6G−8　コンパイラ生成系Copeとその意味評価法

小池　太　　井上　謙蔵
(東京理科大学)

1. はじめに

我々は、属性文法に代わる意味記述法として、プール変数、ノード変数と呼ぶ2つの意味評価変数を利用するコンパイラ生成系について研究を行なってきた[1][2]。今回、この意味評価法に基づいて作成されたコンパイラ生成系Cope、及びその特徴について報告する。

2. Copeの構成

Copeの構成をFig.1に示す。Copeは字句解析部生成系、構文解析部生成系、意味解析部生成系、及び、コード生成部生成系から成る。

言語仕様ファイルには、後に述べる意味評価変数を利用し、定められた書式に従って言語の字句規則、構文規則、意味規則をまとめて記述する。また、機械仕様ファイルには、コード生成部のうち特に目的計算機に依存する部分の情報を記述する。

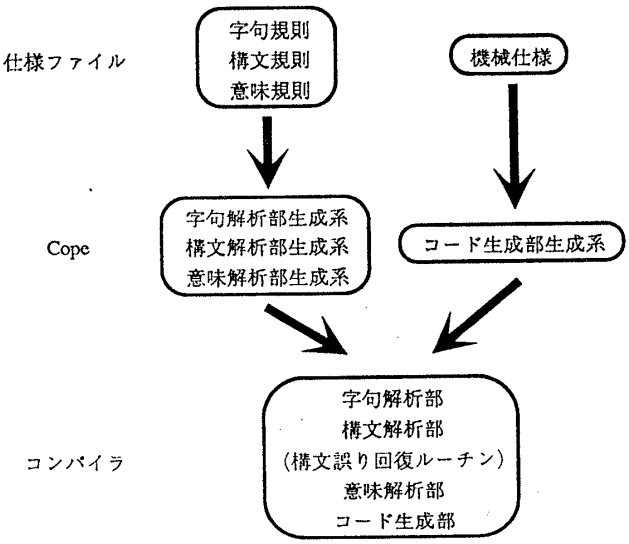

Fig.1　Copeの構成

3. 意味評価法

意味記述法として属性文法を利用する場合、意味評価の局所性から、属性のコピールールの増大が問題となる。そこで、プール変数、ノード変数の2種類の意味評価変数を導入することによって意味記述を行なう。

プール変数は、構文解析中に発生する特定の非端記号を根とする部分木内で有効な変数である。プール変数は、構文解析において、特定の端記号が移動されたときに発生し、プール変数の付与された非端記号の還元により消滅する。プール変数の評価の過程をFig.2に示す。ここで、太字はプログラムテキストであり、下線部はプール変数とその値を示す。

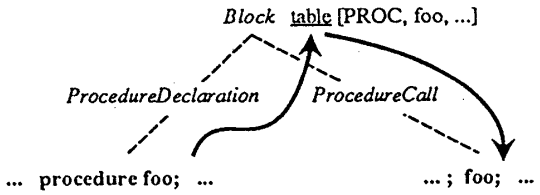

Fig.2　プール変数の評価の過程

部分木内の情報は、すべてプール変数tableに集められる。この情報は、部分木内のどこからでも直接利用することが出来る。したがって、プログラム中の広範囲に必要な情報をその範囲で有効なプール変数の値として評価することが可能である。

ノード変数は、構文解析中に発生する特定の非端記号、端記号に付与できる変数であり、属性文法の合成属性に相当する。このノード変数により、局所的な情報を効率的に評価することができる。

これら2つの意味評価変数を利用することによって、効率的な意味評価、並びに、簡潔な意味記述を行なうことが可能となった。

記述例をFig.3に示す。書式はyaccのそれをを参考にしている。

```
%{ /* プール変数、ノード変数の型の定義 */
   typedef struct { char name[32]; int value; } *TABLE;
   typedef char SYMBOL[32];
   ...
%}
%pool Program, Block: table: TABLE
 /* プール変数の宣言 */
   %{ table = make_table(100); %}  /* 開始手続き */
   %{ write_table(table); %}  /* 終了手続き */
%node 'identifier': symbol: SYMBOL;
 /* ノード変数の宣言 */
   ...
%% /* 以下構文、意味規則 */
Program : ProgramHeading Block '.' ;
   ...
VariableList : 'identifier'
   %{ enter('identifier'.symbol, table); %}
   ...
%% /* 以下ユーザー定義関数記述 */
TABLE make_table(int size)
   ...
```

Fig.3　言語記述例

A Compiler Generator 'Cope' and the Method of Its Semantic Evaluation
Futoshi KOIKE, Kenzo INOUE
Science University of Tokyo

4. 構文解析法

Cope では与えられた構文規則のクラスに応じて SLR (1)、LALR(1)、LR(1) 構文解析表を作り出す。以下にその直感的な説明を述べる。

LALR(1) 項集合の集まり（以下、LALR(1) 集とする）は、核が一致するすべての LR(1) 項集合の集合和を計算したものと言える。しかし、この処理によって還元還元競合を生じ、LALR(1) 文法ではなくなる場合がある。そこで、このような集合和を計算しないように LR(1) 集を計算すれば、最小状態数の LR(1) 集が得られることになる。これを LR(0) 集から出発し、SLR(1)、LALR(1)、そして、状態を分割することによって LR(1) 集にまで発展させるというものである。

5. 構文誤り回復

Cope は、与えられた構文規則から自動的に構文誤り回復ルーチンを生成する。

LR 構文解析の性質上、誤った移動が行なわれることはない。しかし、SLR(1)、LALR(1) 構文解析では、誤りを発見するまでに余分な還元動作を行なってしまうことがある。

Cope における構文誤り回復の戦略は以下の通りである。

1. 構文解析スタックの最上段の状態に対し、それ以外の残りのスタック、入力記号を構文解析表と照らし合わせ、修正を行なう。
2. 1. の操作に失敗した場合、修正以前の状態に戻り、今度は入力記号に対して修正を行なう。
3. 以上に失敗した場合、パニックモードによる修正を行なう。

また、使用者が誤り回復のための記述を行なうことも可能である。これには、yacc で利用できる誤り回復法の他、入力記号の種類別に処理を記述するというものである。記述例を Fig.4 に示す。

```
Expression : Expression '+' Expression
/* 還元を行なえない全ての記号が対象 */
  %error %{ /* 誤り記述 */ %}
  %{ /* 生成規則に対する意味記述 */ %}
Expression : 'number'
/* 入力記号が '(' 、 'number' に対する処理 */
  %error '(', 'number' %{ /* 誤り記述 */ %}
        ...
```

Fig.4　誤り記述例

6. 曖昧な構文に対する動的な解決

Cope は、曖昧な構文を動的に解決する手段をもつ。例えば、

Variable → 'identifier'
FunctionName → 'identifier'

という 2 つの生成規則が還元還元競合を引き起こしたとする。このとき、この情報を構文解析部から意味解析部に送り、記号表に登録されている識別子を調べることによって競合動作を解決する。記述例を Fig.5 に示す。こ

```
Variable : 'identifier'
  %conflict %{ ref('identifier'.name, VAR, table); %}
  %{ /* 生成規則に対する意味記述 */ %}
FunctionName : 'identifier'
  %conflict %{ ref('identifier'.name, FNC, table); %}
        ...
```

Fig.5　曖昧な構文に対する記述例

こで、ユーザー関数 ref は、記号表に登録されている識別子の種類が一致するか否かを判定する関数であり、その真偽値によって対応する生成規則が適用される。

移動還元競合についても同様な記述法を利用する。この場合、あてはまる還元動作が存在しなかったときに移動動作を適用することになる。

7. コード生成部生成系

コード生成部において、本質的に目的計算機に依存した部分とそうでない部分に分離する。そして、目的計算機に関する情報（使用できる命令の種類や、レジスタの特性等）を機械記述ファイルとしてコード生成部生成系に与えることによってコード生成部を生成する。

コード生成は、中間コードに対し、コード生成のための生成規則を適用することにより行なわれる。

記述例を Fig.6 に示す。

```
/* レジスタの特性記述 */
%register {
  D0 { 0, "d0", B|W|L, {} }
  D1 { 1, "d1", B|W|L, {} }
  ...
%% /* 以下、命令記述 */
...
Reg : Op Reg Reg
  %{  put_asm(Op.op, Reg[2].num, Reg[3].num);
      Reg[1].num = Reg[2].num;
  %}
...
```

Fig.6　機械記述例

8. おわりに

効率的な意味評価や様々な機能の導入を行なったとは言え、未だ実験段階であることは否めない。今後さらなる改良を加え、実用に耐えうるツールにしたいと考えている。

現在、Cope は SONY/NEWS、UNIX 4.3BSD (CPU:68020)、ならびに PC-9801、MS-DOS 3.3 (CPU:80386) 上の C 言語で実現されている。また、この Cope を用いて Pascal コンパイラが記述されている。

参考文献

[1] 藤井則久, 井上謙蔵: 多段階コンパイラ生成系, 情報処理学会第32回(前期)全国大会講演論文集 (pp.553-554) (Mar, 1986)

[2] 清水靖, 井上謙蔵: コンパイラにおけるもう一つの意味評価方法, 情報処理学会第38回(前期)全国大会講演論文集 (pp.911-912) (Mar, 1989)