PL/I の形式的定義について (3)*

情報処理学会・PL/I 研究委員会**

6. 抽象機械

6.1 概 要

ALGOL 60 あるいは PL/I のような高水準プログラミング言語の意味を形式的に定義するために，抽象機械による方法が採用された．この機械の特徴は，状態の集合とその状態推移関数とにある．与えられた言語で書かれた特定のプログラム（入力データを伴う）は，機械の初期状態を決め，機械のその後の動きが，与えられた入力データに対して，そのプログラムの解釈を規定する．

6.2 抽象機械の概念

一般に，抽象順序機械は，機械がとりうる**状態の集合** Σ と，与えられた任意の状態 ξ に対して**後続状態**を指定する**状態推移関数** Λ とを規定することによって説明できる．状態推移関数はある状態を他の状態に写す関数である．状態の集合 Σ の部分集合 Σ_E（機械の終了状態）を規定することにより，機械がいつストップするかを決める標識が与えられる．

与えられた任意の**初期状態** ξ_0 に対して

$$\xi_0, \xi_1, \cdots, \xi_i, \xi_{i+1} \cdots \quad (\xi_{i+1} = \Lambda(\xi_i))$$

の如く次々に推移する状態の系列として，機械の動きをとらえることができる．この状態系列は与えられた初期状態 ξ_0 に対する**計算**と呼ばれ，もし終了状態 $\xi_n \in \Sigma_E$ に到達すれば，機械はストップする．その場合，計算は**終結型**であるといわれる．終了状態に到達しない計算は，非終結型であるといわれる．

PL/I あるいは ALGOL 60 のようなプログラミング言語の解釈には，ある種の演算の遂行される順序が適切であっても，定義されずに残されている部分がいくつかある．そのような例として

・式内のオペランドの評価
・データ属性の一部で指定する式の評価
・手続き呼出しあるいは関数呼出しにおけるアーギュメントの評価

などがあげられる．そこで抽象機械の概念をこのような要素を持つ言語の定義にも適合できるように拡張する．

プログラミング言語の定義のために使用される抽象機械は，つぎのような4組ベクトル

$$\langle EO, S, \text{is-state}, \Lambda \rangle$$

で定義される．ここに

EO	：基本対象の無限集合
S	：単純セレクタの無限集合
is-state	：機械がとりうる状態を表現するために使う対象の類を定義する述語
Λ	：状態推移関数

ある対象の類を使って状態の集合を識別すれば，この集合は **4.6** で述べた手段で常に定義できる．また与えられた状態から後続状態への写像は，μ 演算子を使って明記することができる．

さて，状態推移関数 Λ は，与えられた状態 ξ に対 Λ はあるての後続状態の集合を規定する．すなわち，するすべ状態を状態の集合へ写像する関数である（関数 Λ は，先に述べたプログラミング言語の未定義部分の説明に役立つように拡張されている）．

与えられた初期状態 ξ_0 に対する**計算**は，つぎのような状態の系列と考えられる．

$$\xi_0, \xi_1, \cdots, \xi_i, \xi_{i+1} \cdots \quad (\xi_{i+1} \in \Lambda(\xi_i))$$

これは計算が，列内の最近の状態に関数 Λ を適用することにより左から右へ1つずつ作り出されるということを意味している．後続状態は Λ を適用した結果としての状態集合の中から，1つの要素を選出して決められる．

状態推移関数 Λ を適用すると，空集合を生ずる状態 ξ を**終了状態**という．すなわち，$\Lambda(\xi) = \{\ \}$．終了状態に到達したところで計算は終わりである．このような計算は状態の有限系列で表わされ，終結型の計算であると呼ばれる．

状態の無限系列となる計算は，非終結型の計算と呼ばれる．

---

* On the formal definition of PL/I (2), by a resort of the PL/I research commnittee of the ISPJ

** 情報処理 Vol 11, No. 8, p. 457 参照

状態推移関数 A は，その変域となりうる状態の部分集合に対してのみ定義される関数であってもよい．だから計算には第3の型，すなわち，最近の状態に関数 A を適用しても，値が存在しないために継続しようのない型がある．

6.2 抽象機械の制御

6.2.1 制御木

すべての状態 ξ は，**制御部**といわれる直属成分 s-c(ξ) を持ち，is-c なる述語を満足する．すなわち

is-c(s-c(ξ))

機械のある状態 ξ の制御部は，おのおのの節がひとつの**命令**と関連した有限木によって説明できる．このような木を**制御木**という，制御木の**終端節**と関連したいくつかの命令のみがつぎに実行される資格を持ち，それらの間では，順番は規定されない．与えられた状態 ξ の後続状態は，s-c(ξ) の終端節の1つを選出し，それと関連する命令を実行することによって決められる．

たとえば**第 6.1 図**は，ある状態 ξ の制御部を表わす制御木である．第 6.1 図では，つぎに実行する資格がある命令は，instr 2, instr 4, instr 5 である．

制御木 ct の終端節を指定するセレクタの集合を tn(ct) とすれば，状態推移関数 A はつぎのように定義できる．

$$A(\xi)=\{\psi(\xi,\tau)\mid\tau\in tn(\text{s-c}(\xi))\}$$

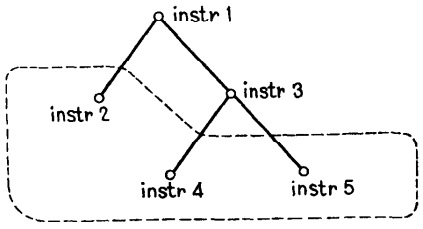
ここに $\psi(\xi,\tau)$ は終端節 $\tau(ct)$ と関連する命令を実行することにより後続状態を指定する．

命令は**命令名**とアギュメントな命令リスト（オプション）から構成され，つぎのように表現される．

$\mathbf{in}(\text{arg}_1,\cdots,\text{arg}_n)$

命令名は下線の引かれた語で，命令がどのようかを識別するものである．アーギュメントはそれぞれひとつの対象である．

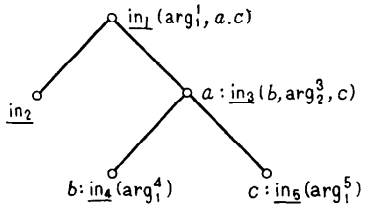
命令が実行されると状態が変わるが，命令実行の効果として2つの型が考えられる．1つは**値戻し**であり，もう1つは**自己書換え**である．まず値戻しは，現在実行している命令を制御木から消去し，実行結果の値を先行する節と関連する命令のアーギュメント部に代入するという型で，自己書換えは，現在実行中の命令が，もうひとつの制御木と置き換わるという型である．後者はマクロ展開と非常によく似ている．



第 6.1 図

関数や命令の効果について述べるには，制御木の構造についてもっと説明する必要がある．

制御木の位置関係を説明するために，つぎのような表記法を考える．まず，各節と関連する命令が実行した結果の値を先行する節と関連した命令のアーギュメントに代入するという関係を示すために，ダミーの名前を使う．制御木の各節と関係した命令の型が値戻しならば，その命令の前にコロンを伴ったダミーの名前を置き，結果の値が代入されるべき命令のアーギュメント部にはダミーの名前を書く．このようにして，命令とその実行結果の値がどこに代入されるべきかという関係が表わされる．1つの命令から，制御木内の数個所，数レベルにわたり値が返されることもある．**第 6.2 図**は上記の説明の理解に役立つであろう．第 6.2 図では，a, b, c がダミーの名前である．



第 6.2 図

ダミーの名前は，それが指定される制御木内でしか意味を持たないということに注意する必要がある．

制御木を図ではなく，式で表現することを考えよう．もし，制御木が1つの節だけから成るならば，その節と関連する命令で制御木を表わす．制御木が1つ以上の節から成るならば，先行する節と関連する命令の後にセミコロンを付け，その後にその節の直属成分である節と関連する命令の集合を続ける．第 6.2 図はつぎのように書くことができる．

$\mathbf{in}_1(\text{arg}_1{}^1, a, c)$; {$\mathbf{in}_2, a : \mathbf{in}_3(b, \text{arg}_2{}^3, c)$;
{$b : \mathbf{in}_4(\text{arg}_1{}^4), c : \mathbf{in}_5(\text{arg}_1{}^5)$}}

もし，中括弧内の命令が1つだけなら，中括弧は省略してもよい．中括弧の代わりにつぎのようにも書ける．すなわち，木の中の同じレベルにある命令は，左

のヘリをそろえて書き，レベルが低い命令は，1段ずらして書くという表記法が考えられる．上の例の場合，つぎのようになる（**第6.3図**）．

$\mathbf{in}_1(\mathrm{arg}_1^1, a, c)$;
　　$\mathbf{in}_2$,
a : $\mathbf{in}_3(b, \mathrm{arg}_2^3, c)$;
　　b : $\mathbf{in}_4(\mathrm{arg}_1^4)$,
　　c : $\mathbf{in}_5(\mathrm{arg}_1^5)$

↑ 実行方向

第 6.3 図

第6.3図に示すように，実行の方向が下から上に向いていることに注意する必要がある．

これまで述べてきた値戻しでは，アーギュメント部に代入される値は，1つの命令だけから返されていた．しかしある場合には，中間結果が複数個の命令から作られることがある．このような場合の説明のために，ある命令が実行した結果の値をアーギュメント全体にでなく，その成分に返すような方法を導入する．

ある制御木内のいくつかのアーギュメントとして，ダミーの名前 a を使用している場合

$\chi(a)$

なる表記法により，それらのアーギュメント部の χ 成分の位置を示す．たとえば

$\mathbf{in}_1(a)$; $\{\chi_1(a) : \mathbf{in}_2,\ \chi_2(a) : \mathbf{in}_3\}$

なる例では，命令 $\mathbf{in}_2$ は命令 $\mathbf{in}_1(a)$ のアーギュメント部の χ_1 成分に値を返し，命令 $\mathbf{in}_3$ は命令 $\mathbf{in}_1(a)$ のアーギュメント部の χ_2 成分に値を返すことを意味している．

6.2.2　制御木を対象として規定すること

まず，後で使用される対象とセレクタの集合を定義する．

（1）　基本対象の集合．

$\hat{\text{is-intg}}$,	$\{1, 2, \cdots\}$
$\hat{\text{is-in}}$	命令名の無限集合
$\hat{\text{is-name}}$	メタ変数の無限集合
$\hat{\text{is-sel}}$:	すべてのセレクタの集合（すなわち，ie, $\hat{\text{is-sel}} = S^*$）

（2）　述語 is-ob はすべての対象に対して成り立つ．

（3）　述語 is-sel-pair はセレクタのすべての対に対して成り立つ．すなわち

is-sel-pair = (⟨elem(1) : is-sel⟩,
　　⟨elem(2) : is-sel⟩).

（4）　s-in, s-al, s-ri, s-sel, s-dum は単純セレクタである*．

（5）　R は s-in, s-al, s-ri を除いた単純セレクタの集合である．

（6）　R^* は R にセレクタ I を加えたセレクタの集合である．

これらの対象，セレクタあるいは抽象構文による表記法を使って，制御木はつぎのような述語 is-ct を満たす対象であると定義できる．

is-ct = (⟨s-in : is-in⟩,
　　⟨s-al : ({⟨elem(i) : is-ob⟩ | is-intg(i)})⟩,
　　⟨s-ri : is-sel-pair-set⟩,
　　{⟨r : is-ct⟩ ‖ $r \in R$})

値戻しの仕組として，各節に対してセレクタの対の集合を対応さる．すなわち，対をなすセレクタの1つは，値がどこからとられるかを示し，他の1つは，値がどこに返されるかを示している．

6.2.1 節では，ダミーの名前を使って値戻しの仕組を説明したが，対象の類を使って，もっと厳密に表現しよう．これらの対象は，**中間の制御木**と呼ばれ，つぎのような述語 is-int-ct を満たす．

is-int-ct
　= (⟨s-in : is-in⟩,
　　⟨s-al : ({⟨elem(i) : is-ob⟩ | is-intg(i)})⟩,
　　⟨s-ri : (⟨S-sel : is-sel⟩,
　　⟨s-dum : is-name ∨ is-Ω⟩)⟩,
　　{⟨r : is-int-ct⟩ | $r \in R$})

ここでは，中間の制御木から制御木への変換を考える過程で，値戻しの仕組で使うダミーの名前の表記法を導入する．変換のための関数 tr(ct) は，つぎのいくつかの関数を使って定義される．

（1）　nd(ct) = {χ | $\chi \in R^*$ & $\chi(ct) \neq \Omega$}.
　　ここに　is-int-ct(ct).

（2）　arg(ct) = {elem(i)∘s-al∘χ | $\chi \in$ nd(ct) &
　　elem(i)∘s-al∘$\chi(ct) \neq \Omega$}.
　　ここに　is-int-ct(ct).

（3）　dum(ct) = {s-dum∘s-ri∘$\chi(ct)$ | $\chi \in$ nd(ct) &.
　　s-dum∘s-ri∘$\chi(ct) \neq \Omega$}.
　　ここに　is-int-ct(ct).

（4）　ri(ct, χ) = {⟨χ_1, $\chi_1 \circ \chi_2$⟩ | χ_1

---

* 省略形：
s-in	命令名を選択する
s-al	アーギュメント列を選択する
s-ri	戻り情報を選択する
s-sel	セレクタを選択する
s-dum	ダミーの名前を選択する

$$=\text{s-sel}\circ\text{s-ri}\circ\chi(ct)\ \&$$
$$\chi_2\in\text{arg}(ct)\ \&$$
$$\chi_2(ct)=\text{s-dum}\circ\text{s-ri}\circ\chi(ct)\}$$

ここに is-int-ct(ct), $\chi\in$nd(ct).

関数 tr(ct) は上記(1)～(4)の関数を使って，つぎのように定義される．

(5) $\text{tr}(ct)=\mu(ct):\{\langle\chi:\Omega\rangle\mid\chi\in\text{arg}(ct)\ \&\ \chi(ct)\in\text{dum}(ct)\}\cup\{\langle\text{s-ri}\circ\chi:\text{ri}(ct,\chi)\rangle\mid\chi\in\text{nd}(ct)\}$

6.2.3 制御木表現

制御木表現とは，与えられた自由変数の値に対する制御木を示すメタ式である．これを ct-rep あるいは自由変数を添えて ct-rep($\chi_1,\cdots,\chi_n,\xi$) という省略形で示す．制御木表現は次節の命令図式で使用される．

以下に使う制御木表現の意味は，制御木表現の各構成要素を構文の面から範疇分けした一覧表**第 6.1 表**によって定義される．第 6.1 表の一番最後のランには，制御木表現自体がのせられており，範疇分けされた各要素ごとに，省略形・書式・意味が明記されている．

「意味」の欄では何の注釈もなしに省略形 instr, pref-instr などが使われている．ある与えられたテキストが何に変換されるかは，テキスト自身にもよるが，テキストがどの範疇に属する要素と考えられるかにも依存する．

表中の関数 sel(*ob*, *succ-set*) は集合 *succ-set* 内の対象から R 内のセレクタへの1対1写像を規定している．関数 tr(ct) については **6.2.2** を参照されたい．

書式(1)の後続集合が制御木表現内に現われる場合，中括弧の代わりに段落をつける方法を使ってもよい．

6.2.4 命令図式

6.2.1 で述べたように，各命令名はこの名前を持つ命令を実行したときの効果（これはそのときのアーギュメントと状態 ξ に従う）を規定する命令図式を識別する．

ある命令名 ***in*** で識別される命令図式は，一般にはつぎのような形式をしている．

$$\textbf{\textit{in}}(X_1,\cdots,X_n)=P_1(X_1,\cdots,X_n,\xi)\rightarrow \text{group}_1(X_1,\cdots,X_n,\xi)$$
$$\cdots\cdots$$
$$P_m(X_1,\cdots,X_n,\xi)\rightarrow \text{group}_m(X_1,\cdots,X_n,\xi)$$

ここに，$P_i(X_1,\cdots,X_n,\xi)$ $(1\leqq i\leqq m)$ は真偽値を示すメタ式である．$\text{group}_i(X_1,\cdots,X_n,\xi)$ $(1\leqq i\leqq m)$ はグループと呼ばれ，次の2形式のいずれかで表現される．

(1) 値戻し

PASS: $\varepsilon_0(X_1,\cdots,X_n\xi)$
s-sc$_1$: $\varepsilon_1(X_1,\cdots,X_n,\xi)$
……
s-sc$_r$: $\varepsilon_r(X_1,\cdots,X_n,\xi)$

ここに，$\varepsilon_i(X_1,\cdots,X_n,\xi)$ $(1\leqq i\leqq r)$ は任意のメタ式．**s-sc**$_i$ $(1\leqq i\leqq r)$ は状態の直属成分を指定する単純セレクタ（必ずしも直属成分全部が参照されること

第 6.1 表

構文中の範疇	省略形	書式	意味
メタ変数	χ	χ	χ
無条件メタ式	expr	任意の無条件メタ式	メタ式の意味
命令の名前	**in**	**in**	**in**
命令	instr	(1) **in** (2) **in**(expr$_1$, ⋯, expr$_m$)	(1) $\mu_0(\langle\text{s-in}:\textbf{in}\rangle)$ (2) $\mu_0(\langle\text{s-in}:\textbf{in}\rangle,\langle\text{s-al}:\mu_0(\langle\text{elem}(1):\text{expr}_1\rangle,\cdots,\langle\text{elem}(m):\text{expr}_m\rangle))$
接頭辞のついた命令	pref-instr	(1) instr (2) χ : instr (3) expr(χ) : instr	(1) $\mu(\text{instr};\langle\text{s-ri}:\mu_0(\langle\text{s-sel}:\text{I}\rangle)\rangle)$ (2) $\mu(\text{instr};\langle\text{s-ri}:\mu_0(\langle\text{s-sel}:\text{I}\rangle,\langle\text{s-dum}:\chi\rangle)\rangle)$ (3) $\mu(\text{instr};\langle\text{s-ri}:\mu_0(\langle\text{s-sel}:\text{expr}\rangle,\langle\text{s-dum}:\chi\rangle)\rangle)$
後続（成分）	succ	(1) pref-instr (2) pref-instr; succ-set	(1) pref-instr (2) $\mu(\text{pref-instr};\{\langle\text{sel}(\chi,\text{succ-set}):\chi\rangle\mid\chi\in\text{succ-set}\})$[1]
後続集合	succ-set	(1) {succ$_1$, ⋯, succ$_k$} (2) {succ|expr} (3) succ-set$_1\cup\cdots\cup$succ-set$_r$	(1) {succ$_1$, ⋯, succ$_k$} (2) {succ|expr} (3) succ-set$_1\cup\cdots\cup$succ-set$_r$
制御本表現	ct-rep	succ[2]	tr(succ,

1) χ は succ-set の中に自由に現われることはない．
2) succ の最初の pref-instr は書式(1)のみである．

はない.).

(2) 自己書換え

ct-rep($X_1, \cdots, X_n, \xi$)

命令図式 $\boldsymbol{in}(X_1, \cdots, X_n, \xi) = T \rightarrow \text{group}(X_1, \cdots, X_n, \xi)$ は $\boldsymbol{in}(X_1, \cdots, X_n, \xi) = \text{group}(X_1, \cdots, X_n, \xi)$ と書いてもよい.

ここで命令図式と関連する関数を定義しよう. 前に述べた一般的な形の命令図式と関連する関数は, つぎのように定義される.

$$\phi_{in}(X_1, \cdots, \xi, \tau, ri) =$$
$$P_1(X_1, \cdots, X_n, \xi) \rightarrow \text{group}_1^*(X_1, \cdots, X_n, \xi, \tau, ri)$$
$$\cdots\cdots$$
$$P_m(X_1, \cdots, X_n, \xi) \rightarrow \text{group}_m^*(X_1, \cdots, X_n, \xi, \tau, ri)$$

ここに, $\text{group}_i^*(X_1, \cdots, X_n, \xi, \tau, ri)$ $(1 \leqq i \leqq m)$ は次のような方法で $\text{group}_i(X_1, \cdots, X_n, \xi)$ から得られる.

・もし $\text{group}_i(X_1, \cdots, X_n, \xi)$ が形式(1)をもてば $\text{group}_i^*(X_1, \cdots, X_n, \xi, \tau, ri)$ の形式は

$\mu(\mu(\xi;\ \{\langle \chi_2 \circ (\tau \in \chi_1) \circ \text{s-c} : \varepsilon_0(X_1, \cdots, X_n, \xi)\rangle \mid \langle \chi_1, \chi_2 \rangle \in ri\});$
$\langle \textbf{s-sc}_1 : \varepsilon_1(X_1, \cdots, X_n, \xi)\rangle, \cdots,$
$\langle \textbf{s-sc}_r : \varepsilon_r(X_1, \cdots, X_n, \xi)\rangle)$.

である. (ただし, $\chi_1 - \chi_2 = (\iota\chi)(\chi_1 = \chi_2 \circ \chi)$)

・もし $\text{group}_i(X_1, \cdots, X_n, \xi)$ が形式(2)をもてば $\text{group}_i^*(X_1, \cdots, X_n, \xi, \tau, ri)$ の形式は

$\mu(\xi;\ \langle \tau \circ \text{s-c} : \mu(\text{ct-rep}(X_1, \cdots, X_n, \xi);\ \langle \text{s-ri} : ri\rangle)\rangle)$

である.

6.2.5 状態推移関数 Λ

6.2.1 で与えられた状態推移関数 Λ の定義, すなわち

$$\Lambda(\xi) = \{\psi(\xi, \tau) \mid \tau \in \text{tn}(\text{s-c}(\xi))\}$$

は関数 tn(ct) と $\psi(\xi, \tau)$ の形式的定義により完全なものにされる.

関数 $\text{tn}(ct)$ は $\text{is-c}(ct)$ なる ct に対して定義され, ct の終端節を指定するセレクタの集合を与える. この関数はつぎのように定義される.

$$\text{tn}(ct) = \{\tau \mid \tau \in \text{nd}(ct)\ \&\ (\forall \text{r})(r \in R \supset \text{r} \circ \tau(ct) = \Omega)\}$$

関数 $\psi(\xi, \tau)$ は $\tau \in \text{tn}(\text{s-c}(\xi))$ に対して定義され, $\text{s-c}(\xi)$ の $\tau(ct)$ の位置にある命令の実行に関して後続状態を与えるものである. この関数はつぎのように定義される.

$$\psi(\xi, \tau) = \phi_{in}(\text{elem}(1, al), \cdots, \text{elem}(n, al), \delta(\xi;\ \tau \circ \text{s-c}), \tau, ri).$$

ただし, $\boldsymbol{in} = \text{s-in} \circ \tau \circ \text{s-c}(\xi)$

$n = \boldsymbol{in}$ が伴うパラメタの数

$al = \text{s-al} \circ \tau \circ \text{s-c}(\xi)$

$ri = \text{s-ri} \circ \tau \circ \text{s-c}(\xi)$

6.2.6 例 題

命令図式 **int-expr** は, 命令 **int-expr**(e) が, 与えられた式 e を評価して, その結果の値をもどすというようなやり方で定義される. ここでいう式とは

・ある値を示す定数

・変数

・値の集合に関して定義される単項, 2項演算

などで形成されるものである.

式に対する抽象構文はつぎのように規定できる.

基本対象の集合:

$\hat{\text{is}}$-const	定数の集合
$\hat{\text{is}}$-var	変数の集合
$\hat{\text{is}}$-unary-rt	単項演算子の集合
$\hat{\text{is}}$-bin-rt	2項演算子の集合

式の集合:

is-expr = is-const $\vee$ is-var $\vee$ is-bin $\vee$ is-unary

is-bin = ⟨s-rd 1 : is-expr⟩, ⟨s-rd 2 : is-expr⟩, ⟨s-op : is-bin-rt⟩)

is-unary = (⟨s-rd : is-expr⟩, ⟨s-op : is-unary-rt⟩)

関数:

value(c)	この関数は与えられた定数 c の示す値を取り出す.
content(v, ξ)	この関数は機械の状態 ξ における与えられた変数 v の値を取り出す.

命令:

int-bin-op(op, a, b)	この命令は値 a, b に対して2項演算子 op を作用させた結果の値を返す.
int-un-op(op, a)	この命令は値 a に対して単項演算子 op を作用させた結果の値を返す.

ここまで規定すると命令図式 **int-expr** はつぎのように定義できる.

int-expr(e) =

is-bin(e)→**int-bin-op**(s-op(e), a, b);
a: **int-expr**(s-rd 1(e)),
b: **int-expr**(s-rd 2(e))
is-unary(e)→**int-un-op**(s-op(e), a);
a: **int-expr**(s-rd(e))
is-var(e)→PASS: content(e, ξ)
is-const(e)→PASS: value(e)

命令 **int-expr**(e) の実行効果が値戻し，あるいは自己書換えのいずれであるかは，この例ではアーギュメント e だけに依存し機械の状態には依存しない．

命令 **int-expr**(e) の4つの選択項目のうち初めの2つは自己書換えで，後の2つは指定された状態がさらに変更されることのない値戻しである．選択項目 is-var(e) が満たされたときもどされる値は，その時点の機械の状態 ξ と与えられた変数 e とによって決まる．

6.3 状態の構成と抽象機械の命令に関する注釈

6.3.1 一意名の生成

プログラムの解釈中，あるものに名前をつける必要のある場合がある．このために，一意名（すでに生成されている名前と異なる名前）を生成する機構を機械に導入する．

一意名を生成する機構は，つぎのように定義できる．まず，互いに異なる名前の無限リスト

$\langle n_1, n_2, n_3, \cdots \rangle$

があり，任意の状態 ξ に対して，つぎのような直属成分 s-n(ξ) がある．

s-n(ξ) は自然数で，つぎに使われるべき上記のリスト内の一意名を示すものである．

つぎに使用すべき一意名を取り出し，カウンタを上げるために命令 **un-name** が使われる．

un-name=
PASS: $n_{\text{s-n}(\xi)}$
s-n: s-n(ξ)+1

機械の状態と関係づけて一意名を使用するのは，つぎの2つの理由からである．

（1） 型の共有　いくつかの情報を表わすある対象を2つ以上の状態で使用できる場合がしばしばある．もしこの情報が，解釈中に変更されることがけっしてないならば，状態のそれぞれの部分で使うコピーを用意すれば十分である．しかし解釈の途中で，この情報が変更されるおそれがある場合は，情報を表わす対象を一意的に名づけると，この名前を仲介にして使用可能になる．以下の考察は型の共有についての簡単な例である．あるプログラムで，2変数 x, y が同一の記憶領域を占めるものとする．すなわち，x の値を変更することは，同時に y の値の変更をも意味し，その逆も成り立つものとする．これを図式で

x-value
y-value

と書く．これだけでは記憶領域の共有について十分表現していない．そこで間接的表現を一段設ければ，型の共有についての例として十分なものとなる．

x-n ＼
y-n ／ n-value

（2） 自己を参照する情報構造　ある対象の成分の1つにおいて，その対象の名前で自分自身を参照するような対象を考えよう．名前を対象自身のコピーで書き換える過程は終結しない．このような場合には，ある名前を使う必要がある．

簡単な例として関数 Fact の再帰的定義を考えよう．

Fact(n)=(n=0→1, T→n＊Fact(n-1))

右辺の Fact のこの関数定義に基づく書換えは，関数 Fact の参照として，何ら変わることはない．

Fact(n)=(n=0→1, T→n＊(n=0→1,
T→(n-1)＊Fact(n-2)))

6.3.2 対象で関数を表わすこと

A をある対象とし，その直属成分を $A_1, A_2, \cdots, A_n$ とする．すなわち

$A=\mu_0(\langle s_1 : A_1\rangle, \langle s_2 : A_2\rangle, \cdots, s\langle_n : A_n\rangle)$

関数 f_A をこのような対象と関連づけるには

$$f_A(s_j)=\begin{cases} A_i & 1\leqq i\leqq n \text{ の場合} \\ \Omega & \text{その他} \end{cases}$$

と定義すればよい．これらの関数は S 内のセレクタからある対象への写像である．

f_A: S→対象

上述の方法である対象と関連しているすべての関数は，変域がセレクタの集合で，アーギュメントの有限集合に対してだけ Ω 以外の値を生ずるような関係である．

機械の状態は関連する対象によって表わされるこのようないくつかの写像を含んでいる．仮定すべきことはもちろん，このような写像の変域がセレクタの集合の部分集合であるということである．

つぎに適用例を2つ示そう．

（1） **環境成分**（Environment）

この節で述べる環境成分の説明は 5. 章の例で必要となる．状態 ξ の環境成分 s-env(ξ) は識別子（これ

は与えられた状態の中で参照される）から一意名への写像である．環境を対象として表現するために，プログラム内のすべての識別子をセレクタの集合 S の要素とみなす．いま，もしある環境に対して識別子 $id_1, id_2, \cdots, id_m$ がそれぞれ $n_1, n_2, \cdots, n_m$ に写されるものとすれば，写像は ENV という対象で表現される．

$$ENV=\mu_0(\langle id_1 : n_1\rangle, \langle id_2 : n_2\rangle, \cdots, \langle id_m : n_m\rangle)$$

与えられた識別子 id_i に対する一意名を取り出すには，環境に識別子を適用すればよい．すなわち

$$id_i(ENV)=n_i.$$

環境は任意の状態 ξ の直属成分 s-env(ξ) であるから，与えられた状態 ξ に関して，どの一意名が与えられた識別子と関連しているかという問題は，次式で答えられる．

$$id_i \circ \text{s-env}(\xi)$$

環境として扱われる対象すべてを述語 is-env で定義することができる．

$$\text{is-env}=(\{\langle id : \text{is-n}\rangle \mid \text{is-name}(id)\})$$

ここに，$\hat{\text{is-n}}$ は一意名すべての集合であり，$\hat{\text{is-name}}$ はあらゆる識別子の集合である．

（2） **登録簿** (Directory)

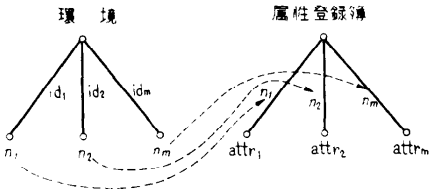
機械の状態の1つの成分は登録簿である．登録簿とは一意名からある対象への写像である．一意名とそれと関連した対象との対を登録簿に関する**記載事項**という．登録簿を表現している対象は，常につぎのような構造をしている．すなわち，一意名はセレクタとして働き，このセレクタを登録簿に適用すると，それと関連する対象を生じる．

たとえば，いわゆる属性登録簿を考えよう．これは一意名とそれに対応する識別子に対して宣言された属性との対から構成されている．識別子 id とある状態 ξ を与えると，次式によってこの識別子の属性を取り出すことができる．

$$(id \circ \text{s-env}(\xi))(\text{s-at}(\xi))$$

これは，まず状態 ξ の直属成分 s-env(ξ) を取り出し，それに識別子 ie を適用して，一意名 $id \circ$s-env(ξ) を得る．この一意名を状態 ξ の直属成分である属性登録簿 s-at(ξ) に適用して id に対して宣言された属性を得る．

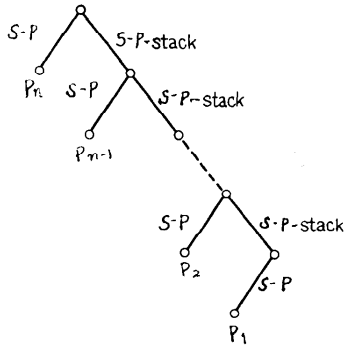
この例では，一意名が2つの役割を果たしていることに注意する必要がある．1つは一意名が環境の成分として使われており，もう1つは，属性登録簿に関するセレクタとして使われている．第6.4図はこの関係を示している．



第 6.4 図

6.3.3 スタックの実現

PL/I のような入れ子構造を持っているプログラミング言語を解釈するために，スタックは重要な役割を果たす．スタックとは $P_1, P_2, \cdots, P_{n-1}, P_n$ なる要素 (P_n をトップの要素という）の並びのことである．スタックを取り扱う演算は2つある．1つは新しい要素 P_{n+1} をスタックのトップに加えるプッシュダウン演算であり，もう1つはスタックからトップの要素を削除するポップアップ演算である．これはスタックをトップの要素と残りの部分とに分けて，これらを直属成分として持つ対象でスタックを表わすのが有効であることを意味している．つぎの図式はスタックを対象として表わすのに使われる．



第 6.5 図

セレクタ s-p はスタックのトップの要素に対して使われ，セレクタ s-p-stack はスタックの残りの部分に対して使われている．

スタック演算は，つぎのように公式化できる．

プッシュダウン：

$$\mu_0(\langle \text{s-p} : p_{n+1}\rangle, \langle \text{s-p-stack} : ST\rangle)$$

ポップアップ：s-p-stack(ST)

あゆるスタックの類 $\hat{\text{is-p-stack}}$（これは $\hat{\text{is-}P}$ の要素から形成される）のメンバは次式で定義できる．

$$\text{is-p-stack}=(\langle \text{s-p} : \text{is-p}\rangle,$$

〈s-p-stack : is-p-stack〉)∨is-Ω

6.5.4 命令定義内の状態成分を参照すること

補助的命令を定義しておけば，これが省略形を導入する手段となるばかりでなく，計算の中で状態への参照がある場合に効果的であるということを述べよう．

命令 $\mathbf{in}_1$ のかなり似通った2つの定義を例として考える．

(1) $\mathbf{in}_1$=
$\mathbf{in}_2$;
$\mathbf{in}_3$
$\mathbf{in}_2$=
$\mathbf{in}_4$(s-p(ξ))

ここに，s-p(ξ) は状態 ξ のある成分を参照している．

(2) $\mathbf{in}_1$=
$\mathbf{in}_5$(s-p(ξ));
$\mathbf{in}_3$
$\mathbf{in}_5$(a)=
$\mathbf{in}_4$(a)

いま，計算の途中で $\mathbf{in}_1$ が実行されようとしていると仮定すれば，問題は上記(1),(2)の定義の差異を調べることである．

両者とも，命令は $\mathbf{in}_1, \mathbf{in}_3, \cdots, \mathbf{in}_4$ の順に実行するけれども，参照される状態は異なる．(1)の場合，ξ は $\mathbf{in}_3$ 実行後の状態であり，(2)の場合，ξ は $\mathbf{in}_3$ 実行前の状態である．

6.3.5 空命令

問題は任意の順序で実行する命令 $\mathrm{instr}_1, \mathrm{instr}_2, \cdots, \mathrm{instr}_n$ に対する制御木を構成することである．これらの命令が何も値をもどさず，共通の補助的対象を構成すると仮定する．このような問題に対して，正確な制御木を構成するためには，制御木から自分自身を削除する他に，何もしないような命令を考える必要がある．この命令を空命令と呼び

null=
PASS: Ω

と定義する．

上の問題はこれを使って，つぎのように解決される．

null; {$\mathrm{instr}_1, \mathrm{instr}_2, \cdots, \mathrm{instr}_n$}

6.5.6 パス命令

値として x をもどす命令図式 **pass** を定義する．

pass(x)=
PASS: x

つぎのような問題がしばしば起こる．任意の順序で実行される数個の命令によって，成分が計算される場合，補助的対象が構成するが，渡されるべきものはその対象ではなく，それに関数 f を適用した結果である．

制御木内でダミーの名前が使えるのは，命令のアーギュメント部のみであるから，命令 **pass** だけではこの問題の解決には十分でない．そこで，下記のような特定の省略形を導入すれば，どんな関数 f に対しても，補助的命令を定義せずに済む．

pass-f(x)=
PASS: $f(x)$

ここに，f は x に対して適用可能な任意の関数で書き換えてもよい．

もし，命令 $\mathrm{instr}_1, \mathrm{instr}_2, \cdots, \mathrm{instr}_n$ が補助的対象の $\chi_1, \chi_2, \cdots, \chi_n$ 成分を計算し，f_1 がその対象に適用できる関数ならば，つぎのような制御木を構成することができる．

pass-f_1(x); {$\chi_1(x)$: instr_1, $\chi_2(x)$: $\mathrm{instr}_2, \cdots$,
$\chi_n(x)$: instr_n}

6.3.7 リストの要素ごとの評価

ここでは，自然な順序でリストを要素ごとに評価する方法を示そう．空リストも1つのリストと考え，その評価結果はやはり空リストであるとする．さらに，命令 **eval**(el) は特定の要素 el を評価し，その結果の値を返す命令であると仮定すると，与えられた問題を解くための命令図式 **eval-list** はつぎのように定義できる．

eval-list($list$)=
is-〈 〉($list$)→PASS : 〈 〉
T→**mk-list**(eh, et);
et : **eval-list**(tail($list$));
eh : **eval**(head($list$))

ここに，**mk-list**(x, list)=
PASS: 〈x〉∧$list$

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, 28, 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 the formal definition of PL/I. SJCC 1968, pp. 363-373.
7) Lauer, P.: Abstract syntax and interpretation of ALGON 60.—IBM Lab. Viema, Lab-Report LR 25. 6. 001, 12, April 1968.

(昭和42年2月3日受付)