5P-3

コンパイラにおける もう一つの意味評価方法

清水 靖、井上 謙蔵
〔東京理科大学〕

1．はじめに

現在使用されている手続き型のプログラミング言語では、前方に定義のある手続き名や型名を使用する場合、前もって不自然な宣言（Pascal の forward 宣言、Ada の不完全宣言）を必要としている。これらの宣言は、意味解析を容易にするための構文上に設けられた冗長な宣言であると考えられる。

ポインタを用いる特殊なデータ構造を利用すれば、前方の意味情報を参照することができる。しかし、そのようなデータ構造を利用すると、意味解析処理が複雑になるだけでなく、そのケースに応じたデータ構造を使用しなければならないので一般的ではない。そこで、データ構造の代わりに制御構造を用いることによって、前方の意味情報を参照する意味評価方法を考案した。ここでは、意味評価の制御を中心に報告する。

2．意味評価の方法

意味記述の方法として、属性文法がしばしば使用されている。しかし、意味解析中に広範囲で必要な情報を評価する場合、属性文法は、複数の生成規則中の属性を経由してその情報を評価するので効率が悪い。そこで、ノード変数とプール変数の２種類の変数を用いて意味評価を行なう[1]。

ノード変数とは、属性文法の合成属性の性質を持つ変数である。上昇型の構文解析を用いる場合、ノード変数によって生成規則中の局所的な情報を効率的に評価することができる。

プール変数とは、構文解析中に発生する非端記号を根とする部分木内で有効な変数である。その部分木内では、プール変数の値を直接利用することができる。Fig.1 にプール変数の評価の過程を示す。但し、太字はプログラムテキストである。また、下線部はプール変数を示し、[] はその値を示す。

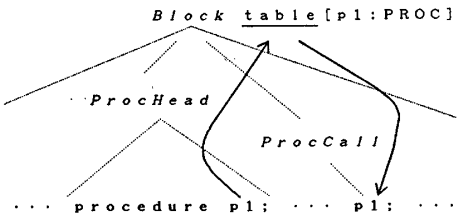

Fig.1 プール変数の評価の過程

プール変数を使用することによって、広範囲で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できる。

3．意味評価の制御

通常、識別子の宣言情報等が記号表（プール変数）に登録された後で、その記号表の情報をもとに意味解析が行なわれる。しかし、前方に宣言のある識別子を使用する場合には、まだ記号表にその識別子の情報が登録されていないので、意味解析を行なうことができない。そこで、Fig.2 のように、プール変数の有効範囲を利用することによって、前方の意味情報を参照する。

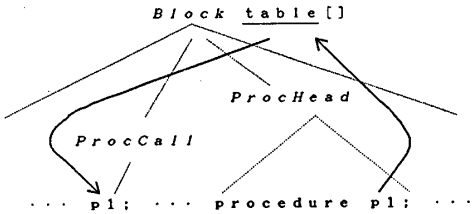

Fig.2 プール変数を利用した意味情報の参照

意味解析部は、生成規則に付けられた意味記述部分を複数の意味関数として実行する。プール変数の有効範囲を利用して、前方の意味情報を参照するので、プール変数をパラメータとして持つ意味関数は、SUCCESS （成功）か FAILURE （失敗）を意味解析部に返す。意味解析部は、FAILURE を返す意味関数を、そのプール変数の関数待ち行列に入れる。そして、そのプール変数が有効範囲から抜け出す直前に、関数待ち行列に入っている意味関数を実行する。

しかし、再帰的なブロック構造を持つプログラミング言語では、同じプール変数がいくつも入れ子状に発生する。すると、現在参照可能なプール変数ではなく、より外側のプール変数に意味情報が登録されている場合がある。この状態では、関数待ち行列に入っている意味関数を実行しても、また、FAILURE を返してしまう。その場合、意味解析部は、一段階外側の同じプール変数の待ち行列にその意味関数を入れる。以後同様に、関数待ち行列に入っている意味関数が FAILURE を返す場合は、一段階外側のプール変数の関数待ち行列に入れる。

Yet Another Semantic Evaluation of Compiler
Yasushi Shimizu, Kenzou Inoue
Science University of Tokyo

Fig.3 に、プール変数と意味解析の過程を示す。但し、太字は、プログラムテキストを示している。また、左側はプール変数の範囲を示し、右側は、意味関数の実行を示している。code は中間コードの順番を制御するプール変数で、table はブロックごとの記号表の役割を果たすプール変数である。また、[] はプール変数の内容を示している。そして、ⓝは意味関数を示し、-ⓝ は関数待ち行列の状態を示している。

①は識別子の情報をプール変数（table）へ登録する意味関数で、③はプール変数（table）から識別子の情報を取り出す意味関数である。また、②と④はそれぞれ中間コードを出力する意味関数である。

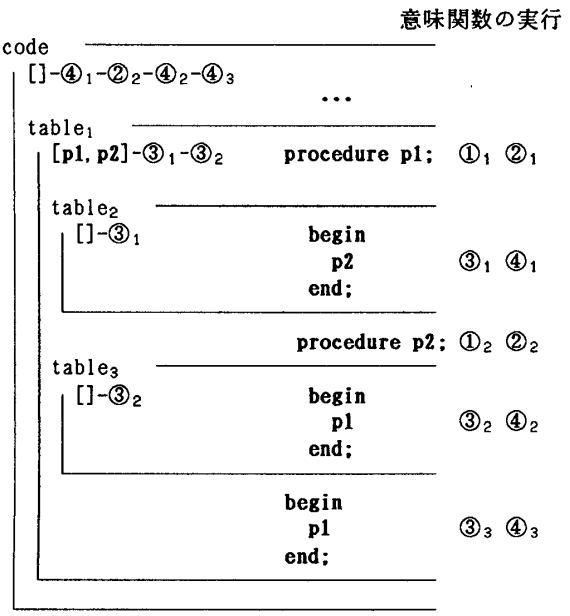

Fig.3 プール変数と意味解析の過程

まず、**procedure p1;** の意味処理で、$①_1$ は、**p1** の情報を $table_1$ に登録する。また、$②_1$ は、**p1** の中間コードを出力する。

次に、**p2** の処理で、$③_1$ は、$table_2$ から **p2** の情報を取り出そうとするが、$table_2$ には **p2** の情報がないので FAILURE を返す。そこで、$③_1$ を $table_2$ の待ち行列に入れる。同様に、**p2** の値が決定していないので、$④_1$ も FAILURE を返す。そこで、$④_1$ を code の待ち行列に入れる。

そして、$table_2$ が有効範囲から抜け出す直前に、再度 $③_1$ を実行するが、このときも FAILURE を返すので、$③_1$ を $table_1$ の待ち行列に入れる。

次に、**procedure p2;** の処理で、$①_2$ は、$table_1$ に **p2** の情報を登録する。また、$②_2$ は、**p2** の手続き開始の中間コードを出力しようとするが、code には、まだ実行できない関数が待ち行列に入っているので、中間コードの出力順序を保つという目的から $②_2$ を code の待ち行列に入れる。

次の $table_3$ 内の処理は、$table_2$ 内の処理と同様なので省略する。

最後の **p1** の処理で、$③_3$ は、$table_1$ から **p1** の情報を取り出す。また、$④_3$ は、code の待ち行列に入れられる。

そして、$table_1$ の有効範囲から抜け出す直前に、$table_1$ の待ち行列に入っている $③_1$ $③_2$ を実行する。この時点では、**p1 p2** の情報が $table_1$ にあるので、問題なく実行される。

意味解析の最後に、code の有効範囲が終わるので、code の待ち行列に入っている $④_1$ $②_2$ $④_2$ $④_3$ を実行する。

この様に、プール変数の有効範囲内で意味関数の動作を制御すると、プール変数を経由して前方の意味情報を利用することができる。Fig.4 に、コンパイラ記述例の一部を示す。但し、太字は意味記述の予約語を示す。

```
nonterminal Program : pool code;
nonterminal Program, Block : pool table; {
  table = init_tab( 50);
  /* プール変数の宣言と初期化 */
}
nonterminal ProcHead, ProcCall : node value;
  /* ノード変数の宣言 */
rule Program -> ProgHead Block;
rule Block -> VarDecPart ProcDecPart StatePart;
  .....
rule ProcHead -> 'procedure' 'id' ';'; {
  ProcHead.value = 'id'.value;
  add_tab( PROC, 'id'.value, table);      ①
  put_code( code, BEGIN, 'id'.value);     ②
}
  .....
rule ProcCall -> 'id'; {
  ProcCall.value = mk_value( 'id'.value);
  ref_id( table, PROC, ProcCall.value);   ③
  put_code( code, CALL, ProcCall.value);  ④
}
```

Fig.4 コンパイラの記述例

4．おわりに

意味関数の動作をプール変数の有効範囲内で制御することによって、前方の意味情報を参照することができる。また、コンパイラ作成者は、意味評価のタイミングを意識しないでコンパイラを記述することができる。現在、このようなコンパイラを自動生成する生成系と、その生成系を用いて forward 宣言を必要としない Pascal コンパイラをそれぞれ作成した。生成系は、SONY/NEWS を使用し、UNIX 4.2BSD 上の C 言語で実現されている。

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)