これはデータ構造の処理に関する記述

をデータ構造の記述の一部として行うことを意味している．この考えは，オブジェクト指向プログラミングの考えと強い関連をもつものである．事実，非終端記号をクラス，属性名をメッセージ・セレクタ，属性定義式をメソッドと対応させることにより，属性評価の過程をメッセージパッシングとして定式化することは容易である．

記述が構文指向的であることは，属性文法によるプログラミングが複雑な構造の処理に適していることを示している．通常，このような処理では，まず対象となるデータ構造の形を大づかみに認識し，そのうえで処理の記述が行われるが，これはまさに構文指向的記述である．また，データ構造の形をそのままにして処理の内容を変えたり，追加したりするような場合にも属性文法的記述は適している．たとえば，図-4 では数字と2項演算子からなる算術式の評価のための記述が与えられ，また，図-8 には変数名を含むことを許した算術式の評価のための属性文法が与えられている．両者を比較してみると，図-8 の記述では名前表の処理のための構文と意味記述，式中の変数名を処理するための構文 F→*n* とそれに関する意味記述が追加されているが，式の処理に関する本質的な部分では構文および意味規則は基本的には同一であり，ただ名前表を表す属性 tab の伝播のための属性定義式が新たに付け加わっているだけである．

3.2 関数的記述

属性定義が関数的に行われることは記述の理解性や保守性を高める上で非常に重要である．すなわち，属性が関数的に定義されることにより，属性の評価順序や評価戦略とは無関係にその値が一意に定まることになり，これにより高い読解性が得られるというわけである．当然のことながら属性定義式の順序は属性値には無関係であり，また，一つの構文規則に関する意味記述のなかでは，同一の名前の属性はそれがどこに現れようとも同一の属性値を表している．

同じように構文指向的記述をするものでも Yacc のような方法では，各構文規則ごとに動作ルーチンが割り当てられ，その中での大域変数の操作をとおして属性計算が行われる．**図-10** には，変数を含んだ式の評価に関する動作ルーチン型の記述が与えられている．この形の記述では動作ルーチンの実行によって大域変数の内容が変化し，この変化を利用して計算が行われるので，動作ルーチンの実行順序，すなわち，構文木をたどる順序の指定が本質的である．図-10 は構文木

大域変数　*s*: スタック
tab: 名前表

構文規則	動作ルーチン
P→DE	output (top (*s*))
D→D*n*=*d*	addtable (tab, *n*, *d* の値)
D→ε	tab=emptytable
E→E+T	plus
E→T	
T→T*F	mult
T→F	
F→*d*	push (*s*, *d* の値)
F→*n*	push (*s*, findvalue (tab, *n*))

ただし，push (*s*, *e*) はスタック *s* に *e* をプッシュする手続き，top (*s*) は *s* の最上の要素を取り出す関数，plus, mult は *s* の上から二つのデータの和，積によってそれらを置き換える演算を表す．また，addtable (tab, *n*, *e*) は表 tab に対 (*n*, *e*) を追加するための手続きである．

図-10 変数を含んだ式の評価のための動作ルーチン形記述

を左から右，下から上にたどったときに正しく動作するように記述されている．図-10 は一見図-8 に比べて簡単なようにみえるが，常にこの順序を意識して読まなければならず，図-8 の属性文法による記述のように規則ごとに単独で理解するのが困難であり，属性文法に比べて読解性や保守性の点で劣っていると考えられる．

4. 属性評価と属性評価器

4.1 属性評価

構文木 T が与えられたとき，それ上の属性の値を決定することを属性評価という．また，属性評価を行うプログラムのことを属性評価器という．属性評価には，T 上の属性すべてを決定する場合と，特定の属——通常は T の根（初期記号）の合成属性——およびそれが直接・間接に依存している属性のみを求める場合とがある．コンパイラなどの場合には後者であり，初期記号の合成属性であるオブジェクト・コードおよびその評価に必要な他の属性が求められる．一方，構造エディタや CAD システムなどの場合には，T 上のすべての属性を評価することが要求される．

属性評価の基本は構文木 T 上の属性依存関係 D (T) である．これは T に現れる属性（のインスタンス）を属性定義式をもとにして関係づけたものであ

る．図-2，図-9 では，点線でこれが示されている．直観的には D(T) は次のようにして求められる．構文木は構文規則を貼り合わせた形になっているが，T を構成する構文規則（のインスタンス）を $p_1, p_2, \cdots, p_n$ とするとき，D(T) はそれらの各構文規則ごとの依存関係 $D(p_1), D(p_2), \cdots, D(p_n)$ を合成することにより得られる．構文規則 p に関する依存関係 $D(p)$ とは，p に関する属性を属性定義式にもとづいて関係づけたものである．図-3 は $D(p)$ の例である．

さて，D(T) が与えられたとき，T 上の属性を評価するには，値の分かっている属性からはじめて，D(T) を利用しながら順次属性値を決定してゆけばよい．もし，D(T) にサイクルがなければ T 中のすべての属性をこの方法で決定することができる．なお，任意の構文木 T に対して D(T) がサイクルを含まないような属性文法は，非循環であるといわれる．

逐次型計算機の上で属性評価を行うには，D(T) をトポロジカルソートすればよい．評価すべき属性集合 A の要素を D(T) に反しないように整列させる，すなわち，A の要素を

$a_1, a_2, \cdots, a_n$

ただし，D(T) において a_i は $a_k (k>i)$ に依存しない

のように一列に並べ，この順に a_i を評価すればよい．

4.2 属性評価戦略

上の方法は簡単であり，意味のある（非循環な）すべての属性文法に適用可能な方法であるが，効率的には問題がある．それは，$D(p_1), \cdots, D(p_n)$ を貼り合わせて D(T) を作る手間と，D(T) をトポロジカルソートする手間が T が大きいときには問題となるからである．属性文法のクラスを実用上問題ない範囲に制限する代りに属性評価の効率を上げるような工夫がいろいろと研究され，それに関連して属性文法のクラスが種々提案されている．それらは，大きく二つに分類することができる．

一つの方法は，構文木が与えられる前にできるだけの作業を行っておこうとするものである．すなわち，属性文法が与えられた時点で，各規則 p に関して $D(p)$ を作っておき，そのトポロジカルソートを行い，ついでに，$D(p)$ に対する部分的属性評価器 $\mathrm{eval}(p)$ を構成する．そして，具体的に一つの構文木 T が与えられれば，T を構成する規則 $p_1, \cdots, p_n$ に対応して，部分評価器群 $\mathrm{eval}(p_1), \cdots, \mathrm{eval}(p_n)$ を結合して T の属性評価を行おうとするものである．

この方法が可能であるためには，各 p に対して $D(p)$（正確には，構文規則 p の右辺の非終端記号の属性に関する可能なすべての依存関係を $D(p)$ に付加したもの）にサイクルが存在しないことが必要であるが，通常使われる属性文法に関してはこの条件は満たされている．このようなものとしては，絶対非循環属性文法や順序つき属性文法などが有名である．部分評価器 $\mathrm{eval}(p)$ の構成法やその結合の仕方などにより種種の属性評価器が考え出されている．

もう一つの方法は，属性依存関係 D(T) を構文木 T の適当な traversal のなかに埋め込んでしまい，その順に属性評価を行うならば D(T) の構成を不必要にしようとするものである．最も単純な場合には，木の左→右，上→下の traversal が考えられるが，もし T が下降構文解析によって決定的に構成できる場合には，T の構成自体も不必要にしてしまう．このようなことが可能な属性文法のクラスは十分に広いとはいいがたいが，多重の traversal によって評価可能な属性文法のクラスなどに関して多くの研究が行われ，実用的に重要なクラスを構成している．

属性評価戦略と属性文法のクラスとの関連については，たとえば，文献[7),8),9),14)]を参照されたい．

4.3 属性に対する記憶割り当て

これまでは，属性の評価順序に関係した話をしてきたが，属性評価器構成上考えるべきもう一つの問題は属性値を格納する記憶領域の管理の問題である．属性はその値が評価され，次に他の属性の評価に用いられるまでの間どこかに格納しておかなければならない．一方，属性文法は純粋な関数型プログラミング言語であり，論理的には値を格納する場所として変数やその値の変更という概念をもたない言語であり（このために属性文法による記述の明解さが得られているのであるが，その代償として），新しい値のコピーが次々に作られてゆく．したがって，不要な値をいつまでも保持しないようにする工夫や，大きな構造体のコピーをなるべく作らないようにして，できるだけ既存の値の部分的変更などの手段によって済ませるなどの方法が非常に重要になってくる．現在のところこの問題は十分には解決されておらず，また，実用的コンパイラの生成のためには解決しなければならない重要な問題である．具体的な属性値格納法としては，(1)構文木の接点上に格納する，(2)スタック上に格納する，(3)大域領域上に格納する，(4)動的記憶管理によってヒープ上に格納する，などの方法が考え出されている

が[10]，これらの具体的内容については本稿の範囲を越えるので省略する．

5. 属性文法の応用

5.1 構文木の不変なシステム

属性文法の導入以来，それによるコンパイラの記述と生成の研究が続けられてきており，コンパイラの試作などに使用されてきた．現在のところ商用コンパイラの生成にはもう少しというところがあるが，属性文法にもとづくコンパイラ生成系を売りに出す会社が現れるなど，実用的コンパイラ生成に向けて確実に歩んでいると思われる．各所で作られたコンパイラ生成系については文献[7),9)]を参照されたい．

さて，属性文法のコンパイラへの応用について特徴的なことは，構文木が一度だけ構成され，それが属性評価の過程で不変であることである．

構文木→[属性評価]→属性つき構文木

この形の応用としては，コンパイラのほかに，最適化器やテキスト清書系，自然言語も含めた翻訳系などがある．

5.2 構文木が変化するシステム

コンパイラとならんで，属性文法の応用が最近盛んに行われているのが構造エディタである．これは属性文法で書かれたプログラミング言語の仕様（実は，エディタからみた仕様）から構造エディタを自動生成しようとするものであり，Cornell Synthesizer が有名である[4]．これは単に構文指向的にプログラムの編集ができるばかりでなく，型のチェックや名前のスコープのチェックなどを編集時に行ってくれるものである．

このシステムの中心は属性つき構文木で，使用者によるプログラムの変更はただちに構文木に反映され，その結果として新しい構文木上で属性値の再評価が行われる．

構文木→[属性評価]→属性つき構文木
→[編集]→変更された構文木→[属性評価]
→属性つき構文木→[編集]→…

編集作業は，部分木の切り出しと，そこへの新しい部分木の挿入という形で行われるが（図-11），その結果，接合点で属性値の不一致が生じる．新しい木全体に関して属性の再評価を行うことは大変であるので，

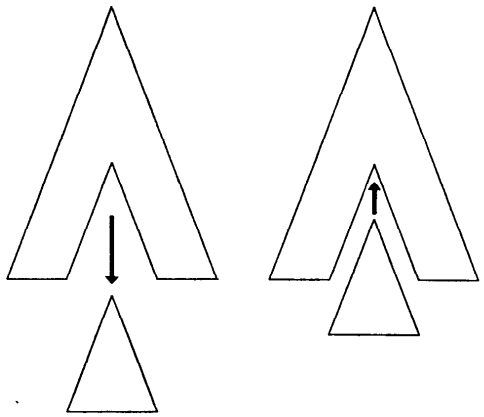


図-11 部分木の切り出しと挿入

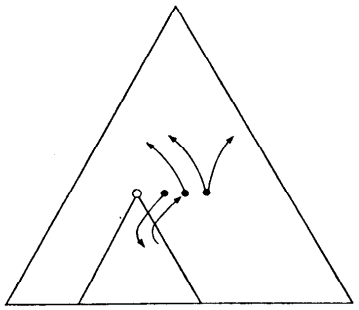


図-12 属性伝播による再評価

接合点での不一致がどこまで波及するかを分析し，不一致の検出された部分に関してのみ incremental な属性評価を行う（図-12）．これにより編集作業中常に正しい属性つき構文木が保持されるようになっている．

この形の属性文法の応用には広いものがある．属性文法を適用して作られた商用の CAD システムの例や，Unix などの階層的ファイルシステムの記述の試み，会話的定理証明システムに適用した例などが報告されている[15]．そのほかにも，バージョン管理システムやソフトウェアデータベースなどの広範囲の応用が考えられる[6]．

この種の応用を考える際には，評価された属性値を保持しておく機構，それを変化させ，さらには，構文木自身も変化できるようなメカニズムが必要になる．これは本来の属性文法にはない部分である．現在は属性文法とは切り離したところでこれらに関する記述が行われており，このために記述できない機能があったり，あるいは不透明な記述になったりしている．属性文法の長所は残したままその拡張を行う研究が行われているが，属性文法とオブジェクト指向パラダイムと

の結合などが試みられている[6].

5.3 構文木が陽には存在しないシステム

構文木がデータとしては存在せず，属性評価の過程のみを利用した計算モデルを考えることができる．この計算モデルは，モジュールとモジュール分解という概念にもとづいている[11].

モジュールは入力と出力のあるブラックボックスで，多入力・多出力の関数を表している．モジュール M の行うべき仕事が簡単な場合には，その出力は入力データを使って書き下せるが，そうでない場合には，M は下位モジュール $M_1, M_2, \cdots, M_n$ に分解される．

$$M \rightarrow M_1 M_2 \cdots M_n$$

このようなモジュール分解を繰り返し，自明なモジュールに到達したらプログラムができ上がったと考える．

属性文法との関連でいえば，モジュールは非終端記号に対応し，モジュール分割は構文規則に対応する．属性定義式が構文規則に付随していたように，モジュール分解にも属性定義式が付随する．これは，(1) 親モジュール M の入力からいかにして子モジュール M_i の入力を定めるかと (2) 子モジュールの出力から親モジュールの出力をいかに計算するかを定めるのに用いられる．

一般に，あるモジュールはいく通りもに分解される可能性があるが，どのモジュール分解が適用されるかはモジュール分解に付随している分解条件によって決められる．属性計算が行われる木構造は計算が始まる前には存在せず，モジュール分割が進んでいくようすを表すモジュール分割木がこの木に対応しており，これは，属性計算と並行して incremental に構成される．

この計算モデルは関数的プログラミングのための一つのスタイルを提案している．通常の関数的プログラミング言語では，関数適用や関数合成などによって関数形として関数が定義されるのに対し，この計算モデルでは，関数は対応として定義されている．プログラミングのスタイルだけにかぎっていえば，論理型プログラミングにおける，ゴールのサブゴールへの分解，というプログラムの書き方に似ている．作業の部分作業への分解という原理にもとづくこの計算モデルは，複雑な処理の階層的記述に適している．この計算モデルにもとづく関数型言語 AG とそのプログラミング環境が構成されているが[12]，次のものは AG によるプログラムの例である．

```
/*フィボナッチ数の計算*/
type x, y=int
module fibonacci (x|y)=
  case x≥2
       ⇒fibonacci (x−1|y.1)
         fibonacci (x−2|y.2)
         where
           y=y.1+y.2
     x=0 or x=1
       ⇒return
         where
           y=1
  end
end fibonacci
```

fibonacci $(x|y)$ は入力属性 x，出力属性 y をもつモジュール fibonacci を表している．case 句は分解条件を，また，where 以下では属性定義式を記述する．return はモジュール分解を停止させるための空モジュールである．

この言語は階層的分解という観点から関数的プログラムを記述しようとするものであり，構造指向的処理や階層的処理を明解に記述するのに適している．ソフトウェアの設計プロセスの記述をこの言語で行う試みが行われている[5].

6. おわりに

属性文法について，特に，その構造指向的かつ関数的計算モデルという側面に焦点をあてて解説した．コンパイラの自動生成への応用もそろそろ実用化を考えても良いころになり，また，構文エディタや CAD などへの応用も行われ，その有用性が認識されてはいるが，その良さが一般にはよく知られていないことも考えて，原理的面や可能な応用などについてやさしく述べたつもりである．その基本的性質の良さを考えると，今後ともその有用性が増すと考えられるが，この小論により属性文法に興味をもたれる方が増えることを望むものである．

参考文献

属性文法に関する文献は非常に多数あるが，ここでは本稿の主題に関係あるもののなかからいくつかを選んだ．属性文法一般については，文献 7), 8), 9), 14) に多数の文献が整理されている．

1) Knuth, D. E.: Semantics of Context-Free Languages, Math. Syst. Th., Vol. 2, No. 2 (1968), pp. 127-145. 訂正, ibid, Vol. 5, No. 1, pp. 95-96 (1971).
2) Knuth, D.E.: On the Translation of Languages from Left to Right, Information and Control, Vol. 8, No. 6, pp. 607-639 (1965).
3) ICAD System, Computer Aided Design Report, Vol. 5, No. 12 (1985)
4) Reps, T. W.: Generating Language-Based Environments, MIT Press, 138 pp., 1984. または, Reps, T. and Teitelbaum, T.: The Synthesizer Generator, Proc. ACM Soft. Eng. Symp. on Practical Soft. Dev. Env., SIGPLAN Notices, Vol. 19, No. 5, pp. 42-48 (1984).
5) 片山卓也, 生田淳三: 階層的・関数的アプローチによるソフトウェア設計プロセスの記述, 日本ソフトウェア科学会第4回大会論文集 pp. 79-82 (1987).
6) 篠田陽一, 片山卓也: オブジェクトベースの属性文法による記述, 同上, pp. 83-86.
7) 佐々政孝: 属性文法, コンピュータソフトウェア, Vol. 3, No. 4, pp. 73-91 (1986).
8) Aho, A. V., Sethi, R. and Ullman, J. D.: Compilers, Adison Wesley (1986).
9) Deransart, P., Jourdon, M. and Lorho, B.: A Survey on Attribute Grammars, INRIA Research Report, Part I, II, III (1985).
10) Farrow, R. and Yellin, D.: A Comparison of Storage Optimizations in Automatically-Generated Attribute Evaluators, Acta Inf. Vol. 23, No. 4, pp. 393-427 (1986).
11) Katayama, T.: HFP: A Hierarchical and Functional Programming Based on Attribute Grammar, 5th Int. Conf. Soft. Eng., pp. 343-352 (1981).
12) Shinoda, Y., Katayama, T.: Attribute Grammar Based Programming and its Environment, 21th HICSS., pp. 612-620 (1988).
13) Irons, E. T.: A Syntax Directed Compiler for Algol 60, Comm, ACM, Vol. 4, No. 1, pp. 51-55 (1961).
14) Lorho, B. (ed): Methods and Tools for Compiler Construction, Cambridge Univ. Press (1984).
15) Reps, T. and Alpern, B.: Interactive Proof Checking, 11th ACM Symp. on POPL pp. 36-45 (1984).

(昭和63年1月5日)