属性文法　－　チュートリアル　－

佐々　政孝　（筑波大学　電子情報工学系）

１．はじめに

コンパイラやプログラミング言語に関係するソフトウェア・ツールにおいて、属性文法に基づくものが目にとまるようになってきた。属性文法（attribute grammar）は、プログラミング言語の意味（semantics）の記述のために Knuthによって提案されたもので[Knu]、一口でいえば、

属性文法＝文脈自由文法＋意味規則

である[片山]。本論では、この属性文法について、例をまじえながら基本的事項と主な理論的結果を解説する。（なお、本稿は[佐々$_{84}$]に負う部分のあることをお断りする。）

２．属性文法とは

２．１　簡単なプログラミング言語の例

まず、属性文法について例を用いて説明しよう。ここでは変数の宣言と変数の使用だけからなる簡単なプログラミング言語を考える。簡単のため変数のスコープに入れ子はないものとする。変数の宣言は「declare V;」の形、使用は「use V;」の形とする。宣言した変数だけが使用できるものとする。同じ変数を２度宣言したり、宣言しない変数を使用すると、エラーメッセージが出される。たとえば、

declare x; declare y; declare x;　　　(S1)
　　　　　　　　　　　　　　↑
　　　　　　　　　　二重宣言のエラー

declare x; declare y; use y; use z;　　　(S2)
　　　　　　　　　　　↗　　　　↑
　　　　　　　（正しい）　未宣言変数使用のエラー

である。

この言語の文法（構文）は、次のように表せる。

```
program → dcllist stlist
dcllist → dcllist declare id ;
dcllist → ε
stlist  → stlist use id ;
stlist  → ε
```

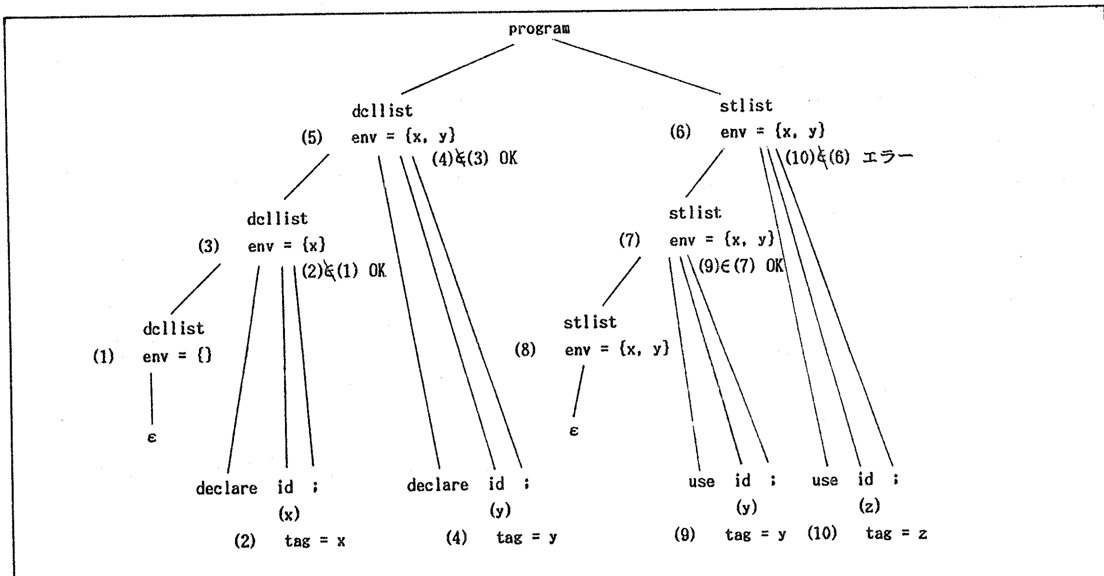

図１．　文法Ｇ１によるソースプログラム(S2)の属性つき構文木

ここで、program, dcllist, stlistは非終端記号、declare, use, id, ; は終端記号である。εは空列を表す。

ソースプログラム(S1)や(S2)は、この文法に対して構文的には正しいが、意味的な誤りがある。これを検査するために、変数の名前や宣言ずみの変数などの情報を記憶しながら意味解析を行なうとよい。これらの情報のことを属性と呼ぶ。変数の名前を表す属性をtag、宣言ずみの変数の集合（記号表）を表す属性を env (environment 環境の意）と名付けよう。たとえば、ソースプログラム(S2)に対して、構文木を作り、その各ノードに上記の属性を付加すると図1のようになる。

この図で、属性（正確には属性出現という）の値はどのように決まっていくのであろうか。大まかには、dcllistの下で変数の宣言を集め、それを stlist の方へ渡していけばよい。詳しくいうと、属性の値は(1)から(10)の順に決まる。まず(1)では何も変数宣言がないので env={} とする。(3)では、(1)の env に(2)の tag=x を加えて env={x} とする。(5)ではさらに(4)の tag=y を加えて env={x,y} とする。このように、構文木の下から上へ決まっていく属性を合成属性という。 dcllist の env は合成属性である。

次に、宣言全体の記号表である(5)の env={x,y} を、文列の env (6) の値とする。同様にして(7)(8)の env も同じ値をコピーしていく。このように、木の上から下へ、あるいは兄弟間で値が決まっていく属性を相続属性という。stlist の env は相続属性である。

さて、(9)の箇所では変数 yを使用しているが、(9)の tag=y が(7)の env={x,y} に含まれているので宣言ずみの変数を使用したことがわかり、これは正しい。(10)の箇所では変数 z を使用しているが、(10)の tag=z が(6)の env={x,y} に含まれていないので、未宣言変数使用のエラーであることがわかる。

以上は、属性を用いた意味解析を構文木の上で行なった訳であるが、この構文と意味を形式的に規則の形で書いたものが属性文法である。上の例を属性文法で書くと次のようになる。

文法G1

```
program   →  dcllist  stlist
  { stlist.env = dcllist.env }          (図1の(6))
dcllist1  →  dcllist2  declare  id  ;
  { dcllist1.env = dcllist2.env ∪ {id.tag}
                                        ((3)、(5))
    condition  id.tag ∉ dcllist2.env
      message  二重宣言のエラー          }
dcllist   →  ε
  { dcllist.env = {}  }                 ((1))
stlist1   →  stlist2  use  id  ;
  { stlist2.env = stlist1.env           ((7)、(8))
    condition  id.tag ∈ stlist1.env
      message  未宣言変数使用のエラー     }
stlist    →  ε
```

属性文法は、構文規則とそれに対応する意味規則とからなるが、ここでは意味規則を { と } ではさんで表した。意味規則の例として、たとえば

dcllist1.env = dcllist2.env ∪ {id.tag}

は、構文規則

dcllist1 → dcllist2 declare id ;

に対し、左辺の dcllist1 の属性 env の値が、右辺の dcllist2 の属性 env と id の属性 tag の和集合によって決まることを表している。1つの構文規則中に同じ記号がいくつか現れたときは、その区別のために 1,2などの添字を用いた。また、

condition …　　message …

は、文脈条件と呼ばれ、言語の静的意味として満たされるべき条件とそれに違反したときのエラーメッセージを表している。

先程、図1によって意味解析のあらましを述べたが、正確には上で示した属性文法に従って意味解析が行われる。参考のため、図1で属性の値が決まっていった箇所を属性文法G1の右側に示しておく。図1のように、属性評価の結果、各節点の属性値が定まった構文木を属性つき構文木という。

2．2　2進数の例

もう1つ例をあげよう。これは、2進小数の値を計算する属性文法で、[Knu]にある例を一部変更した[Kat]より引用した。2進小数の値を計算する属性文法の書き方はこれだけではないが、後述の属性評価法の説明のため、この例を取上げる。

文法G2

1: F → . L
　{ F.val = L.val; L.pos = 1 }

2: L → B
　{ L.val = B.val; B.pos = L.pos }

3: L_0 → B L_1
　{ L_0.val = B.val + L_1.val;
　L_1.pos = L_0.pos + 1; B.pos = L_0.pos }

4: B → 0
　{ B.val = 0 }

5: B → 1
　{ B.val = 2↑(-B.pos) }

Fが開始記号、F.val, L.val, B.valは合成属性、L.pos と B.pos は相続属性である。数「. 101」についての属性つき構文木を図2に示す。

3．　属性文法の定義

属性文法の定義法には、いくつかの流儀があるが、本質的な差は殆どない。ここでは、[Eng81]流の定義に若干の変更を加えたものを示す。

定義　属性文法Gは、3つ組<Gu, A, R>として、次の(1)～(3)により定義される。

(1)　文脈自由文法Gu＝(N, T, P, Z) はGの基底文脈自由文法(underlying context-free grammar)と呼ばれ、属性文法のうち、構文の部分を表している。

ここで、Nは非終端記号の集合、Tは終端記号の集合、Pは生成規則の集合、Zは開始非終端記号である。V＝N∪Tとする。

生成規則p∈Pは、

p:　X_0 → X_1 ･･･ X_n

の形とする。

Guは既約(reduced)と仮定する。

(2) 各記号X∈Vに、2つの有限集合S(X)とI(X)、S(X)∩I(X)＝φ、を与える。S(X)、I(X)はそれぞれXの合成属性(synthesized attribute)、相続属性(inherited attribute)の集合である。

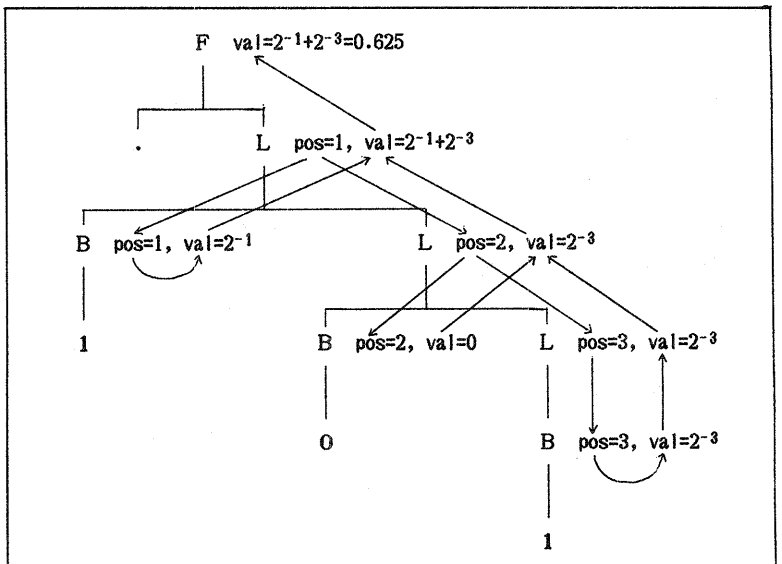

図2．　文法G2による「. 101」の属性つき構文木

但し、開始記号Zと終端記号については、$I(Z)=\phi$、$X \in T$に対し$S(X)=\phi$とする。

記号Xのすべての属性の集合を$A(X)=I(X) \cup S(X)$で表す。Gの属性全体の集合を$A=\cup_{X \in V} A(X)$で表す。

記号Xの属性aを$X.a$と表すこともある。

(3) 各生成規則$p \in P$に対し、意味規則(semantic rule)の集合$R(p)$が与えられる。これは、$S(X_0) \cup I(X_1) \cup \cdots \cup I(X_n)$に含まれるすべての属性の値を定めるものである。各意味規則を

$$X_k.a = f(X_{i1}.a_1, \cdots, X_{im}.a_m)$$

のように記す。 このとき、$X_k.a$ は $X_{i1}.a_1, \cdots, X_{im}.a_m$ に依存する(depend)という。

すべての$X_{ij}.a_j \in I(X_0) \cup S(X_1) \cup \cdots \cup S(X_n)$のとき、意味規則(あるいは文法)はBochmann正規形(Bochmann normal form)であるという。

$X_k.a$、$X_{ij}.a_j$を属性生起(attribute occurrence)とよぶ。

集合$R=\cup_{p \in P} R(p)$はGの意味規則の集合である。

(注1) 属性の計算のされ方は、関数的で、単一代入である。一旦、意味規則によって属性の値が決められれば、その値は不変である。

(注2) 文法記号が異なれば、その属性は別物である。つまり

$$X \neq Y \quad \text{なら} \quad A(X) \cap A(Y) = \phi$$

である。たとえば、文法G1の dcllist.env と stlist.env は別物である。

(注3) [Wai]では、属性文法の定義に文脈条件の集合$B=\cup_{p \in P} B(p)$を加えている。

(注4) (2)において、終端記号の属性をどの程度許すかには、色々な変種がある。 たとえば、 文法G1の id.tag は、コンパイラでは字句解析ルーチンから値が与えられるが、これを合成属性とみなすこともある。

例 文法G2の場合

$Gu = (N, T, P, F)$

$N = \{ F, L, B \}$

$T = \{ ., 0, 1 \}$

$P = \{ F \rightarrow .L,\ L \rightarrow B,\ L_0 \rightarrow B\ L_1,\ B \rightarrow 0,\ B \rightarrow 1 \}$

$I(F)=\phi$, $S(F)=\{ F.val \}$

$I(L)=\{ L.pos \}$, $S(L)=\{ L.val \}$

$I(B)=\{ B.pos \}$, $S(B)=\{ B.val \}$

$A=\{ F.val, L.pos, L.val, B.pos, B.val \}$

$R=\{ F.val = L.val,\ L.pos = 1, \cdots \}$

属性文法Gの文(sentence)の解析は、(原理的に)次のように行なわれる。Gの文が与えられると、それをGの基底文法によって構文解析し、構文木を作成する。その構文木の各節に対応する生成規則がわかるので、それに付随する意味規則に従って、各節の属性の値を定める。もちろん、属性間の依存関係があるので、属性の値が直ちに決まらない節もあるが、決められる所から順に値を決めていって、最終的にすべての属性に対して値が定められればよい。

定義 属性文法Gが well-definedとは、Gの言語$L(G)$の任意の文に対する構文木上で、すべての属性が計算できることである[Knu]。

属性文法Gの well-defined性は、構文木上の属性依存関係にサイクルがないことと同等であることが知られている。これをもう少し詳しく述べるために、まず属性依存関係を表すグラフ的な定義をいくつか示そう。

定義 生成規則pに対する依存グラフ(dependency graph) DG_P は、pの属性生起間の依存関係を表し、次で与えられる。

$$DG_P = \{ (X_i.a, X_j.b) \mid X_j.b = f(\cdots X_i.a \cdots) \in R(p) \}$$

例　文法Ｇ２の生成規則についてのＤＧｐを図３に示す。たとえばＤＧ１では、生成規則１に「F.val=L.val」という意味規則があるので、L.val から F.val へ矢印が引かれる。

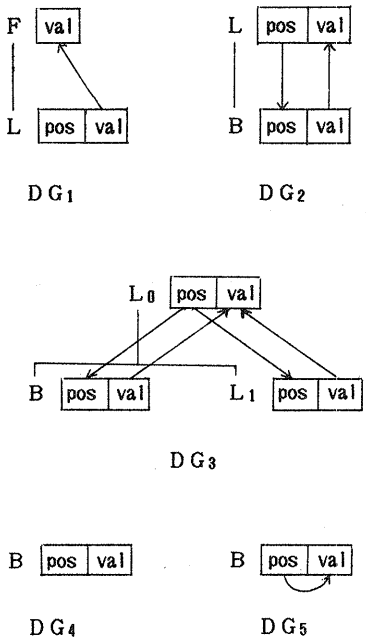

図３．　文法Ｇ２の生成規則に対するＤＧP

定義　構文木Ｔが与えられたとき、Ｔに対する依存グラフＤＧTは、Ｔのすべての節の属性の間の依存関係を表す。ＤＧTは、Ｔの構文の構造に従ってＤＧPをのりづけしたものである。

例　図２の矢印は、文法Ｇ２の文「．１０１」に対するＤＧTを表す。

定理　属性文法Ｇが well-defined（このような属性文法を well-defined 属性文法 Ｗ－ＡＧという）
<==> Ｌ(Ｇ)中の各文に対する構文木ＴのＤＧTに
サイクルがない
（これを非循環属性文法(noncircular attribute grammar) ＮＣ－ＡＧという）。

属性文法の非循環性の判定アルゴリズムは[Aho]などにある。

４．　属性評価法と属性文法のクラス

属性の値を計算することを属性評価(attribute evaluation) という。

３節では、構文木上で属性の値が計算できさえすれば、属性文法は well-defined であると述べたが、実際に属性文法を用いる場合は、もっと効率のよい属性評価法が必要とされる。たとえば、コンパイラへ適用する場合は、コンパイラのパス数などが関係してくるからである。そこで、様々な属性評価法と、それに対応した属性文法の部分クラスが提案されてきた。

属性文法のある部分クラスＸ－ＡＧは、Ｘ型の属性評価器（attribute evaluator）が存在するようなクラスとして設定されるのがふつうである。一方、与えられた属性文法がＸ－ＡＧかどうかを判定する際は、属性評価器を作ってみるまでもなく、属性間の依存関係などの簡単な解析で判定できることが多い。

そこで、属性文法のクラスを考える場合は、対応する属性評価器（その属性文法クラスの動的性質ともいう）と判定法（その属性文法クラスの静的性質ともいう）とに分けて考えると都合がよい[Eng84]。以下でもこの考え方に沿って解説する。

この節全体の参考文献としては[Eng84]が役に立つ。

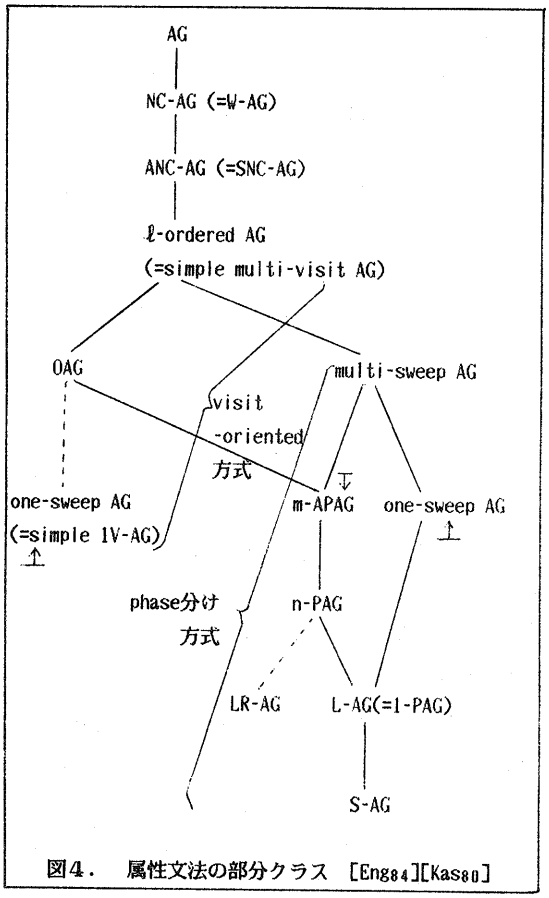

図4．　属性文法の部分クラス　[Eng84][Kas80]

↥ ここから上は、評価戦略 が 属性依存性から計算される.
↧ ここから下は、評価戦略 が アプリオリに決まっている.

AG　　= attribute grammars
NC-AG　= noncircular (非循環) AG
W-AG　= well-defined AG [Knu]
ANC-AG = absolutely noncircular (絶対非循環) AG
　　[Ken][Saa]
SNC-AG = strongly noncircular (強非循環) AG
ℓ-ordered AG [Kas80][Eng84]
simple multi-visit AG [Eng82]
OAG　　= ordered AG [Kas80]
one-sweep AG [Eng84]
simple 1V-AG = simple one-visit AG [Eng81]
multi-sweep AG [Eng84]
m-APAG = m-Alternating-pass (m-交互パス) AG [Jaz75]
n-PAG　= n-Multi-pass from left to right AG [Boc]
LR-AG [Jon][Sas85]
L-AG　[Boc][Lew]
S-AG　= Only S-AG [Lew]

属性文法の代表的な部分クラスとその階層関係を図4に示す。このうちのいくつかについて述べよう。

非循環属性文法(ＮＣ－ＡＧ)
あるいはWell-defined属性文法(Ｗ－ＡＧ)

３節で述べたように、意味のある属性文法のクラスはこのクラスから下（このクラスの部分クラス）である。

ＮＣ－ＡＧは、その判定に、文法サイズに対して指数関数的な時間が必要であることが知られている [Jaz75a] [Jaz81]。

絶対非循環属性文法(ＡＮＣ－ＡＧ)

ＮＣ－ＡＧは、その判定に指数時間が必要で、実用的ではない。そこで、判定が多項式時間で行える絶対非循環属性文法ＡＮＣ－ＡＧ(absolutely noncircular AG)（ＳＮＣ－ＡＧ　strongly noncircular AGともいう）が考えられた[Ken][Saa][Aho]。ＡＮＣ－ＡＧの判定法は、各生成規則ｐに対し、依存グラフＤＧpを重ね合わせた格好の拡張依存グラフ(augmented dependency graph) ＤＧp*というものを考え、すべての生成規則ｐについてＤＧp*にサイクルがないことを確かめればよい。

ＡＮＣ－ＡＧの属性評価器としては種々のものが知られている[Ken][Saa][Jou][Kat][Aho]。そのうちで Katayama の評価器[Kat]について述べよう。これは、属性文法を手続きの群に変換する方法であり、その手続き群が評価器となる。各非終端記号に対し、１つの手続きが生成される。文法Ｇ２に対する属性評価器の例を図5に示す。非終端記号Ｘに対する手続きの引数は、

（Ｘ.ｓを評価するのに必要な属性群、構文木Ｔ；
　合成属性Ｘ.ｓ）

の形をしている。

Ｏｒｄｅｒｅｄ属性文法(ＯＡＧ)

詳細は省くが、属性評価器の効率が余り悪くなく、しかも一般の手続き型プログラミング言語をほぼ表現できるのがＯｒｄｅｒｅｄ属性文法（ ordered attribute grammar）である[Kas80]。これはコンパイラ生成系ＧＡＧで採用されており、Pascal，Ada等がこれで記述されている[Kas82]。

```
program
  procedure RF(T; F.val)
    L.pos ← 1;
    call RL(L.pos, T[2]; L.val);
    F.val ← L.val
  end;
  procedure RL(L.pos, T; L.val)
    case production(T) of
      p2: B.pos ← L.pos;
          call RB(B.pos, T[1]; B.val);
          L.val ← B.val
      p3: B.pos ← L.pos;
          call RB(B.pos, T[1]; B.val);
          L1.pos ← L.pos + 1;
          call RL(L1.pos, T[2]; L1.val);
          L.val ← B.val + L1.val
    end
  end;
  procedure RB(B.pos, T; B.val)
    case production(T) of
      p4: B. val ← 0
      p5: B.val ← 2 ↑ (−B.pos)
    end
  end;
  input_derivation_tree(T0);
  call RF(T0; F.val);
  output_attribute(F.val)
end
```

図5．　文法Ｇ２に対する属性評価器[Kat]

1パス型の属性文法

最近のプログラミング言語は、１パスでコンパイルできるように設計されているものが多い。そのようなクラスの属性文法として、Ｌ属性文法やＬＲ属性文法が知られている。適当な構文解析法と組み合せると、構文木を作ることなく、構文解析と同時に属性評価が行なえる。

Ｌ属性文法(Ｌ－ＡＧ)

Ｌ属性文法(L-attributed grammar) [Boc][Lew]は、構文木を左から右、上から下へと一回たどるだけで属性評価のできるような属性文法のクラスである。基底文法がＬＬ(k)文法であれば、ＬＬ(k)構文解析あるいは再帰的下向き構文解析と同時に属性評価が行なえ、構文木を作成する必要がない。Ｌ属性文法の定義は簡単で、これがそのまま判定法として使える。

定義　（Ｌ属性文法）[Wai][Boc][Lew]

属性文法Ｇは、任意の生成規則　$X_0 \to X_1 \cdots X_n$ について次が成り立つとき、Ｌ属性文法である。

$I(X_k)$ $(1 \leqq k \leqq n)$ 中の属性生起は、

$$I(X_0) \cup \bigcup_{i=1}^{k-1} A(X_i) \cup I(X_k)$$

中の属性生起だけに依存する。

これは、属性依存関係に「右→左」のものがないことを意味する。

Bochmann正規形を仮定すると、Ｌ属性文法の定義は次のようになる。

$I(X_k)$ $(1 \leqq k \leqq n)$ 中の属性生起は、$I(X_0) \cup \bigcup_{i=1}^{k-1} S(X_i)$中の属性生起だけに依存する。

例　文法Ｇ１、Ｇ２はＬ属性文法である。

Ｌ属性文法に対する属性評価器は、構文解析譜の中に属性評価を埋め込んだ形のものがふつうである。文法Ｇ１は左再帰性があり、ＬＬ(k)文法ではないが、文法Ｇ２はＬＬ(２)文法であるので、Ｇ２については再帰的下向き構文解析譜に埋め込んだ形の属性評価器を作成できる。それを図６に示す[Kat]。ここで、lookahead[1] と lookahead[2] は、第１,第２の先読み記号、 read は入力記号を読み進めるものである。生成される手続きのパラメタには、構文木に対応するものが不要である。

ＬＲ属性文法(ＬＲ－ＡＧ)

詳細は省くが、ＬＲ属性文法(LR-attributed grammar) [Jon][Sas85]は、ＬＲ(k)構文解析と同時に属性評価が行えるようなクラスである。

Ｌ属性文法との関連については

Ｌ属性ＬＬ(１)文法 ⊂ ＬＲ属性ＬＲ(１)文法

という性質が示されたので[Nak86]、 ＬＲ属性文法は１パス型の属性文法の中では実用上一番広いクラスといえよう。

ＬＲ属性文法に基づき、コンパイラ生成系 Rie が作られている[石塚]。

```
program
  var F.val;
  procedure RF( ; F.val)
    var L.pos
    L.pos ← 1;
    read;
    call RL(L.pos; F.val)
  end;
  procedure RL(L.pos; L.val);
    procedure Qp2
      call RB(L.pos; L.val);
    end;
    procedure Qp3
      var B.val, L1.pos, L1.val;
      call RB(L.pos; B.val);
      L1.pos ← L.pos + 1;
      read;
      call RL(L1.pos; L1.val);
      L.val ← B.val + L1.val
    end;
    case lookahead[2] of
          ' ': Qp2
        '0', '1': Qp3
    end
  end;
  procedure RB(B.pos; B.val)
    case lookahead[1] of
      '0': B.val ← 0
      '1': B.val ← 2 ↑(−B.pos)
    end
  end;
  input_string;
  call RF( ; F.val);
  output_attribute(F.val)
end
```

図６．　文法Ｇ２の再帰下向き属性評価器[Kat]

５．　属性文法の課題

属性文法は、プログラミング言語の意味を形式的に記述するために提案されたので、形式性、明せき性にすぐれている。しかし、実用上はいくつかの問題点も指摘されている。

まず、良い点をあげよう。

（１）　局所性.

意味の記述が構文規則ごとに局所的であるので、モジュール性が良い。

（２）　関数性.

属性の値は意味規則によって決められるが、各属性の値は一度だけ評価され、変えられることはない。つまり、単一代入である。その意味で属性文法は宣言的、あるいは関数的であって、明解である。

（３）　自動生成に向く.

文法から構文解析譜を生成できるのと同様に、属性文法から属性評価器を機械的に生成できる。これは、コンパイラ作成を容易にする。

このように、属性文法はすぐれた形式的道具であるが、実際にこれを用いてシステムを作成するときには、次のような問題がある。

（１）　記述量の増大.

長所の(１)と裏腹であるが、変化の少ない属性を用いようとすると、記述量が増大し、見通しが悪くなる。たとえば、コンパイラにおける記号表は、宣言部を解析しているときは、新しい宣言が次々に加わって行くが、実行部を解析しているときは一般に不変である。ところが、実行部の解析中でも、識別子などの同定のために記号表は必要であるから、記号表という属性を実行部の奥深くまで次々にコピーして渡してやらなければならない。そのため、単にコピーを行なうだけの意味規則(コピー規則という)が多数現れる。このようなコピー規則の数は５０%～７０%と言われ（[Far]では５７%）、記述量が倍増する。

実用的なシステムでは、記述量の増加を避けるため、明らかなコピー規則は省略できるようにしたり、簡単に記述できるようにしたりしている。また、属性文法の記述法として、もともとの記述よりも文脈条件などを簡潔に表現できるＥＡＧ(Extended Attribute Grammar)という記法も用いられている[Wat]。

（2） 意味規則として記述しにくい処理.

これも、長所の(2)と裏腹であるが、コンパイラ作成者にとっては、単一代入性がネックになって、記述しにくい処理がある。たとえば、バックパッチや、「ひとまず箱(レコード)を作っておいてあとで中身を埋める」処理はうまく記述できない。

このような点から、実際には、純粋な属性文法をはみ出した用い方をすることもある。たとえば、上記のような副作用のある処理を意味規則から呼ばれる関数の中で行なったり、一部大域変数を導入する場合もある。

（3） 属性評価器の最適化.

素朴に作られた属性評価器では、意味規則を評価するときに左辺の属性への代入が起こるのがふつうである。すると、記号表などのような大きな領域を占める属性についても、実体全体のコピーがしばしば起こって属性評価の効率が悪くなる。

このため、大きな属性については、実体コピーの代りにポインタコピーで実現したり、属性の寿命を解析して、複数の属性出現を1つの大域変数を用いて実現して、属性評価の効率を上げる研究が盛んに行なわれている（[Aho][Kas82]他)。上記の(2)と(3)とは無関係ではなく、属性文法の記述法の条件を緩めることで属性評価器の効率が上がることもある。

（4） 動的意味の取扱い.

プログラミング言語の意味には静的意味（static semantics）と動的意味（dynamic semantics）がある。静的意味とは、コンパイル時にわかるが文脈自由文法では表しきれない性質、たとえば型チェックや識別子の同定などのことをいう。動的意味とは、実行時になってはじめてわかる性質、たとえば実際の計算内容そのものや、0（ゼロ）による除算や値が未定義な変数の参照が起こらないことの実行時チェックなどをいう。本来の属性文法では構文木の各節での属性値が一意に決まる必要があるので、意味として扱うのは主に静的意味であるが、インタプリタなどの実行時の性質を形式的に取扱いたいという要請も強い。

これについては、表示的意味論からのアプローチ（[Pau]など）があるが、表示的意味論では、静的意味と動的意味とが分化されていず、実際のシステムのモデルとしては不十分なところがある。

実行時の性質を属性文法で表すのは、同じ変数に何度も異なった値が代入されたり、飛び越しが起こったりするので、属性規則に対する負担が重くなり、難しい点もある。しかし、動的意味を属性文法で記述したり、必要なら属性文法の拡張を行なうことは、興味あるテーマであると思われる［松田］。

6. おわりに

属性文法の基本的事項と主な理論的結果について述べた。

紙面の関係で、触れることのできなかったものも多い。記述例については[Pag][Kas82]をみられたい。属性文法を用いたコンパイラやコンパイラ生成系、応用、関連分野については殆ど述べられなかった。全体的な参考文献としては、[Lor],[Wai],[Aho],[佐々85]、および本研究会の他の論文を参照されたい。

7. 謝辞

コメントを頂いた中田育男先生と石塚治志氏に感謝する。

8. 参考文献

[Aho] Aho,A.V.,Sethi,R.and Ullman,J.D. : Compilers—Principles,Techniques,and Tools, Addison-Wesley, 1986.

[Boc] Bochmann,G.V. : Semantic evaluation from left to right, Comm. ACM, Vol.19 (1976), pp.55-62.

[Eng81] Engelfriet,J.and Filè,G.; The Formal Power of One-Visit Attribute Grammars, Acta Inf. Vol.16 (1981),pp.275-302.

[Eng82] Engelfriet,J. and Filè.G. : Simple multi-visit attribute grammars, J. Comput. Syst. Sci., Vol.24(1982), pp.283-314

[Eng84] Engelfriet,J.: Attribute Grammars: Attribute Evaluation Methods, in [Lor], pp.103-138.

[Far] Farrow,R.: Generating a Production Compiler from an Attribute Grammar, IEEE Software, Vol.1, No.4 (1984),pp.77-93.

[石塚] 石塚治志, 佐々政孝 : 属性文法によるコンパイラ生成系, 情報処理学会第26回プログラミング・シンポジウム報告集, 1985, pp.69-80.

[Jaz75] Jazayeri,M. and Walter,K.G. : Alternating semantic evaluator, Proc. ACM 1975 Ann.Conf.(1975), pp.230-234.

[Jaz75a] Jazayeri,M., Ogden,W. and Rounds,W. : The intrinsically exponential complexity of the circularity problem for attribute grammars, Comm.ACM, Vol.18 (1975),pp.697-706.

[Jaz81] Jazayeri,M. : A Simpler Construction for Showing the Intrinsically Exponential Complexity of the Circular Problem for Attribute Grammars, J.ACM, Vol.28, No.4(1981), pp.715-720.

[Jon] Jones, N.D. and Madsen, M. : Attribute-influenced LR Parsing, LNCS 94, Springer, 1980, pp.393-407.

[Jou] Jourdan,M. : Recursive evaluators for attribute grammars, an implementation, in [Lor], pp.139-163.

[Kas80] Kastens,U. : Ordered attributed grammars, Acta Inf., Vol.13(1980),pp.229-256. ([Lor]中の The GAG-Systemも見よ)

[Kas82] Kastens,U., Hutt,B. and Zimmermann,E. : GAG: a practical compiler generator, LNCS 141,Springer,1982. (同)

[片山] 片山卓也 : 属性文法型計算モデル, 情報処理, Vol.24, No.2 (1983)pp.147-155.

[Kat] Katayama,T. : Translation of Attribute Grammars into Procedures, ACM Trans. Prog. Lang. Syst., Vol.6,No.3 (1984),pp.345-369.

[Ken] Kennedy,K. and Warren,S.K. : Automatic generation of efficient evaluators for attribute grammars, Proc. 3rd ACM Symp. on POPL(1976), pp.32-49.

[Knu] 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 (1971), pp.95-96.

[Lew] Lewis,P.M., Rosenkrantz,D.J. and Stearns,R.E. : Attributed translations, J Comput. Syst. Sci., Vol.9(1974),pp.279-307.

[Lor] Lorho,B.(ed.) : Methods and Tools for Compiler Construction, Cambridge Univ. Press, 1984.

[Nak86] Nakata,I. and Sassa,M. : L-attributed LL(1) grammars are LR-attributed, 本研究会報告SF-16(1986),または to appear in Inf. Process. Lett.

[松田] 松田裕幸 : 属性文法形式によるセマンティックス記述の問題点, 情報処理学会プログラミング言語研究会, 3-3 (1985).

[Pag] Pagan,F.G. : Formal Specification of Programming Languages : A Panoramic Primer ,Prentice-Hall, 1981の2.3節, 3.2節.

[Pau] Paulson,L. : Compiler Generation from Denotational Semantics, in [Lor], pp.219-250.

[Saa] Saarinen,M. : On constructing efficient evaluators for attribute grammars, 5th ICALP, LNCS 62, Springer,1978,pp.382-397.

[佐々84] 佐々政孝 : コンパイラ生成ツール ーーYacc+属性文法ーー, Computer Today, 1984/9 No.3 (1984) pp.41-45.

[佐々85] 佐々政孝 : 読書・文献案内 属性文法, コンピュータソフトウェア, Vol.2, No.3 (1985) pp.560-562.

[Sas85] Sassa,M., Ishizuka,H. and Nakata,I.: A Contribution to LR-attributed Grammars, J.Inf.Process., Vol.8, No.3 (1985), pp.196-206.

[Wai] Waite,W.M. and Goos,G. : Compiler Construction, Ch.8 & 9, Springer,1984.

[Wat] Watt,D.A. and Madsen,O.L., Extended Attribute Grammars, Comp.J., Vol.26, No.2 (1983), pp.142-153.