Vol. 35 No. 4　　情　報　処　理　　Apr. 1994

解　説

属性文法によるコンパイラの記述例†

佐々　政孝††

1. はじめに

本稿では，属性文法に関する連載の一環として，属性文法によるコンパイラ記述の一例について述べる．

属性文法の応用の一つとして属性文法に基づくコンパイラ生成系が種々作成されている[2),5),7),11)]．これらによるコンパイラの記述例は参考文献5),7)などにもあるが，本稿では具体例として，小さな言語 PL/0 のコンパイラを，1パス型属性文法に基づくコンパイラ生成系 Rie（りえ）で記述したものを取り上げ，属性文法によるコンパイラの記述手法の一例を示す．

生成系 Rie による記述を取り上げた理由は，ファイルの入手が容易であること，1パス型属性文法を用いているので効率のよいコンパイラが生成できること，後述のように Rie が UNIX 上のコンパイラ生成系 Yacc の自然な拡張になっていること，である．

紙面の関係ですべてを説明することはできないが，記述の全体像を示すことがたいへん有意義であると考えたので，属性文法による記述のトップレベルの部分の全リストを掲載する．記述の残りの部分（下請関数など）は anonymous ftp で入手できる(**5.** 参照)．

なお，コンパイラと属性文法の予備知識については，それぞれ参考文献1)，10)と，本連載のI，IV，VIや参考文献2)，10)を参照されたい．

2. 1パス型属性文法に基づく生成系 Rie

前述のように，属性文法に基づくコンパイラ生成系が種々作成されている．その中で，Rie[11),12)]は，1パス型属性文法に基づいたコンパイラ生成系の一つである．Rie は，ECLR 属性文法とよばれる属性文法を基礎としている．この属性文法は，LR 構文解析と同時に，解析木を作らずに1パスで属性評価ができるようなクラスの一つである．したがって，Rie により生成されたコンパイラは，LR 構文解析をしながら同時に属性の値を決めていき，コンパイル時の効率がよい．

一方，Rie の記述スタイルは，UNIX 上のコンパイラ生成系 Yacc[4)]（や GNU Bison[3)]，以下同じ）に親しんでいるユーザがたやすく属性文法を利用できるように，これらの記述法の拡張の形とした．また，Rie 自身が Bison を拡張して実現された．

Rie システムは，属性文法に基づくコンパイラの記述（これを Rie 記述という）を入力して，C言語のプログラムを生成する．生成されたC言語のプログラムに字句解析器や下請関数群を加えたものがコンパイラとなる．

3. 属性文法による PL/0 コンパイラの記述

3.1 PL/0 言語

本稿で対象とする PL/0 言語は，小さな言語であるが，手続き型言語の基本的機能をもった言語である[13),10)]．PL/0 は Pascal の小さなサブセットで次のようなものである．

- 型は整数型のみで，配列やレコード，ポインタはない．定数定義はあるが，型定義はない．
- ふつうの算術，関係演算子をもつ．
- if 文，while 文がある．goto 文はない．
- 再帰呼出しのできるパラメタなしの手続きがある．関数はない．
- スコープは，入れ子型静的スコープ規則に従う．

もともとの PL/0 言語には入出力文がなくて不便だったので，本稿では簡単な入出力文 read（変数），write(式)，writeln(式)，writeln を追加した．

† An Example of Compiler Description Using an Attribute Grammar by Masataka SASSA (Dept. of Information Science, Tokyo Institute of Technology).

†† 東京工業大学理学部情報科学科

ソースプログラムの例と実行結果を図-1 に示す．これは参考文献 13) にある例に出力を加えたものであるが，シフト (2*a や b/2) と加算を用いて乗算を行うアルゴリズムを表現したものである．図-1(b) が実行結果で，7×85=595 を示している．

3.2 PL/0 処理系の構成

PL/0 処理系は，参考文献 13) を踏襲してコンパイラ・インタプリタ方式とする．これは，コンパイラがソースプログラムをいったん PL/0 コードというPコードふうの中間コードに落とし，それをインタプリタが解釈実行する方式である．

ここで述べるのは，そのうちのコンパイラ部分である．コンパイラは生成系により生成するが，字句解析器に Lex[8] (または Flex, 以下同じ)，構文・意味解析と中間コード生成に Rie を用いる．記号表処理やコード生成のための下請関数群とインタプリタはC言語で記述する．

3.2.1 PL/0 コード

PL/0 コードは，PL/0 マシンという仮想的スタックマシンのコードである[13),10)]．PL/0 マシンは，演算をスタックトップのところを使って行うのが特徴である．一方，手続き呼出しの実現は，多くの計算機と同様に，スタック領域に各手続き呼出しの活性レコード (activation record) を積んでいく方式である．

図-1(a) のソースプログラムに対する PL/0 コードを図-2 に示す．

PL/0 コードには次のようなものがある．以下で「静的レベル差」とは，ソースプログラムの現在実行中の箇所と参照される識別子の宣言された箇所の静的レベル（字面上の入れ子のスコープの深さ）の差のことである．

```
const m=7,n=85;
var x,y,z;
procedure multiply;
  var a,b;
  begin
    a:=x; b:=y; z:=0;
    while b>0 do
      begin
        if odd b then z:=z+a;
        a:=2*a; b:=b/2;
      end
  end;
begin
  x:=m; y:=n; call multiply;
  write(x); writeln(y); writeln(z)
end.
```

(a) PL/0 ソースプログラム

```
start PL/0
          7          85
        595
end PL/0
```

(b) 実行結果

図-1 PL/0 ソースプログラム例と実行結果

```
PL/0コード                    説明
 1 [ JMP,   0,  31 ]    31番地へ飛ぶ
 2 [ JMP,   0,   3 ]    手続きmultiply:
 3 [ INT,   0,   5 ]      活性レコードを確保
 4 [ LOD,   1,   3 ]      xをロード
 5 [ STO,   0,   3 ]      aにストア
 6 [ LOD,   1,   4 ]      yをロード
 7 [ STO,   0,   4 ]      bにストア
 8 [ LIT,   0,   0 ]      0をロード
 9 [ STO,   1,   5 ]      zにストア
10 [ LOD,   0,   4 ]      bをロード
11 [ LIT,   0,   0 ]      0をロード
12 [ OPR,   0,  12 ]      b>0か？
13 [ JPC,   0,  30 ]      偽なら30番地へ飛ぶ
14 [ LOD,   0,   4 ]      bをロード
15 [ OPR,   0,   6 ]      oddか？
16 [ JPC,   0,  21 ]      偽なら21番地へ飛ぶ
17 [ LOD,   1,   5 ]      zをロード
18 [ LOD,   0,   3 ]      aをロード
19 [ OPR,   0,   2 ]      +
20 [ STO,   1,   5 ]      zへストア
21 [ LIT,   0,   2 ]      2をロード
22 [ LOD,   0,   3 ]      aをロード
23 [ OPR,   0,   4 ]      *
24 [ STO,   0,   3 ]      aへストア
25 [ LOD,   0,   4 ]      bをロード
26 [ LIT,   0,   2 ]      2をロード
27 [ OPR,   0,   5 ]      /
28 [ STO,   0,   4 ]      bへストア
29 [ JMP,   0,  10 ]      10番地へ飛ぶ
30 [ OPR,   0,   0 ]      リターン
31 [ INT,   0,   6 ]    メイン:活性レコードを確保
32 [ LIT,   0,   7 ]      7(m)をロード
33 [ STO,   0,   3 ]      xへストア
34 [ LIT,   0,  85 ]      85(n)をロード
35 [ STO,   0,   4 ]      yへストア
36 [ CAL,   0,   3 ]      multiplyを呼ぶ
37 [ LOD,   0,   3 ]      xをロード
38 [ CSP,   0,   1 ]      write
39 [ LOD,   0,   4 ]      yをロード
40 [ CSP,   0,   1 ]      write
41 [ CSP,   0,   2 ]      改行
42 [ LOD,   0,   5 ]      zをロード
43 [ CSP,   0,   1 ]      write
44 [ CSP,   0,   2 ]      改行
45 [ OPR,   0,   0 ]      リターン (ストップ)
```

図-2 図-1(a)のプログラムに対する PL/0 コード

(1) LOD l,a: 静的レベル差 l, オフセット a の変数の値をロード (スタックにプッシュ).

(2) LIT 0,a: 整数 a をロード.

(3) STO l,a: スタックトップにある値をポップし, 静的レベル差 l, オフセット a の場所にしまう.

(4) OPR 0, a: a で指定した算術・関係演算を行う.

(5) INT 0,a: スタックトップを指すレジスタ t を a だけ増加させる.

(6) JMP 0,a: a 番地へ飛ぶ.

(7) JPC 0,a: スタックトップの値をポップし, それが 0 (偽)なら a 番地へ飛ぶ.

(8) CAL l,a: 静的レベル差 l, コードの先頭番地が a の手続きを呼び出す.

(9) CSP 0,a: a で指定した入出力手続きを呼ぶ. (もとの PL/0 コード[13]にはない)

(10) LAB 0,a: a という整数のラベルを「立てる」(目的コードのこの場所につける)アセンブラ命令. (もとの PL/0 コードにはない)

3.3 Lex による字句解析器の記述

PL/0 の字句解析器は, Lex により生成する. Lex 記述は Yacc と組み合わせて使う場合と同様なので, その一部のみ図-3 に示す. ただし, Lex から Rie へトークンの属性値を渡すところは異なっている. たとえば, 図-3 の 38 行目は, 終端記号 IDENT の合成属性 id に, 読み込まれた識別子の文字列 yytext を下請関数 strsave によりセーブしたものを代入することを表す. rrlval は Yacc の yylval に対応する Rie での大域変数(共用体)で, 字句解析器から意味解析器への属性値の受渡しを行う.

3.4 Rie による構文・意味解析と中間コード生成

この部分は, Rie により生成する (それに C 言語による下請関数群を加える). その Rie 記述を図-4 に示す. この記述は, 参考文献 9), 10)の付録 B, を改訂したものである.

3.4.1 Rie 記述の全体的な構成

Rie 記述は, 一般に 4 つの部分からなる.

(1) **属性の型定義など** (図-4 の 1〜7 行)

ここには, これ以降の記述で用いる属性の型や, 下請関数の型, 定数などの宣言を書く. 図-4 では属性の型の宣言などを記述した env.h と code.h をインクルードしている. なお, この部分は Rie が生成する C 言語によるコンパイラにそのまま組み込まれる.

(2) **文法記号と属性の宣言** (図-4 の 9〜56 行)

ここでは, 文法記号とそれに付随する属性とそ

```
1  %{ /* ---------------------- pl0.l ---------------------------- */
2     /* Jul. 1992. revised by M.Sassa */
3  #include "lex.c"
4  %}
5
6  %%
7  "/*".*"*/"       { ECHO; }
8  begin            { ECHO; return( BEGINN ); }
9  end              { ECHO; return( END ); }
.....
22 "+"              { ECHO; return( PLUS ); }
23 "-"              { ECHO; return( MINUS ); }
.....
38 [a-z][0-9a-z]*   { ECHO; rrlval.rrIDENT.id = strsave( yytext );
39                    return( IDENT ); }
40 [0-9]+           { ECHO; sscanf( yytext,"%d", &rrlval.rrNUMBER.val );
41                    return( NUMBER ); }
42 [ \t]            { ECHO; }
43 "\n"             { ECHO; }
44 .          { fprintf(stdout,"lexical error: '%s' omitted\n",yytext); }
```

図-3 PL/0 の字句解析器の Lex 記述 (一部)

```
%{ /* ------------------------ pl0.rie ------------------------ */
/* Jan. 1994 version. Originally by M. Sassa and H. Ishizuka (Sep. 1984).
   Revised by M. Sawatani, K. Hayashi, T. Hirai and M. Sassa. */
        /* data types of attributes */
#include "env.h"
#include "code.h"
%}

        /* nonterminal symbols and attributes */
%nonterm pl0:
        code:   codeptr         synt;    /* code */
%nonterm block:
        lab:    int             inh,     /* label */
        I_env:  envptr          inh,     /* environment (symbol table) */
        code:   codeptr         synt;
%nonterm constdefpart, constdeflist, constdef:
        I_env:  envptr          inh,
        S_env:  envptr          synt;    /* environment (symbol table) */
%nonterm vardeclpart, vardecllist, vardecl:
        I_env:  envptr          inh,
        S_env:  envptr          synt;
%nonterm procdeclpart, procdecl:
        I_env:  envptr          inh,
        S_env:  envptr          synt,
        code:   codeptr         synt;
%nonterm prochead:
        I_env:  envptr          inh,
        S_env:  envptr          synt,
        lab:    int             synt;
%nonterm statement, statementlist:
        I_env:  envptr          inh,
        code:   codeptr         synt;
%nonterm expression, term, factor, condition:
        I_env:  envptr          inh,
        code:   codeptr         synt;
%nonterm addsub, muldiv:
        ope:    mathope         synt;    /* operation */
%nonterm relop:
        code:   codeptr         synt;
%nonterm number:
        val:    int             synt;    /* value */
%nonterm ident:
        I_env:  envptr          inh,
        ent:    tblptr          synt;    /* symbol table entry */
%nonterm dot1, then1, do1;                    /* dummy nonterminals */
%nonterm semi123, semi4, comma12, eq1, coleq1, rpar1;

        /* terminal symbols and attributes */
%terminal IDENT:
        id:     string          synt;
%terminal NUMBER:
        val:    int             synt;
%terminal CONST, VAR, PROCEDURE, CALL, BEGINN, END,     /* key words */
          IF, THEN, WHILE, DO, ODD, READ, WRITE, WRITELN;
%terminal LPAR, RPAR, COLEQ, DOT, SEMI, COMMA,    /* special symbols */
          EQ, NE, LT, GT, LE, GE, PLUS, MINUS, TIMES, SLASH;

        /* equivalence class of inh. attributes */
%equiv  constdefpart.I_env,     constdeflist.I_env,     constdef.I_env,
        vardeclpart.I_env,      vardecllist.I_env,      vardecl.I_env,
        procdeclpart.I_env,     procdecl.I_env,         prochead.I_env,
        statement.I_env,        statementlist.I_env,
        expression.I_env,       term.I_env,             factor.I_env,
        condition.I_env,        ident.I_env;

%%      /* syntactic and semantic rules */
pl0     : block dot1
          { block.I_env = nullenv();
```

図-4 PL/0 コンパイラの Rie 記述 (続く)

```
69              block.lab = genlabel();
70              pl0.code = linearize( block.code ); } ;
71  block    : constdefpart vardeclpart procdeclpart statement
72              { %thread I_env S_env;
73                %except constdefpart.I_env = newenv( block.I_env );
74                block.code = concat6( gen( O_JMP, 0, block.lab ),
75                                      procdeclpart.code,
76                                      gen( O_LAB, 0, block.lab ),
77                                      gen( O_INT, 0, procdeclpart.S_env->dx ),
78                                      statement.code,
79                                      gen( O_OPR, 0, 0 ));
80                %action printenv( procdeclpart.S_env );
81                %action printcode( block.code ); } ;
82  constdefpart : CONST constdeflist semi123
83              { %thread I_env S_env; }
84            | /* empty */
85              { %thread I_env S_env; } ;
86  constdeflist : constdeflist comma12 constdef
87              { %thread I_env S_env; }
88            | constdef
89              { %thread I_env S_env; }
90            | error
91              { %thread I_env S_env;
92                %message "'const' must be followed by an identifier"; } ;
93  constdef : IDENT eql number
94              { constdef.S_env = enter( CONSTANT, IDENT.id, number.val,
95                                        constdef.I_env );
96                %condition !exist( IDENT.id, constdef.I_env )
97                %message "redeclaration of constant '%s'",IDENT.id; }
98            | IDENT EQ error
99              { %thread I_env S_env;
100               %message "number expected after '='"; } ;
101 vardeclpart : VAR vardecllist semi123
102             { %thread I_env S_env; }
103           | /* empty */
104             { %thread I_env S_env; } ;
105 vardecllist : vardecllist comma12 vardecl
106             { %thread I_env S_env; }
107           | vardecl
108             { %thread I_env S_env; }
109           | error
110             { %thread I_env S_env;
111               %message "'var' must be followed by an identifier"; } ;
112 vardecl : IDENT
113             { vardecl.S_env = enter( VARIABLE, IDENT.id, 0, vardecl.I_env );
114               %condition !exist( IDENT.id, vardecl.I_env )
115               %message "redeclaration of variable '%s'",IDENT.id; } ;
116 procdeclpart : procdeclpart procdecl
117             { %thread I_env S_env;
118               procdeclpart[1].code = concat2( procdeclpart[2].code,
119                                               procdecl.code); }
120           | /* empty */
121             { %thread I_env S_env;
122               procdeclpart.code = nullcode(); } ;
123 procdecl : prochead semi123 block semi123
124             { %thread I_env S_env;
125               block.lab = prochead.lab;
126               %transfer code; } ;
127 prochead : PROCEDURE IDENT
128             { %local int plab = genlabel();
129               prochead.S_env = enter( PROC, IDENT.id, plab, prochead.I_env );
130               prochead.lab = plab;
131               %condition !exist( IDENT.id, prochead.I_env )
132               %message "redeclaration of procedure '%s'",IDENT.id; }
133           | PROCEDURE error
134             { %local int plab = genlabel();
135               %thread I_env S_env;
136               prochead.lab = plab;
```

図-4 PL/0 コンパイラの Rie 記述 (続く)

```
137             %message "identifier expected after 'procedure'"; } ;
138 statement: /* empty */
139           { statement.code = nullcode(); } ;
140 statement: ident coleql expression
141           { %transfer I_env;
142             statement.code = concat2( expression.code,
143                                       gen(O_STO,
144                                           statement.I_env->lev-ident.ent->level,
145                                           ident.ent->adr ));
146             %condition ident.ent != SENTINEL
147             %message "undeclared identifier";
148             %condition ident.ent->kind == VARIABLE
149             %message "assignment to constant/procedure is not allowed"; } ;
150 statement: CALL ident
151           { %transfer I_env;
152             statement.code = gen( O_CAL,
153                                   statement.I_env->lev - ident.ent->level,
154                                   ident.ent->adr );
155             %condition ident.ent != SENTINEL
156             %message "undeclared identifier";
157             %condition ident.ent->kind == PROC
158             %message "call of a constant or a variable is meaningless"; }
159         | CALL error
160           { statement.code = nullcode();
161             %message "'call' must be followed by an identifier"; } ;
162 statement: IF condition then1 statement
163           { %local int ilab = genlabel();
164             %transfer I_env;
165             statement[1].code = concat4( condition.code,
166                                          gen( O_JPC, 0, ilab ),
167                                          statement[2].code,
168                                          gen( O_LAB, 0, ilab )); } ;
169 then1     : THEN
170           | error
171           { %message "'then' expected"; } ;
172 statement: WHILE condition do1 statement
173           { %local int wlab1 = genlabel();
174             %local int wlab2 = genlabel();
175             %transfer I_env;
176             statement[1].code = concat6( gen( O_LAB, 0, wlab1 ),
177                                          condition.code,
178                                          gen( O_JPC, 0, wlab2 ),
179                                          statement[2].code,
180                                          gen( O_JMP, 0, wlab1 ),
181                                          gen( O_LAB, 0, wlab2 )); } ;
182 do1       : DO
183           | error
184           { %message "'do' expected"; } ;
185 statement: WRITE LPAR expression RPAR
186           { %transfer I_env;
187             statement.code = concat2( expression.code, gen(O_CSP,0,WRI)); }
188         | WRITELN LPAR expression RPAR
189           { %transfer I_env;
190             statement.code = concat3( expression.code,
191                                       gen(O_CSP,0,WRI), gen(O_CSP,0,WRL)); }
192         | WRITELN
193           { statement.code = gen(O_CSP,0,WRL); }
194         | READ LPAR ident RPAR
195           { %transfer I_env;
196             statement.code = concat2( gen(O_CSP,0,RDI),
197                                       gen(O_STO,
198                                           statement.I_env->lev-ident.ent->level,
199                                           ident.ent->adr ));
200             %condition ident.ent->kind == VARIABLE
201             %message "read to constant/procedure is not allowed"; } ;
202 statement: BEGINN statementlist END
203           { %transfer I_env, code; } ;
204 statementlist: statementlist semi4 statement
```

図-4 PL/0 コンパイラの Rie 記述 (続く)

```
205           { %transfer I_env;
206             statementlist[1].code = concat2( statementlist[2].code,
207                                              statement.code);  }
208         | statement
209           { %transfer I_env, code; } ;
210 expression: term
211           { %transfer I_env, code; }
212         | PLUS term
213           { %transfer I_env, code; }
214         | MINUS term
215           { %transfer I_env;
216             expression.code = concat2( term.code, gen( O_OPR, 0, P_NEG )); }
217         | expression addsub term
218           { %transfer I_env;
219             expression[1].code = concat3( expression[2].code,
220                                           term.code,
221                                           addsub.ope == ADD
222                                               ? gen( O_OPR, 0, P_ADD )
223                                               : gen( O_OPR, 0, P_SUB )); } ;
224 addsub  : PLUS    { addsub.ope = ADD; }
225         | MINUS   { addsub.ope = SUB; } ;
226 term    : factor
227           { %transfer I_env, code; }
228         | term muldiv factor
229           { %transfer I_env;
230             term[1].code = concat3( term[2].code,
231                                     factor.code,
232                                     muldiv.ope == MUL
233                                         ? gen( O_OPR, 0, P_MUL )
234                                         : gen( O_OPR, 0, P_DIV )); } ;
235 muldiv  : TIMES   { muldiv.ope = MUL; }
236         | SLASH   { muldiv.ope = DIV; } ;
237 factor  : ident
238           { %transfer I_env;
239             factor.code = ident.ent->kind == CONSTANT
240                           ? gen( O_LIT, 0, ident.ent->val )
241                           : gen( O_LOD, factor.I_env->lev-ident.ent->level,
242                                         ident.ent->adr );
243             %condition ident.ent != SENTINEL
244             %message "undeclared identifier";
245             %condition ident.ent->kind != PROC
246             %message "expression must not contain a procedure identifier"; }
247         | number
248           { factor.code = gen( O_LIT, 0, number.val );
249             %condition number.val <= AMAX
250             %message "this number '%d' is too large",number.val; }
251         | LPAR expression rpar1
252           { %transfer I_env, code; } ;
253 condition: ODD expression
254           { %transfer I_env;
255             condition.code = concat2( expression.code, gen(O_OPR,0,P_ODD)); }
256         | expression relop expression
257           { %transfer I_env;
258             condition.code = concat3( expression[1].code,
259                                       expression[2].code, relop.code ); } ;
260 relop   : EQ      { relop.code = gen( O_OPR, 0, P_EQ ); }
261         | NE      { relop.code = gen( O_OPR, 0, P_NE ); }
262         | LT      { relop.code = gen( O_OPR, 0, P_LT ); }
263         | GT      { relop.code = gen( O_OPR, 0, P_GT ); }
264         | LE      { relop.code = gen( O_OPR, 0, P_LE ); }
265         | GE      { relop.code = gen( O_OPR, 0, P_GE ); }
266         | error   { relop.code = nullcode();
267                     %message "relational operator expected"; } ;
268 ident   : IDENT
269           { ident.ent = lookup( IDENT.id, ident.I_env ); } ;
270 number  : NUMBER
271           { number.val = NUMBER.val; } ;
272
```

図-4 PL/0 コンパイラの Rie 記述 (続く)

```
273         /* dummy nonterminals for syntax error handling */
274 dot1     : DOT
275         | error  { %message "period '.' expected"; } ;
276 semi123 : SEMI
277         | error  { %message "semicolon ';' missing"; }  ;
278 semi4    : SEMI
279         | error  { %message "semicolon ';' between statements is missing"; } ;
280 comma12 : COMMA
281         | /* empty */  { %message "comma ',' missing"; } ;
282 eq1      : EQ
283         | COLEQ  { %message "use '=' instead of ':='"; }
284         | error  { %message "identifier must be followed by '='"; } ;
285 coleq1   : COLEQ
286         | EQ     { %message "use ':=' instead of '='"; } ;
287 rpar1    : RPAR
288         | error  { %message "right parenthesis ')' missing"; }  ;
289
290 %%      /* user defined functions */
291         /* main program */
292 static int err = 0;
293 extern FILE *yyin, *yyout;
294 static FILE *yyerr;
295
296 yyerror( s )
297   char *s;
298 {
299     fprintf( yyerr, "%s\n", s );
300 }
301
302 main( argc, argv )
303   int argc;
304   char *argv[];
305 {
306     yyin = stdin;  yyout = stdout;  yyerr = stdout;  rrmessage = stdout;
307     switch( argc ) {
308     case 1:
309         break;
310     case 3:
311         if(( yyout = fopen( argv[2], "w" )) == NULL ) {
312             fprintf( stderr, "%s: cannot open `%s'.\n", argv[0], argv[2] );
313             exit( 1 );
314         }
315     case 2:
316         if(( yyin = fopen( argv[1], "r" )) == NULL ) {
317             fprintf( stderr, "%s: cannot open `%s'.\n", argv[0], argv[1] );
318             exit( 1 );
319         }
320         break;
321     default:
322         fprintf( stderr, "usage: %s [infile] [outfile]\n", argv[0] );
323         exit( 1 );
324     }
325     yyparse();
326     if( err == 0 ) {
327         interpreter();
328     } else {
329         fprintf( yyout, "errors in PL/0 program\n" );
330     }
331     if( yyin != stdin ) fclose( yyin );
332     if( yyout != stdout ) fclose( yyout );
333     exit( 0 );
334 }
```

図-4 PL/0 コンパイラの Rie 記述（終わり）

の型の宣言を行う.(属性には所与のものがあるのではなく，コンパイラ記述者が自由に決める.) **%nonterm** は非終端記号,**%terminal**は終端記号, **synt** は合成属性, **inh** は継承属性を表す. たとえば 10〜11 行は，pl0 という非終端記号が code というcodeptr型の合成属性をもつことを宣

言している．59〜64 行の **%equiv** は，ECLR 属性文法の特徴の一つである属性の同値類を宣言するものであるが，これは効率向上のための記述であり，意味内容には直接関係しないので，説明を省く．

（3） 生成規則と意味規則（図-4 の 66〜288 行）

ここは属性文法の本体であり，生成規則とそれに対応する意味規則を記述する．意味規則は { } で囲んで表す．たとえば，67 行目の

```
pl0: block dot1
```

は生成規則で，68〜70 行目が対応する意味規則である．70 行目の

```
pl0.code = linearize( block. code );
```

は，文法記号 pl0 の属性 code が，文法記号 block の属性 code をパラメタとする関数呼出し linearize の結果の値として決まることを記述している．生成規則中の「|」は「または」を表す．

この属性文法の本体部分の記述内容については，**3.4.2** で詳述する．

（4） C言語のプログラム（図-4 の 290 行〜最後）

ここには，メインプログラムなどをC言語のプログラムにより記述する．この例では 325 行で構文解析・属性評価器 yyparse を呼び，誤りがなければ 327 行でインタプリタを呼んでいる．

この最後の部分には，字句解析器や下請関数群を直接記述してもよい．しかしこの例では，図-3 の Lex 記述から生成された字句解析器や下請関数群と一緒にリンクするようにして，それらは記述しなかった．

3.4.2 1パス型属性文法による記述の要点

図-4 の Rie 記述を例に，1パス型属性文法によるコンパイラ記述の方針と要点について述べる．主な属性として，記号表およびコードを用いて記述した．

（1） 記号表属性

記号表を表す属性として，継承属性 I_env と合成属性 S_env (env は環境 environment の意) を用いた．

記号表属性の流れを**図-5** に示す．block に入った直後はスコープの入れ子が深くなるので，レベルを一つ上げる操作（関数 newenv）を行う．点線の部分では定義や宣言が追加され，実線の部分

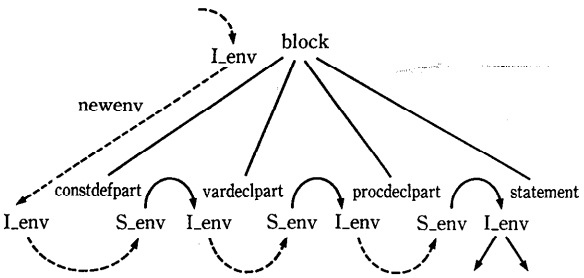


図-5 記号表属性の流れ

では記号表がコピーされる．実行文 statement から下では記号表はたんに上から下へコピーされる．

図-4 の Rie 記述に戻ろう．PL/0 プログラムの処理の先頭では記号表を空とする（68 行目の nullenv( )）．

Rie では図-5 のような「縫い糸」のような属性の渡し方を略記するために **%thread** という記法を設けている (72, 83, 85 行など)．ただし **%except** 指定があるとそちらが優先する．たとえば 72〜73 行の記述は次と同等である．

```
constdefpart.I_env=newenv (block.I_env);
vardeclpart.I_env=constdefpart.S_env;
procdeclpart.I_env=vardeclpart.S_env;
statement.I_env=procdeclpart.S_env;
```

また，宣言があると記号表に追加する（定数定義では 94 行目，変数宣言では 113 行目，手続き宣言では 129 行目の関数 enter)．

141 行目の **%transfer** は属性のコピーの略記法で，「expression.I_env=statement.I_env;」と同じである．141, 151, 164 行目などにより，statement から下では記号表が親から子へコピーされる．

%condition... **%message**... は意味誤りの処理のための記述で，満たすべき文脈条件とそれに違反したときのメッセージを表す．たとえば 96〜97 行目は，定数の二重定義をチェックしている．

（2） コード属性

コードを表す属性として合成属性 code を用いた．これは，命令の線形リストとして実現した．ただし飛越命令の前方参照 (forward reference, あとで決まる番地などを参照すること) をどう扱うかは問題である．手書きの1パスコンパイラでは前方参照をバックパッチング (backpatching, 後埋め) で処理することが多い．しかし属性文法で

は，前方参照は属性の依存関係に右文脈依存性(右から左への依存関係が存在すること）があることとなり，1パス属性文法では表しにくい．

そこで，Rie 記述では本体でのバックパッチングをやめ，目的コードに **3.2.1** の(10)で導入したラベルを立てるアセンブラ命令「LAB 0, a」を用いることとした．これに対応して，命令「JMP 0, a」，「JPC 0, a」，「CAL l, a」の a もさし当りラベルとしておく．以後の (2.1)，(2.2)，(2.3) 項では必要に応じてラベルを立てる．

これらのラベルは，最後にまとめてC言語の関数 (70 行目の関数linearize)の中でバックパッチングとアセンブルを行って，ラベルを本当の番地に直す．この結果，最終的な目的コードでは LAB 命令は除去され，命令「JMP 0, a」，「JPC 0, a」，「CAL l, a」の a は番地となる．前述の図-2 は，これを行った結果の最終的な目的コードである．

(2.1) block (71〜81 行) のコード生成

block の構文は，71 行目にある．block 中に手続き宣言部 procdeclpart があるかもしれないので，block のコードは手続き宣言部を飛び越す次のようなものとする．

JMP 0, l	(a)
procdeclpart のコード	(b)
LAB 0, l	
INT 0, 活性レコードの大きさ	(c)
statement のコード	(d)
OPR 0, 0 (return を表す)	(e)

たとえば(a)，(b)，(c)，(d)，(e)は，それぞれ図-2（ラベルを除いた最終的な目的コード）の 1，2〜30，31，32〜44，45 行に対応する．

このため，上のコードのラベル l に対応する継承属性 block. lab を設けることにした．block. lab の値は，block が構文規則の右辺に現れる所で，呼ばれるたびに異なるラベルを発生する関数 genlabel を呼んで設定する (69 行目)．それを用いて，上のようなコードを 74〜79 行で生成する (concat 6 (…)は 6 個のコードの連接を返す)．手続き内の block については次の(2.2)項で述べる．

(2.2) 手続き (116〜137 行，150〜161 行) のコード生成

手続きは入れ子になっているかもしれない．そこで，入れ子の手続き宣言と手続き呼出しを含むソースプログラムの例と，その目的コードを**図-6** に示す．（詳しく言うと手続きのコードでは(カ)や(キ)の命令は不要であるが，これは block のコードをそのまま流用したためである．）

1パスでコードを生成するには，(ウ)や(エ)で手続き A の入口(ク)が分からなければならない．このために，各手続きの頭の部分(ア)や(イ)でそれぞれ l_A や l_B のラベルを発生することとし，かつ自分と同じか自分より内側の block でそれが参照可能になるように記号表に登録する．手続きは記号表に

(PROC, 手続き名，入口のラベル)

の形で登録する．そしてその手続きの block 中で(ク)のようなラベルを立てる．

具体的な Rie 記述では，prochead の部分 (127 行〜) でその手続きの入口のラベル(図-6 の手続き A では l_A に当たる) を発生し (128 行目の genlabel)，整数型の局所属性 plab に保持する．(局所属性とは，その生成規則内だけで局所的に有効な属性で **%local** により定義する)．それを記号表に登録する(129 行)とともに，prochead. lab に与える(130 行)．そして，そのラベル prochead. lab を手続きの内側の block の block. lab へ渡し (125 行)，この block 中でラベルを立てる (76 行)．

一方，手続き呼出し (150 行〜) では呼び出す手続き ident に対する記号表のエントリ（登録場所）が ident.ent に得られているので，その番地へ飛ぶコードを生成する (152〜154 行)．

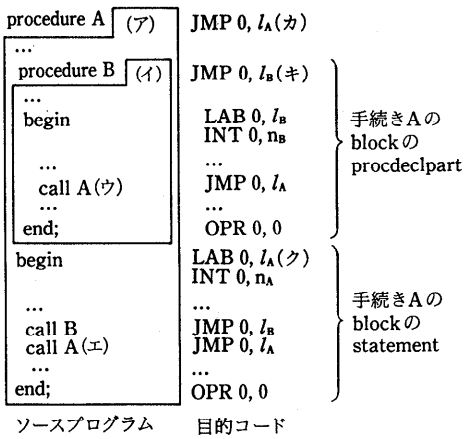


図-6 手続き宣言と呼出し

(2.3) if 文 (162～171 行) のコード生成

PL/0 言語での if 文には else 節がない．if 文のコードは，

condition のコード　　(a)
JPC 0, ilab　　(b)
statement のコード　　(c)
LAB 0, ilab

とする．たとえば図-2 での 14～15，16，17～20 行が(a)，(b)，(c)に対応する．

図-4 の 165～168 行は上のようなコードの生成を表している．163 行目は上記のラベル ilab を genlabel により発生する．

while 文 (172～184 行) のコードも同様である．

(2.4) 式 (210～225 行) のコード生成

たとえば 217 行目の生成規則

expression : expression addsub term

に対するコードは，PL/0 マシンがスタックマシンなので後置コードの形で，

(生成規則の右辺の)expression のコード
term のコード
OPR 0,P_ADD か ORP 0,P_SUB

とする (219～223 行)．(219 行目の[1]，[2]は，一つの生成規則に同じ文法記号が何回か現れたとき，それらを区別するためにその文法記号に左から順に番号をふったものである．)

term (226 行～)，factor (237 行～)，condition(253 行～) のコードも同様である．

定数や変数の識別子 ident については，記号表 ident.I_env 中でその識別子を lookup により探し，記号表の対応するエントリを ident.ent に得る (269 行)．そして，それを利用してコードを生成する (239～242 行)．

3.4.3 誤り処理

構文誤り処理については，Yacc (Bison) と同じ error トークン[4]を利用する．たとえば図-4 の 67 行目の dot1 は本来は DOT であるが，ソースプログラムに DOT が来ないときの誤り処理のため，274～275 行のように生成規則の右辺に error トークンを用いた非終端記号とした．45～46 行に記述した非終端記号はすべて同様の目的のために導入したものである．結果として，もとの PL/0 コンパイラ[13]と同程度の構文誤り検出を行うことができた[9]．

一方，意味誤りの処理は前述の %condition を用いて記述する．

4. 属性文法によるコンパイラ記述の評価

3. を振り返りつつ，属性文法によるコンパイラの記述について議論しよう．

(1) 記述量．PL/0 処理系（インタプリタを含む）の Rie 記述は 334 行（うち属性の宣言など 56 行，C言語部分 52 行)，Lex 記述は 44 行，C言語関数などは 587 行，計 965 行であった．参考文献13)にある Pascal で書かれた処理系(ただし read, write などなし，読みやすさのための空行もなし）が約 450 行であったことを考えると，量的には長い．しかし，読解性，形式性，保守性は格段に良くなった．

(2) コンパイル時間．Rie が生成した PL/0 処理系のコンパイル時間は，手書きのものの約 1.8 倍遅い[12]．Yacc と Lex を用いるとコンパイル時間が約 1.9 倍遅くなるというデータもあるので[6]，形式的な記述から生成されたコンパイラとしてはこれは許容範囲であろう．

(3) 属性文法の特徴と言われている点．属性文法は，構文ごとに局所的に記述，理解できる，と言われている．しかし，**3.** での説明でもそうであったが，記号表の扱いなどは全体的な流れを決めないと記述に取り掛かれない．また，属性文法では，属性は一回だけ評価され，その後その値は変わらず(単一代入性)，副作用はないという．しかし，前述の Rie 記述では，呼ばれるたびに異なるラベルを返す関数 genlabel を方々で用いた．これは一種の副作用であるが，この程度のたちの良い副作用は認めざるをえない．

(4) 1パス型属性文法について．1パス型では当然であるが，右文脈依存の属性が使えない．図-4 の Rie 記述で前方参照の処理をC言語の関数に任せたのはその帰結である．これを避けるには，もうすこし広いクラスの属性文法を用いることが考えられるが，たとえば GAG[5] が生成するコンパイラは手書きの 5 倍くらい遅いと言われるので，記述性とコンパイル時間とのトレードオフとなる．

5. おわりに

属性文法によるコンパイラ記述の実例を示し，記述手法や実際上の問題について論じた．全体と

して，属性文法を用いることで，読解性，形式性が良く，実用的な効率をもったコンパイラが作成できることを実感していただければ幸いである．

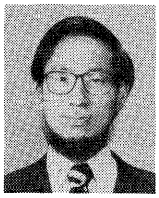
なお，Rie は GNU 規約に基づき配布されており，Rie 自身と PL/0 処理系を含む Rie 記述の例は ftp. is. titech. ac. jp : pub/Rie から anonymous ftp で入手可能である．

謝辞 Rie と PL/0 記述に多大な貢献をした石塚治志，黒石和宏，平井輝久の各氏に感謝する．

参考文献

1) Aho, A. V., Sethi, R. and Ullman, J. D. : *Compilers—Principles, Techniques, and Tools*, Addison-Wesley (1986). (邦訳) 原田：コンパイラ，サイエンス社.
2) Deransart, P. et al. : *Attribute Grammars: Definitions, Systems and Bibliography*, Lec. Notes in Comp. Sci., Vol. 323, Springer (1988).
3) Donnelly, C. and Stallman, R. : Bison Reference Manual, 1990-.
4) Johnson, S. C. : Yacc—Yet Another Compiler Compiler, Computer Science Tech. Rep. 32, AT &T Bell Lab. (1975).
5) Kastens, U., Hutt, B. and Zimmermann, E. : *GAG: A Practical Compiler Generator*, Lec. Notes in Comp. Sci., Vol. 141, Springer (1982).
6) 木村，武市：コンパイラ構成法の比較，情報処理学会第 27 回大会，6 E-9 (1983).
7) Koskimies, K. and Paakki, J. : *Automating Language Implementation—a Pragmatic Approach*, Ellis Horwood (1990).
8) Lesk, M. E. : Lex—A Lexical Analyzer Generator, Computer Science Tech. Rep. 39, AT & T Bell Lab. (1975).
9) 佐々，石塚：ECLR 属性文法による PL/0 コンパイラの記述，Tech. Memo PL-2, 筑波大学プログラミング言語研究室 (1984).
10) 佐々政孝：プログラミング言語処理系，岩波書店 (1989).
11) Sassa, M., Ishizuka, H., Sawatani, M. and Nakata, I. : *Rie—Introduction and User's Manual*, Tech. Rep. ISE-TR-90-82, Inst. of Inf. Sci., Univ. of Tsukuba (1990). 最新版は ftp で入手されたい.
12) 佐々，石塚，中田：1 パス型属性文法に基づくコンパイラ生成系 Rie, コンピュータソフトウェア, Vol. 10, No. 3, pp. 20-36 (1993).
13) Wirth, N. : *Algorithms+Data Structures=Programs*, Prentice-Hall (1976). (邦訳) 片山：アルゴリズム＋データ構造＝プログラム，日本コンピュータ協会.

(平成 6 年 2 月 1 日受付)



佐々 政孝 (正会員)

1948 年兵庫県西宮生．1970 年東京大学理学部物理学科卒業．1974 年同大学院理学系研究科博士課程中退．東京工業大学理学部情報科学科助手．1981 年筑波大学電子・情報工学系，1992 年東京工業大学理学部情報科学科教授．理学博士．1987～1988 年ヘルシンキ大学客員研究員．プログラミング言語，属性文法，コンパイラ生成系，プログラミング環境に興味をもっている．著書「プログラミング言語処理系」(岩波書店) など．日本ソフトウェア科学会，ACM, IEEE 各会員.