# PL/I の形式的定義について (4)*

情報処理学会・PL/I 研究委員会**

## 7. EPL 解釈に関する定義

本章では，これまでに導入した抽象機械の概念を用いて，EPL の解釈に関する定義を述べる．前半で解釈機械の状態を定義し，ついでほとんど説明なしに形式的定義を展開する．最後に，これら定義の意味について具体的な説明を行なう．

### 7.1 解釈機械の状態

ここでは，解釈機械で扱われる状態の集合 is-state を定義する．すでに定義した基本対象の集合，セレクタの集合，および EPL 抽象構文で規定されているセレクタの集合のほかに，つぎの基本対象とセレクタが必要となる．

| | |
|---|---|
| is-n | 名前の無限集合 |
| {FUNCT} | 関数名を示す属性 |
| {s-env, s-c, s-at, s-dn, s-d, s-n} | 解釈機械の成分に対するセレクタ |

解釈機械の状態 $\xi$ は，以下の S1～S7 で定義される．

(S1) is-state＝(⟨s-env : is-env⟩, ⟨s-c : is-c⟩, ⟨s-at : is-at⟩, ⟨s-dn : is-dn⟩, ⟨s-d : is-d⟩, ⟨s-n : is-integer-value⟩†)

(S2) is-env＝({⟨*id* : is-n⟩|is-id(*id*)})

(S3) is-c＝{6章で述べた性質を持つ制御木}

(S4) is-at＝({⟨*n* : is-type⟩|is-n(*n*)})

(S5) is-type＝{INT, LOG, FUNCT}

(S6) is-dn＝({⟨*n* : (⟨s-env : is-env⟩, ⟨s-attr : is-funct-attr⟩)∨is-value⟩|is-n(*n*)})

(S7) is-d＝(⟨s-env : is-env⟩, ⟨s-c : is-c⟩, ⟨s-d : is-d⟩)∨is-Ω

つまり解釈機械の状態とは，環境部 (s-env)，制御部 (s-c)，属性部 (s-at)，値表示部 (s-dn)，ダンプ部 (s-d) および一意名生成のカウンタ (s-n) の各成分の組として表わされる．これら各部の意味については後述する．

任意に与えられたプログラム $t$∈is-progr に対して，

$$\mu_0=(\langle\text{s-c}:\ \textbf{int-progr}(t)\rangle, \langle\text{s-n}:\ 1\rangle)$$

なる状態を，そのプログラムの初期状態と呼ぶ．すなわち，制御部だけからなり制御部に命令 **int-progr**($t$) を持つ状態が，初期状態であり，命令 **int-progr** を次節の如く定義することによって，プログラム $t$ の解釈が定義される．

状態 $\xi$ の制御部 s-c($\xi$) がΩであるとき，$\xi$ を終了状態と定義する．

### 7.2 言語の解釈

ここでは，プログラムをパラメタに持つ命令図式，**int-progr** を定義する．命令 **int-progr**($t$) の実行によって，プログラム $t$ の仕事が抽象機械の言葉で定義されるのである．

定義を読みやすくするために現在状態 $\xi$ の直属成分 s-env($\xi$), s-c($\xi$), s-at($\xi$), s-dn($\xi$), s-d($\xi$) などをそれぞれ，**ENV, C, AT, DN, D** と略記する．

また定義中で，定義されない条件が存在するとき，それが誤まりであることを

T→error

と表示する．

以下に述べる関数は，形式的定義なしで使われる．

(1) convert ($v$, *attr*)：変数 $v$ を必要なら *attr* で指定される型 (INT または LOG) に変換する関数．

(2) int-bin-op (*op*, *a*, *b*)：演算子 *op* を *a* と *b* とに作用させた結果を返す関数．

(3) value($a$)：定数 $a$ の値を返す関数．

上記の約束をもとに，(I1)～(I16) により **int-progr** ($t$) の形式的定義を与える．

(I1) **int-progr**($t$)＝**int-block**($t$)
ただし，$t$∈is-progr($t$)

(I2) **int-block**($t$)＝
**s-d** : $\mu_0$(⟨s-env : **ENV**⟩, ⟨s-c : **C**⟩, ⟨s-d : **D**⟩)
**s-c** : **exit**;
**int-st-list** (s-st-list($t$));

---

* On the formal definition of PL/I (4), a report of the PL/I research committee of the ISPJ

** 情報処理 Vol. 11, No. 8, p. 457 参照

† これは一意名生成に使うカウンタである．

**int-decl-part** (s-decl-part ($t$));
**update-env** (s-decl-part ($t$));
ただし, $t \in$ is-block ($t$)

(I 3) **update-env** ($t$) =
**null**;
{**update-id** ($id$, $n$);
$n$ : **un-name** | $id(t) \neq \Omega$}
ただし, $t \in$ is-decl-part ($t$)

(I 4) **update-id** ($id$, $n$) =
s-env : $\mu$(**ENV**; ⟨$id$ : $n$⟩)
ただし, is-id ($id$), is-($n$)

(I 5) **int-decl-part** ($t$) =
**null**;
{**int-decl** ($id$(**ENT**), $id(t)$) | $id(t) \neq \Omega$}
ただし, is-del-part ($t$)

(I 6) **int-decl** ($n$, $attr$)
is-var-attr ($attr$) ⟶
**s-at** : $\mu$(**AT**; ⟨$n$ : $attr$⟩)
is-funct-attr ($attr$) ⟶
**s-at** : $\mu$(**AT**; ⟨$n$ : FUNCT⟩)
**s-dn** : $\mu$(**DN**; ⟨$n$ : $\mu_0$(s-attr : $attr$⟩, ⟨s-env : **ENV**⟩)⟩)
ただし, is-n ($n$), is-attr ($attr$)

(I 7) **int-st-list** ($t$) =
is-⟨⟩($t$) ⟶ **null**
T ⟶ **int-st-list** (tail ($t$));
**int-st** (head ($t$))
ただし, is-st-list ($t$)

(I 8) **int-st** ($t$) =
is-assign-st ($t$) ⟶ **int-assign-st** ($t$)
is-cond-st ($t$) ⟶ **int-cond-st** ($t$)
is-block-st ($t$) ⟶ **int-block** ($t$)
ただし, is-st ($t$)

(I 9) **int-assign-st** ($t$) =
is-var-attr ($n_t$(**AT**)) ⟶
**assign** ($n_t$, $v$);
$v$ : **int-expr** (s-right-part ($t$))
T ⟶ error
ここに, $n_t$ = (s-left-part ($t$)) (**ENV**)
ただし, is-assign-st ($t$)

(I 10) **assign** ($n$, $v$) =
**s-dn** : $\mu$(**DN**; ⟨$n$ : convert ($v$, $n$(**AT**))⟩)
ただし, is-n ($n$), is-value ($v$)

(I 11) **int-cond-st** ($t$) =
**branch** ($v$, s-then-st ($t$), s-else-st ($t$)) :
$v$ : **int-expr** (s-expr ($t$))
ただし, is-cond-st ($t$)

(I 12) **branch** ($v$, $st1$, $st2$) =
convert ($v$, LOG) ⟶ **int-st** ($st1$)
¬convert ($v$, LOG) ⟶ **int-st** ($st2$)
ただし, is-value ($v$), is-st ($st1$), is-st ($st2$)

(I 13) **exit** =
**s-env** : s-env (***D***)
**s-c** : s-c (***D***)
**s-d** : s-d (***D***)

(I 14) **int-expr** ($t$) =
is-bin ($t$) ⟶ **int-bin-op** (s-op ($t$), $a$, $b$);
$a$ : **int-expr** (s-rd 1 ($t$)),
$b$ : **int-expr** (s-rd 2 ($t$))
is-funct-des ($t$) & (at$_t$ = FUNCT) ⟶
**pass-value** ($n$);
**int-funct-call** ($t$, $n$);
$n$ : **un-name**
is-var ($t$) & is-var-attr ($n_t$(**AT**))
⟶ PASS : $n_t$(**DN**)
is-const ($t$) ⟶ PASS : value ($t$)
T ⟶ error
ここに, $n_t = t$(**ENV**),
at$_t$ = ((s-id ($t$)) (**ENV**)) (**AT**)
ただし, is-expr ($t$)

(I 15) **pass-value** ($n$) = PASS : $n$(**AT**)

(I 16) **int-funct-call** ($t$, $n$) =
(length (arg-list$_t$) = length (p-list$_t$)) ⟶
**s-env** : $\mu$(env$_t$; {⟨elem ($i$, p-list$_t$) : elem ($i$, arg-list$_t$) (**ENV**)⟩ | $1 \leq i \leq$ length (p-list$_t$)}
**s-d** : $\mu_0$(⟨s-env : **ENV**⟩, ⟨s-c : ***C***, s-d : ***D***⟩)
**s-c** : exit;
**assign** ($n$, $v$);
$v$ : **int-expr** (expr$_t$);
**int-st** (st$_t$)
T ⟶ error
ここに, $n_t$ = (s-id ($t$)) (**ENV**),
p-list$_t$ = s-param-list ∘ s-attr ∘ $n_t$ (**DN**),
env$_t$ = s-env ∘ $n_t$ (**DN**),
arg-list$_t$ = s-arg-list ($t$),
st$_t$ = s-st ∘ s-attr ∘ $n_t$ (**DN**),

$expr_t$=s-expr∘s-attr∘n (**DN**)

ただし, is-funct-des($t$), is-n($n$)

### 7.3 形式的定義の直観的説明

#### 7.3.1 状態の成分

**(1) 環境部とダンプ成分**

環境部 (environment, **ENV**) は, 状態$\xi$で参照できるすべての識別子 (identifier) を含み, 識別子と一意名$n$とを対応づける. 抽象機械は, 各識別子に対応する一意名によって, 値表示簿 (denotation directory, **DN**) および属性登録簿 (attribute directory, **AT**) を索引して, 現在状態$\xi$での識別子の値や意味を知ることができる.

EPL のブロック構造の意味をみるためには, 状態の環境成分の書き換えをする全命令 (s-env を含む命令) を調べればよい. 初期状態においては環境成分は$\Omega$である. EPL プログラムは1つのブロックであり, プログラムの最初の動作はブロックの起動である. I1, I2 によれば, ブロックの起動では, 現在状態の環境部と制御成分とをダンプの頭に乗せ, 新しい制御部を作り, さらに環境部を更新する. 環境部の更新では, 当ブロック内で宣言された各識別子 *id* に対し一意名 $n$ を生成し (I.3), セレクタ *id* を持つ成分 $n$ が$\mu$関数により環境に組み込まれる. その *id* がすでに存在すれば対応する成分は新しい $n$ で書きかえられる.

ブロックの出口では, (I2), (I14) の定義にしたがってダンプの先頭に乗せられている環境部と制御部が, 再生され新しい状態での環境部・制御部となる. ダンプはポップアップされる.

あるブロックの実行中に, さらに, 他のブロックの起動や関数呼出しがあった場合には, 環境部を更新する前にそのコピーがダンプの先頭に乗せられ, 復帰に際して, 上述の機構で再生される. これによりブロック構造による *id* の有効範囲の管理が行なわれるのである. (I2) では, ブロックの解釈によって環境部の更新を行ない, ついで宣言部の解釈が行なわれることを示している. 関数宣言を解釈すると, その状態での環境部が当関数名の値表示の一部として値表示簿 **DN** に保存される. つまり関数宣言に現われた非局所変数の意味は, その宣言が解釈された時点で凍結される. 一般に与えられたあるテキストにおいて, その意味が, テキスト内のある識別子に対する参照の時点に依存する場合, 当 *id* を含む環境部とテキストとを対応づけて保存することにより解決される.

関数呼出し時に行なわれるアーギュメントの受授は, 単にパラメタをアーギュメントの一意名に対応づければよく, パラメタをアーギュメントの同義語とすればよい.

**(2) 値表示簿** (Denotation Directory, **DN**)

値表示簿は, 一意名 $n$ と表示子 $dn$ を対応づける, このことを $n$—$dn$ と略記する.

EPL の場合この対応として, 変数の場合 $n$-*value*, 関数の場合, $n$-(*param-eist*, *st*, *expr*), *env* の如き対応となる. (S6)

**DN** に登録されるこれら各エントリは, 関数の場合は, ブロックの起動により, 変数の場合は代入の実行により作られる. 関数対応のエントリは登録後の変更はないが, 変数対応の値は, 当変数への代入により変化する〔(I6), (I10)〕. 関数の値は, **DN** の補助エントリを介して返される〔(I14), (I16)〕.

**(3) 属性登録簿** (Attribute Directory, **AT**)

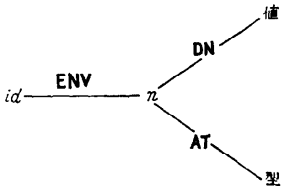
属性登録簿は, 一意名と名前の型とを対応づける. EPL では, 整数変数に対し, $n$—INT, 論理変数に対して $n$—LOG, 関数に対して, $n$—FUNCT の場合が生ずる. 抽象構文ではある型の識別子が, その型と矛盾しない文脈においてしか参照できないとは規定していない. この点を解釈機械では (I14) で規定している.

属性登録簿中のエントリは, ブロック起動によって作られ, 以後変更, 削除などは行なわれない.

#### 7.3.2 識別子の型と動的な意味

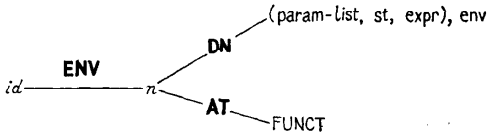
前節で述べたように, ある状態$\xi$で参照できるすべての *id* は, 環境成分を介して一意名 $n$ と対応づけられ, $n$ はさらに, 値表示簿の表示子 $dn$ および属性登録簿中の型と対応づけられる. この関係を図示すると EPL では, つぎの場合が生ずる.

変数のとき: (**第 7.1 図**)




第 7.1 図

関数のとき: (**第 7.2 図**)

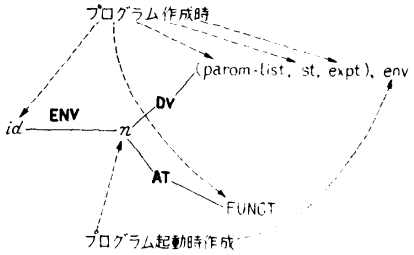



第 7.2 図

このような情報および関係が，いつ，どんな場合に作られ，いつ解除されるかを明確にする必要がある．

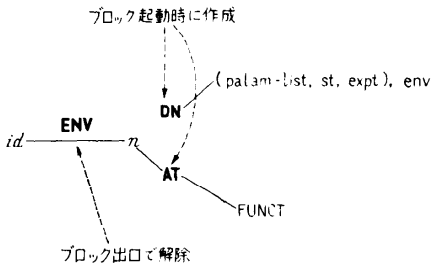
この点については，EPL の場合は単純であるが，PL/I の形式的定義の理解には非常な参考となる[1]．関数の場合を例にとり図で示せば，下図のとおりである．

（1） 情報の作製（**第 7.3 図**）




第 7.3 図

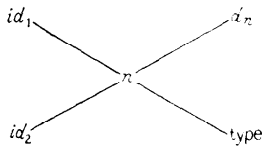
（2） 情報間の対応づけ（**第 7.4 図**）




第 7.4 図

実際の言語では，異なる *id* 間で，これら制御情報の共用が必要な場合がよくある．PL/I では，きわめて複雑な共用パターンが現われるが，EPL ではきわめて単純で，関数のアーギュメントとパラメタの関係づけの場合のみ現われる．

ある関数呼出しの場合，アーギュメント $id_1$ がパラメータ $id_2$ に送られる場合，（I 16）に示すように，$id_1$ の一意名 $n$ を介して，つぎの如き共用パターンがアーギュメントとパラメタの対応づけの後に現われる（**第 7.5 図**）．




第 7.5 図

### 7.3.3 制御の流れ

これまでは，ブロック構造と識別子の動的な意味づけの観点からのみ説明したが，ここでは，プログラムで指示された動作が遂行される順序に焦点を合わせる．

定義（I 2）では，ブロックの解釈は，環境部の更新，宣言部の解釈，文の解釈，および出口での動作を意味することを規定している．

また，定義（I 5）では，個々の宣言は任意の順序で解釈してもよいことを規定している．これは，EPL の場合意味をもたないが，PL/I などにおいては，宣言の解釈に式の評価が含まれる場合があり，副作用をもたらすことがあるので，問題となる．

文リストの解釈は，（I 7）で規定されているように，与えられた順序に従って，リストを順に解釈することを意味する．

つぎに個々の文の解釈について簡単に説明しよう．

代入文については，定義（I 9）によって，右辺の式が評価され，その結果が左辺の変数に代入されることが規定されている．式のオペランドは，定義（I 14）によれば任意の順序で評価してよいが，順序の選択によって，意味の変わるような式がありうることには注意する必要がある．式の評価の途中で関数が起動されると定義（I 16）に従って，そのときの制御部はダンプに乗せられ，新しい制御部が組み入れられる．つまり，関数呼出しによって他のオペランドの評価が，関数の処理の完了するまで待たされることになる．

たとえば

$$f_1(X)+f_2(Y)*f_3(Z)$$

の如き式では，このオペランドはどんな順序で評価してもよい．しかし，どの 2 つの関数の評価動作も，互いに入り組むことはできない．

条件文の解釈は説明するまでもないが，定義は（I 11）に与えられている．

## 9. あとがき

以上，IBM ウイーン研究所でまとめられた，PL/I の形式的定義の手法を，簡単な例題言語 EPL をもとに解説した．

PL/I の如ききわめて単純な言語では，ここに述べたように，かなりコンパクトに，その解釈を定義できるが，実際に PL/I を定義するとなると文献（1）の如く，きわめて大部なものとなってしまうのが実状である．

現在，PL/I の形式的定義は，コンパイラ設計や PL/I の応用に際し，自然語による文法書に疑義が生じた場

合に統一的解釈を得る手段として利用されているようである.

ここで述べたような,意味論も含めた形式的言語定義の研究に残された問題としては,いかにしてよりコンパクトな定義を与えるか,また,それによって,実在する言語を解析的に取り扱う手法の開発などであると考えられる.

本稿での紹介は,ULD の1つの方向についての現状の解説であるが,これが今後この方面での研究の,1つの契機となることを期待したい.

終わりに,PL/I 研究会の発足に努力を払われた,前情報処理学会理事,岸上利秋氏に感謝の意を表する. (終)

## 参考文献

1) Walk, K. et al: Abstract syntax and interpretation of PL/I.—IBM Lab. Vienna, Techn. Report TR 25.082, 28, June 1968.
2) Lucas, P. et al: Informal introduction to the abstract syntax and interpretation of PL/I. IBM Lab. Vienna, Techn. Report TR 25.083, June 1968.
3) Lucas, P. et al: Method and notation for the formal definition of programming languages. IBM Lab. Vienna, Techn. Report TR 25.087, 28, June 1968.
4) McCarthy, J.: Towards a mathematical science of computation.—Information Processing 1962, pp. 21-28; Amsterdam 1963.
5) Landin, P. J.: Correspondence between ALGOL 60 and Church's Lambda- Notation. Part I.—C. ACM, 8(1965), No. 2, pp. 89-101, Part II.—C. ACM 8(1965), No. 3, pp. 158-165.
6) Bandat, K.: On tde formal definition of PL/I. SJCC 1968, pp. 363-373.
7) Lauer, P.: Abstract syntax and interpretation of ALGOL 60.—IBM Lab. Vienna, Lab. Report LR 25.6.001, 12, April 1968.

(昭和42年2月3日受付)