PL/I の形式的定義について (2)*

情報処理学会 PL/I 研究委員会**

5. 抽象構文への適用と言語の表示法

以下の章でいままでに与えられた手段の言語への適用法を EPL の構文と意味を定義することによって示す．構文の2つの基本的なタイプ，抽象構文と具象構文には，言語の2つの定義法が対応する．

言語の構文が抽象構文によって記述されると仮定すると，言語は，抽象構文によって記述される式に，直接意味を付け加えることによって解釈される．しかしこの場合，式の意味が何であるかだけでなく，その式に対する具象表現を文字列として，いかに記述するかが問題になる．

一方，言語の構文が具象構文で記述されるなら，具象表現の問題は，具象構文自身が，この表現を定義しているので生じない．しかし，言語の式の意味の問題は，まだ解かれずに残っており，この場合は構文によって作られる文字列に意味をつけ加えることによって，解釈が完全に与えられる．複雑なプログラミング言語の意味の形式的な定義の場合には，このような直接的な解釈はむしろ実際的でなく，意味を定義しようとする前に，具象構文によって記述される式をある種の抽象標準形へ翻訳する形式をとるのが一般的である．

5.1 EPL の抽象構文

抽象構文を記述する方法として，前章で述べた対象の類の定義を利用する．この目的のために，集合 EO と S の規定，**4.6** で述べた定義図式による述語の規定，および定義すべき言語の式の集合，すなわち，対象の類に対応する特定の述語を決める．

EPL の抽象構文を定義する目的は，EPL のプログラムと同一視できるような対象の類を定義することである．すなわち，対象がそれに対応するプログラムの構造をそのまま反映していなければいけない．そのため，対象の構造の選定には，本質的に任意性はない．しかし，特有なセレクタ，述語の名前，および基本対象を表わすための記号の選定は任意であり，覚えやすさのみを考慮すればよい．

EPL で書くことができるプログラムの集合を表わす対象の集合を is-prôgr と表わすことにする．

表記上の約束として，すべての述語は前置語 "is-" を持ち，セレクタとして使われている識別子を除いたすべてのセレクタは，前置語 "s-" を持つ．

is-id	識別子の無限集合
is-log	真偽値を示す定数の集合
is-int	整数値を示す定数の無限集合
is-binary-rt	2項演算子の集合
{INT, LOG}	整数変数と論理変数を区別するために使われる2つの属性

これらの集合は，相互に排他的であると仮定される．集合 EO は上にあげたものの和集号である．

EO = is-id ∪ is-log ∪ is-int ∪ is-binary-rt ∪ {INT, LOG}

EPL の抽象構文の記述に必要なセレクタ s の集合は，無限集合である．しかしながら，識別子と対応しないセレクタの集合は有限で，実際少ししかないため列挙することができ，つぎのように集合 S を定義できる．

S = {s-decl-part, s-st-list, s-param-list, s-st, s-expr, s-left-part, s-right-part, s-id, s-arg-list, s-op, s-rd, s-rd1, s-rd2, s-then-st, s-else-st} ∪ is-id.

EPL の抽象構文の記述に必要な述語は，

(A 1) is-progr = is-block

(A 2) is-block = (<s-decl-part : is-decl-part>, <s-st-list : is-st-list>)

(A 3) is-decl-part = ({<id : is-attr> | is-id(id)})

(A 4) is-attr = is-var-attr ∨ is-funct-attr

(A 5) is-vâr-attr = {INT, LOG}

(A 6) is-funct-attr = (<s-param-list : is-id-list>, <s-st : is-st> <s-expr : is-expr>)

(A 7) is-st = is-assign-st ∨ is-cond-st ∨ is-block

---

* On the formal definition of PL/I (2), a report of the PL/I research committee of the ISPJ

** 情報処理 Vol. 11, No. 8, p. 457 参照

(A 8) is-assign-st=(<s-left-part : is-var>,
<s-right-part : is-expr>)

(A 9) is-expr=is-const∨is-var∨is-bin∨
is-funct-des

(A10) is-const=is-log∨is-int

(A11) is-var=is-id

(A12) is-funct-des=(<s-id : is-id>,
<s-arg-list : is-id-list>)

(A13) is-bin=(<s-rd 1 : is-expr>,
<s-rd 2 : is-expr>,
<s-op : is-bin-rt>)

(A14) is-cond-st=(<s-expr : is-expr>,
<s-then-st : is-st>,
<s-else-st : is-st>)

EPL のプログラムは is-progr のメンバと同一である.

EPL のプログラムの抽象構文を定義する場合，要素や特別の句読号の順序に関しての指定は何も必要としない.

ある類が相互に排他的であるということは，後で抽象構文を使用するために重要である．それゆえ，結果に影響させずに任意の順序で与えられた式が定数か，変数か，2項式であるかどうか知ることができる.

5.2 具象表現の定義

抽象構文によって記述される式に対する具象表現を定義することは，有限の長さを持った1つ以上の文字列を言語の式と関連させることを意味している．したがって，いままで言語の式を，対象と同一にみなしてきたので，表現を定義することは，対象と文字列をとくに関連させることと考えられる.

もし，与えられた文字列が式の表現であるかどうか決定が可能であり，さらに式がその表現によって一意に決定されるなら，その表現は意味がある．これらの条件を満足する表現は"あいまい"でないと呼ばれる.

5.2.1 表現方式 言語に対する表示方式を記述する第1のステップは，**終端記号**のアルファベットを与えることである．このアルファベットは2つの集合の和集合と考えられる．初めの集合の要素は，言語の抽象構文の基本対象を一意に表わし，この集合は一般に無限集合である．もう1つの集合は，区切り記号の有限集合であり，列として表わされた言語の対象の構造を反映するのに用いられる．ここでは，終端記号は原子文字として考えられる.

終端記号による基本対象の一意な表現 eo を関数 rep(eo) で定義し，区切り記号の集合を $\mathcal{J}$ によって示す．そのとき終端記号$\mathcal{T}$の集合はつぎのように定義される.

$$\mathcal{T}=\{\mathrm{rep}(eo)\mid eo\in EO\}\cup\mathcal{J}$$

終端式は式の中で生じる自由変数に関係した終端記号を示す任意のメタ式である．終端記号は終端式の特別な場合であると考えられる.

表現方式の書換え図式を公式化するために，非終端および非終端式の概念をつぎのように導入する.

非終端は非終端名と対象であるアーギュメントのリスト（空も可）からなる．非終端はつぎの形をしている.

NONTERM $[ob_1,\cdots,ob_n]$

すなわち，**非終端名**は下線を引いた太文字でアーギュメントリストはかっこでかこまれている（アーギュメントが空の場合はかっこが省略される）.

非終端式は式の中で生じる自由変数に関係する非終端を示すメタ式であり，つぎのように書かれる.

NONTERM $[expr_1,\cdots,expr_n]$

ここで，$expr_1,\cdots,expr_n$ は対象を示す任意のメタ式である.

各非終端名 ***NONTERM*** に対し，つぎの一般形を持つ条件付き書換え図式がいくつか存在する.

$$P(X_1,\cdots,X_m,t_1,\cdots,t_n):\ \textbf{\textit{NONTERM}}\ [t_1,\cdots,t_n]\Rightarrow r(X_1,\cdots,X_m,t_1,\cdots,t_m)$$

$t_1,\cdots,t_n$ はパラメータで，$X_1,\cdots,X_m$ はこの規則の補助変数である．Pはメタ式でこの規則の条件と呼ばれ，自由変数 $X_1,\cdots,X_m,t_1,\cdots,t_n$ によって決まる真偽値を示す．r は自由変数 $X_1,\cdots,X_m,t_1,\cdots,t_n$ の非終端式と終端式からなる列を示す.

上のタイプの図式はつぎのような意味を持つ，変数 $X_1,\cdots,X_m,t_1,\cdots,t_n$ の値として特定の対象 $X_1^\circ,\cdots,X_m^\circ,t_1^\circ,\cdots,t_n^\circ$ が与えられたとき，もし，条件Pが満足されるなら，与えられた列内にそれが現われた場合，非終端 ***NONTERM*** $[t_1^\circ,\cdots,t_n^\circ]$ は，列 $r(X_1^\circ,\cdots,X_m^\circ,t_1^\circ,\cdots,t_n^\circ)$ によって書き換えられる.

特別の場合として，条件に無関係な図式の形は，

NONTERM $[t_1,\cdots,t_n]\Rightarrow r(t_1,\cdots,t_n)$

表現方式は5組ベクトル$<\mathcal{A},\mathcal{T},\mathcal{N},\mathcal{H},\mathcal{R}>$によって定義される．ここで$\mathcal{A}$は表現される言語の抽象構文，$\mathcal{T}$は終端記号の集合，$\mathcal{N}$は非終端名の集合，$\mathcal{H}$は$N$に含まれる一つの非終端名であり，この表現方式の出発点を表わす．そして$\mathcal{R}$は条件付き書換え図式の集合である.

抽象構文によって記述されるそれぞれの抽象式t^0に関し，少なくとも1つのt^0の具象表現が書き換え過程によってみつけられる．**書換え過程**は列 $r_0, r_1, \cdots, r_k$ の系列である，ここでr_0は非終端 $\mathcal{H}[t^0]$で，r_{i+1}は$\mathcal{R}$の条件書換え図式の適用によって$r_i (0 \leqq i \leqq k)$から得られる．系列の最後の列，r_kは終端記号のみからなり，t^0の具象表現を構成する．

5.2.2 EPL のプログラムの具象表現方式

基本対象，たとえば，識別子・定数・演算子へ適用されるとき，対応する終端記号を生じる関数 rep は，つぎのように定義される†．

rep(t) =
- is-id(t) → rep-id(t)
- is-const(t) → rep-const(t)
- is-bin-rt(t) → rep-rt(t)
- t=INT → **INTEGER**
- t=LOG → **LOGICAL**

EPL のために必要とされる書換え方式を構成する5組ベクトルのメンバを以下に定義する．

非終端名の集合Nは要素を列挙することによって定義される．$\mathcal{H}$は，初めの非終端名であり，それゆえ

$\mathcal{H}$=**PROGR**

(N1)	**PROGR**	プログラム
(N2)	**BLOCK**	ブロック
(N3)	**DECL**	宣言部分
(N4)	**STL**	文リスト
(N5)	**ST**	文
(N6)	**EXPR**	式
(N7)	**ID**	識別子
(N8)	**ID′**	識別子

終端記号の集合$\mathcal{T}$は区切り記号の集合と関数 rep の値域の和集合により定義される．

$\mathcal{T}$ = {**BEGIN, END, FUNCTION, IF, THEN, ELSE, RETURNS,**

(,), ;, ,, =} ∪ {rep(t) | $t \in EO$}

終端記号は下線が引いてないので，下線が引いてある非終端名とはっきり区別できる．例外はかっこ，セミコロン，カンマと等号で，それ自身を表わす終端記号を示すため太く書かれる．

† 関数 rep-id rep-const, rey-rt は1対1である．rep-id の変域は識別子の集合 $\hat{s}$id で，値域は具象構文によって記述される識別子の集合である．rep-const の変域は集合 is-log と is-int の和集合で，値域は具象構文によって記述される論理定数と整数定数の集合の和集合である．rep-rt の変域は集合 is-bin-rt で，値域はある具象構文によって記述されるような2項演算子の集合である．

抽象構文$\mathcal{A}$は，定義 (A1)～(A14)からなる．

条件付き書換え図式Rの集合の図式は，つぎに列挙される．

(R1) **PROGR** 〔t〕 ⇨ **BLOCK** 〔t〕

(R2) **BLOCK** 〔t〕 ⇨ **BEGIN**
DECL 〔s-decl-pt(t)〕
STL 〔s-st-list(t)〕**END**

(R3) $t=\Omega$: **DECL** 〔t〕 ⇨ λ
is-var-attr∘id(t) : **DECL**〔t〕
⇨ rep∘id(t)rep(id);
DECL 〔$\delta(t$; id)〕
is-funct-attr∘id(t) : **DECL**〔t〕
⇨ **FUNCTION** rep(id)
ID 〔s-param-list∘id(t)〕;
ST 〔s-st∘id(t)〕
RETURNS
EXPR 〔s-expr∘id(t)〕;
DECL 〔$\delta(t$; id)〕

(R4) length$(t)=0$: **ID**〔t〕⇨λ
length$(t)>0$: **ID**〔t〕⇨(**ID′**〔t〕)

(R5) length$(t)=1$: **ID′**〔t〕
⇨ rep∘head(t)
length$(t)>1$: **ID′**〔t〕
⇨ rep∘head(t) ; **ID′**〔tail(t)〕

(R6) length$(t)=1$: **STL**〔t〕
⇨ **ST**〔head(t)〕
length$(t)>1$: **STL**〔t〕
⇨ **ST**〔head(t)〕; **STL**〔tail(t)〕

(R7) is-assign-st(t) : **ST**〔t〕
⇨ rep∘s-left-part(t) =
EXPR 〔s-right-part(t)〕
is-cond-st(t) : **ST**〔t〕
⇨ **IF EXPR** 〔s-expr(t)〕
THEN ST 〔s-then-st(t)〕
ELSE ST 〔s-else-st(t)〕
is-block(s) : **ST**〔t〕
⇨ **BLOCK** 〔t〕

(R8) is-const(t) : **EXPR**〔t〕 ⇨ rep(t)
is-var(t) : **EXPR**〔t〕 ⇨ rep(t)
is-funct-des(t) : **EXPR**〔t〕
⇨ rep∘s-id(t) **ID** 〔s-arg-list(t)〕
is-bin(t) : **EXPR**〔t〕
⇨ (**EXPR** 〔r-rd1(t)〕

rep∘s-op(t) **EXPR** [s-rd2(t)])

以上により，集合 is-progr のメンバ（EPL のプログラム）の可能な表現の集合を順に構成する書換え方式は完全に定義される．

5.3 抽象構文と Backus 標準形との相互関係

ここでは，Backus 標準形による構文の定義（具象構文と呼ばれる）と抽象構文による構文の定義との相互関係について述べる．構文の定義は終端記号の集合アルファベット T，T 上の列の集合を規定する規則の集まり，および列の集合を1つ選ぶことによって与えられる．この選ばれた列の集合が，定義される言語の良形な式の集合となる．

$\alpha, \alpha_1, \alpha_2, \cdots\cdots$アルファベット$T$の任意の要素
$M, M_1, M_2, \cdots\cdots$列集合のための任意な名前
$W, W_1, W_2, \cdots\cdots T$に関する任意の列
$W_1 \cap W_2 \cap \cdots\cdots W_1$ と W_2 の連結

構文定義の規則は，つぎの特別な形を仮定する．

(1) $M=\{\alpha\}$

(2) $M=M_1 \cup M_2 \cup \cdots\cdots \cup M_n$

(3) $M=M_1 M_2 \cdots\cdots M_n$

ここで $M_1 M_2 \cdots\cdots M_n = \{W_1 \cap W_2 \cap \cdots \cap W_n \mid W_1 \in M_1 \ \& \ W_2 \in M_2 \ \& \cdots \& \ W_n \in M_n\}$

Backus 標準形による定義は2つの目的がある．第1の目的は，列の集合（良形の式）を定義することであり，第2は，列に対し，それぞれの句構造を定義することである．同じ目的を達成するために，抽象構文の定義図式に対しても，同様な見方をすることによって，相互関係が明らかとなる。形式化するには，集合 s に対する順序の定義を付け加えればよい．すなわち，順序の関係 $s_1 < s_2$ はセレクタ s_1, s_2 の対に対して定義される．

セレクタに対して順序が定義されるならば，複合セレクタ $x = s_n \circ s_{n-1} \circ \cdots\cdots s_1$ に対し，すぐに順序を関連させることができる．すなわち，最上位を s_1，最下位を s_n とするアルファベット的な順序である．

$\{<x_1 : e_1>, <x_2 : e_2>, \cdots, <x_n : e_n>\}$

と $x_1 < x_2 < \cdots < x_n$ によって特色づけられる対象を考えると，関連ある記号の列は $e_1 \cap e_2 \cap \cdots \cap e_n$ であると定義され，与えられた対象の終端列と呼ばれる．このくふうによって，抽象構文は，一方では列の集合（記号の集合 EO に関して）を定義し，他方では各対象の構造によって，それぞれの終端記号に対し構造を定義する．

Backus 標準形での各構文定義に対し，対応する抽象構文を定義することができる．

具象構文がアルファベット A，列の集合に対する名前の集合，そして規則の集合によって与えられるとする．対応する抽象構文は，そのときつぎのように定義される．

(a) $EO=T$

(b) セレクタのある順序づけられた集合 s:

(c) 具象構文の中の集合のそれぞれの名前 M に対し，抽象構文における述語 P_M を対応づけ，$M_2 \neq M_2$ なら $P_{M_1} \neq P_{M_2}$ であるようにする:

(d) $M=\{\alpha\}$ の形の規則は $P_M=\{\alpha\}$ に変換する:

(e) $M=M_1 \cup M_2 \cup \cdots\cdots \cup M_n$ の形の規則は $P_M = P_{M_1} \vee P_{M_2} \vee \cdots\cdots \vee P_{M_n}$ に変換する:

(f) $M=M_1 M_2 \cdots M_n$ の形の規則は $P_M = (<s_1 : P_M>, <s_2 : P_{M_2}>, \cdots, <s_n : P_M>)$ に変換する．ここで s_1 は最初のセレクタで $s_2, \cdots, s_n$ は s に与えられた順序による直属後行成分である:

(g) もし，M が具象構文の先頭であるなら，P_M は抽象構文の先頭である．

具象構文によって定義される列の集合は，対応する抽象構文によって定義される終端列の集合と同一である．具象構文によって定義される句構造は，それぞれの規則が1対1であるため，対応する抽象構文によって定義される句構造と同一である．与えられた列の具象構文に対する解剖木の構造は，余分な節を除けば，それぞれの対象の構造と同じである．

5.4 EPL の具象構文

EPL の具象構文を述べる前に，よく知られている Backus 標準形をいくつかの都合のよい簡略記法を加えることによって拡張する．

5.4.1 拡張された Backus 記法 一般に，Backus 標準形は，つぎのような形で表現される．

$V ::= S_1 | S_2 | \cdots | S_n |$

（V は $S_1, S_2, \cdots, S_n$ の中のどれか1つと置き換えられる）

以下の定義により，メタ言語記号である '{', '}', '[', ']', '.' が導入される．ただし，

U 任意の構文単位（終端記号または非終端記号）
V 非終端記号
S_i 任意の列

T_j 空でない列

(1) $V::=S_1T_1S_2|S_1T_2S_2|\cdots|S_1T_nS_2$
はつぎのものと同等である.
$V::=S_1\{T_1|T_2|\cdots|T_n\}S_2$
$(n=1,2,\cdots)$

(2) $V::=S_1S_2|S_1T_1S_2|\cdots|S_1T_nS_2$
はつぎのものと同等である.
$V::=S_1[T_1|T_2|\cdots|T_n]S_2$

(3) $V::=U|VU$ または $V::=U|UV$
はつぎのものと同等である.
$V::=U...$
(下の形から上の形へ書き換える場合に,'$V::=U...$' なる形の生成則をもたない文法の中に'$U...$' が現われた場合は,まだ使われてない非終端記号Vを,その文法につけ加えてから書き換えればよい.).

(4) $V::=S_1T_1[\{T_2T_1\}...]S_2$
はつぎのものと同等である.
$V::=S_1\{T_2.T_1...\}S_2$
('$S_1[\{T_2.T_1...\}]S_2$' は '$S_1[T_2.T_1...]S_2$' と書き換えられる.).

5.4.2 具象構文 上述の拡張された Backus 記法を使って, EPL の具象構文を以下に述べる.

(C 1) program::=
block

(C 2) block::=
BEGIN declaration-list statement-list END

(C 3) declaration-list::=
{;.declaration...}

(C 4) declaration::=
variable-declaration|function-declaration

(C 5) variable-declaration::=
{INTEGER|LOGICAL} variable

(C 6) function-declaration::=
FUNCTION identifier [parameter-list] ; statement RETURNS expression

(C 7) parameter-list::=
({,.identifier...})

(C 8) statement-list::=
{;.statement...}

(C 9) statement::=
assignment-statement|conditional-statement|block

(C10) assignment-statement::=
identifier=expression

(C11) expression::=
constant|variable|function-designator|binary

(C12) constant::=
logical-constant|integer-constant

(C13) variable::=
identifier

(C14) function-designator::=
identifier[argument-list]

(C15) argument-list::=
({,.identifier...})

(C16) binary::=
(expression binary-operator expression)

(C17) conditional-statement::=
IF expression THEN statement ELSE statement

EPL のプログラム例

```
BEGIN INTEGER X;
    FUNCTION RENRITSU 1
          (A1, A2, B1, B2, C1, C2);
    BEGIN INTEGER P;
        P=((A1*B2)-(A2*B1));
        IF P=0 THEN X=0;
        ELSE X=(((C1*B2)-(C2*B1))/P)
    END
    RETURNS X;
    .....
END
```

5.5 具象プログラムから抽象プログラムへの翻訳

具象プログラム(具象構文によって記述されるプログラム)から**抽象プログラム**(抽象構文によって記述されるプログラム)への翻訳を考える際に,**4.**で述べた方法と概念を適用するため,まず,具象プログラムは**文字の値** (character value) のリストであると考える.ここで文字の値とは,具象プログラムを記述するのに使われている具象文字と,1対1に対応する抽象基本対象であり,EPL におけるこの1対1対応は,**第5.1表**で示されるとおりである.

第5.1表

具象文字	対応する文字の値によって満たされる述語
BEGIN	is-BEGIN
END	is-END
FUNCTION	is-FUNCT
INTEGER	is-INT
LOGICAL	is-LOG
IF	is-IF
THEN	is-THEN
ELSE	is-ELSE
RETURNS	is-RETURNS
(	is-LEFT-PAR
)	is-RIGHT-PAR
;	is-SEMIC
,	is-COMMA
=	is-EQ
identifir	is-c-id
integer-constant	is-c-int
logical constant	is-c-log
binary operator	is-c-bin-op

これらの述語に対してはただ1つの基本対象が存在する

具象プログラムから抽象プログラムへの翻訳は，2つの関数 parse と translate によって2段階にわたってなされる．すなわち，関数 parse により具象プログラム(txt)から抽象表示（txtの解剖木と考えられる.）が生成され，つぎに関数 translate により抽象プログラム（translate∘parse(txt)）が生成される．

5.5.1 具象プログラムの抽象表示 (abstract representation)　具象プログラムの抽象表示 t は述語 is-c-program を満たす対象である．t の構造は具象プログラムの構文上の構造を反映し，その基本成分は具象プログラムを構成する文字の値である．述語 is-c-program は具象プログラムの生成則を書き換えることによって得られる一群の述語の定義によって記述される．

これらの述語の定義を公式化する際に，標準的なセレクタ s_1, s_2, … が使われる．それらは互いに異なり，セレクタ関数 s の値 $s(i)$ として考えられる．セレクタ s_i によって対象が形成され，それはセレクタ elem(i) によって形成されるリストと同じような構造を持つ．これらの s-list に対して，関数 **slength** を定義する．

$$\text{slength}(X) = (\forall_i)(\text{is-}\Omega\circ s_i(X))\to 0\quad T\to(\iota j)(\text{is-}\Omega\circ s_i(X)\,\&\,(\forall j)(j>i\supset \text{is-}\Omega\cdot s_j(X)))$$

さらに，**4.** で導入された定義図式(1)～(5)に加えて，定義図式(6)を付加する必要がある．

(6) $$\text{is-pred}(X) = (\exists X_1,\cdots,X_n,m,y_1,\cdots,y_m)\left(m\geq 1\ \&\ \overset{m}{\underset{i=1}{E}}t\,\text{is-pred}_i(X_t)\ \&\ \overset{m}{\underset{i=1}{E}}t\,\text{is-pred}_0(y_t)\ \&\ X=\mu_0(\{<\text{s-sel}_1:X_1>,\cdots,<\text{s-sel}_n:X_n>,<\text{sel-fct}(1):y_1>,\cdots,<\text{sel-fct}(m):y_m>\})\right)$$

sel-fct は，たとえば elem や s のような，整数値をセレクタに写すセレクタ関数である．$n=0$ とした省略形がしばしば使われる．すなわち

$$\text{is-pred} = (<\text{sel-fct}(1):\text{is-pred}_0>,\cdots)$$

この記法を使って is-pred-list をつぎのように定義する．

$$\text{is-pred-list} = \text{is-}<> \vee (<\text{elem}(1):\text{is-pred}>,\cdots)$$

EPL の抽象表示

以下の述語の定義は**第 5.1 表**を使って **5.4.2** の（C 1)～(C 17)を機械的に書き換えることによって得られる．

(AR 1) is-c-progr = is-c-block

(AR 2) is-c-block = (<s_1 : is-BEGIN>, <s_2 : is-c-decllist>, <s_3 : is-SEMIC>, <s_4 : is-c-stlist>, <s_5 : is-END>)

(AR 3) is-c-decllist = (<s-del : is-SEMIC>, <s_1 : is-c-decl>, …)

(AR 4) is-c-decl = is-c-var-decl ∨ is-c-funct-decl

(AR 5) is-c-var-decl = (<s_1 : is-INT ∨ is-LOG>, <s_2 : is-c-var>)

(AR 6) is-funct-decl = (<s_1 : is-FUNCT>, <s_2 : is-c-id>, <s_3 : is-c-par-list v is-Ω>, <s_4 : is-SEMIC>, <s_5 : is-c-st>, <s_6 : is-RETURNS>, <s_7 : is-c-expr>)

(AR 7) is-c-parlist = (<s_1 : is-LEFT-PAR>, <s_2 : (<s-del : is-COMMA>,

$<s_1$: is-c-id>, ···) >,
$<s_3$: is-RIGHT-RAR>)

(AR 8) is-c-stlist = (<s-del : is-SEMIC>,
$<s_1$: is-c-st>, ···)

(AR 9) is-c-st = is-c-assign-st ∨ is-c-cond-st
∨ is-c-block

(AR 10) is-c-assign-st = (<s_1 ·· is-c-id>,
<s_2 : is-EQ>,
<s_3 : is-c-expr>)

(AR 11) is-c-expr = is-c-const v is-c-var v
is-funct-des v is-c-bin

(AR 12) is-c-const = is-c-log v is-c-int

(AR 13) is-c-var = is-c-id

(AR 14) is-c-funct-des = (<s_1 : is-c-id>,
<s_2 : is-c-arglist v
is-Ω>)

(AR 15) is-c-arglist = (<s_1 : is-LEFT-PAR>,
<s_2 : (<s-del :
is-COMMA>,
<s_1 : is-c-id>, ···) >,
<s_3 : is-RIGHT-PAR>)

(AR 16) is-c-bin = (<s_1 : is-LEFT-PAR>,
<s_2 : is-c-expr>,
<s_3 : is-c-bin-rt>,
<s_4 : is-c-expr>,
<s_5 : is-RIGHT-PAR>)

(AR 17) is-c-cond-st = (<s_1 : is-IF>,
<s_2 : is-c-expr>,
<s_3 : is-THEN>,
<s_4 : is-c-st>,
<s_5 : is-ELSE>
<s_6 : is-c-st>)

関数 generate と関数 parse

具象プログラムに対して，その解剖木（抽象表示）を生成する構文解剖機を直接定義するのは大変なので，最初に抽象表示を具象プログラムに写す関数 generate を定義し，つぎに，関数 parse を関数 generate の逆関数として暗黙的に定義する．

generate(t) =
is-Ω(t) → < >
slength(t) = 0 → <t>
T ← generate∘s_1(t) $\overset{\text{slength}(t)}{\underset{i=2}{\cap}}$CONC generate∘s-del($t$) ∪
generate∘s_i(t))

parse(txt) =
(ιt)(txt = generate(t) & is-c-program(t))

(注) s-del リストの区切り記号を選択する特別なセレクタ．

5.5.2 翻訳機 (translator) この節は具象プログラムの抽象表示から抽象プログラムへの翻訳について述べる．この翻訳はつぎの関数によってなされる．

translate(t)

この関数は具象構文の抽象表示によって述べられる述語 is-c-program を満たす対象 t を抽象構文によって記述される述語 is-program を満たす対象に写す．

7. の抽象プログラムの解釈 (interpretation) の場合におけると同様に，ここでは翻訳をだんだんにプログラムテキスト t の各要素の翻訳に置き変えていく．しかしながら，解釈機と翻訳機の概念の間には2つの基本的な違いが存在する．

(a) 翻訳機は抽象表示で表現された具象プログラムを対応する抽象プログラムに写す関数によって定義されるのに対し，解釈機は機械の状態をつぎの状態に写す命令によって定義される．

(b) EPL よりも複雑なプログラミング言語（たとえば，PL/I）のプログラムの一部を翻訳することは，このプログラムテキストの一部ばかりでなく，それが完全なプログラムテキスト内で出現している文脈に依存する．一方，プログラムの一部を解釈することは，このプログラムの一部それ自身（それと必要ならば，文脈情報を反映する機械の現在状態）に依存する．

これらの2つの違いをつぎの約束に吸収させて，翻訳機と解釈機の取り扱いの統一をはかる．

(a) 解釈機のおのおのの命令のかくれたアーギュメントである機械の現在状態 ξ と同様に，翻訳される完全なプログラムテキスト t を翻訳機のおのおのの関数のかくれたアーギュメントとみなす．

(b) t の要素である翻訳されるべき対象の代わりに，一般に t からそれらを選択するセレクタが関数のアーギュメントとして指定される．これらの 'text pointer' または 'pointer' と呼ばれるセレクタは elem(i) と s(i) という形のセレクタから構成され（i は整数），通常，文字 p, q と r で記述される．

2つのアーギュメント，すなわち，かくれたテキス

トtと陽に指定されたpointer pによって必要なすべての情報，すなわち，tの一部である$p(t)$と，それのt内での前後関係が与えられる．

一般に翻訳機の仕事は，具象プログラム内のすべての宣言を認識し，それらの属性をテストし，組み立てることである．そして，その他の部分に対しては，本質的には解剖木から抽象プログラムへの写像を行なうことである．この写像は具象プログラム内の順序によってのみ決められるセレクタの代わりに，具体的に名前がつけられたセレクタから構成された対象を生成する．

翻訳機にはチェック機構が組み込まれていて，もしプログラミング言語の定義によって指定された判定規準に関係する文脈が無視された場合には，チェックが可能となる，このように翻訳の最中に実行される誤りチェックは'静的である'といわれ，一方，解釈の最中にチェックされえる誤りは'動的な誤り'といわれる．

EPL に対する関数 translate

(T 1) translate(t) =
is-c-progr$(t)\rightarrow$trans-block(p)
$T\rightarrow$error

(T 2) trans-block(p) =
μ_0(<s-del-part : trans-decllist$(s_2\circ p)$>,
<s-st-list : trans-stlist$(s_4\circ p)$>)

(T 3) trans-decllist(p) =
$\neg(\exists i,j)(i\neq j\ \&\ s_2\circ s_i\circ p(t)$
$=s_2\circ s_j\circ p(t)\neq$
$\Omega)\rightarrow$
μ_0({<id : trans-decl$(s_1\circ p)$>|id
$=s_2\circ s_i\circ p(t)\neq\Omega$})
$T\rightarrow$error

この条件は重複宣言がないことをたしかめる．

(T 4) trans-decl(p) =
is-INT$\circ s_1\circ p(t)\rightarrow$INT
is-LOG$\circ s_1\circ p(t)\rightarrow$LOG
is-FUNCT$\circ s_1\circ p(t)\rightarrow$
μ_0(<s-param-list :
trans-parlist$(s_3\circ p)$>,
<s-st : trans-st$(s_5\circ p)$>,
<s-expr : trans-expr$(s_7\circ p)$>)

(T 5) trans-parlist(p) =
is-$\Omega\circ p(t)\rightarrow$< >
$\neg(\exists i,j)(i\neq j\ \&\ s_i\circ s_2\circ p(t)$
$=s_j\circ s_2\circ p(t)$
$\neq\Omega)\rightarrow$
μ_0({<elem(i) : $s_i\circ s_2\circ p(t)$>|
$1\leq i\leq$slength$\circ s p_2\circ(t)$>)
$T\rightarrow$error

この条件は与えられたパラメタリストの中で同一パラメタが一度以上現われないことをたしかめる．

(T 6) trans-stlist(p) =
μ_0({<elem(i) : trans-st$(s_i\cdot p)$>|
$1\leq i\leq$slength$\circ p(t)$})

(T 7) trans-st(p) =
is-c-assign-st$\circ p(t)\rightarrow$
μ_0(<s-left-part : $s_1\circ p(t)$>,
<s-right-part :
trans-expr$(s_3\circ p)$>)
is-c-cond-st$\circ p(t)\rightarrow$
μ_0(<s-expr : trans-expr$(s_2\circ p)$>,
<s-then-st : trans-st$(s_4\circ p)$>,
<s-else-st : trans-st$(s_6\circ p)$>)
is-c-block$\cdot p(t)\rightarrow$trans-block(p)

(T 8) trans-arglist(p) =
is-$\Omega\cdot p(t\rightarrow$< >
$T\rightarrow\mu_0$({<elem(i) : $s_i\circ s_2\circ p(t)$>|
$1\leq i\leq$slength$\circ s_2\circ p(t)$})

(T 9) trans-expr(p) =
is-c-const$\cdot p(t)\rightarrow p(t)$
is-c-var$\cdot p(t)\rightarrow p(t)$
is-c-funct-des$\cdot p(t)\rightarrow$
μ_0(<s-id : $s_1\circ p(t)$>,
<s-arg-list :
trans-arglist$(s_2\circ p)$>)
is-c-bin$\cdot p(t)\rightarrow$
μ_0(<s-rd 1 : trans-expr$(s_2\circ p)$>,
<s-rd 2 : trans-expr$(s_4\circ p)$>,
<s-op : $s_3\cdot p(t)$>)

具体例

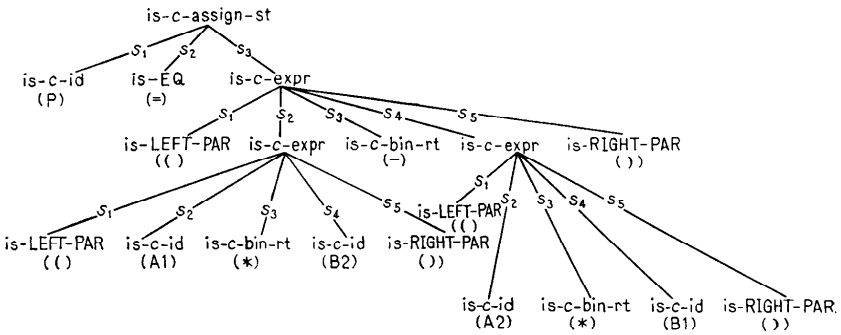
5.4.2 で与えられた EPL のプログラム例の中の代入文 $P=((A1*B2)-(A2*B1))$; の抽象表示および抽象プログラムは**第 5.1 図**および**第 5.2 図**のようになる．

(つづく)

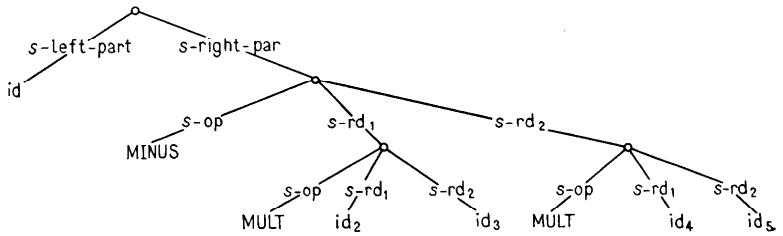
参考文献

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)



第 5・1 図 抽象表示



第 5・2 図 抽象プログラム