A Semantic Meta-Language

TAN WATANABE*

1. *Introduction*

Flowcharting of a problem would proceed from an overall sketch of its computation procedure to the detailed description of its component parts. This paper presents a programming language in which the process of programming closely resembles that of flowcharting. The language has a few basic expressions and the capability of refining the syntax and semantics of itself.

In this language, the first step of programming is to describe a general flow of computation introducing many phrases and clauses which are suitable to represent the flow intuitively. And then, the flow is to be refined by explaining the meanings of the phrases and clauses introduced. This process is continued repeatedly until all words are explained completely in terms of basic expressions.

If we prepare many expressions which would make it easy to do programming in some application field, and build them in an expression library, then we can obtain a problem oriented language in that field. In this way, the language presented here can be a kernel language common to many problem oriented languages.

After a brief illustration of programming, the syntactic and semantic formulation of the language follows. An effort has been made to keep the logical consistency of the language by introducing some mappings from symbol to symbol or from symbol to set of symbols.

Compared to other extensible languages such as ALGOL 68 [1], GPL [2], and PLAN [3], this language has differences from them in such points as: (1) the process of programming proceeds from global to detail, (2) the syntax and semantics of the language is explained mathematically by using mappings.

2. *Programming Example*

Fig. 1 shows a sample program written in this language. In this case, following expressions are assumed to be library expressions.

ARRAY (u, m) $[v_1, \ldots, v_n]$;	INTEGER $[v_1, \ldots, v_n]$;
IF *bool* THEN a ELSE c;	GO TO l;
RETURN (e);	LET $v=e$;
e_1+e_2 e_1 EQ e_2	e_1 GE e_2
GETDATA $(v_1, \ldots, v_n)$;	PUTDATA $(v_1, \ldots, v_n)$;

This paper first appeared in Japanese in Joho Shori (the Journal of the Information Processing Society of Japan), Vol. 11, No. 8 (1970), pp. 439-448.

* Central Research Lab., Hitachi, Ltd.

```
defineoperator[MAZE SOLVER;
      ARRAY(INTEGER,100,10)[XPATHNUMBER];
        $ XPATHNUMBER(I,J) REPRESENTS THE J-TH PATH WHICH GOES OUT FROM ROOM I.
      ARRAY(INTEGER,200)[XENTRYCOUNT,XFROMROOM,XTOROOM];  $ FOR EACH PATH
      INTEGER[XENTRYROOM,XEXITROOM,XI,XC,XR,XP];
        $ XR IS THE CURRENT ROOM NUMBER.  XP IS THE CURRENT PATH NUMBER.
      INITIATION;
  L1: IF EXIT THEN GO TO LAST;
      IF LOOP THEN GO TO L2;
      IF (UNSEARCHED PATH IS LEFT) THEN GO TO L3;
      IF ENTRANCE THEN GO TO LAST;
  L2: GO BACK;    GO TO L1;
  L3: GO ON;      GO TO L1;
  LAST: DISPLAY RESULTS;

defineoperator[INITIATION;
      GETDATA(XPATHNUMBER,XFROMROOM,XTOROOM,XENTRYROOM,XEXITROOM);
      LET XENTRYCOUNT=0;
      LET XR=XENTRYROOM;
    ];
defineoperator[UNSEARCHED PATH IS LEFT;
      LET XI=1;
  L3: IF XPATHNUMBER(XR,XI) EQ 0 THEN RETURN(FALSE);
      IF XENTRYCOUNT(XPATHNUMBER(XR,XI)) EQ 0 THEN GO TO L4;
      LET XI=XI+1;    GO TO L3;
  L4: LET XP=XPATHNUMBER(XR,XI);
      RETURN(TRUE);
    ];
defineoperator[LOOP;
      LET XC=0;    LET XI=1;
  L5: IF XENTRYCOUNT(XPATHNUMBER(XR,XI)) EQ 1 THEN LET XC=XC+1;
      LET XI=XI+1;    GO TO L5;
  L6: IF XC GE 2 THEN RETURN(TRUE) ELSE RETURN(FALSE);
    ];
defineoperator[GO ON;
    LET XENTRYCOUNT(XP)=XENTRYCOUNT(XP)+1;
      IF XR EQ XFROMROOM(XP) THEN (LET XR=XTOROOM(XP))
                             ELSE  LET XR=XFROMROOM(XP);
    ];
defineoperator[GO BACK;   GO ON;];
defineoperator[EXIT;    XR EQ XEXITROOM;];
defineoperator[ENTRANCE;    XR EQ XENTRYROOM;];
defineoperator[DISPLAY RESULTS;    PUTDATA(XR,XP,XENTRYCOUNT);];
    ]
```

Fig. 1.

3. *Outline of Programs*

In this language, a program, say p, takes the form

defineoperator $[f(x_1, \ldots, x_n)$; *exptype* $[m_0, m_1, \ldots, m_n]$;
set definition part
declaration definition part
declaration expression part
l_1: $f_1(t_{11}, t_{12}, \ldots)$;
$\ldots$
l_k: $f_k(t_{k1}, t_{k2}, \ldots)$;
operator definition part].

This formula declares following: (1) The string $f(x_1, \ldots, x_n)$ represents the same operation as specified by (l_1: $f_1(t_{11}, t_{12}, \ldots)$; $\ldots l_k$: $f_k(t_{k1}, t_{k2}, \ldots)$;). (2) m_0 is the name of the symbol class which contains the string $f(x_1, \ldots, x_n)$. (3) $m_1, \ldots,$

m_n are the names of the symbol class to which the corresponding parameters $x_1, \ldots, x_n$ should be contained. (4) f is called the operator defined by the program p.

The set definition part is either null or a sequence of expressions of the following form:

defineset $[m(t_1, \ldots, t_n)$; *exptype* $[u_1, \ldots, u_n]$; $s]$.

An expression of the above type defines that the symbol class named $m(x_1, \ldots, x_n)$ is a compound set specified by the string s. $u_1, \ldots, u_n$ specify the symbol classes containing the corresponding parameters $x_1, \ldots, x_n$, if they were not omitted.

The declaration definition part is either null or a sequence of expressions of the following form:

definedeclaration $[m(x_1, \ldots, x_n)\ [x]$; *exptype* $[u_1, \ldots, u_n]$; d_1; $\ldots d_l$;$]$;.

This defines that the declaration expression $m(x_1, \ldots, x_n)\ [x]$ is equivalent to a sequence of declaration expressions d_1; $\ldots d_l$;. A declaration expression takes the form $u[v]$ and it means that the variables specified by the string v belong to the symbol class u.

The operator definition part is either null or a sequence of expressions which take the same form as the program p. The program p itself is an operator definition.

A string of the form l: $f(t_1, \ldots, t_n)$; is called an operation statement. l is called the label of an operation expression $f(t_1, \ldots, t_n)$.

Some symbols are called known symbols. They are:

constant $\mathbf{C}=\{1, 3.14, -1.0E6, \text{'ABC'} \ldots\}$,

known operator $\mathbf{F}_k=\{+, -, *, /, \text{ASSIGN, JUMP, } \ldots\}$,

known set-name $\mathbf{M}_k=\{\text{INTEGER, REAL, SETEXPRESSION, } \ldots\}$,

set function $\mathbf{G}=\{\text{AND, OR, NOT, QUOTE, REPRO}\}$.

In this paper, $\mathbf{I}$ represents the set of identifiers. An element of the set

$$\mathbf{J}=\mathbf{I}\cup\{a(b_1, \ldots, b_n)\mid a\in\mathbf{I},\ b_i\in\mathbf{C}\cup\mathbf{J},\ i=1, \ldots, n\}\cup\{x\cdot y\mid x\in\mathbf{J},\ y\in\mathbf{J}\}$$

is called a variable type string.

Relations between expressions are specified by two mappings, φ_k and χ_k. $\varphi_k(a)=\alpha$ implies that a, an element of $\mathbf{M}_k$, is the name of α, a set of expressions. When $h\in\mathbf{F}_k$, $\chi_k(h)$ is a set of strings $m_0m_1\ldots m_n$, where $m_i\in\mathbf{M}_k$, $i=0, 1, \ldots, n$. It specifies that an expression $h(e_1, \ldots, e_n)$ is legal provided $e_i\in\varphi_k(m_i)$, $i=1, \ldots, n$, and that $h(e_1, \ldots, e_n)\in\varphi_k(m_0)$. The relations of φ_k and set functions are as follows:

$\varphi_k(\text{AND } [e_1, e_2])=\varphi_k(e_1)\cap\varphi_k(e_2)$,

$\varphi_k(\text{OR } [e_1, e_2])=\varphi_k(e_1)\cup\varphi_k(e_2)$,

$\varphi_k(\text{NOT } [e])=$complement of $\varphi_k(e)$,

$\varphi_k(\text{QUOTE } [e])=\{e\}$,

$$\varphi_k(\text{REPRO } [i, m, n, p])=\bigcup_{i=m}^{n}\varphi_k(p_i),$$

where, p_m, p_{m+1}, ⋯, p_n are the strings obtained by substituting m, $m+1$, ..., n for i contained in the string p each respectively.

4. *Detail of Programs*

The preceding explanations give only the vague idea of the language. Let's make them clear with the help of the following sets and mappings introduced by rules R1 through R19 concerning a program p: (1) **F**: set of operators, (2) **M**: set of names of symbol classes, (3) **T**: set of operation expressions, (4) **U**: set of set expressions, (5) **W**: set of variable forms, (6) **D**: set of declaration forms, (7) **L**: set of lables, (8) **V**: set of variables, (9) φ: mapping which relates symbol classes to their names, (10) χ: mapping which specifies the parameter types of operators and set functions, (11) ϕ: mapping which expands variable forms into variables and declaration forms into declarations.

R1) $\mathbf{F}\supset\mathbf{F}_k$, $\mathbf{M}\supset\mathbf{M}_k$.

R2) If $a\in\mathbf{M}_k$, then $\varphi(a)\supset\varphi_k(a)$.

R3) If h is contained in the domain of χ_k, then $\chi(h)=\chi_k(h)$.

R4) $\mathbf{T}\supset\mathbf{C}$, $\mathbf{T}\supset\mathbf{V}$, $\mathbf{T}\supset\mathbf{L}$.

R5) $\mathbf{U}\supset\mathbf{M}$.

R6) If $g\in\mathbf{G}$, $m_1...m_n\in\chi(g)$, and $t_i\in\varphi(m_i)$, $i=1, ..., n$, then $g[t_1, ..., t_n]\in\mathbf{U}$.

R7) $\mathbf{W}=\mathbf{J}\cup\{u\in\mathbf{U}\mid\varphi(u)\subset\mathbf{J}\}$.

R8) If $w\in\mathbf{W}$ and $w\in\mathbf{U}$, then $\phi(w)=\varphi(w)$.
If $w\in\mathbf{W}$ and $w\notin\mathbf{U}$, then $\phi(w)=\{w\}$.

Note) $\phi(w)$ is called the expansion of w.

R9) If $m\in\mathbf{I}$, $x_i\in\mathbf{I}$, $u_i\in\mathbf{U}$, $i=1, ..., n$, $s\in\mathbf{U}$, and the string *defineset* $[m(x_1, ..., x_n)$; *exptype* $[u_1, ..., u_n]$; $s]$ is contained in p as a set definition, then

$$m\in\mathbf{M},\ u_1...u_n\in\chi(m),\ x_i\in\varphi(u_i),\ i=1, ..., n,$$

and, if $t_i\in\varphi(u_i)$, $i=1, ..., n$, then

$$m(t_1, ..., t_n)\in\mathbf{M},$$

$$\varphi(m(t_1, ..., t_n))=\varphi(s'),$$

$$\varphi(m)\supset\bigcup_{\substack{t_i\in\varphi(u_i)\\ i=1,...,n}}\varphi(m(t_1, ..., t_n)),$$

where, s' is a string obtained by substituting $t_1, ..., t_n$ for the corresponding $x_1, ..., x_n$ in the string s. In case of $n=0$, this rule is shortened as follows: If $m\in\mathbf{I}$, $s\in\mathbf{U}$, and the string *defineset* $[m; s]$ is contained in p as a set definition, then $m\in\mathbf{M}$, $\varphi(m)\supset\varphi(s)$.

Note) In such cases, explanations will be shortened by saying "the same rule applies to parameterless situations".

R10) $\mathbf{D}'=\{u[w]\mid u\in\mathbf{U},\ w\in\mathbf{W}\}$,
$\mathbf{D}=\mathbf{D}'\cup\{u\in\mathbf{U}\mid\varphi(u)\subset\mathbf{D}'\}$,
that is, a declaration is either a string of the form $u[w]$ or a set expression

which generates the strings of the form $u[w]$.

R11) If $d \in \mathbf{D}'$, then $\phi'(d) \supset \{d\}$. If $d \in \mathbf{D} \cap \mathbf{U}$, then $\phi'(d) = \bigcup_{u[w] \in \varphi(d)} \phi'(u[w])$.

R12) If $m \in \mathbf{I}$, $x \in \mathbf{I}$, $x_i \in \mathbf{I}$, $i=1, \ldots, n$, $d_j \in \mathbf{D}$, $j=1, \ldots, l$, and if the string *definedeclaration* $[m(x_1, \ldots, x_n)[x]$; *exptype* $[u_1, \ldots, u_n]$; d_1; $\ldots$, d_l;$]$ is contained in p as a declaration definition, then

$m \in \mathbf{M}$, $u_1 \ldots u_n \in \chi(m)$, $x_i \in \varphi(u_i)$, $i=1, \ldots, n$,

$m(t_1, \ldots, t_n) \in \mathbf{M}$,

$\phi'(d_j') \subset \phi'(m(t_1, \ldots, t_n)\,[x])$, $j=1, \ldots, l$,

where, $t_i \in \varphi(u_i)$, $i=1, \ldots, n$, and d_j' is a string obtained by substituting $t_1, \ldots, t_n$ for the corresponding $x_1, \ldots, x_n$ contained in the string d_j.

R13) The mapping ϕ' is defined by rules R11 and R12 only.

R14) If $d \in \mathbf{D}$, then

$$\phi(d) = \bigcup_{u[w] \in \phi'(d)} \{u[a] \mid a \in \phi(w)\}.$$

R15) If $u \in \mathbf{U}$, $w \in \mathbf{W}$, and the string $u[w]$; is contained in p as a declaration statement, then

$\mathbf{V} \supset \phi(w)$, $\mathbf{V} \supset \{a \mid m[a] \in \phi(u[w])\}$,

$\varphi(u) \supset \phi(w)$,

and, if $m[a] \in \phi(u[w])$, then $a \in \varphi(m)$.

Note) An illustration of variable declaration would help to understand the preceding rules. Consider the following strings as contained in p:

definedeclaration [ARRAY (U, L) [X];
exptype [SETEXPRESSION, INTEGER];
U [REPRO [I. 1. L, QUOTE [X (I)]]];]
definedeclaration [POINT [P];
INTEGER [P.X]; INTEGER [P.Y];]

In this case, if the string ARRAY (POINT, 5) [A]; is contained in p as a variable definition, then the following relations hold:

ϕ (ARRAY (POINT, 5) [A])
= {ARRAY (POINT, 5) [A], POINT [A(1)], ..., POINT [A(5)],
INTEGER [A(1). X], ..., INTEGER [A(5). X],
INTEGER [A(1). Y], ..., INTEGER [A(5). Y]},
$\mathbf{V} \supset$ {A, A(1), ..., A(5),
A(1). X, ..., A(5). X, A(1). Y, ..., A(5). Y},
φ(ARRAY) $\supset \varphi$(ARRAY (POINT, 5)) $\supset$ {A},
φ(INTEGER) $\supset$ {A(1). X, ..., A(5). X, A(1). Y, ..., A(5). Y},
φ(POINT) $\supset$ {A(1), ..., A(5)}.

R16) If $l \in \mathbf{J}$, $f \in \mathbf{F}$, $t_i \in \mathbf{T}$, $i=1, \ldots, n$, and the string $l : f(t_1, \ldots, t_n)$; is contained in p as an operation statement, then $l \in \mathbf{L}$. l is called the label of this operation statement.

R17) If $f \in \mathbf{F}$, and $u_0u_1 \ldots u_n \in \chi(f)$, then

$f(t_1, \ldots, t_n) \in \mathbf{T} \cap \varphi(u_0)$,

where, $t_i \in \varphi(u_i)$, $i=1, \ldots, n$. The same rule is applied to parameterless situations.

R18) If a string having the same form as (1) in the previous section is contained in p as an operator definition, and if it satisfies

$f \in \mathbf{I}$, $x_i \in \mathbf{I}$, $i=1, \ldots, n$, $m_i \in \mathbf{U}$, $i=0, 1, \ldots, n$,

$l_j \in \mathbf{J}$, $j=1, \ldots, k$, $f_j(t_{j1}, t_{j2}, \ldots) \in \mathbf{T}$, $j=1, \ldots, k$,

then

$f \in \mathbf{F}$, $m_0m_1 \ldots m_n \in \chi(f)$, $x_i \in \mathbf{V} \cap \varphi(m_i)$, $i=1, \ldots, n$.

The same rule is applied to parameterless situations. Some of the label parts l_1:, ..., l_k: may be omitted.

R19) Let q be an operator definition contained in the program p, and let each $\mathbf{F}_p$, $\mathbf{M}_p$, $\mathbf{V}_p$, and $\mathbf{L}_p$ stand for **F**, **M**, **V**, and **L** of p; and let each $\mathbf{F}_q$, $\mathbf{M}_q$, $\mathbf{V}_q$, and $\mathbf{L}_q$ stand for **F**, **M**, **V**, and **L** of q respectively. Then the following relations hold:

$\mathbf{F}_p \subset \mathbf{F}_q$, $\mathbf{M}_p \subset \mathbf{M}_q$, $\mathbf{L}_p \subset \mathbf{L}_q$, $\mathbf{V}_p \subset \mathbf{V}_q$,

that is, the symbols defined in the outer program p are available in the inner program q, also.

5. *Compiler*

Operation expressions are in the form of $f(t_1, \ldots, t_n)$ in the above discussions. This bracketed notation is, however, inconvenient to human beings. For practical purposes, expressions are changed in a readable form by introducing operator precedence, by splitting operator names, and by a rule of abbreviations. In order to avoid confusions, variable names are restricted to those symbols which have the character X as their initials.

Programs are translated into the standard bracketed form by the preprocessing part of the compiler of this language. The main part of the compiler performs a substitution operation repeatedly, and it changes the source program into a string of known symbols and variables.

6. *Concluding Remarks*

We have formulated a syntactic and semantic definition of an extensible programming language, in which, one can write down programs from global to detail. Programs written in this language would require no farther explanations in flowchart.

A basic compiler of the language has been implemented and it is, now, in experimental use. Some slight modifications have been done for the convenience of implementation and for the refinement of the language.

Acknowledgment

The author wishes to express his deep gratitude to H. Murata of St. Sophia University, K. Asai of Japan Atomic Energy Research Institute, and I. Nakata and K. Yoshimura of Hitachi Ltd. for their valuable comments.

References

[1] A. van Wijngarden (editor): Draft Report on the Algorithmic Language, ALGOL 68; Mathematisch Centrum (1968).

[2] Garwick, J.: GPL, A Truly General Purpose Language; *Comm. ACM, 11,* 9, pp. 634-638 (1968).

[3] IBM Corp.: Problem Language Analyzer (PLAN) Program Description Manual, Form H20—0594 (1969).