プログラミング言語 13－3
（1987・9・16）

属性文法の
ソースレベルの変換

渡辺 喜道　　徳田 雄洋
山梨大学工学部計算機科学科

本論文は、属性文法を別の属性文法に変換する方法や、属性文法を効率のよい動作ルーティンに変換する方法について述べたものである。この動作ルーティンは標準的なボトムアップ構文解析を基礎としている。
属性文法の関数移動を用いて、ある属性文法の記述から別の属性文法の記述を導き出す例を示す。これは、データフロー解析を基礎としている。
また、テーブル型属性と呼ぶ属性を用いた属性文法の記述が、実は後置的合成属性の変形であることを示す。これは、記号表等のデータベースを扱うときの簡単なモデルである。

SOURCE-LEVEL TRANSFORMATION OF ATTRIBUTE GRAMMARS

Yoshimichi WATANABE and Takehiro TOKUDA

Department of Computer Science,
Faculty of Engineering, Yamanashi Univ.,
4-3-11 Takeda, Kofu 400, Japan

This paper gives methods for transforming some type of attribute grammars into other attribute grammars and efficient action routines. These action routines are based on the canonical bottom-up syntax analysis.
Using function motion of attribute grammars, we can obtain transformed attribute grammars from source attribute grammars. This technique is based on data-flow analysis techniques.
Then we introduce table type attributes. We show table type attributes are essentially equivalent to postfix synthesized attributes. This technique is applicable when we deal with simple databases such as symbol tables.

1．　はじめに

現在、コンパイラ生成系や構造エディタ生成系等に広く用いられている基本的方法に構文主導型翻訳がある。構文主導型翻訳は、構文記述とその意味記述の２つの部分からなる。意味記述の方法には、属性文法による記述や動作ルーティンによる記述などがある。

属性文法による記述は、構文解析の順序を仮定せず、各文法規則ごとに、その規則にローカルに属性の値を求める記述をするだけでよい。この方法は、構文解析木の上で属性の値の依存関係から属性の値が計算されるので、記述することは比較的容易である。

これに対して動作ルーティンによる記述は、各文法規則に対して、プログラムの断片を対応させ、文法規則が還元されるときに、その文法規則に対応する動作ルーティンを実行するものである。この方法は、ボトムアップ構文解析の還元の順序を仮定すると、その文法規則にローカルな変数だけでなく、グローバルな変数も扱うことができ、効率のよい記述ができる。しかし、その記述は必ずしもやさしくはない。

本論文は、属性文法を動作ルーティンに変換するいくつかの変換法[7-9]を応用し、新たに属性文法から属性文法に変換する関数移動法、属性文法を効率的な動作ルーティンに変換するコード移動法やテーブル型属性と呼ぶ属性の変換について述べたものである。

本論文の構成は次の通りである。２章では、属性文法から動作ルーティンに変換する既存の方法として、同期スタック法と単純後置翻訳法を説明する。３章では、属性文法から属性文法へ関数移動を用いて変換し、属性文法から効率的な動作ルーティンへコード移動を用いて変換する。４章では、単純後置翻訳法の発展として得られたテーブル型属性と呼ぶ属性を用いた属性文法について、例題をまじえて示す。５章では、今後の課題についてまとめる。

なお、構文主導型翻訳、属性文法、動作ルーティンの基本的な用語については文献［１、４－６］を参照されたい。

さらに、以下では基となる文脈自由文法の一般形の開始記号Ｓ’は、ただ１つの文法規則Ｓ’→Ｓ（文法規則番号０）に出現するものとする。

2．　属性文法を動作ルーティンに変換する伝統的方法

属性文法を動作ルーティンに変換するための伝統的な方法として、同期スタック法と単純後置翻訳法を説明する。

２．１　同期スタック法

同期スタック法は、属性がすべて合成属性のときのみ使うことができる伝統的な方法である。属性の値を評価するとき、その各属性生起のインスタンスにメモリを割り当てればよいが、属性の評価が下から上の方向で、構文解析木の節の訪問順序が後行順（postorder ）の順序で行われるので、すべての属性生起のインスタンスにメモリを割り当てることなしにスタックで十分評価できる。同期スタック法の概要を以下述べる。

各属性ごとに、その属性の値をしまうスタックを用いて属性の値を計算する。この属性をしまうスタックを属性用スタックと呼ぶ。同期スタック法は、ボトムアップ構文解析用のスタックと属性用スタックとを同期させて、構文解析を行う方法である。すなわち、文法記号の構文解析用スタックへのポップ、プッシュ操作と属性用スタックの属性の値へのポップ、プッシュ操作とを同期させる方法である。文法規則の右辺が左辺に還元されるときに、属性の計算は属性用スタック内で計算され、左辺の属性の値が求められる。

属性用スタックは、完全に構文解析用スタックに同期して動くので、構文解析用スタックに入れる文法記号を構造体にして、構造体の各フィールドに各属性の値を入れることもよく行われる。

算術式を４つ組コードに翻訳する属性文法の記述は図２．１のようになる。

```
0)S→E
        S.place := E.place
        S.code := E.code
1)E0→E1 + T
        E0.place := newtemp()
        E0.code := E1.code + T.code +
          ("+",E1.place,T.place,E0.place)
2)E0→E1 - T
        E0.place := newtemp()
        E0.code := E1.code + T.code +
          ("-",E1.place,T.place,E0.place)
3)E→T
        E.place := T.place
        E.code := T.code
4)T0→T1 * F
        T0.place := newtemp()
        T0.code := T1.code + F.code +
          ("*",T1.place,F.place,T0.place)
5)T0→T1 / F
        T0.place := newtemp()
        T0.code := T1.code + F.code +
          ("/",T1.place,F.place,T0.place)
6)T→F
        T.place := F.place
        T.code := F.code
7)F→( E )
        F.place := E.place
        F.code := E.code
8)F→i
        F.place := lex(i)
        F.code := ""
```

図２．１　算術式の４つ組への変換

属性ｐｌａｃｅとｃｏｄｅは文字列を値としてとり、関数ｎｅｗｔｅｍｐは呼ばれるごとに毎回新しい作業用の変数名を返す関数である。非終端記号ＥやＴの生起を区別するためにE_0、E_1とT_0、T_1を用いる。非終端記号E_0とT_0は文法規則の左辺の生起を表し、E_1とT_1は右辺の生起を表す。これを同期スタック法を用いて変換すると、図２．２のようになる。

```
0)S→E
        t := (top).place
        reduction()
        (top).place := t
1)E0→E1 + T
        t := newtemp()
        print( "+", (top-2).place, (top).place, t )
        reduction()
        (top).place := t
```

```
2)E0→E1 - T
        t := newtemp()
        print( "-", (top-2).place, (top).place, t )
        reduction()
        (top).place := t
3)E→T
        t := (top).place
        reduction()
        (top).place := t
4)T0→T1 * F
        t := newtemp()
        print( "*", (top-2).place, (top).place, t )
        reduction()
        (top).place := t
5)T0→T1 / F
        t := newtemp()
        print( "/", (top-2).place, (top).place, t )
        reduction()
        (top).place := t
6)T→F
        t := (top).place
        reduction()
        (top).place := t
7)F→( E )
        t := (top-1).place
        reduction()
        (top).place := t
8)F→i
        reduction()
        (top).place := lex(i)
```

図２．２　算術式の４つ組への変換

記法（ｔｏｐ－ｉ）．ｐｌａｃｅは構文解析用スタックのｔｏｐ－ｉ番目のｐｌａｃｅフィールドの値を表し、手続きｒｅｄｕｃｔｉｏｎは還元動作を行う手続きである。

２．２　単純後置翻訳法

同期スタック法を発展させ効率を良くした方法の１つが単純後置翻訳法である。以下の形をした合成属性を、後置的合成属性と呼ぶことにする。

すべての文法規則における属性Ａの意味関数が、次の形をしているとき、属性Ａを後置的合成属性という。ただし、属性Ａのとる値は文字列で、演算子”＋”は文字列を結合する演算子、記号X_0，X_1，・・・，X_n（ｎは０以上の整数）はそれぞれ１つの非終端記号、記号a_0，a_1，・・・，a_nは長さ０以上の終端記号列、そして、ｔａｉｌは文字列である。

１）０番目のルールの場合
　Ｓ’→Ｓ
　　出力（Ｓ．Ａ）

２）０番目のルール以外の場合
　$X_0 \rightarrow a_0\ X_1\ a_1\ \cdots\ X_n\ a_n$
　　$X_0.A\ :=\ X_1.A\ +\ X_2.A\ +\ \cdots\ +\ X_n.A\ +\ tail$

このとき、属性Ａのための属性用スタックは不要となり、上記意味関数を、次のように置き換えることができる。
１）０番目のルールの場合
　　動作なし
２）０番目のルール以外の場合
　　出力（ｔａｉｌ）

中置記法の算術式から後置記法の算術式へ変換する属性文法の記述を例にとる。属性文法の記述は図２．３のようになる。

```
0)S→E
        S.code := E.code
1)E0→E1 + T
        E0.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２．３　中置記法から後置記法への変換

ここで、属性ｃｏｄｅは後置的合成属性である。これを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２．４のようになる。

```
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２．４　中置記法から後置記法への変換

ｔａｉｌの部分が空の文字列の出力の場合、動作はなくなることに注意されたい。

３．　属性文法の関数移動と動作ルーティンのコード移動

ここでは、属性文法における関数移動と動作ルーティンにおけるコード移動について例を１つずつ用いて説明する。

３．１　関数移動やコード移動の例

図３．１は２進小数の値を評価する属性文法の記述である。これは、属性文法における関数移動の例である。

```
0)S→0 . L
        S.val := L.val / 2
1)L0→B L1
        L0.val := B.val + L1.val / 2
2)L→B
        L.val := B.val
3)B→1
        B.val := 1
4)B→0
        B.val := 0
```

図３．１　２進小数の値の評価

ここで、属性ｖａｌは実数の値をとる。文法番号１で非終端記号Ｌの生起を区別するために、L_0 とL_1 を用いる。非終端記号L_0 は文法規則の左辺の生起を表し、非終端記号L_1 は、右辺の生起を表す。

関数移動のテクニックを用いると図３．２のように変換することができる。

```
0)S→0 . L
        S.val := L.val
1)L0→B L1
        L0.val := (B.val + L1.val) / 2
2)L→B
        L.val := B.val / 2
3)B→1
        B.val := 1
4)B→0
        B.val := 0
```

図３．２　２進小数の値の評価

次に、動作ルーティンにおけるコード移動の例を示す。図３．３は、与えられたかっこの列が釣り合っているかどうかをチェックする属性文法の記述である。

```
0)N→L
        N.test := (L.test) and (L.count = 0)
1)L0→L1 B
        L0.count := L1.count + B.count
        L0.test := (L1.test) and
                        (L0.count ≧ 0)
2)L→B
        L.count := B.count
        L.test := (L.count ≧ 0)
3)B→(
        B.count := +1
4)B→)
        B.count := -1
```

図３．３　かっこの釣合のチェック

ここで、属性ｃｏｕｎｔとｔｅｓｔの値はそれぞれ整数と論理型である。ｔｅｓｔの値がｔｒｕｅのとき、釣合がとれている。この属性文法をグローバル変数導入法[9] を用いると図３．４のようになる動作ルーティンの記述になる。

```
0)N→L
        var2 := var2 and (var0 = 0)
1)L0→L1 B
        var0 := var0 + var1
        var2 := var2 and (var0 ≧ 0)
2)L→B
        var0 := var1
        var2 := (var0 ≧ 0)
3)B→(
        var1 := +1
4)B→)
        var1 := -1
```

図３．４　かっこの釣合のチェック

しかし、この動作ルーティンもコード移動等の最適化テクニックを用いると図３．５のように自然で効率的な動作ルーティンの記述が得られる。

```
{初期時点：count := 0 }
0)N→L
        if count = 0 then yse else no
1)L0→L1 B
        if count < 0 then no
2)L→B
        if count < 0 then no
3)B→(
        count := count + 1
4)B→)
        count := count - 1
```

図３．５　かっこの釣合のチェック

３．２　一般的原理

ここでは、属性文法における関数移動の一般的原理と動作ルーティンにおけるコード移動の一般的原理について述べる。

３．２．１　属性文法における関数移動

どんな構文解析木に対しても、その依存グラフから求められる属性の値のすべての評価順序について、ある一連の属性の値の評価が、ある属性の値の評価の列の先頭にあった場合、その一連の属性の値の評価をそれより前の属性の値の評価の列に移動することができる。例えば、属性値の評価列の集合AE_{11}，AE_{12}，・・・，AE_{1n}，AE_{21}，AE_{22}，・・・，AE_{2m}があり、これらは次のような内容であったとする。

AE_{11}：
　　$f_{11}()$
AE_{12}：
　　$f_{12}()$
　・・・
AE_{1n}：
　　$f_{1n}()$
AE_{21}：
　　$g_{21}(h())$

AE_{22}:
　　$g_{22}(h())$
　・・・
AE_{2m}:
　　$g_{2m}(h())$
ここでf_{1i}やg_{2i}やhは意味関数である。さらに、これらの属性評価列は次の条件を満たしていたとする。
I）属性値の評価列AE_{11}，AE_{12}，・・・，AE_{1n}のうちの1つが全属性評価値の列の中にあるならば、その属性値の評価列の直後に必ず属性値の評価列AE_{21}，AE_{22}，・・・，AE_{2m}のうち1つが存在する。
II）属性値の評価列AE_{21}，AE_{22}，・・・，AE_{2m}のうちの1つが全属性値の評価列の中にあるならば、その属性値の評価列の直前に必ず属性値の評価列AE_{11}，AE_{12}，・・・，AE_{1n}のうち1つが存在する。
　このような状況下では、次のように変形できる。
AE_{11}:
　　$h(f_{11}())$
AE_{12}:
　　$h(f_{12}())$
　・・・
AE_{1n}:
　　$h(f_{1n}())$
AE_{21}:
　　$g_{21}()$
AE_{22}:
　　$g_{22}()$
　・・・
AE_{2m}:
　　$g_{2m}()$

3．2．2　動作ルーティンにおけるコード移動

　どんな解析にもあらわれる一連の動作列があれば、その動作列のある生成規則から別の生成規則にその一連の動作列を移動することができる。
1）代入文の初期時点への移動
　変数の初期化のための代入は、実行時に行われる最初の動作に移動することができる。例えば、次のような動作ルーティンがあり、どんな入力に対しても変数ｖａｒへの参照（代入）はこの動作ルーティンが最初であるとする。
　動作ルーティン：
　　　　・・・
　　　　var := 定数
　　　　・・・
このとき、この変数ｖａｒへの代入はこの動作ルーティンから除去され、実行時における最初の動作に移動できる。
　初期時点：
　　　　var := 定数
　動作ルーティン：
　　　　・・・
　　　　・・・
2）コードの前方移動
　ある一連の動作が、それより後にくるコードの先頭にあった場合、その一連の動作をそれより前のコードに移動することができる。例えば、動作ルーティンの集合AR_{11}，AR_{12}，・・・，AR_{1n}，AR_{21}，AR_{22}，・・・，AR_{2m}があり、これらは次のような内容であったとする。
AR_{11}:
　　A_{11}
AR_{12}:
　　A_{12}
　・・・
AR_{1n}:
　　A_{1n}
AR_{21}:
　　B
　　C_{21}
AR_{22}:
　　B
　　C_{22}
　・・・
AR_{2m}:
　　B
　　C_{2m}
さらに、これらの動作ルーティンは次の条件を満たしていたとする。
I）動作ルーティンAR_{11}，AR_{12}，・・・，AR_{1n}のうちの1つが全動作列の中にあるならば、その動作ルーティンの直後に必ず動作ルーティンAR_{21}，AR_{22}，・・・，AR_{2m}のうち1つが存在する。
II）動作ルーティンAR_{21}，AR_{22}，・・・，AR_{2m}のうちの1つが全動作列の中にあるならば、その動作ルーティンの直前に必ず動作ルーティンAR_{11}，AR_{12}，・・・，AR_{1n}のうち1つが存在する。
　このような状況下では、Bを次のように移動することができる。
AR_{11}:
　　A_{11}
　　B
AR_{12}:
　　A_{12}
　　B
　・・・
AR_{1n}:
　　A_{1n}
　　B
AR_{21}:
　　C_{21}
AR_{22}:
　　C_{22}
　・・・
AR_{2m}:
　　C_{2m}

3．3　変換の詳細

　まず、図3．1の属性文法を図3．2のような属性文法に変換する場合について述べる。
1）はじめに、依存グラフから属性の値の評価順序を人手で解析する。各文法規則の意味関数はすべて1つであるから意味関数の代わりに文法規則番号を用いて表現することにする。
①任意の構文解析木に対して、文法番号0の属性評価は最後に行われる。
②次に、文法番号1の属性の評価の順序に注目すると、どんな構文解析木においても木の高さの高い順から属性の値が評価されていく。また、最も木の高さが高い位置にある文法番号1よりも先に文法番号2の属性の値の評価がされなければならない。
③さらに、文法番号3と4の属性の値の評価は文法番号0または1または2の属性の値の評価が行われる時点で、その文法番号0または1または2のある構文解析木の高さより高い位置にあるものはすべて評価されていなければなら

ない。

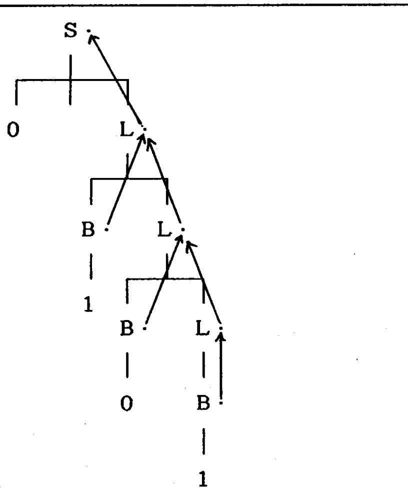

図３．６　構文解析木の例

ここでは、文法番号０から２までを変形する（すなわち、Ｓ．ｖａｌとＬ．ｖａｌの評価の変形をする）ので、これらの番号の属性の評価が行われるときには文法番号３と４の属性の評価（Ｂ．ｖａｌの評価）は既に行われているとする。

２）関数移動を用いて関数の移動を行う。

図３．１の記述を関数div2（引数を２で割った値を返す関数）とplus（２つの引数の和を返す関数）を用いて次のように書き換える。

```
0)S→0 . L
        S.val := div2(L.val)
1)L0→B L1
        L0.val := plus(B.val, div2(L1.val))
2)L→B
        L.val := B.val
3)B→1
        B.val := 1
4)B→0
        B.val := 0
```

図３．７　２進小数の値の評価

I)構文解析列に１が含まれない場合

文法番号２の属性の値の評価の直後に文法番号０の属性の値が評価されるので、文法番号０の関数div2を文法番号１に移動する。

```
2) L.val := B.val
0) S.val := div2(L.val)
        ↓
2) L.val := div2(B.val)
0) S.val := L.val
```

II) 構文解析列に１が含まれる場合

文法番号２の属性の値の評価の直後に文法番号１の属性の値が評価されるので、その直後の文法番号１の関数div2を文法番号２に移動する。

```
2) L.val := B.val
1) L0.val := plus(B.val, div2(L1.val))
1) L0.val := plus(B.val, div2(L1.val))
        ...
1) L0.val := plus(B.val, div2(L1.val))
1) L0.val := plus(B.val, div2(L1.val))
0) S.val := div2(L.val)
        ↓
2) L.val := div2(B.val)
1) L0.val := plus(B.val, L1.val)
1) L0.val := plus(B.val, div2(L1.val))
        ...
1) L0.val := plus(B.val, div2(L1.val))
1) L0.val := plus(B.val, div2(L1.val))
0) S.val := div2(L.val)
```

さらに、その文法番号１の直後の文法番号１との間で、関数div2が直前に移動できる。この動作を繰り返す。

```
2) L.val := div2(B.val)
1) L0.val := div2(plus(B.val, L1.val))
1) L0.val := plus(B.val, div2(L1.val))
        ...
1) L0.val := plus(B.val, div2(L1.val))
1) L0.val := plus(B.val, div2(L1.val))
0) S.val := div2(L.val)
        ↓
2) L.val := div2(B.val)
1) L0.val := div2(plus(B.val, L1.val))
1) L0.val := div2(plus(B.val, L1.val))
        ...
1) L0.val := plus(B.val, div2(L1.val))
1) L0.val := plus(B.val, div2(L1.val))
0) S.val := div2(L.val)
        ↓
2) L.val := div2(B.val)
1) L0.val := div2(plus(B.val, L1.val))
1) L0.val := div2(plus(B.val, L1.val))
        ...
1) L0.val := div2(plus(B.val, L1.val))
1) L0.val := plus(B.val, div2(L1.val))
0) S.val := div2(L.val)
```

最後に、文法番号１の属性の値の評価の直後に文法番号０の属性の値の評価があるので、文法番号１に文法番号０の関数div2を移動する。

```
2) L.val := div2(B.val)
1) L0.val := div2(plus(B.val, L1.val))
1) L0.val := div2(plus(B.val, L1.val))
        ...
1) L0.val := div2(plus(B.val, L1.val))
1) L0.val := plus(B.val, div2(L1.val))
0) S.val := div2(L.val)
        ↓
2) L.val := div2(B.val)
1) L0.val := div2(plus(B.val, L1.val))
1) L0.val := div2(plus(B.val, L1.val))
        ...
1) L0.val := div2(plus(B.val, L1.val))
1) L0.val := div2(plus(B.val, L1.val))
0) S.val := L.val
```

I),II)より、新しい属性文法の記述、図３．８を得る。

```
0)S→0 . L
        S.val := L.val
1)L0→B L1
        L0.val := div2(plus(B.val,L1.val))
2)L→B
        L.val := div2(B.val)
3)B→1
        B.val := 1
4)B→0
        B.val := 0
```

図３．８　変換後の属性文法

これは、関数div2とplusを用いずに記述すると図３．２と同一である。これは、図３．１よりも自然で分かりやすい記述である。また、この変換により、属性文法から属性文法への変換ができ、変換の種類によっては、効率のよい動作ルーティンに変換しやすい属性文法の記述が得られる可能性がある。

次に図３．４の属性文法の記述を図３．５のような記述に変換する。

図３．４をショートサーキット型にし、初期代入の導入[9]を行うと図３．９の記述が得られる。

```
0)N→L
        if var0 = 0 then yes else no
1)L0→L1 B
        var0 := var0 + var1
        if var0 < 0 then no
2)L→B
        var0 := 0
        var0 := var0 + var1
        if var0 < 0 then no
3)B→(
        var1 := +1
4)B→)
        var1 := -1
```

図３．９　かっこの釣合のチェック

生成規則の還元の順序を人手で解析すると図３．１０のようになる。

```
始め→{3,4}→2→0→終わり
始め→{3,4}→2→{3,4}→1→0→終わり
始め→{3,4}→2→{3,4}→1→{3,4}→1→0→終わり
始め→{3,4}→2→{3,4}→1→・・・→{3,4}→1
                →0→終わり
```

図３．１０　還元の順序

変数var0への代入は、文法番号２が最初であるから、その代入を初期時点に移動する。結果を図３．１１に示す。

```
{初期時点：var0 := 0 }
0)N→L
        if var0 = 0 then yes else no
1)L0→L1 B
        var0 := var0 + var1
        if var0 < 0 then no
2)L→B
        var0 := var0 + var1
        if var0 < 0 then no
3)B→(
        var1 := +1
4)B→)
        var1 := -1
```

図３．１１　かっこの釣合のチェック

次に、文法番号１や２の還元は文法番号３や４の還元よりも必ず後にあるので、文法番号１と２の
  var0 := var0 + var1
を文法番号３と４に移動する。結果は図３．１２のようになる。

```
{初期時点：var0 := 0 }
0)N→L
        if var0 = 0 then yes else no
1)L0→L1 B
        if var0 < 0 then no
2)L→B
        if var0 < 0 then no
3)B→(
        var1 := +1
        var0 := var0 + var1
4)B→)
        var1 := -1
        var0 := var0 + var1
```

図３．１２　かっこの釣合のチェック

これに、定数たたみ込みを行うと図３．１３の記述を得る。これは、図３．５の記述と変数名が異なるだけであり、同一である。

```
{初期時点：var0 := 0 }
0)N→L
        if var0 = 0 then yes else no
1)L0→L1 B
        if var0 < 0 then no
2)L→B
        if var0 < 0 then no
3)B→(
        var0 := var0 + 1
4)B→)
        var0 := var0 - 1
```

図３．１３　かっこの釣合をチェックする属性文法

４．テーブル型属性

記号表等のデータベースを扱うときに、有効となる単純後置翻訳法の変形を示す。このとき、構文解析はボトムアップ構文解析で行われると仮定する。

４．１　テーブル型属性の例

図４．１は変数値表を作成するための属性文法の記述である。

```
0)P→B
        B.old := ""
        print(B.new)
1)B→S
        S.old := B.old
        B.new := S.new
2)B0→B1 ; S
        B1.old := B0.old
        S.old := B1.new
        B0.new := S.new
3)S→A
        A.old := S.old
        S.new := A.new
4)A→L = N
        A.new := A.old + ( L.code, N.val )
5)N→int_val
        N.val := (int) lex(int_val)
6)L→letter
        L.code := lex(letter)
```

図４．１　変数値表を作成する属性文法

ここで、属性ｏｌｄは相続属性で属性ｎｅｗとｖａｌとｃｏｄｅは合成属性である。属性ｖａｌは整数の値をとり、その他の２つの属性は文字列の値をとる。演算子”＋”は文字列の結合を表す演算子である。

この属性文法は図４．２のような動作ルーティンに変換することができる。

```
{初期時点：clear()}
0)P→B
1)B→S
2)B0→B1 ; S
3)S→A
4)A→L = N
        print("( L.code, N.val )")
5)N→int_val
        N.val := (int) lex(int_val)
6)L→letter
        L.code := lex(letter)
```

図４．２　変数値表を作成する文法

もう１つ例を示す。図４．３は論理式を４つ組に変換する属性文法の記述である。

```
0)Z→E
        E.start := 2
        E.true := E.next + 1
        E.false := E.next
        Z.code := (:=,1,,Z) + E.code + (:=,0,,Z)
1)E→T
        T.start := E.start
        E.next := T.next
        T.true := E.true
        T.false := E.false
        E.code := T.code
2)E0→T or E1
        T.start := E0.start
        E1.start := T.next
        E0.next := E1.next
        T.true := E1.next := E0.true
        E1.false := E0.false
        T.false := T.next
        E0.code := T.code + E1.code
3)T→F
        F.start := T.start
        T.next := F.next
        F.true := T.true
        F.false := T.false
        T.code := F.code
4)T0→F and T1
        F.start := T0.start
        T1.start := F.next
        T0.next := T1.next
        T1.true := T0.true
        F.false := T1.false := T0.false
        F.true := F.next
        T0.code := F.code + T1.code
5)F→i
        F.next := F.start + 1
        F.code := (Br,lex(i),F.true,F.false)
6)F→( E )
        E.start := F.start
        F.next := E.next
        E.true := F.true
        E.false := F.false
        F.code := E.code
```

図４．３　論理式を４つ組に変換する属性文法

ここで、属性ｓｔａｒｔとｎｅｘｔは次の出力位置を計算するための補助的な属性である。

この属性文法は、パッチ導入法と還元導入法を用いると図４．４のように書き換えることができる。[8]

```
{初期時点：generate(:=,1,,Z)}
0)Z→E
        patch(E.true, true, nextlocation()+1)
        patch(E.false, false, nextlocation())
        generate(:=,0,,Z)
1)E→T
        E.true := T.true
        E.false := T.false
2a)E0→TOR E1
        E0.true := TOR.true + E1.true
        E0.false := E1.false
2b)TOR→T or
        TOR.true := T.true
        patch(T.false, false, nextlocation())
3)T→F
        T.true := F.true
        T.false := F.false
4a)T0→FAND T1
        T0.false := FAND.false + E1.false
```

```
        T0.true := T1.true
4b)FAND→F and
        patch(F.true, true, nextlocation())
        FAND.false := F.false
5)F→i
        F.true := F.false := {nextlocation()}
        generate(Br, lex(i), 0, 0)
6)F→( E )
        F.true := E.true
        F.false := E.false
```

図4．4　論理式の4つ組への変換

ここで、関数nextlocationは間接出力領域の、次の出力位置を返す関数であり、関数generateは間接出力領域への出力である。また、関数patch は第一引数（集合型）の各アドレスの第二引数で示されるフィールドに第三引数の値をを代入する関数である。各属性のとる値はアドレスの集合であり、演算子”＋”は和集合を表す。

また、図4．4の記述は非同期スタック法[7] を用いると図4．5のような記述になる。記述が非常に簡単になった。

```
{初期時点 : generate(:=, 1, , Z)}
0)Z→E
        pop(TRUE, op)
        patch(op, true, nextlocation()+1)
        pop(FALSE, op)
        patch(op, false, nextlocation())
        generate(:=, 0, , Z)
1)E→T
2a)E0→TOR E1
        pop(TRUE, op2)
        pop(TRUE, op1)
        push(TRUE, op1 + op2)
2b)TOR→T or
        pop(FALSE, op)
        patch(op, false, nextlocation())
3)T→F
4a)T0→FAND T1
        pop(FALSE, op2)
        pop(FALSE, op1)
        push(FALSE, op1 + op2)
4b)FAND→F and
        pop(TRUE, op)
        patch(op, false, nextlocation())
5)F→i
        push(TRUE, {nextlocation()})
        push(FALSE, {nextlocation()})
        generate(Br, lex(i), 0, 0)
6)F→( E )
```

図4．5　論理式の4つ組への変換

4．2　一般的原理と説明

テーブル型属性と呼ぶ属性は、実は後置的合成属性の変形であることを示す。見かけ上2つの属性が働いているが、実は1つの後置的合成属性と同じふるまいをする。以下の形をした2つの属性の組をテーブル型属性（文字列版テーブル型属性）と呼ぶ。

属性ｏｌｄとｎｅｗは、文字列を値としてとる属性で、それぞれ相続属性と合成属性である。演算子”＋”は文字列の結合演算子、ｔａｉｌは文字列である。

１）０番目のルールの場合
S’→S
　　Ｓ．ｏｌｄ　：＝　初期文字列
　　出力（Ｓ．ｎｅｗ）
２）０番目のルール以外でかつｎ＞０の場合
$X_0 \rightarrow a_0 \ X_1 \ a_1 \ X_2 \ \cdots \ X_n \ a_n$
　　X_1．ｏｌｄ　：＝　X_0．ｏｌｄ
　　X_2．ｏｌｄ　：＝　X_1．ｎｅｗ
　　X_3．ｏｌｄ　：＝　X_2．ｎｅｗ
　　　・・・
　　X_n．ｏｌｄ　：＝　X_{n-1}．ｎｅｗ
　　X_0．ｎｅｗ　：＝　X_n．ｎｅｗ＋ｔａｉｌ
３）０番目のルール以外でかつｎ＝０の場合
$X_0 \rightarrow a_0$
　　X_0．ｎｅｗ　：＝　X_0．ｏｌｄ＋ｔａｉｌ

上記のような属性文法で、初期文字列が空列のとき、テーブル型属性は、合成属性１つに変換できる。また、初期文字列が空列でないとき、初期動作を許す単純後置翻訳と意味的には同一となり、これらの意味関数は次の動作ルーティンに変換できる。
０）初期動作
　　出力（初期文字列）
１）０番目のルールの場合
　　動作なし
２）０番目のルール以外でかつｎ＞０の場合
　　出力（ｔａｉｌ）
３）０番目のルール以外でかつｎ＝０の場合
　　出力（ｔａｉｌ）

さらに、属性ｏｌｄとｎｅｗが整数の値をとるように応用することができる。これを整数版テーブル型属性と呼ぶことにする。属性文法の記述が以下のようなとき整数版テーブル型属性と呼ぶ。ただし、演算子”＋”は加法演算子、ｔａｉｌは整定数である。

１）０番目のルールの場合
S’→S
　　Ｓ．ｏｌｄ　：＝　初期整定数
　　出力（Ｓ．ｎｅｗ）
２）０番目のルール以外でかつｎ＞０の場合
$X_0 \rightarrow a_0 \ X_1 \ a_1 \ X_2 \ \cdots \ X_n \ a_n$
　　X_1．ｏｌｄ　：＝　X_0．ｏｌｄ
　　X_2．ｏｌｄ　：＝　X_1．ｎｅｗ
　　X_3．ｏｌｄ　：＝　X_2．ｎｅｗ
　　　・・・
　　X_n．ｏｌｄ　：＝　X_{n-1}．ｎｅｗ
　　X_0．ｎｅｗ　：＝　X_n．ｎｅｗ＋ｔａｉｌ
３）０番目のルール以外でかつｎ＝０の場合
$X_0 \rightarrow a_0$
　　X_0．ｎｅｗ　：＝　X_0．ｏｌｄ＋ｔａｉｌ

この場合も、実は単純後置翻訳の拡張したものと意味的に同一となり、これらの意味関数は次のような動作ルーティンに変換することができる。ただし、変数ｖａｒはグローバル変数で初期化を行う。
０）初期動作
　　ｖａｒ　：＝　初期整定数
１）０番目のルールの場合

出力（ｖａｒ）
２）０番目のルール以外でかつｎ＞０の場合
　　ｖａｒ　：＝　ｖａｒ　＋　ｔａｉｌ
３）０番目のルール以外でかつｎ＝０の場合
　　ｖａｒ　：＝　ｖａｒ　＋　ｔａｉｌ

次にこのテーブル型属性が単純後置翻訳と意味的に同一であることを説明する。このことを説明するためには次の３つが単純後置翻訳と同じであることをいえばよい。
ここでは、文字列版テーブル型属性について説明をする。整数版テーブル型属性の場合も同様に説明できる。
１）構文解析木の各記号を訪問する順序
２）還元時の動作
３）初期時の動作

１）構文解析木をたどる順序
テーブル型属性の場合、ルールが
$X_0 \rightarrow a_0 \ X_1 \ a_1 \ X_2 \ \cdots \ X_n \ a_n$
の時、$X_1 \rightarrow X_2 \rightarrow \cdots \rightarrow X_n \rightarrow X_0$の順序つまり後行順で各非終端記号をたどる。単純後置翻訳の場合は、意味関数は
$X_0.code := X_1.code +$
$X_2.code + \cdots +$
$X_n.code + tail$
であるから、やはり後行順に各非終端記号をたどる。
２）還元時の動作
テーブル型属性の場合、ルール
$X_0 \rightarrow a_0 \ X_1 \ a_1 \ X_2 \ \cdots \ X_n \ a_n$
が還元されるとき、X_0.ｎｅｗに追加されるものは、文字列ｔａｉｌだけである。つまり、合成属性として親に追加され渡されるものは文字列ｔａｉｌだけである。単純後置翻訳の場合も、意味関数の記述から分かるように同様である。
３）初期時の動作
単純後置翻訳とテーブル型属性の最大の違いは、テーブル型属性ではルートで初期値を渡すところである。しかし、単純後置翻訳で初期時点での動作を許すように拡張すればよく（拡張後置翻訳法[8]）、問題とはならない。
以上より、単純後置翻訳とテーブル型属性は意味的に等しいと言える。
したがって、単純後置翻訳と同様に動作ルーティンに変換することができる。

５．おわりに

本論文の中で、属性文法を属性文法に変換したり、属性文法を効率のよい動作ルーティンに変換する方法を述べた。
属性文法における関数移動により、属性文法を別の属性文法に変換することができた。これを発展させると、効率のよい動作ルーティンを導くための属性文法の記述ができ、有益であると思われる。また、動作ルーティンにおけるコード移動により、属性文法を効率のよい動作ルーティンに変換することができた。しかし、今の段階では、これらの変形を行うときのデータフローの解析を人の手を介して行っている。これを一般化することが今後の課題である。
次に、テーブル型属性により、記号表等のデータベースを扱うときの、効率のよい動作ルーティンを導くような属性文法の記述を得た。しかし、本論文の方法では、テーブル参照に制限が伴う。今の状態では、テーブル参照をしようとしたとき、その位置より構文解析木において左側で登録されたものは参照できるが右側出登録されたものは参照できない。このテーブル型属性が一般化されることが今後の課題であり、一般化されると効率のよい属性文法の記述や動作ルーティンの記述ができるであろう。

《参考文献》

[1]Aho, A.V., Sethi, R., and Ullman, J.D. Compilers, Principles Techniques, and Tools, Addison-Wesley (1986).
[2]Aho, and Ullman, J.D. Principles of Compiler Desigin, Addison-Wesley (1977).
[3]Bochmann, G.V. Semantic evaluation from left to right, Comm. ACM, 19, 2(Feb. 1976), 55-62
[4]Gries, D. Compiler Construction for Digital Computers, Jhon Wiley and Sons (1971).
[5]Knuth, D.E. Semantics of context-free languages, Math. Systems Theory 2,2 (1968) 127-145; Correction 5,1 (1971), 95-96.
[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. Information Processing 9,2 (1986), 79-85.
[8]Tokuda T. Transformation of attributre grammars into efficient action routines by patch introduction, Trans. IECEJ E69,9(1986), 980-987.
[9]Tokuda, T. Code improvement techniques in the transformation of attribute grammars into efficient action routines, J. Information Processing 10,1 (1986), 20-26.