B5　問題向き言語に対する意味論的メタ言語とそのコンパイラ

渡辺　坦（日立製作所中央研究所）

1. まえがき

人間は込入った事柄を伝達する場合，まず全体の概観を説明し，ついで細部の説明に移ることが多い．また，正確な伝達が要求される場合は，話が複雑になってくると，その場に応じた種々の記号や専門用語を導入して，説明をコンパクトにする．プログラムは人間の意図を計算機に伝えるための記号列といえる．そこで，プログラミング言語に記号の意味を説明する機能を持たせるならば，問題の種類に応じて，直感的にわかりやすい言葉やその分野の専門用語を導入して，プログラムを人間にわかりやすい形に書くことができる．この行き方をとると，専門用語の定義パッケージを取りかえることにより，一つの基本となる言語から，さまざまの問題向き言語を派生させ得ることになる．

すでに文献1で，プログラミングの段階で次々と新しい用語を導入してゆき，人間の思考形態に即して，大局から細部へとプログラミングを進めるという方法について論じた．ここでは，それを実用化するために，1ページの紙面に図示できる単純な「基本構文」で定められる言語を提示し，そのコンパイラの作用を明確に定める．この基本構文は，あとで図2に例示するように，多種多様な，人間にとってわかりやすい表現形式を許すものであり，多くの問題向き言語をこれに沿って作ることができる．その場合，基本構文に対するコンパイラは，それらの問題向き言語に対する共通コンパイラとなる．

以下では，プログラム形式を例示したあと，まず，プログラムの標準形式とコンパイラの作用を定め，ついで，プログラムを見やすく書きやすくするように表現形式を変える規則を定めて，基本構文を作る．

2. プログラム例

人間が歩くのを図1のように模式的に図示するプログラムを考えよう．この言語の形式に従

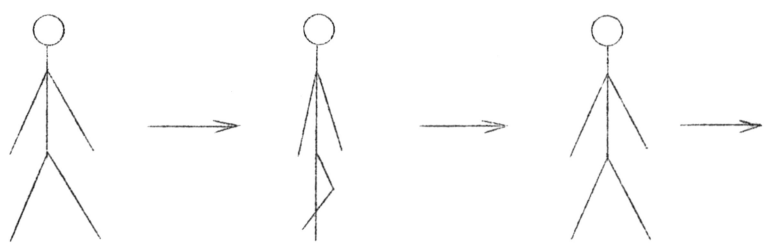


図1.　例題のプログラムが出力する図

うと，そのプログラムは図2のようになる．使われている記号のうち，次のものはその意味が

```
DE(WALK
      INITIATION
      LET OX=0.0
    L SET POSITION (OX,0.0)
      DRAW FIGURE
      LET OX=OX+DX
      IF OX LT 50.0 THEN GO TO L

MS(INITIATION
      LINE FIGURE FIG1
      LINE FIGURE FIG2
      LET FIG1.NP=9
      LET FIG2.NP=8
      LET FIG1.ELEM= 0.0 7.0 0  0.0 6.5 1 -1.0 3.0 1  1.0 3.0 0
1         0.0 6.5 1  0.0 4.5 1 -1.0 0.0 1  1.0 0.0 0  0.0 4.5 1
      LET FIG2.ELEM= 1.0 0.0 0  1.0 7.0 1  0.5 3.0 0  1.0 6.5 1
1         1.5 3.0 1  1.0 4.5 0  1.5 2.5 1  0.5 0.5 1
      LET DX=2.0
    )
MS(LINE FIGURE XVAR XN
      DECLARE(XVAR.NP,INTEGER)
      DECLARE(XVAR.ELEM,ARRAY(3,XN))
    )
MS(DRAW FIGURE
      DRAW CIRCLE (0.0,7.5.0.5)
      DRAW LINEFIG FIG1
      DRAW CIRCLE (DX/2.0,7.5,0.5)
      DRAW LINEFIG2
    )
DE(DRAW LINEFIG XFIG
      LINE FIGURE XFIG
      FOR IP=1 TO XFIG.NP REPEAT
1       MOVE TO (XFIG.ELEM(1,IP),XFIG.ELEM(2,IP),XFIG.ELEM(3,IP))
    )
ME(FOR XP=X1 TO X2 REPEAT XE
      LET XP=X1
    L1 IF XP GT X2 THEN GO TO L2
       XE
       LET XP=XP+1
       GO TO L1
    L2 NULL
    )
MS(LET XVAR=XVAL
      ASSIGN(XVAR,VALUE(XVAL))
    )
MS(IF BOOL THEN XE
      SELECT(XBOOL,XE,NULL)
    )
MS(GO TO XL
      JUMP(XL)
    )
    )
```

図2　プログラム例

既知であって，プログラム内で説明する必要のない記号であるとする．

DE (…), DS (…) ME (…), MS (…)	WALK, INITIATION などの記号の表わす内容を定義することを意味する．
DECLARE	記号の属性を宣言する．
SET POSITION	相対座標系の原点を定める．
DRAW CIRCLE	円を描く（中心の座標は相対座標系に対するもの）．
MOVE TO	指定された点へ，第3引数が1なら線を書きながら，0なら線を書かずに，移動する（点の座標は相対座標系に対するもの）．
ASSIGN (v, e)	v の値を e とする．
VALUE (e)	e の値をとりだす．
SELECT (e_0, e_1, e_2)	e_0 が真なら e_1 を，偽なら e_2 を選ぶ．
JUMP (e)	e で指定されたところへ飛ぶ．
GT, LT	大小を比較する．
NULL	空な記号列を表わす．

カラム1にある文字1は，その行が前の行の続きであることを示す．

3. プログラムの標準形

この言語では，プログラムは「式」で構成され，式の形は

定数　1，3.14，−1.0E6，'ABC'，　etc.

変数　x，$x(e_1, \cdots, e_n)$，$v_1, v_2, \cdots, v_n$

関数　f，$f(e_1, \cdots, e_n)$

リスト　$(e_1, \cdots, e_n)$

ブロック　$(l_1 : e_1; \cdots; l_n : e_n;)$

の5種類に限られる．ここで，x，f，および l_1，…，l_n は標識（英字で始まる英数字列）であり，それぞれ，変数名，関数名，およびラベルと呼ばれる．v_1，…，v_n，および，e_1，…，e_n は，それぞれ，変数および一般の式を表わす．以下，とくに断わらなければ c，c_0，c_1，… で定数を表わし，x，x_0，x_1，… で変数名を，f，f_0，f_1，… で関数名を，e，e_0，e_1，… で一般の式を，l，l_0，l_1，… でラベルを表わす．また，s，s_0，s_1，… でそれぞれ $l : e$，$l_0 : e_0$，$l_1 : e_1$，… 形の記号を表わす．さらに，$x(e_1, \cdots, e_n)$，$f(e_1, \cdots, e_n)$ などと書いたときには，これらはそれぞれ（引数の省略された）x，f などの形の式も代表しており，$l : e;$ と書いたときには，これは（ラベルのつかない）$e;$ 形の式も代表している．　これらの式の計算機による内部表現は，図3のようにする．

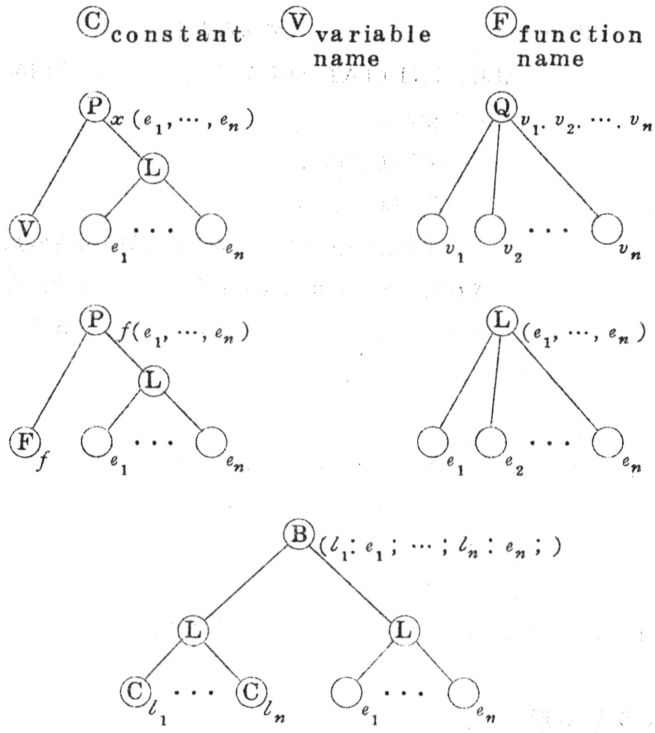


図3. 式のtree表現

式は，その意味があらかじめ定められている「基底式」と，プログラム内でその式の持つ意味が定義されるものとに分かれる．基底式には最小限以下の式を含める．

定数	（定数には，TRUE，FALSE，UNKNOWN，NULLも含める．）
DEFINE (e_0, $f(x_1, \cdots, x_n)$, e)	式 e を $f(x_1, \cdots, x_n)$ と表現することを定義する．e_0 はEXPRESSIONまたはSTATEMENTという記号であり，後者の場合，第3項 e はブロック形の式でなければならない．
VALUE (e)	式 e の値を求める．
ASSIGN (v, e)	変数 v の値を e とする．
EQUAL (e_1, e_2)	e_1 と e_2 が等しいか否かを判定する．
SELECT (e_0, e_1, e_2)	e_0 が真ならば e_1 を選び，偽ならば e_2 を選ぶ．
JUMP (e)	e で示される場所へ飛ぶ．
RETURN (e)	ブロック形の式に対して，その値を e とする．
DECLARE (e_0, e)	e_0 が e という属性リストを持つことを宣言する．
ALIST (e_0)	e_0 の属性リストをとり出す．

これらの基底式の持つ意味は，あとでトランスレータとインタープリタを説明するとき明確に

なる．基底式はこれ以外にも必要に応じて追加してよい．たとえば次の式などを含めてよい．

$+(e_1, e_2)$ $-(e_1, e_2)$ $*(e_1, e_2)$ $/(e_1, e_2)$	VALUE (e_1) と VALUE (e_2) との間の加減乗除演算
GT (e_1, e_2)	VALUE (e_1) ＞ VALUE (e_2) の判定
LT (e_1, e_2)	VALUE (e_1) ＜ VALUE (e_2) の判定
QUOTE (e)	e そのもの．
EXPTYPE (e)	式 e のタイプ
COMBINE (e_0, e_1, e_2)	e_0 で指定されたタイプに合わせて，式 e_1 と e_2 を結合する．
SUBEXP (e_0, e)	e の第 e_0 番目の部分式（次項参照）をとりだす．

式 e の「直接の部分式」とは，e が $x(e_1, \cdots, e_n)$，$f(e_1, \cdots, e_n)$，$(e_1, \cdots, e_n)$，$(l_1 : e_1 ; \cdots ; l_n : e_n ;)$ のいずれかの形であれば $e_1, \cdots, e_n$ のおのおのを指し，$v_1 . v_2 . \cdots . v_n$ の形であれば各 $v_1, v_2, \cdots, v_n$ のことを指す．式 e の「部分式」 $Subexp(e)$ といえば，e の直接の部分式 e_i のほかに，e_i の部分式，および e 自身も含める．

$$Subexp(e) = \{ e \} \cup \bigcup \{ Subexp(e_i) \mid e_i : e \text{ の直接の部分式} \}$$

一つの式に含まれる部分式に対して，次の規則による順序づけを行なう．

(1) e' が e の部分式であれば，e' は e の先行式である．

(2) $x(e_1, \cdots, e_n)$，$f(e_1, \cdots, e_n)$，または $(e_1, \cdots, e_n)$ において，$2 \leqq i \leqq n$ なる i をとるとき，$e_1, \cdots, e_{i-1}$ の各々は e_i の先行式である．

(3) 式 $(l_1 : e_1 ; \cdots ; l_n : e_n ;)$ において，$2 \leqq i \leqq n$ なる i に対して $l_1, \cdots, l_{i-1}$ が省略されていて，この式が $(e_1 ; \cdots ; e_{i-1} ; l_i : e_i ; \cdots ; l_n : e_n ;)$ という形であるならば，$e_1, \cdots, e_{i-1}$ の各々は，$e_i, \cdots, e_n$ の各々に対する先行式である．このとき，$l_i, \cdots, l_n$ の中にもさらに省略されているものがあってもよい．

e' が e の先行式のとき，e を e' の後続式という．

4. トランスレータ

トランスレータ σ は，DEFINE式によって定義された式をその定義内容でおきかえることによって，式を基底式に変える写像であり，通常のアセンブラにおけるマクロ命令の展開に相当する．σ の説明に移る前に 2，3 の用語を準備する．

いま，記号列 p に含まれる標識 $x_1, \cdots, x_n$ をそれぞれ同時に記号列 $a_1, \cdots, a_n$ でおきかえてできる記号列を $Subst \begin{matrix} x_1, \cdots, x_n \\ a_1, \cdots, a_n \end{matrix} (p)$ と表わすことにする．また，式 p の中に，あるいは式 p を部分式として含む式 q の中に，DEFINE (e_0, $f(x_1, \cdots, x_n)$, e) が直接の部分式として含まれているとき，式 $f(e_1, \cdots, e_n)$ を p における「導入された式」と呼ぶ．そして，e_0 がEXPRESSIONであるとき，e に含まれる $x_1, \cdots, x_n$ にそれぞれ $e_1, \cdots, e_n$ を代入した

$Subst\ {}^{x_1,\cdots,x_n}_{e_1,\cdots,e_n}(e)$ を $Text(f(e_1,\cdots,e_n))$ と表わす．e_0 がSTATEMENTであって，$Subst\ {}^{x_1,\cdots,x_n}_{e_1,\cdots,e_n}(e)=(s_1;\cdots;s_k;)$ であるときは，両端のかっこをはずした記号列 $s_1;\cdots;s_k$ を $Text(f(e_1,\cdots,e_n))$ と表わすことにする（この場合，f は必ずブロック形の式の中に，$(\cdots;\ell:f(e_1,\cdots,e_n);\cdots)$ という形で現われているものとする）．

a 形または $a(e_1,\cdots,e_n)$ 形の式で a が標識のとき，これが基底式でも導入された式でもなければ，それを変数と呼ぶ．

ではここで，一般の式に対するトランスレータ σ の作用を次のように定める．

$$\sigma(c) = c$$

$$\sigma(x(e_1,\cdots,e_n)) = x(\sigma(e_1),\cdots,\sigma(e_n))$$

$$\sigma(v_1.v_2.\cdots.v_n) = \sigma(v_1).\sigma(v_2).\cdots.\sigma(v_n)$$

$$\sigma(f(e_1,\cdots,e_n)) = \begin{cases} \sigma(Text(f(\sigma(e_1),\cdots,\sigma(e_n)))) & f\text{が導入された式のとき} \\ f(\sigma(e_1),\cdots,\sigma(e_n)) & f\text{が基底式のとき} \end{cases}$$

$$\sigma((e_1,\cdots,e_n)) = (\sigma(e_1),\cdots,\sigma(e_n))$$

$$\sigma((\ell_1:e_1;\cdots;\ell_n:e_n;)) = (\ell_1:\sigma(e_1);\cdots;\ell_n:\sigma(e_n);)$$

ただし，$f(e_1,\cdots,e_n)$ がrecursiveに定義された関数の場合には，それに対する σ の作用を基底式に対する作用と同じにする．

この変換規則に従って式をトランスレートすると，結果としてできる式は，定数，変数，基底式，および，recursiveに定義された式のみから構成される．

5. インタープリタ

インタープリタ π は，σ でトランスレートした結果に対して作用させる写像であり，一般に，

$$\pi(c) = c$$

$$\pi(x(e_1,\cdots,e_n)) = x(\pi(e_1),\cdots,\pi(e_n))$$

$$\pi(v_1.v_2.\cdots.v_n) = \pi(v_1).\pi(v_2).\cdots.\pi(v_n)$$

$$\pi(f(e_1,\cdots,e_n)) = f(\pi(e_1),\cdots,\pi(e_n))$$

$$\pi((e_1,\cdots,e_n)) = (\pi(e_1),\cdots,\pi(e_n))$$

となる．ブロックに対する π の作用についてはあとで説明する．しかし，関数形の基底式に対しては，必ずしもこの規則通りにならず，インタープリトすると結果式が簡略化されることも多い．以下，基底式と π との関係を個別に説明する．

(1) $\pi(\mathrm{DEFINE}(e_0, f(x_1,\cdots,x_n), e)) = \mathrm{DEFINE}(e_0, f(x_1,\cdots,x_n), \pi(e))$

(2) $\mathrm{VALUE}(e)$

式には，「値」と呼ばれる式が対応していることがあり，VALUEはそれをとり出す働きをする．式の値は，定数では固定されているが，変数ではASSIGN式によって定められ，ブロ

ックではRETURN式によって定められる．

π(VALUE (c)) $= c$

π(VALUE (v)) $= \pi(v)$の値

π(VALUE ($f(e_1, \cdots, e_n)$)) $= \pi(f(e_1, \cdots, e_n))$の値

π(VALUE (($e_1, \cdots, e_n$)))

$= (\pi(\text{VALUE}(e_1)), \cdots, \pi(\text{VALUE}(e_n)))$

π(VALUE ((s_1; $\cdots$; s_n ;)))

$= \pi((s_1; \cdots ; s_n ;))$ 内のRETURN式によって定められる式

値の定められていない式eに対しては，π(VALUE (e)) ＝UNKNOWN である．

(3) π(ASSIGN (v, e)) = ASSIGN ($\pi(v), \pi(e)$)；

これは，変数$\pi(v)$の値を$\pi(e)$とおく働きをする．

(4) EQUAL (e_1, e_2)

$\pi(e_1)$と$\pi(e_2)$がいずれもUNKNOWNでない場合は，$\pi(e_1)$と$\pi(e_2)$を記号列として比較してみて，互いに等しいか否かにしたがって，π(EQUAL (e_1, e_2))をTRUEかFALSEとする．$\pi(e_1)$と$\pi(e_2)$の一方または双方がUNKNOWNであれば，π(EQUAL (e_1, e_2)) = EQUAL ($\pi(e_1), \pi(e_2)$) とする．

(5) SELECT (e_0, e_1, e_2)

πを作用させると，π(VALUE (e_0))がTRUEかFALSEかにしたがって，結果式は$\pi(e_1)$か$\pi(e_2)$となる．TRUEでもFALSEでもなければ SELECT ($\pi(e_0), \pi(e_1), \pi(e_2)$) となる．

(6) π(JUMP (e)) = JUMP ($\pi(e)$)

(7) π(RETURN (e)) = RETURN ($\pi(e)$)

(8) ブロック

ブロック $e = (l_1 : e_1; \cdots ; l_n : e_n ;)$ に対するπの作用を図4のように定める．ここで，*enblock* という作用素は

$$enblock((l_1 : e_1; \cdots ; l_n : e_n ;), l, e) = (l_1 : e_1; \cdots ; l_n : e_n ; l : e ;)$$

と，ブロックに式を継ぎ足して延長するものである．引数にNULLがあれば，結果式において対応する部分が省略された形になる．ブロックeのインタープリテーションは，e内の式を順次インタープリトしてゆき，JUMPに出合えばそこで指定された飛び先に行き，RETURN (a)に出合えば式aの値π(VALUE (a))をeの値としてインタープリテーションを終えるというように進む．このとき，$\pi(e)$はインタープリトした式の列を表わす．RETURNによって値の示されていない場合は，π(VALUE (e)) をUNKNOWN とする．

(9) DECLARE (a, e)

f形，x形，およびx_1．x_2．$\cdots$．x_n形の式aには，「属性リスト」と呼ばれる式が対応していることがある．属性リストは一般に，$(f_1(e_{11}, \cdots, e_{1n_1}), \cdots, f_k(e_{k1}, \cdots, e_{kn_k}))$

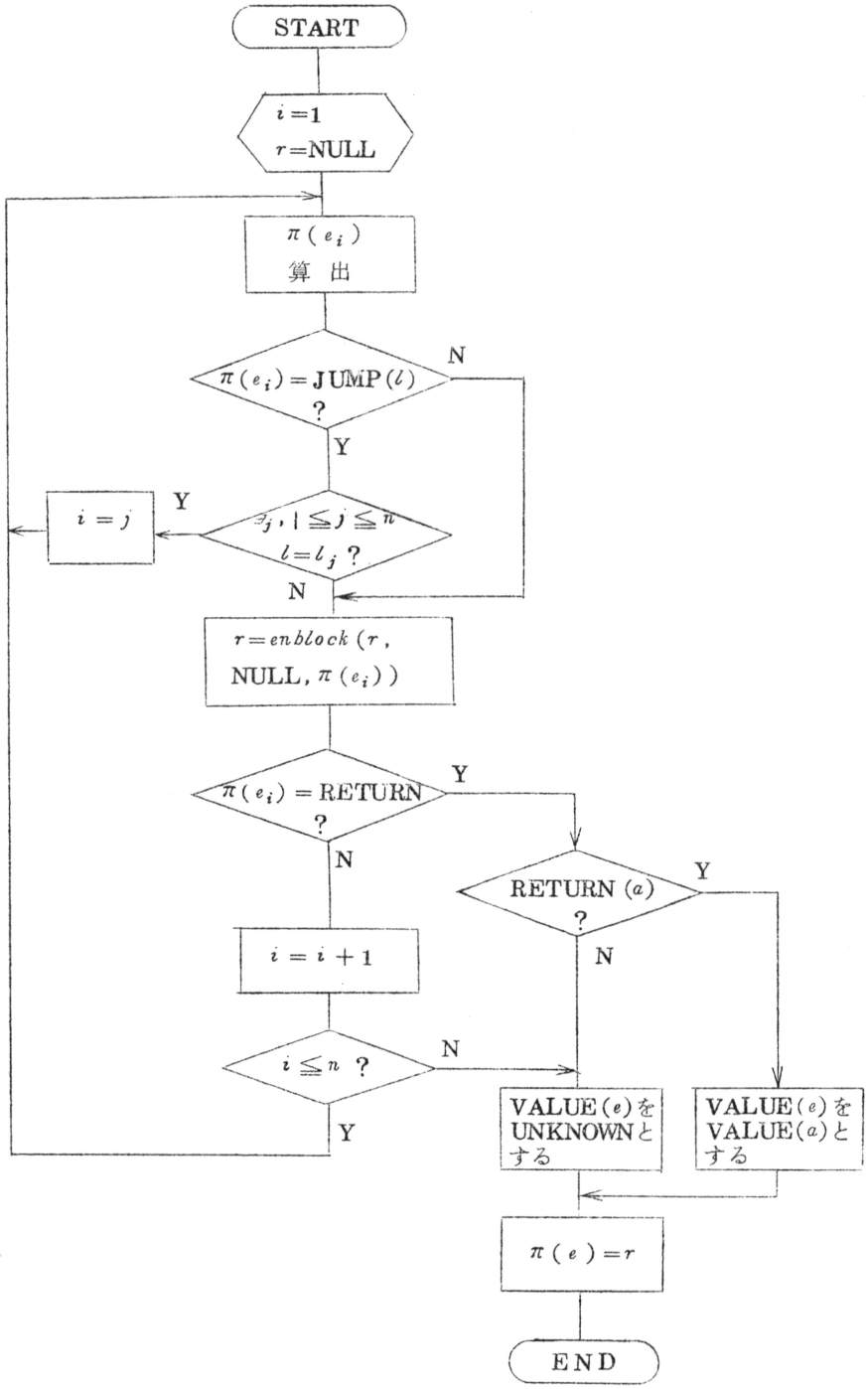


図4　$e=(l_1: e_1; \cdots; l_n: e_n;)$のインタープリテーション

という形であり，これは，対応する a が $f_i(e_{i1}, \cdots, e_{in_i})$，$i=1, \cdots, k$ で示される属性を持つことを表わす．DECLARE (a, e) は，a の属性リストに e で示される属性を追加することを表わす．π に対する作用は通常の通りである．

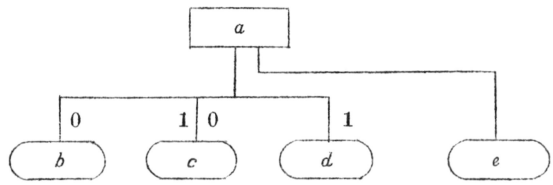
$$\pi(\text{DECLARE}(a, e)) = \text{DECLARE}(a, \pi(e))$$

(10) ALIST (a)

a の属性リストが $(e_1, \cdots, e_n)$ であれば，$\pi(\text{ALIST}(a)) = (\pi(e_1), \cdots, \pi(e_n))$ となる．a に属性リストがなければ結果はNULLとなる．ここで，a は f 形，x 形，または x_1．x_2．…．x_n 形の式とする．

6. プログラミング上の便法

以上のことを踏まえて，ある処理操作を基底式に帰着させるために書かれたDEFINE式がプログラムである．しかし，この「標準形」の式では，かっこが多くて実用的でない．そこで，式を書きやすくわかりやすい形にするための変形規則を定めて，「修正形」の式を作り，実際のプログラミングでは，図5に示した形の表現形式をとる．標準形で使うか修正形で使うかは，その式をDEFINE式でどちらの形として定義したかで決まる．図5で使っている記号を説明すると，たとえば，



という図があったとすれば，これはバッカス記法による次の表現と同等である．

$$<a> ::= <b_1><c_1><d_1> \mid <e>$$
$$<b_1> ::= <null> \mid <b> \mid <b_1><b>$$
$$<c_1> ::= <null> \mid <c>$$
$$<d_1> ::= <d> \mid <d_1><d>$$

(1) DEFINE $(e_0, f(x_1, \cdots, x_n), e)$ における $x_1, \cdots, x_n$ を仮引数というが，仮引数の名前は頭文字をXとし，関数名は頭文字をXとしない．

(2) △で省略の可能性のある空白を表わすことにして，$f(e_1)$ を $f \triangle e_1$，$g(e_1, e_2)$ を $e_1 \triangle g \triangle e_2$ と表わすときには，このように変形する関数名相互の間に優先順位を定めておいて，一意的にもとの標準形の式に戻せるようにする．

(3) $f(e_1, \cdots, e_n)$ に対して，標識や特殊演算子からなるキイ $k_0, k_1, \cdots, k_n$ を1組定め，$f(k_1 \triangle e_1, \cdots, k_n \triangle e_n)$，あるいは，$f \triangle k_0 \triangle k_1 \triangle e_1 \triangle \cdots \triangle k_n \triangle e_n$ と変形するときには，キイによって何番目の引数かが示されるので，引数の一部を省略して使ってもよい．

(4) 前項(2)，(3)における空白△は，前後の記号がともに定数か標識，あるいは，ともに特殊

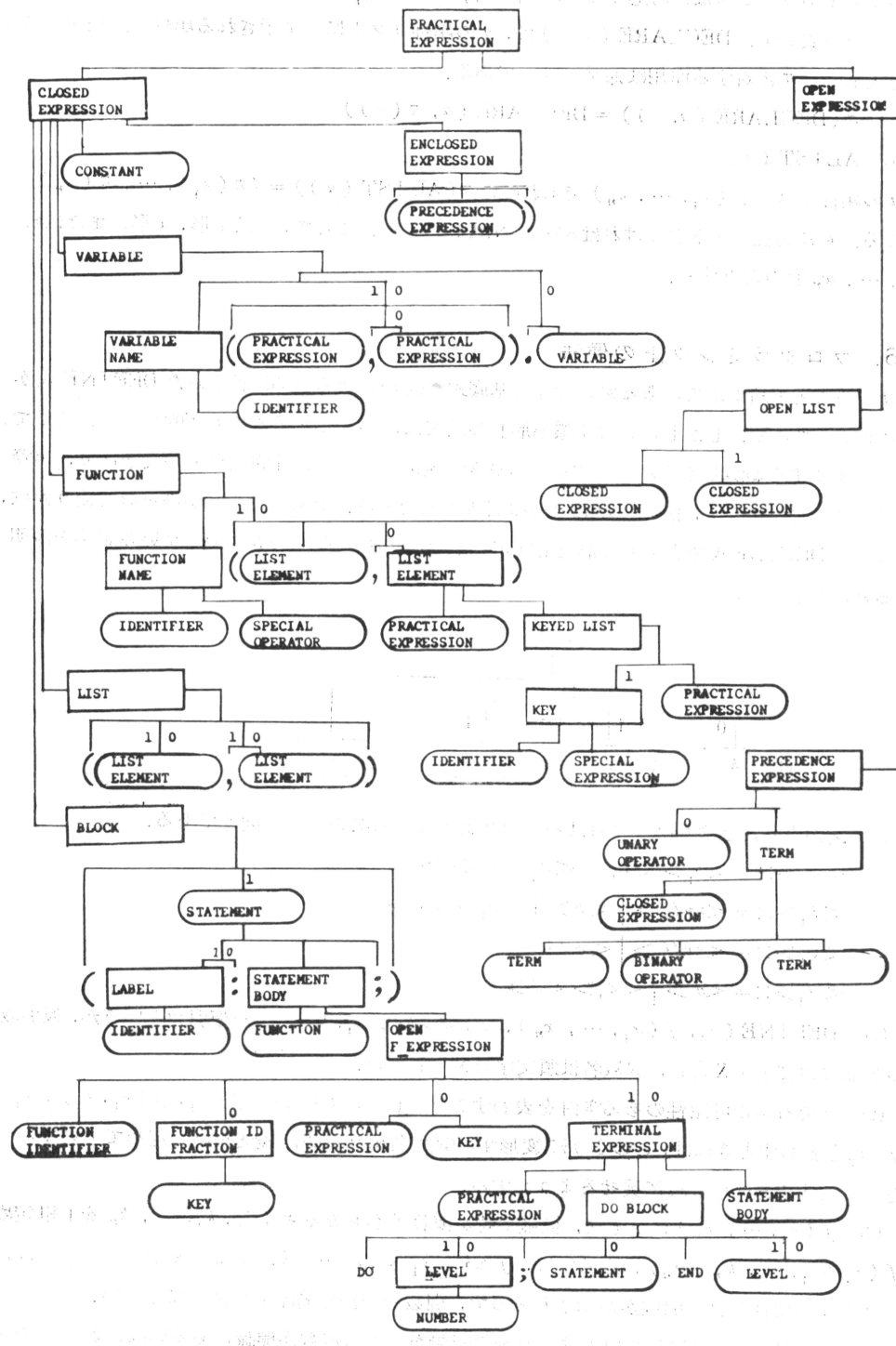

PRACTICAL EXPRESSION
CLOSED EXPRESSION
OPEN EXPRESSION
CONSTANT
ENCLOSED EXPRESSION
PRECEDENCE EXPRESSION
VARIABLE
VARIABLE NAME
PRACTICAL EXPRESSION
PRACTICAL EXPRESSION
VARIABLE
IDENTIFIER
OPEN LIST
CLOSED EXPRESSION
CLOSED EXPRESSION
FUNCTION
FUNCTION NAME
LIST ELEMENT
LIST ELEMENT
IDENTIFIER
SPECIAL OPERATOR
PRACTICAL EXPRESSION
KEYED LIST
KEY
PRACTICAL EXPRESSION
IDENTIFIER
SPECIAL EXPRESSION
LIST
LIST ELEMENT
LIST ELEMENT
PRECEDENCE EXPRESSION
BLOCK
UNARY OPERATOR
TERM
STATEMENT
CLOSED EXPRESSION
LABEL
STATEMENT BODY
TERM
BINARY OPERATOR
TERM
IDENTIFIER
FUNCTION
OPEN F_EXPRESSION
FUNCTION IDENTIFIER
FUNCTION ID FRACTION
PRACTICAL EXPRESSION
KEY
TERMINAL EXPRESSION
KEY
PRACTICAL EXPRESSION
DO BLOCK
STATEMENT BODY
DO
LEVEL
STATEMENT
END
LEVEL
NUMBER

図5. 基本構文

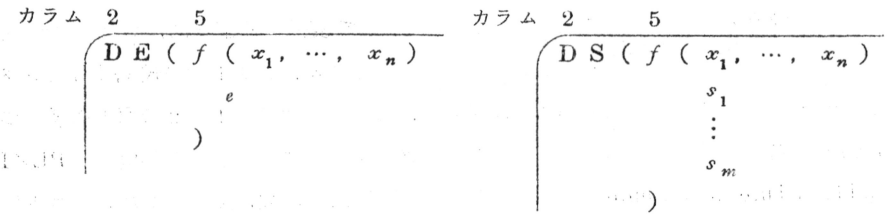
演算子となるのでなければ省略してよい．

(5) かっこを少なくするために，ブロック（s_1；…；s_n；）をDO；s_1；…；s_n；END；と表わしてもよい．このDO形ブロックが多重になっているときは，DO n．n；…END n．n；と，DOとENDの対応を数字n．nで表わすことにより，多重のENDを一つのENDで表わしてもよい．

(6) 記号を書くカラム位置を利用して，プログラムをもっとわかりやすい形にする．そのために，カラム1～4を特殊用途にあて，一般の式では使わないことにする．

a) 改行をセミコロンと同格に扱う．式の途中で改行するときは，カラム1に1～9の数字を入れて，改行によるセミコロンの働きを打ち消す．

b) DEFINE (EXPRESSION, $f(x_1, \cdots, x_n)$, e)，および，DEFINE (STATEMENT, $f(x_1, \cdots, x_n)$, (s_1；…；s_m；)) をそれぞれ，



と表わすことにする．

c) ラベルは，標準形の式のように，l：と標識lのうしろにコロンをつけて示してもよいが，lをカラム5から書き，lのうしろに一つ以上の空白をおくことによって示してもよい．ラベルと区別するために，通常の式は必ずカラム7かそれ以降に書く．

d) 関数名の優先順位を指定するときは，カラム4からPRIORITY (n．n)と書いて，その優先度n．nを示す．

(7) 記号の重複使用を認めるために，記号の通用範囲（scope)を定め，同一記号でも通用範囲が違えば異なる記号として扱う．

a) 変数の通用範囲は，PL/Iにおけるブロックをこの言語におけるブロックに対応させて考えた場合と同じである．

b) 関数fを定義するDEFINE式があるとき，fの通用範囲は，そのDEFINE式を直接の部分式として含む式pに限定される．

c) ブロックe＝（l_1：e_1；…；l_n：e_n；）のラベルl_1, …, l_nの通用範囲は，eの真の部分式のうち，ブロック形の部分式を除いた部分に限定される．

7. コンパイラ

プログラムpをトランスレータσで基底式に帰着させて式$\sigma(p)$を作り，それにインタープリタπを作用させて$\pi(\sigma(p))$を作ることが，プログラムpの実行に相当する．ところで，

導入された式をすべてそれを定義している式でおきかえると，$\sigma(p)$が著しく長くなる．そこで，指定した式に対してのみσによるおきかえを行ない，そのほかについては，対応するDEFINE式へのリンケージを作るだけにとどめることにし，このように変形したσをσ_0と表わす．とくに，recursiveに定義する式はリンケージを作るにとどめることにする．

さきに，DEFINE(e_0, e_1, e_2)をDE(e_1, e_2)またはDS(e_1, e_2)と表わすことにしたが，σ_0によって代入を行なうものをそれぞれME(e_1, e_2)またはMS(e_1, e_2)と表わすことにし，DE(…)，DS(…)はリンケージを作るものを表わすことにする．

プログラムの実行に関与するがプログラム中に表面上表われていないもの(入力データなど)を，インタープリタに対する外部環境と仮りに呼ぶことにしよう．プログラム実行時の効率を考えると，プログラムのうち，外部環境に左右されない部分をあらかじめインタープリトしておくとよい．しかし，外部環境に左右されない部分をすべてπによって変換すると，結果としてできる式が非常に長くなる．そこで，指定した部分式のみをπによって変換し，残りをそのままにする写像を考えてそれをπ_0と表わす．π_0によって変換する部分は，ブロック形の式pの直接の部分式のうち，カラム1に%がついている式とする．%が1つの式e_iは，インタープリトした結果である$\pi_0(e_i)$を$\pi_0(p)$に含める．しかし，カラム1と2の双方に%のついた式は，結果を$\pi_0(p)$に含めず，π_0によってインタープリトする時点でのみ働く(PL/Iのcompile time statementに相当する)式とする．定数，変数，関数，リストに対するπ_0の作用はπと全く同じである．

このようにすると，プログラムpの実行は次のようになる．

$$p \rightarrow \sigma_0(p) \rightarrow \pi_0(\sigma_0(p)) \rightarrow \pi(\pi_0(\sigma_0(p)))$$

σ_0で変換したあとπ_0を作用させる写像$\pi_0\sigma_0$がこの言語のコンパイラである．

図2の例は，以上の変形規則にしたがって書いたプログラムである．

8. あとがき

記号の意味を定義する機能を持った計算機言語を導入し，そのコンパイラの作用を記号列から記号列への写像として定めた．この言語は，1ページで図示できる単純な構文を持つことと，その構文の範囲内で，人間にとってわかりやすい表現形式をとるさまざまの問題向き言語を構成できること，および，プログラムをフローチャートのように大局から細部へと書き進める形態をとること，この3点を特徴とする．現在はこの言語に対するコンパイラの作成を進めている段階である．

9. 参考文献

(1) 渡辺： 意味論的メタ言語の形をした計算機言語，情報処理，Vol.11，No.8(1970)