case 文翻訳のための属性文法から動作ルーティンへの変換†

渡辺 喜道††

属性文法の記述を効率のよい動作ルーティンの記述に機械的に変換する方法を提案する．ここで用いる動作ルーティンは標準的なボトムアップ構文解析系を基礎としている．テーブル型属性と呼ばれるある種の相続属性と合成属性の組の属性は，属性文法の記述中では，よく用いられている．いままで，このテーブル型属性を参照する意味関数を記述中に含む属性文法の記述を動作ルーティンの記述に機械的に変換することはできなかった．この論文で提案する特殊整数版テーブル型属性除去法を導入することにより，いままでできなかった機械的な変換が可能となった．しかし，この変換法には厳しい制約条件があるが，既存の変換法と組み合わせることにより，制約が緩和される．この変換法の適用例として，C言語の switch 文の構文に似た case 文を4つ組コードに翻訳する属性文法の記述を動作ルーティンの記述に変換する．さらに，その他の最適化技術を組み合わせて，属性文法の記述から効率のよい動作ルーティンの記述に変換する．

1. はじめに

構文主導型翻訳[1),2),4),6)]は，コンパイラ生成系や言語ベースのソフトウェア開発環境の構築などでよく用いられている基本的な翻訳方法である．構文主導型翻訳は2つの部分から成る．1つは文法の記述であり，BNF 記法がよく用いられる．もう1つは意味の記述であり，属性文法による方法や動作ルーティンによる方法，表示的意味論による方法などが用いられる．

属性文法[3),5)]による記述は，Knuth により導入された方法である．構文解析木上の各ノードに属性を持たせ，意味の評価を行う方法である．この方法は，構文解析木上の親と子の間の依存関係を基に，各文法規則に局所的に，属性の値を求める記述をするため，記述することは比較的容易である．

一方，動作ルーティンによる方法は各文法規則に動作ルーティンであるプログラムの断片を対応させ，各文法規則が適用されるときに，その文法規則に付随した動作ルーティンを実行し，意味の評価を行う方法である．この方法は，各文法規則に局所的に記述することはできず，記述するのは容易ではないが，効率のよい意味の評価ができるという利点がある．

本論文では，比較的記述しやすい属性文法の記述から，効率のよい動作ルーティンの記述に機械的に変換する方法[10)~12)]をさらに拡張する1つの方法を提案する．今回新たに，整数版テーブル型属性と呼ばれる属性を定義する意味関数以外の意味関数が，文法規則中のある一部分の文法記号の整数版テーブル型属性の参照を含んでいる属性文法の記述を動作ルーティンの記述に変換する方法を提案する．この変換法の適用例として，case 文を4つ組コードに翻訳する属性文法の記述を動作ルーティンの記述に機械的に変換する．既存の変換法の組合せだけでは機械的に変換することはできなかったが，今回の変換法の導入により機械的な変換が可能になった．この変換法は，一般の場合にも役立つ変換法であると考えられる．

なお，本論文の基となる文法規則の一般形の0番に $S' \rightarrow S$ を加え，それを文法規則とする．また構文解析は標準的なボトムアップ構文解析系を用いるものとする．さらに，文法記号Xの属性Aを X.A で表記し，記号←は，属性文法の記述の中ではその左辺が右辺によって定義されることを意味するが，動作ルーティンの中では代入を意味する．

本論文の構成は次のとおりである．2章で，属性文法の記述を動作ルーティンの記述に変換する既存の変換法について説明する．3章では，今回新たに提案する整数版テーブル型属性の参照を含む記述を動作ルーティンに変換する方法について説明する．4章では，具体的な変換例として，case 文を取り挙げ，機械的な変換過程を順次示す．最後に5章で，本論文をまとめる．

2. 既存の変換法

属性文法の記述を動作ルーティンの記述に変換する既存の方法として，単純後置翻訳法とパッチ導入法について説明する．

† Transformation of an Attribute Grammar to Translate case Statements into the Action Routines by YOSHIMICHI WATANABE (Department of Electrical Engineering and Computer Science, Faculty of Engineering, Yamanashi University).

†† 山梨大学工学部電子情報工学科

2.1 単純後置翻訳法

単純後置翻訳法は，1960年代に発表されたある種の属性文法の記述を動作ルーティンの記述に変換する伝統的な方法の1つである．

[後置的合成属性の定義] 次の形をしている文字列を値として取る属性Aを後置的合成属性と呼ぶ．ただし，$a_0, a_1, \cdots, a_n$ は長さ0以上の終端記号列で，$X_0, X_1, \cdots, X_n$ は非終端記号であり，文字列 tail は文法規則に依存した定文字列である．さらに，演算子「＋」は文字列を結合する演算子である．

1) 0番目の文法規則

文法規則：

$S' \rightarrow S$

意味関数：

出力 (S. A)

2) 0番目以外の文法規則

文法規則：

$X_0 \rightarrow a_0\ X_1\ a_1\ X_2\ a_2 \cdots X_n\ a_n \quad (n \geqq 0)$

意味関数：

$X_0.A \leftarrow X_1.A + X_2.A + \cdots + X_n.A + \text{tail}$

単純後置翻訳法は，すべての非終端記号に後置的合成属性Aが存在するとき適用できる方法である．

一般に，属性がすべて合成属性であるように属性文法の記述は，各属性の代わりにスタックを用いて評価することができる．特に，属性が後置的合成属性である場合には，次のような動作ルーティンの記述に変換することができる．この変換法を単純後置翻訳法と呼ぶ．

1) 0番目の文法規則

文法：

$S' \rightarrow S$

意味関数：

動作なし

2) 0番目以外の文法規則

文法：

$X_0 \rightarrow a_0\ X_1\ a_1\ X_2\ a_2 \cdots X_n\ a_n \quad (n \geqq 0)$

意味関数：

出力 (tail)

中置記法の算術式を後置記法に変換する属性文法の記述を**図1**に示す．図1の中で関数 print は引数の値を出力する関数である．ここで，属性 code は後置的合成属性であるから，単純後置翻訳法を用いて動作ルーティンに変換すると**図2**のような簡単な記述になる．ここで，文法規則番号1番や2番，4番，5番で非終端記号EとTの文法規則の左辺と右辺の生起を区別するために，それぞれ E^0 と T^0，E^1 と T^1 を用いた．

```
0)S' → E
     print(E.code)
1)E⁰ → E¹ + T
     E².code ← E¹.code + T.code + "+"
2)E⁰ → E¹ - T
     E⁰.code ← E¹.code + T.code + "-"
3)E → T
     E.code ← T.code
4)T⁰ → T¹ * F
     T⁰.code ← T¹.code + F.code + "*"
5)T⁰ → T¹ / F
     T⁰.code ← T¹.code + F.code + "/"
6)T → F
     T.code ← F.code
7)F → ( E )
     F.code ← E.code
8)F → i
     F.code ← lex(i)
```

図1 中置記法を後置記法に変換する属性文法の記述
Fig. 1 Infix-to-posifix translation in an attribute grammar.

```
0)S' → E
1)E⁰ → E¹ + T
       print("+")
2)E⁰ → E¹ - T
       print("-")
3)E → T
4)T⁰ → T¹ * F
       print("*")
5)T⁰ → T¹ / F
       print("/")
6)T → F
7)F → ( E )
8)F → i
       print(lex(i))
```

図2 中置記法を後置記法に変換する動作ルーティンの記述
Fig. 2 Infix-to-posifix translation in action routines.

2.2 パッチ導入法[8]

1986年に発表された，放送型相続属性[8]と呼ばれる属性を含んだ属性文法の記述からその放送型相続属性を除去する方法である．

[放送型相続属性の定義] 属性Aの値を求めるための補助的属性で，意味関数が次の3つのうちいずれかの形をしている属性Bを放送型相続属性と呼ぶ．ただし，$a_0, a_1, \cdots, a_n$ は長さ0以上の終端記号列で，$X_0, X_1, \cdots, X_n$ は非終端記号であり，関数 F は一変数整数値関数である．

文法規則：

$X_0 \rightarrow a_0\ X_1\ a_1\ X_2\ a_2 \cdots X_n\ a_n \quad (n \geqq 0)$

意味関数：

（1） 属性の値が左辺の非終端記号から右辺の非終端記号にコピーされる場合.

$X_i.B \leftarrow X_0.B$

ただし，$1 \leqq \exists i \leqq n$

（2） 属性の値が後置的合成属性に初期文字列を与えることができるように拡張した属性である拡張後置的合成属性Aの値の部分文字列になる場合.

この場合，この部分文字列は属性Aの値のB欄とみなされる．また，$n=0$ となっていることに注意されたい.

$X_0.A \leftarrow \cdots + X_0.B + \cdots$

（3） 属性の値が属性Aの部分文字列の相対的位置の関数の場合.

$X_k.B \leftarrow F(\text{next})$

ただし，$1 \leqq \exists k \leqq n$，next は変数

パッチ導入法は，放送型相続属性Bを整数値の集合を値とする合成属性Cに変換する方法である．この変換された合成属性はさらに，グローバル変数導入法[9]や非同期スタック法[7]を用いて効率のよい動作ルーティンに変換することができる.

パッチ導入法を用いると，上記の意味関数は次のように変換される.

意味関数：

（1） $X_0.C \leftarrow X_1.C \vee X_2.C \vee \cdots \vee X_n.C$

演算子∨は和集合を表す.

（2） $X_0.C \leftarrow \{\text{next}\}$

generate $(\cdots, 0, \cdots)$

B欄が定まる前に属性Aの仮の値（この場合0）を生成する.

（3） patch (X_k.C, B欄, F (next))

属性Aの仮の値を入れた位置である X_k.C のB欄に値 F (next) を後埋めする.

3. 特殊整数版テーブル型属性除去法

ここでは，整数版テーブル型属性と呼ばれる，ある種の属性の組の定義とその属性が生起する意味関数を動作ルーティンに変換する方法について述べる.

3.1 整数版テーブル型属性の定義[11),12)]

次の形をした2つの属性の組を整数版テーブル型属性と呼ぶ.

[整数版テーブル型属性の定義] 2つの属性 old と new は，整数値を値として取る属性で，それぞれ相続属性，合成属性である．また，記号「+」は，算術加法演算子，記号 tail は，文法規則に依存する整数である．このとき，次の条件を満足する2つの属性 old と new に関する意味関数が，各文法規則の意味関数群中に含まれるとき，その2つの属性 old と new の組を整数版テーブル型属性と呼ぶ．ただし，英大文字は非終端記号で，英小文字は空列を含む終端記号列とする.

1） 0番目の文法規則の場合

文法規則：$S' \rightarrow S$

意味関数：S. old←初期整数定数

2） 0番目以外の文法規則かつ $n>0$ の場合

文法規則：$X_0 \rightarrow a_0\ X_1\ a_1\ X_2\ a_2 \cdots X_n\ a_n$

意味関数：$X_1.\text{old} \leftarrow X_0.\text{old}$

$X_{k+1}.\text{old} \leftarrow X_k.\text{new}$

$(1 \leqq \forall k \leqq n-1)$

$X_0.\text{new} \leftarrow X_n.\text{new} + \text{tail}$

3） 0番目以外の文法規則かつ $n=0$ の場合

文法規則：$X_0 \rightarrow a_0$

意味関数：$X_0.\text{new} \leftarrow X_0.\text{old} + \text{tail}$

また，属性oldを整数版テーブル型相続属性，属性newを整数版テーブル型合成属性と呼ぶ.

3.2 整数版テーブル型属性の除去

ここでは，整数版テーブル型属性がそれを定義する意味関数以外で参照されるのは，文法規則の右辺の最後の非終端記号の整数版テーブル型属性のみである属性文法の記述を考える．この記述から，整数版テーブル型属性を除去し，動作ルーティンに変換する方法を提案する．この変換法を特殊整数版テーブル型属性除去法と名付ける.

次の形をした意味関数が存在すると仮定する.

文法規則：

$X_0 \rightarrow a_0\ X_1\ a_1\ X_2\ a_2 \cdots X_n\ a_n \quad (n \geqq 0)$

意味関数：

属性 old と new は整数版テーブル型属性である．さらに，その整数版テーブル型属性を定義する意味関数以外の整数版テーブル型属性を参照する意味関数群の中では，次の(a)から(c)の3つのうちいずれかの形であると仮定する．ただし，属性Aの整数版テーブル型属性以外の属性である.

（a） $X_i.A \leftarrow f_1(\cdots, X_n.\text{old}, \cdots)$

（b） $X_i.A \leftarrow f_2(\cdots, X_n.\text{new}, \cdots)$

（c） $X_i.A \leftarrow f_3(\cdots, X_n.\text{old}, X_n.\text{new}, \cdots)$

ただし，いずれの場合も，$0 \leqq \exists i \leqq n$，$f_k$ は関数である $(1 \leqq k \leqq 3)$．さらに，各文法規則の右辺の最後の非終端記号を左辺とする文法規則以外の規則におい

て，整数版テーブル型属性の定義中の整数 tail は 0 と仮定する．この仮定は厳しい仮定ではなく，一般の属性文法の記述のほとんどが，還元導入法[8]等を用いて，この形に変形できる．

このとき，整数版テーブル型属性は次の順序で除去できる．

ステップ1　文法規則の還元順序の解析

標準的なボトムアップ構文解析系を仮定しているので，文法規則の還元される順序は，文法規則が

$X_0 \rightarrow a_0\ X_1\ a_1\ X_2\ a_2 \cdots X_n\ a_n \quad (n \geqq 0)$

のとき，$X_1, X_2, \cdots, X_n, X_0$ の順に還元される．この順序を構文解析木に関係する各文法規則に適用することにより，文法規則が還元される順序が機械的に求められる．

ステップ2　整数版テーブル型相続属性の除去

文法規則の右辺の最後の非終端記号の整数版テーブル型相続属性を参照する意味関数を含む文法規則において，整数版テーブル型属性が評価されるときの，整数版テーブル型相続属性の値と整数版テーブル型合成属性の値の差を値とする属性 diff を導入する．つまり，

文法規則：

$X_0 \rightarrow a_0\ X_1\ a_1\ X_2\ a_2 \cdots X_n\ a_n \quad (n \geqq 0)$

に対して

$X_n.\mathrm{new} = X_n.\mathrm{old} + X_n.\mathrm{diff}$

の関係を満足する属性 diff を導入する．

また同時に，関係する文法規則の各 X_n.diff の値を求めるための意味関数も追加する．その意味関数の形は次のようになる．

構文解析木上で，X_n をルートとする部分木に適用されている文法規則の中で，部分木の最右の文法規則に

(1)　$n \neq 0$ のとき，$X_0.\mathrm{diff} \leftarrow X_n.\mathrm{diff} + \mathrm{tail}$

(2)　$n = 0$ のとき，$X_0.\mathrm{diff} \leftarrow \mathrm{tail}$

を追加する．ここで，整数 tail は整数版テーブル型属性を定義する意味関数の中に現れる定数である．

この属性 diff の導入により，属性 old の値の参照は，

$X_n.\mathrm{new} - X_n.\mathrm{diff}$

に置き換えることができ，整数版テーブル型相続属性の参照が除去できる．この操作により，意味関数は次のような形になる．

(a)　$X_i.A \leftarrow f_1(\cdots, X_n.\mathrm{new} - X_n.\mathrm{diff}, \cdots)$

(b)　$X_i.A \leftarrow f_2(\cdots, X_n.\mathrm{new}, \cdots)$

(c)　$X_i.A \leftarrow f_3(\cdots, X_n.\mathrm{new} - X_n.\mathrm{diff}, X_n.\mathrm{new}, \cdots)$

ステップ3　整数版テーブル型属性の除去

ステップ2の属性 diff の導入により，整数版テーブル型相続属性を参照している意味関数が除去できた．ここで，整数版テーブル型属性である属性 old と new をテーブル型属性除去法[12]を用いて除去する．このテーブル型属性除去法は，テーブル型属性を1つのグローバル変数を用いて除去する方法である．このとき，属性 new の参照は，対応する文法規則の意味関数により，グローバル変数の値が更新された後の値と等しい．つまり，属性 new の参照のある意味関数の評価を，グローバル変数の値の更新の後に行うことにより，属性 new の参照をグローバル変数の参照に置き換えることができる．置き換えた後の意味関数の記述は次のようになる．ただし，変数 var は整数版テーブル型属性を除去したときに生成されたグローバルな整数型変数である．

(a)　$X_i.A \leftarrow f_1(\cdots, \mathrm{var} - X_n.\mathrm{diff}, \cdots)$

(b)　$X_i.A \leftarrow f_2(\cdots, \mathrm{var}, \cdots)$

(c)　$X_i.A \leftarrow f_3(\cdots, \mathrm{var} - X_n.\mathrm{diff}, \mathrm{var}, \cdots)$

ステップ4　属性 diff のグローバル変数化

次に，属性 diff をグローバル変数導入法を用いて除去する．属性 diff は合成属性であり，この属性は，ある文法規則で値が求められると，その規則の親を文法規則の右辺の最後の非終端記号とするある文法規則の意味関数で参照され，その後は決して参照されない．つまり，構文解析木上で，親に値を渡し，渡された値は構文解析木上で適用されている文法規則で使用するだけで，その他のところではいっさい使用しない属性である．また，属性 diff は合成属性であるので，スタックを用いて属性の評価ができるが，そのスタック中の属性生起はいつも高々1つである．したがって，グローバル変数 aux を用いて，属性 diff を除去することができ，変換後の記述は次のようになる．

(a)　$X_i.A \leftarrow f_1(\cdots, \mathrm{var} - \mathrm{aux}, \cdots)$

(b)　$X_i.A \leftarrow f_2(\cdots, \mathrm{var}, \cdots)$

(c)　$X_i.A \leftarrow f_3(\cdots, \mathrm{var} - \mathrm{aux}, \mathrm{var}, \cdots)$

また，関連する各文法規則の属性 diff を求める記述は，次の動作ルーティンになる．

(1)　$n \neq 0$ のとき，$\mathrm{aux} \leftarrow \mathrm{aux} + \mathrm{tail}$

(2)　$n = 0$ のとき，$\mathrm{aux} \leftarrow \mathrm{tail}$

以上の手順により，文法規則の右辺の最後の非終端記号の整数版テーブル型属性の参照を含む意味関数から，整数版テーブル型属性の参照を除去し，グローバ

ル変数に置き換えることができる.

4. case 文の変換

ここでは，3章で述べた特殊整数版テーブル型属性除去法を用いた具体的な変換例として，case 文を4つ組コードに翻訳する属性文法の記述を効率的な動作ルーティンの記述に変換する.

4.1 case 文翻訳の属性文法と動作ルーティン

case 文は，多くの言語で用いることができる基本的な構文である．ここでは，C言語の switch 文に似た構文を持つ case 文を4つ組コードに翻訳する属性文法の記述を効率のよい動作ルーティンの記述に変換する．case 文の構文を**図3**(a)に示す．この構文と等価な文を図3(b)に示す．この記述の各行は，ほとんど4つ組コードに対応している．対応する4つ組コードの記述を図3(c)に示す．この図3(c)の形式の4つ組コードの記述を容易に生成できる属性文法の記述を考える．また，簡単のために case 文の入れ子はないものと仮定する.

```
switch expression
  begin
    case value1   : statement1
    case value2   : statement2
        ...
    case valuen-1 : statementn-1
    default       : statementn
  end
```

図 3(a) case 文の構文
Fig. 3(a) Syntax of a case statement.

```
      code to evaluate expression into T
      goto TEST
L1:   code for statement1
      goto END
L2:   code for statement2
      goto END
        ...
Ln-1: code for statementn-1
      goto END
Ln:   code for statementn
      goto END
TEST: if T=value1   goto L1
      if T=value2   goto L2
        ...
      if T=valuen-1 goto Ln-1
      goto Ln
END:
```

図 3(b) case 文と等価な文
Fig. 3(b) Translation of the case statement.

```
address )  ( quadruple )
        ) (code to evaluate expression into T)
        ) (BR, , , TEST)
  L1    ) (code for statement1)
        ) (BR, , , END)
  L2    ) (code for statement2)
        ) (BR, , , END)
          ...
  Ln-1  ) (code for statementn-1)
        ) (BR, , , END)
  Ln    ) (code for statementn)
        ) (BR, , , END)
  TEST  ) (BE, T, value1, L1)
        ) (BE, T, value2, L2)
          ...
        ) (BE, T, valuen-1, Ln-1)
        ) (BR, , , Ln)
  END   )
```

図 3(c) 翻訳する case 文の4つ組コードの形式
Fig. 3(c) Quadruples for the case statement.

```
0) S' → S
       S.start ← 1
       printout(S.code)
1) S  → C begin L T end
       C.start ← S.start
       L.start ← C.next
       T.start ← L.next
       S.code ← C.code + L.code + T.code +
                L.cond + "(BR, , ,T.start)"
       S.next ← T.next + L.count + 1
       L.end ← S.next
       T.end ← S.next
       L.val ← C.val
       C.test ← T.next
2) C  → switch E
       E.start ← C.start
       C.code ← E.code + "(BR, , ,C.test)"
       C.next ← E.next + 1
       C.val ← E.val
3) L  → B
       B.start ← L.start
       L.next ← B.next
       L.code ← B.code
       L.count ← 1
       L.cond ← "(BE,L.val,B.label,B.start)"
       B.end ← L.end
4) L0 → L1 B
       L1.start ← L0.start
       B.start ← L1.next
       L0.next ← B.next
       L0.code ← L1.code + B.code
       L0.count ← L1.count + 1
       L0.cond ← L1.cond +
                 "(BE,L0.val,B.label,B.start)"
       L1.end ← L0.end
       B.end ← L0.end
       L1.val ← L0.val
5) B  → case V : S
       V.start ← B.start
       S.start ← V.next
       B.code ← V.code + S.code + "(BR, , ,B.end)"
       B.next ← S.next + 1
       B.label ← V.label
6) T  → default : S
       S.start ← T.start
       T.code ← S.code + "(BR, , ,T.end)"
       T.next ← S.next + 1
7) V  → constant_value
       V.next ← V.start
       V.code ← ""
       V.label ← lex(constant_value)
8) S  → execute_statements
       S.next ← S.start + the number of quadruples
       S.code ← execute_statements code
9) E  → expression
       E.code ← expression code
       E.next ← E.start + the number of quadruples
       E.val ← the value of expression
```

図 4 case 文を翻訳する属性文法
Fig. 4 Translation for a case statement in an attribute grammar.

図3(a)の文法を基に，正しい4つ組コードが生成されるように，属性文法で記述すると**図4**になる．図4の中で，英大文字は非終端記号を表し，その他の文字記号は終端記号である．また，文法規則番号4番の非終端記号Lの右辺と左辺の生起を区別するために，それぞれ L^1 と L^0 を用いた．さらに，文法規則番号8番と9番の中で，実行文と式に対応する文法規則の一部を省略し，便宜的に終端記号とした．また，一変数関数 lex(x) は，終端記号 x に対応する実際の値を返し，関数 printout は引数の値を出力する．

この記述では，相続属性3つ，合成属性6つの合計9つの属性を用いて記述した．属性 start は，整数版テーブル型相続属性で，属性 next は整数版テーブル型合成属性である．合成属性 code は，目的とする4つ組コードが入る属性で，S. code にこの case 文を4つ組に翻訳した最終的なコードが入る．また，合成属性 cond は，この case 文の中で，式の値により実行される文までの条件分岐文の集合を保持する属性であり，その分岐の数を合成属性 count で保持する．このとき，式の値を評価したのちに，上記の分岐文の集合の4つ組コードのある場所への飛び先が相続属性 test で与えられる．また，その分岐先を決める値 (case 文の構文の $value_i$) は合成属性 label に蓄えられる．合成属性 val は，各記号の実際の値を保持する合成属性である．また，この case 文全体が終了したときに，遷移する位置を相続属性 end が覚えている．

この属性文法の記述を，テーブル型属性除去法やグローバル変数導入法，パッチ導入法等の最適化技術を用いて効率のよい動作ルーティンに変換すると，**図5**の記述が得られる．

4.2 変換の詳細

ここでは，図4に示した属性文法の記述を図5の動作ルーティンに変換する過程を順次示す．

まず，文法規則の還元順序を解析する．構文解析木の各ノードをポストオーダに辿る，または，文法規則の右辺の非終端記号を左から右に順に，その後左辺の非終端記号という順序を各文法規則に適用することにより機械的に求められ，次の順序になる．

始め→9→2→7→8→5→3→
(7→8→5→4)＊→8→6→1→0→終り

ここで，記号＊は0回以上の繰り返しを表す記号である．

次に，属性 val に注目する．文法規則番号9番で最初に値が求められ，その後は変更されることなく，

```
        { initialize() }
0) S' → S
        flush()
1) S  → C begin L T end
        aux ← nextlocation()
        patch(tests, nextlocation())
        generation(flushtmp())
        generation("(BR, , , aux-var4)")
        patch(ends, nextlocation())
2) C  → switch E
        tests ← { nextlocation() }
        generation("(BR, , , 0)")
3) L  → B
        var3 ← 1
        gentmp("(BE, var1, var2, nextloaction()-var4)")
4) L⁰ → L¹ B
        var3 ← var3 + 1
        gentmp("(BE, var1, var2, nextloaction()-var4)")
5) B  → case V : S
        ends ← ends + { nextlocation() }
        generation("(BR, , , 0)")
        var4 ← var4 + 1
6) T  → default : S
        ends ← ends + { nextlocation() }
        generation("(BR, , , 0)")
        var4 ← var4 + 1
7) V  → constant_value
        var2 ← lex(constant_value)
8) S  → execute_statements
        aux ← nextlocation()
        generation(execute_statements code)
        var4 ← nextloaction() - aux
9) E  → expression
        generation(expression code)
        var1 ← the value of expression
```

図5 case 文を翻訳する動作ルーティン
Fig. 5 Translation for a case statement in action routines.

他の記号の属性 val に単純にコピーされたり，参照される．このとき，属性 val は最初にそれを評価するために用いるスタック中に属性生起は高々1つであるから，この属性 val はグローバル変数 var1 に置き換えることができる．同時に，文法規則番号1番，2番，4番の冗長な単純コピーである意味関数を除去することができることに注意されたい．

また，属性 label に注目すると，文法規則番号7番で値が代入され，文法規則番号5番で単純にコピーされ，文法規則番号3番または4番で参照される．属性評価用スタックにおいて，この属性生起も高々1つであるので，グローバル変数導入法を用いて，属性 label をグローバル変数 var2 に置き換えることができる．このとき，文法規則番号5番にある冗長なコピーを除去することができる．

同様に，属性 count に注目すると，文法規則番号3番で値が求められ，構文解析木に文法規則番号4番を含む場合はその意味関数で更新され，その後構文解

析木に文法規則番号4番の適用があるなしにかかわらず，文法規則番号1番で参照される．この場合も属性評価用のスタック中の属性生起が高々1つであるので，グローバル変数導入法を用いてグローバル変数 var3 に書き換えることができる．

以上の操作により，属性 val と label と count を除去した結果を**図6**に示す．

次に，文法規則番号1番の右辺の非終端記号Lの生起を構文解析木のルートとする部分木に注目する．ここで，属性 cond は，この部分木以外には生起せず，この部分木のルートで参照され属性 code へ値を渡すためだけに参照される．また，非終端記号Bをルートとする構文解析木の中には属性 cond の生起はないので，非終端記号Bを非終端記号Lをルートとする部分木の終端記号と考えると，属性 cond は後置的合成属性とみなすことができる．そこで，単純後置翻訳法を用いて，あらかじめ初期化済みの間接出力領域 tmp への出力に変換する．この変換の中で，2つの関数 gentmp と flushtmp を導入した．これらの関数はそれ

```
0) S' → S
       S.start ← 1
       printout(S.code)
1) S  → C begin L T end
       C.start ← S.start
       L.start ← C.next
       T.start ← L.next
       S.code ← C.code + L.code + T.code +
                 L.cond + "(BR, , ,T.start)"
       S.next ← T.next + var3 + 1
       L.end ← S.next
       T.end ← S.next
       C.test ← T.next
2) C  → switch E
       E.start ← C.start
       C.code ← E.code + "(BR, , ,C.test)"
       C.next ← E.next + 1
3) L  → B
       B.start ← L.start
       L.next ← B.next
       L.code ← B.code
       var3 ← 1
       L.cond ← "(BE,var1,var2,B.start)"
       B.end ← L.end
4) L⁰ → L¹ B
       L¹.start ← L⁰.start
       B.start ← L¹.next
       L⁰.next ← B.next
       L⁰.code ← L¹.code + B.code
       var3 ← var3 + 1
       L⁰.cond ← L¹.cond + "(BE,var1,var2,B.start)"
       L¹.end ← L⁰.end
       B.end ← L⁰.end
5) B  → case V : S
       V.start ← B.start
       S.start ← V.next
       B.code ← V.code + S.code + "(BR, , ,B.end)"
       B.next ← S.next + 1
6) T  → default : S
       S.start ← T.start
       T.code ← S.code + "(BR, , ,T.end)"
       T.next ← S.next + 1
7) V  → constant_value
       V.next ← V.start
       V.code ← ""
       var2 ← lex(constant_value)
8) S  → execute_statements
       S.next ← S.start + the number of quadruples
       S.code ← execute_statements code
9) E  → expression
       E.code ← expression code
       E.next ← E.start + the number of quadruples
       var1 ← the value of expression
```

図6 属性 val と label と count を除去した記述
Fig. 6 A description without val, label, and count attributes.

```
0) S' → S
       S.start ← 1
       printout(S.code)
1) S  → C begin L T end
       C.start ← S.start
       L.start ← C.next
       T.start ← L.next
       S.code ← C.code + L.code + T.code +
                 flushtmp() + "(BR, , ,T.start)"
       S.next ← T.next + var3 + 1
       L.end ← S.next
       T.end ← S.next
       C.test ← T.next
2) C  → switch E
       E.start ← C.start
       C.code ← E.code + "(BR, , ,C.test)"
       C.next ← E.next + 1
3) L  → B
       B.start ← L.start
       L.next ← B.next
       L.code ← B.code
       var3 ← 1
       gentmp("(BE,var1,var2,B.start)")
       B.end ← L.end
4) L⁰ → L¹ B
       L¹.start ← L⁰.start
       B.start ← L¹.next
       L⁰.next ← B.next
       L⁰.code ← L¹.code + B.code
       var3 ← var3 + 1
       gentmp("(BE,var1,var2,B.start)")
       L¹.end ← L⁰.end
       B.end ← L⁰.end
5) B  → case V : S
       V.start ← B.start
       S.start ← V.next
       B.code ← V.code + S.code + "(BR, , ,B.end)"
       B.next ← S.next + 1
6) T  → default : S
       S.start ← T.start
       T.code ← S.code + "(BR, , ,T.end)"
       T.next ← S.next + 1
7) V  → constant_value
       V.next ← V.start
       V.code ← ""
       var2 ← lex(constant_value)
8) S  → execute_statements
       S.next ← S.start + the number of quadruples
       S.code ← execute_statements code
9) E  → expression
       E.code ← expression code
       E.next ← E.start + the number of quadruples
       var1 ← the value of expression
```

図7 属性 cond を除去した記述
Fig. 7 A description without cond attribute.

ぞれ，間接出力領域 tmp に出力する関数と間接出力領域 tmp の文字列を吐き出す関数である．この関数 flushtmp をコールすることにより，間接出力領域 tmp は初期化される．

この操作により，case 文の意味記述は図 7 のようになる．

次に属性 start と next に着目する．この2つの属性は整数版テーブル型属性で，この整数版テーブル型属性を定義する意味関数以外の属性の参照は，各文法規則の右辺の最後の非終端記号の整数版テーブル型属性の参照のみである．そこで，今回提案した特殊整数版テーブル型属性除去法を適用し，整数版テーブル型属性を除去すると図 8 の記述が得られる．文法規則の右辺の最後の非終端記号の整数版テーブル型属性の参照がある文法規則番号は，1番と3番と4番にあることに注意されたい．ここでは，整数版テーブル型属性をグローバル変数 var と var4 を用いて除去した．{ }の中の記述はグローバル変数の初期化動作を示す．

```
        { init: var ← 1 }
0) S' →  S
        printout(S.code)
1) S  →  C  begin  L  T  end
        C.test ← var
        S.code ← C.code + L.code + T.code +
                    flushtmp() + "(BR, , ,var-var4)"
        var ← var + var3 + 1
        L.end ← var
        T.end ← var
2) C  →  switch  E
        C.code ← E.code + "(BR, , ,C.test)"
        var ← var + 1
3) L  →  B
        L.code ← B.code
        var3 ← 1
        gentmp("(BE,var1,var2,var-var4)")
        B.end ← L.end
4) L⁰ →  L¹  B
        L⁰.code ← L¹.code + B.code
        var3 ← var3 + 1
        gentmp("(BE,var1,var2,var-var4)")
        L¹.end ← L⁰.end
        B.end ← L⁰.end
5) B  →  case  V  :  S
        B.code ← V.code + S.code + "(BR, , ,B.end)"
        var ← var + 1
        var4 ← var4 + 1
6) T  →  default  :  S
        T.code ← S.code + "(BR, , ,T.end)"
        var ← var + 1
        var4 ← var4 + 1
7) V  →  constant_value
        V.code ← ""
        var2 ← lex(constant_value)
8) S  →  execute_statements
        var ← var + the number of quadruples
        var4 ← the number of quadruples
        S.code ← execute_statements code
9) E  →  expression
        E.code ← expression code
        var ← var + the number of quadruples
        var1 ← the value of expression
```

図 8 テーブル型属性除去後の記述
Fig. 8 A description without table type attributes.

次に，属性 code に注目すると，まさに後置的合成属性である．そこで，単純後置翻訳法を用いてあらかじめ初期化済みの間接出力領域 gen への出力に変換する．このとき，間接出力領域 gen に出力する関数 generation と間接出力領域 gen の次の出力位置を返す関数 nextlocation，そして間接出力領域のデータを直接出力する関数 flush を用いた．関数 nextlocation の導入により，整数版テーブル型属性を除去したとき導入されたグローバル変数 var がそれに置き換わる．

最後に，属性 test と end に注目すると，これら2つの属性は放送型相続属性であるから，パッチ導入法を用いて除去すると最終的な記述である図5が得られる．ここで，整数値の集合を値として取る2つのグローバル変数 tests と ends を用いた．また便宜的に，補助的局所変数 aux を用いて，非終端記号の S の整数版テーブル型相続属性の値に相当する値を保持した．

5. おわりに

本論文では，属性文法の記述中の意味関数に，整数版テーブル型属性を定義する意味関数以外に，文法規則の右辺の最後の非終端記号の整数版テーブル型属性の参照を含む記述から，機械的に効率のよい動作ルーティンに変換する方法である特殊整数版テーブル型属性除去法を提案した．その変換法の例として，C言語の switch 文に似た構文の case 文を翻訳する属性文法の記述を機械的に動作ルーティンの記述に変換した．

今まで，整数版テーブル型属性を定義する意味関数以外の意味関数で整数版テーブル型属性，特にテーブル型相続属性の参照を含む属性文法の記述を機械的に動作ルーティンの記述に変換することは知られていなかった．今回の変換法の提案により，一部ではあるが整数版テーブル型属性を定義する意味関数以外の意味関数中に，整数版テーブル型属性の参照を含む属性文法の記述を変換することに成功した．

しかし，今回提案した特殊整数版テーブル型属性除去法は，文法規則の右辺の最後の非終端記号に関する

整数版テーブル型属性の参照に限ることを仮定している．さらに，各文法規則の右辺の最後の非終端記号を左辺とする文法規則以外の文法規則において，整数版テーブル型属性を定義する意味関数中の整数 tail は0であるという仮定もある．このように，特殊整数版テーブル型属性除去法には制限があり，この一般化が望まれる．この制限を取り除くことにより，より一般的な記述の完全な機械的変換が可能であろう．

謝辞 本研究に関して数々のご指導ならびにご助言を賜りました東京工業大学工学部情報工学科徳田雄洋助教授ならびに山梨大学工学部電子情報工学科今宮淳美教授，坂本忠明文部技官に深く感謝いたします．

参 考 文 献

1) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers—Principles, Techniques, and Tools*, Addison-Wesley (1986).
2) Aho, A. V. and Ullman, J. D.: *Principles of Compiler Design*, Addison-Wesley (1977).
3) Bochmann, G. V.: Semantic Evaluation from Left to Right, *Comm. ACM*, Vol. 19, No. 2, pp. 55-62 (1976).
4) Gries, D.: *Compiler Construction for Digital Computers*, John Wiley and Sons (1971).
5) Knuth, D. E.: Semantics of Context-free Languages, *Math. Systems Theory*, Vol. 2, No. 2, pp. 127-145 (1968); Correction, Vol. 5, No. 1, pp. 95-96 (1971).
6) Lewis, P. M., Rosenkrantz, D. J. and Stearns, R. E.: *Compiler Design Theory*, Addison-Wesley (1976).
7) Tokuda, T.: Two Methods for Eliminating Redundant Copy Operations from the Evaluation of Attribute Grammars, *J. Inf. Process.*, Vol. 9, No. 2, pp. 79-85 (1986).
8) Tokuda, T.: Transformation of Attribute Grammars into Efficient Action Routines by Patch Introduction, *Trans. IECEJ*, Vol. E 69, No. 9, pp. 980-987 (1986).
9) Tokuda, T.: Code Improvement Techniques in the Transformation of Attribute Grammars into Efficient Action Routines, *J. Inf. Process.*, Vol. 10, No. 1, pp. 20-26 (1986).
10) Tokuda, T.: Method for Transforming Attribute Grammars into Efficient Action Routines, *Advances in Software Science and Technology*, Vol. 1, pp. 55-69 (1989).
11) 渡辺喜道，徳田雄洋：属性文法のソースレベルの変換，情報処理学会プログラミング言語研究会，13-3 (1987).
12) 渡辺喜道：FOR ループの属性文法から動作ルーチンへの変換，電子情報通信学会論文誌 D-I, Vol. J 72-D-I, No. 10, pp. 734-741 (1989).

(平成2年2月15日受付)
(平成2年9月11日採録)

渡辺 喜道（正会員）

1964 年生．1986 年山梨大学工学部計算機科学科卒業．1988 年同大学大学院修士課程修了．現在，同大学工学部電子情報工学科助手．ソフトウェア開発環境，対話型ソフトウェアに興味をもつ．ACM, IEEE, 電子情報通信学会，ソフトウェア科学会，人工知能学会各会員．