D 2. 増殖型言語ＳＥＬＦについて

中田育男，浜田穂積，霜田忠孝，野木兼六（日立中研）

1．はじめに

計算機を使用するにあたって，コンパイラを使うことが常識となった現在，われわれはソフトウェア・メーカの立場として，各種の言語の能率のよい，信頼できるコンパイラを短期に，少ない人手で作って，多様なユーザの要求に応えるにはどのようにすればよいかを考えて来た．そのための第一の方法は，システムを記述するに必要な最小限の機能を持つような言語を，広く使われている特定の言語の部分集合とするという意味で，ＰＬ／Ｉの部分集合を選びＰＬ／ＩＷと名付けて，そのコンパイラを作った．[4] そして，ＰＬ／ＩＷによってコンパイラを作った．この方法はコーディングとデバッグの時間を短縮して，コンパイラ製造の生産性を向上させ得ることを示したが，個々のコンパイラの設計段階については，本質的に，従来のアセンブラで書いていたときと同じ手数を必要とすることもわかっている．

そこでわれわれは次のような方針を立てた．まず第一に，多くの計算機言語を構成している基本的構造と，それを基にして拡張できる機能とを持った言語を設定し，そのコンパイラを作る．そして，それから後の，各種の問題向き言語への拡張は，必要に応じて付加できるようにする．そのためには，HITAC 5020 TSS にあるようなファイルシステムを有効に利用して，拡張した部分の目的プログラムをファイルとして保存することによって，いつでも，誰にでも使えるようにすれば，言語を拡張することが，ちょうどコンパイラを大きくするのと同じ効果を持つようにすればよい．こう考えて，そのような言語を増殖型言語と呼ぶことにした．

ところで，ALGOL－Ｎ[1,2,3]が，われわれの求めていた基本言語に対する要求を極めてよく満足させてくれることがわかったので，ALGOL－Ｎを基礎にして，それに，なお不十分だと感じていた，宣言の形式の宣言（declaration declaration）の機能などを付け加えて，一つの言語にまとめて，それをSELF（Self Extensible Language Facility）と名付けた．

2．ＳＥＬＦの設計方針

SELFの設計にあたってわれわれが考慮した機能は次のことである．

(1) SELFの基本言語は拡張型言語（Extensible Language）であること．

(2) SELFコンパイラは増殖型コンパイラであること．

(3) 基本機能として十分大きい機能を持つこと．

(4) いろいろの面で能率のよいこと．

(5) TSSでの会話的機能を持つこと．

(6) サブシステムとしての機能を持つこと．

上に述べた機能を持つ増殖型言語SELFを，われわれは，増殖に都合のよいファイルシステムを持つとか，会話機能を持つなどの理由で，HITAC 5020 TSSで実現することにした．また，基本言語のもとになるものとしてALGOL－Nを選んだ理由は

(1) 基本的機能として十分なデータ構造を持っている．

(2) 拡張機能を持っている．

(3) 言語のsyntaxとsemanticsがよく整理されていて，まとまっている．

(4) ALGOL－N以外の言語との接続も考慮されている．

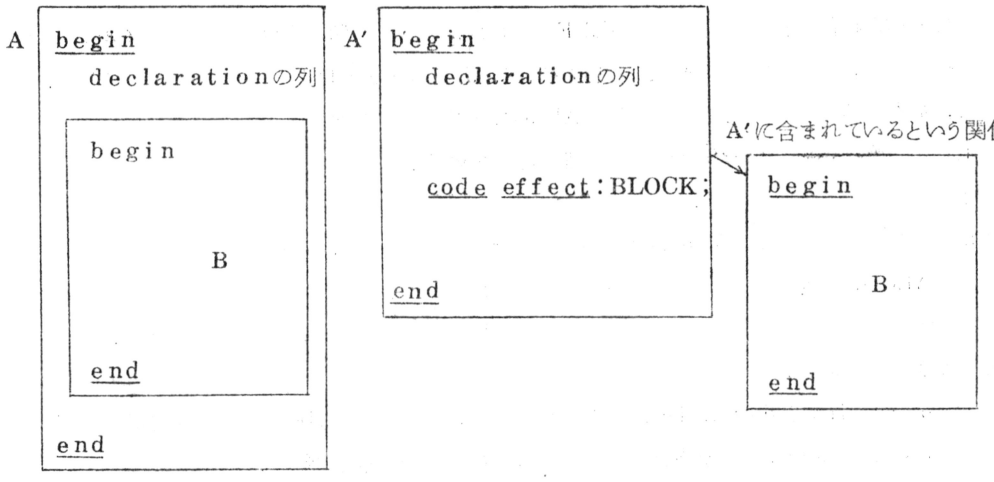
などが優れていると考えたからであるが，われわれの目的に適さない面もなくはない．それは次のような点である．

(5) quantityがdynamicに生成されるため，能率のよい目的プログラムが作りにくい．

(6) expressionの拡張機能は大きいが，宣言の拡張機能が弱い．

これらの点については，われわれの目的を達するためには，変更をいとわなかった．

次に増殖機能をどのようにして実現するかについて述べる．先に述べたことから，プログラムを分割してコンパイルする必要が生じる．そこである部分を別にコンパイルするためには，それは，blockまたはprocedure donorでなければならないと決めた．そうするとその部分を取除いた元のプログラム（これもblockまたはprocedure donorでなければならない）には抜け穴ができるが，プログラムとしての体裁を整えるために，ここにcodeを置く．こうすれば，FORTRAN Ⅳに近い形式で，プログラム単位を認めることができるわけである．procedure donorの方は，formal parameterのtypeを知るために，頭にその情報をつけて，形としてはprocedure notationの形式にする必要がある．次に問題になるのはglobalなdeclarationの有効範囲の問題がある．そのため，一つのプログラムをコンパイルしたとき，目的プログラムと共に，declarationの内容を記録した表もファイルとして，ファイルシステムに登録する．したがって，一つのプログラムをコンパイルするときには，その上のレベルのプログラム（このプログラムを含むことになっているプログラム）をコンパイルしたときにできた表を参照して目的プログラムを作り，その表に，このプログラムのdeclarationの内容を付け加えてファイルにするわけである．プログラムのコンパイル単位の分割の方法を図1に示す．ここでは理解し易いように，SELFの書き方ではなくて，ALGOL－N流の表現にする．



(a)　　(b)

図 1　プログラムの分割

3. SELFの文法

SELFの設計にあたり，増殖型言語として実用上充分な機能とは次のようなことであろうと考えた．

(1) 新しいデータ構造が導入できる．

(2) 新しい演算の形式が導入できる．

(3) 新しい宣言の形式が導入できる．

(4) 以上のものを保存することができる．

我々がSELFの基本言語の範としたALGOL-N version 2は，これらのうちの(1)と(2)の機能は比較的よく具えているがそれでも充分ではなく，(3)の機能はほとんどない．(4)の機能は考慮の対象外となっている．以下では，このために行なった種々の追加，修正についてより詳しく述べる．

(1) ALGOL-Nでは新しいデータ構造を導入するとき，主にstructure を使用する．例えば

```
let complex represent procedure ( ) structure (re:real, im:real):
    structure (re:real, im:real);
let a be complex;
```

などとして，complexという新しい型のデータを導入したと考える．しかしこれは従来からある複雑な型のデータに名前を付けたという効果しかもたない．本当に新しい型のデータを導入するためには，＜type declaration＞のようなものが必要になる．

例えば

let complex typify structure (re:real, im:real);

let a be complex;

などとして，complexはstructure (re:real, im:real) とは異なる全く新しい型と考えるのである．この機能を導入するかどうかで種々の検討を行なったが，結局

(a) これを導入すると基本言語がやや複雑になる．

(b) これを導入しなくても新しいデータ構造の導入に関しては，ALGOL-N で行なっている方法だけで実用上余り困らないであろうと思われる．

などの理由から，この導入は見合わせることになった．

(2) 我々がALGOL-Nを基本言語の範とした主な理由の1つは，新しい演算の形式の導入の方法が非常に優れているということであった．しかしそれでも不満な点がない訳ではなかった．

(i) <mark declaration>

ALGOL-Nの<mark declaration>はあるmarkのfacingとpriority を宣言するものである．このうちfacingの宣言はそのmarkの使い方にある種の制限を課するもので余り好ましくない．更に<mark declaration>のsyntax 自体も直観的に見易いものではない．そこで我々は全てのmarkはいわゆるdouble facedであるとし，<mark declaration>はmarkの対に対するpriority を宣言するものであるという考え方を採用した．(この考え方はALGOL-N の Working Group での議論にもとづいたものである)．このようにすると<mark declaration>の数が増えるのではないかという懸念があるが，それは<formula declaration>によってpriorityをimplicit に宣言する(例えば $\alpha(\ )\beta(\ )\gamma$ というframeに対しては $\alpha \fallingdotseq \beta$, $\beta \fallingdotseq \gamma$, $\alpha \prec ALL$, $ALL \succ \beta$, $\beta \prec ALL$, $ALL \succ \gamma$ を自動的に補う．もし前で宣言されている $\alpha(\ )\delta$ というformulaも使いたいときには，$\alpha \fallingdotseq \delta$ をexplicitに宣言し直す．)ことを許すことと，<mark declaration>のsyntax を次のようにすることでほぼ解決した．

MARK (<marks> (<< | == | >>) <marks>) {,} ...

ここで<marks>とはmarkの列かALLかALL BUTの後にmark の列を書いたものである．

(ii) <formula declaration>

ALGOL-Nの<formula declaration>は1つのskeleton に対してその意味を定義する機能しかもっていない．しかし言語を拡張する場合には，1つのframeに無限個のtype-listが対応しているような"skeleton"をもつformulaを宣言する必要が生じてくる．実際ALGOL-Nのstandard declaration の中にはそのようなものがいくつかある．この種のformulaをstandard declarationに制限して，それだけ特別な処理をするというimplementationも考えられるが，我々は，拡張型言語とい

う観点から見て，このように強力な機能はぜひ一般的に使えるようにしたいと考えた．このような機能があれば，standard declarationをコンパイラから切り離すことができ（というよりはむしろstandard declarationというものはもはや意味がない）コンパイラ作りの労力も軽減する．これを実現するためにtypeを一般化してANY0～ANY9という任意のtypeを示すキーワードを導入した．このようなものをもち込むとコンパイル時にtypeが決定できないのではないかという心配が生ずるが，ANY0～ANY9はskeletonのtype-listを決定するprocedure notationのtypifierの部分でしか使えないという制限によって，これは回避できる．

(3) 既成のコンパイラ言語は，大きく分けて宣言と文という2つの部分から成り立っている．従って拡張型言語を考える際には，これら2つのもののsyntaxをユーザが自由に決められるようにすることが望ましい．しかるにALGOL-Nには，新しい宣言の形式を導入する機能がない．宣言はデータ構造が複雑になればなるほど，その重要度を増すから，この機能の欠除は拡張型言語としてかなり大きな欠点となるように思われる．そこでSELFでは<variable declaration>のsyntaxを定義する<declaration declaration>というものを考えた．そのsyntaxは次の通りである．

DECLARATION <declarator>〔( )〕{<key>} ... WITH(〔(<variable>:
(ID〔SEPARATED BY <key>〕|ANY|<expression>))〔,〕...)
〔REPEATED BY <key>〕MEANS <variable declaration>

ここで<declarator>とはこれによって新たに定義された<variable declaration>の存在を示すものでmarkと同じ形をしており，また<key>とは全体の形を決定するもので，markと同じ形かあるいは(,),〔,〕,:,,である．SELFの標準的な<variable declaration>に対する<declaration declaration>が一番始めに何らかの方法で（例えば

DECLARATION VARIABLE( ):( )WITH(I:ID, E:ANY) MEANS .../*
let I be E*/...

という感じで）宣言されているものとすれば，SELFの<variable declaration>のsyntaxは全て<declaration declaration>によって与えられると考えてよい．

(4) 拡張機能によって新しく導入したものを保存してコンパイラの増殖を行なうために，code文によって別のコンパイル単位として取り出したblockやprocedureの本来の存在位置を示すことにしたことは既に述べた．これを実現するcode文のsyntaxは

CODE〔<primary>〕:<code body>

であるとし，<code body>には次のどちらかを書くことにした．

(a) BLOCK

(b) EXTERNAL ≪file name≫

ここで≪file name≫とはユーザのファイル・システムに登録されている，SELFのprocedure notationの目的プログラムに相当するファイルの名前である．例えば，これは

```
FORMULA COPY( ) REPRESENS CODE(PROCEDURE(ANY0)ANY0):
    EXTERNAL COPY
```

などと使われる．

以上増殖型言語に要求される機能を中心にして述べてきたが，この他にもハードウェア表現あるいはコンパイラおよび目的プログラムの効率向上などの理由によって変更した点も多い．その1つにcall by nameの問題がある．ALGOL-Nのparameter mechanismはcall by nameであるが，これを単純にimplement したのでは非常に能率が悪い．そこでSELFではcall by quantityを基調とすることにし，call by name にしたいparameterはparameter spcificationの部分にその旨を明記することにした．

なおSELFの基本言語のsyntaxが付録に記載されている．

4. SELFのParser

4.1 基本概念

すでに述べてきたように，SELFコンパイラは

(1) 機能の上では，増殖型コンパイラであり，会話処理が行なえ，サブシステムとしても使えることを目ざしており，

(2) コンパラの製造という面からは，その生産性の向上に役立つことを目ざしている．

そのために，コンパイラの作成にあたっては，(a)能率，(b)信頼性，(c)簡潔，(d)作りやすさ，(e)融通性，(f)拡張性，(g)形式化，等を常に考えてゆく必要がある．これらのことはシステム作成においては常識的なことであるが，コンパイラなどの大きなシステムを作るときは，その作成に追われたりして，ややもすれば，忘れられがちになる．我々としては，増殖型コンパイラとしての外面の拡張性と同様に，その中身に対して常に(a)から(g)までのことを考えてゆく必要がある．

4.2 Parsing方法

SELFのparserを作るにあたって次の2つの方法が考えられた．

(1) Contexでparseする方法：京大で佐久間氏が行なっている方法[2]と同じであり，プログラムのContexをながめながらparseする．

(2) Operatorのprecedenceでparseする方法：markはもちろんであるがdeclaratorやkeyやstraight symbolもoperatorと考え，それらの間のprecedence関係によってparseする．[5]

我々はこの2つの方法のうち，結局，(2)のprecedenceでparseする方法を選んだわけ

であるが，それは，(1)のcontexでparseする方法の中でform callのparseは(2)とほとんど同じ処理を行なっており，secondary readやprimary readが互いに呼びあうときには，現在どのsyntactical elementを処理しているのかを覚えておくためにstack操作をせねばならなく，結局，これらの処理も(2)の方法で統一的に処理できると考えたからである．

parserは図2のように構成される．各ブロックの処理は次のようになる．

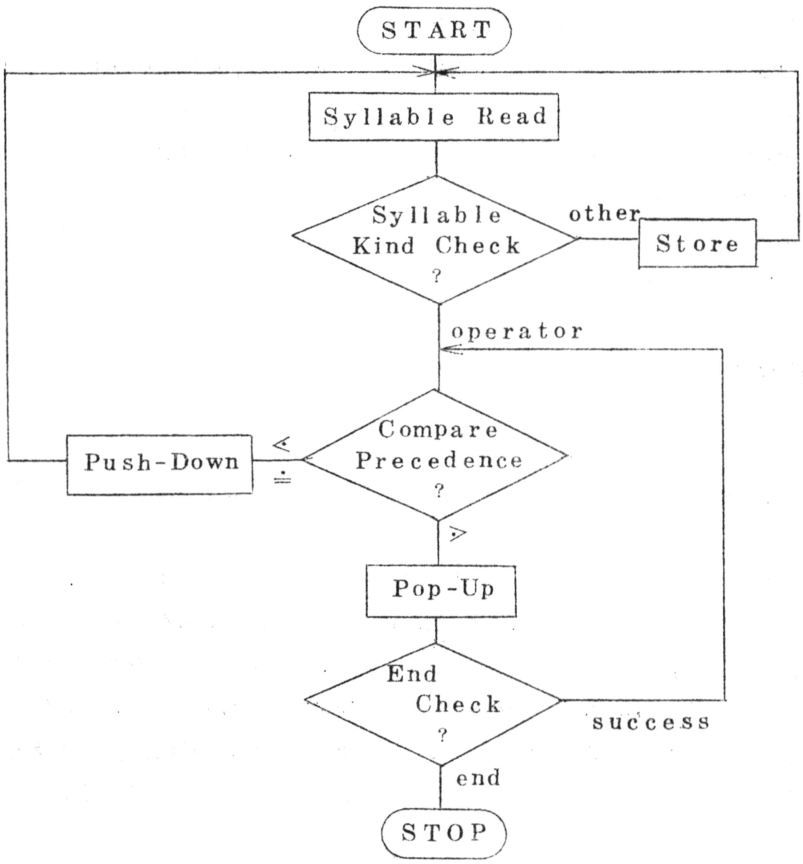


図2　Parserの構成

(a) Syllable Read：source programからone syllable読み込み，それをテーブルに登録して，登録番号およびsyllableの種別(<identifier>，<specifier>,<number>,<bits>,<string>,<straight symbol>等の区別)を返す．

(b) Syllable Kind Check：<identifier>や<specifier>に関しては，通常，<variable>,<label>,<selector>,<mark>,<declarator>

または<key>を表わしているのであるが，declarationでは単なるfigure として扱われたりする．ここでは，これらの区別を行なっている．

(c) Store：今読み込んだsyllableまたは作成したtreeをoperand area にstoreする．syllableをstoreする際，もし前にstoreされたものがoperand areaにstackされずに残っているときは，そのoperandと今読んだsyllable の間にcontexによる仮想的な区切り記号があると考え，Syllable Readerには一syllable読みすぎたという指令を与え，仮想的な区切り記号をoperatorとしてCompare Precedenceに行く．

(d) Compare Precedence：今読込んだoperatorとstackされているものとのprecedenceを比較して，その結果（$\lessdot$, $\doteq$, $\gtrdot$）を返す．

(e) Pop-Up：stackにある「$\cdots \lessdot \alpha_1 \doteq \alpha_2 \doteq \cdots \doteq \alpha_n$」なるoperator $\alpha_1, \alpha_2, \cdots, \alpha_n$（$\alpha_n$はstackのtopにおかれている）の列をとり出して，対応するsyntactical elementまたはskeltonをテーブルから探す．そこで得られた登録番号に従って，適当なaction routine（syntactical element に特有の処理を行なう）をcallして，treeを作成する．

(f) Push-Down：operandとoperator対をstackにpush downする．

(g) End Check：operatorがENDまたは；のときはstackが空かどうか調べる．空のときは処理を終る．

4.3 Precedence Matrix

Operator間のprecedenceは表1のようにmatrixによって表わされる．

ただし，ph, dh, pt1, pt2は次の通り：

ph：GOTO, BEGIN, CODE, EFFECT, REAL, BITS, STRING, REFERENCE, ARRAY, STRUCTURE：

dh：MARK, FORMULA, DECLARATION；

pt1：GOTO, EFFECT, REFERENCE；

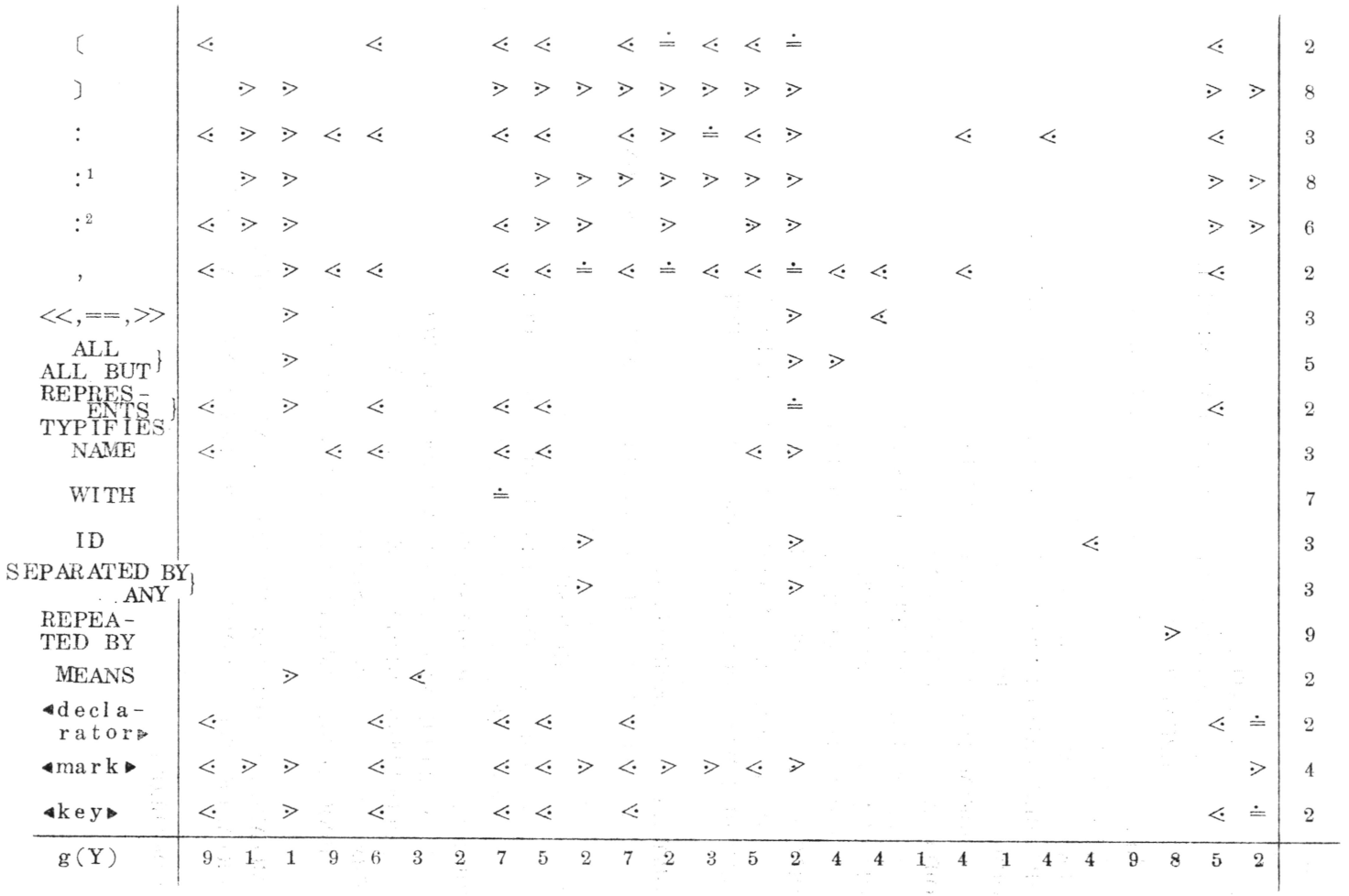
pt2：REAL, SITS, STRING

「〔」と「〕」は<procedure notation>と<procedure call>に現われる「(」または「)」であり，「:[1]」は<code>に現われる「：」であり，「:[2]」は<procedure notation>に現われる「：」である．◂mark▸どうしのprecedenceは別の表によって決められる．

このprecedence matrixはSELF syntaxから忠実に導き出したものでなく，処理の仕方やprecedence functionというものを考慮して作られている．表1のf(X)とg(Y)は対応するprecedence functionである．

表1 SELFのoperatorのprecedence matrixとfunction

X \ Y	pt1	pt2	BEGIN } ;	END	CODE	ARRAY STRUC- TURE	ANY i PROCE- DURE	VARIA- BLE	MARK FORMU- LA	DECLA- RATION ({	) }	f(X)
pt1												6
pt2												7
BEGIN } ;												1
END												8
CODE												1
ARRAY STRUC- TURE												7
ANY i PROCE- DURE												7
VARIA- BLE												3
MARK FORMU- LA												5
DECLA- RATION ({												2
) }												2
) }												1
p h												1
END												2
,												8
ANY i												8
PROCEDURE VARIABLE, ◂declarator▸												
d h												
)												
( )												
{ }												
[ ]												
:												
:1 :2												
,												
REPRESENTS TYPIFIES												
NAME												
WITH												
ID, ANY												
SEPARATED BY												
REPEATED BY												
MEANS												
◂mark▸												
◂key▸												


〔
〕
:
:1
:2
,
<<,==,>>
ALL
ALL BUT
REPRES-
ENTS
TYPIFIES
NAME
WITH
ID
SEPARATED BY
ANY
REPEA-
TED BY
MEANS
◂declarator▸
◂mark▸
◂key▸
g(Y)
9 1 1 9 6 3 2 7 5 2 7 2 3 5 2 4 4 1 4 1 4 4 9 8 5 2

5. 例

5.1 declaration

```
BEGIN
    DECLARATION MATRIX ( )[( ),( )]
        WITH (A:ID SEPARATED BY:,M:REAL,N:REAL)
              REPEATED BY,
        MEANS VARIABLE A:ARRAY[1:M,1:N](REAL);
    MATRIX MTYPE[1,1];
    MARK = ≪ ALL ,   ALL ≪?,+− ≪ */,
    VARIABLE MATOP:PROCEDURE(MTYPE,MTYPE)MTYPE;
    FORMULA ( )+( ) REPRESENTS
        CODE(MATOP):EXTERNAL MATADD;
    FORMULA ( )−( ) REPRESENTS
        CODE(MATOP):EXTERNAL MATSUB;
    FORMULA ( )*( ) REPRESENTS
        CODE(MATOP):EXTERNAL MATMULT;
    FORMULA   /( ) REPRESENTS
        CODE(PROCEDURE(MTYPE)MTYPE):EXTERNAL MATINV;
    VARIABLE MATPROC:PROCEDURE(MTYPE)EFFECT;

    FORMULA ( )? REPRESENTS
        CODE(PROCEDURE(MTYPE)MATPROC):EXTERNAL ASSIGN;
    %% X?の結果をYとすればY(A)はAの値をXに代入すること %%
    FORMULA ( )+( ) REPRESENTS
        PROCEDURE(A:MTYPE,P:MATPROC)MATPROC:
        (PROCEDURE(B:MTYPE)EFFET: P(B−A)  );
    FORMULA ( )*( ) REPRESENTS
        PROCEDURE(A:MTYPE,P:MATPROC)MATPROC:
        (PROCEDURE(B:MTYPE)EFFECT: P(/A*B)  );
    FORMULA ( )=( ) REPRESENTS
        PROCEDURE(P:MATPROC,A:MTYPE)EFFECT:(P(A));

    VARIABLE READ:CODE(MATPROC):EXTERNAL MATREAD;
    VARIABLE PRINT:CODE(MATPROC):EXTERNAL MATPRINT;

    CODE:BLOCK
END
```

5.2 プログラム例

5.1のdeclarationを持ったblock内のプログラム

```
BEGIN
  MATRIX V:W:X:Y〔10,10〕;
  READ(V); READ(W);
  Y?=V*W;
    %% まずY?を実行してMATPROC型のprocedure P1が得られ，次にV*W
       を実行し，P1(V*W)を実行するとYにV*Wの値が入る %%
  V*(Y+X?)=W;
    %% まずX?を実行してprocedure P2が得られ，次にY+P2を実行して
       PROC(B:MTYPE)EFFECT:P2(B-Y)というprocedure P3が
       得られ，次にV*P3を実行して，PROC(B:MTYPE)EFFECT:P3(/V*B)
       というprocedure P4が得られ，最後にP4=Wが実行される．すなわち
       P4(W)が実行される．すなわちP3(/V*W)=P2(/V*W-Y)が実行される．
       すなわち/V*W-Yの値がXに代入される %%
  PRINT(Y);PRINT(X)
END
```

6. むすび

本システムは，バーチャル・メモリ・マシンHITAC 5020のTSSにおいて実現すべく，現在，設計，開発中である．記述用言語としてはPL/IWを用いている．

SELFの設計にあたってはALGOL-Nの思想をほとんどとり入れた．AWG(ALGOL Working Group)の方々には，多くの検討会を通して有益なアイデアをいただいた．また立教大学の島内剛一助教授には，言語の全般にわたり数々の教えをいただいた．ここに深く謝意を表する．

参考文献

1) 京大数理解析研：算法言語の設計，記述，処理の研究――ALGOL N――：
数理解析研究所講究録66(1969年2月)
2) 佐久間紘一：ALGOL N compilerの作成について：
第11回プログラミング・シンポジウム報告集(1970年1月)
3) T. Shimauchi, et al.：ALGOL N, 2nd version：(unpublished)
(Aug. 1969)
4) 浜田他：PL/IWによるシステムの開発：
第11回プログラミング・シンポジウム報告集(1970年1月)
5) R. W. Floyd：Syntactic analysis and operator precedence:
J. ACM. (July 1963)

〔付　録〕　SELFのsyntax

```
<program>=<expression>
<expression>=<secondary>|<formula>
<secondary>=<primary>|<procedure notation>|<procedure call>|
            <array-structure element>
<primary>=<variable>|<go to statement>|<block>|
          <closed expression>|<code>|<effect notation>|
          <real notation>|<bits notation>|<string notation>|
          <reference notation>|<array notation>|<structure notation>|
          <any type>
<go to statement>=GO TO <label>
<block>=BEGIN(<decl>;)。。。/[(<label>:)。。。/<expression>]{;}。。。END
<closed expression>=(<expression>)
<code>=CODE[<primary>]:<code body>
<effect notation>=EFFECT
<real notation>=REAL[<mode>]/<number>
<bits notation>=BITS[<mode>]/<bits>
<string notation>=STRING[<mode>]/<string>
<reference notation>=REFERENCE
<array notation>=ARRAY[([<expression>:/<expression>]{,}。。。)](<entry>{,}。。。)
<structure notation>=STRUCTURE([<item>{,}。。。])
<procedure notation>=PROCEDURE([<typifier>{,}。。。])<primary>|
```

PROCEDURE([<parameter specification>{,}...])[<primary>]:<primary>

<procedure call>=<secondary>([<expression>{,}...])

<array-structure element>=<secondary>[<subscript>{,}...]

<formula>=[<expression>]<mark>[<expression>]{<mark>}...

<declaration>=<variable declaration>|<mark declaration>|
<formula declaration>|<declaration declaration>

<variable declaration>=VARIABLE(<variable>...:<expression>){,}...|
<declarator>([<variable>{<key>}...|<expression>]{<key>}...){<key>}...

<mark declaration>=MARK(<marks>(<<|==|>>)<marks>){,}...

<formula declaration>=FORMULA[()]<mark>[()]{<mark>}...TYPIFIES<expression>{,}...|
FORMULA[()]<mark>[()]{<mark>}...REPRESENTS<expression>{,}...

<declaration declaration>=DECLARATION<declarator>[()]{<key>}...
WITH([(<variable>:(ID[SEPARATED BY <key>]|ANY|<expression>))]{,}...)
[REPEATED BY <key>]MEANS<variable declaration>

<any type>=ANY0|ANY1|ANY2|ANY3|ANY4|ANY5|ANY6|ANY7|ANY8|ANY9

<mode>=([<modifier>])

<modifier>=<number>/,<expression>

<entry>=(<entry>{,}...)|<expression>

<item>=<selector>:<expression>

<typifier>=[NAME]<expression>

<parameter specification>=<variable>:<typifier>

<subscript>=<expression>|<selector>

<marks>=<mark>...|ALL[BUT <mark>...]

<key>=<identifier>|<specifier>|<delimiter*>

<declarator>=<identifier>|<specifier>

<mark>=<identifier>|<specifier>

<variable>=<identifier>

<label>=<identifier>

<selector>=<identifier>

<identifier>=<letter>[(<letter>|<digit>)...]

<specifier>=<special character>...

<number>=(<digit>... / .<digit>...)[E[+|-]<digit>...]

<bits>='(0|1)...'B

<string>='[<character>...]'

<character>=<letter>|<digit>|<delimiter>|<special character>|.|''

<letter>=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit>=0|1|2|3|4|5|6|7|8|9

<delimiter>=<delimiter*>|;|␣

<delimiter*>=(|)|[|]|:|,

<special character>=+|-|*|/|=|<|>|$|#|@|←|&|!|↑|"|?

<code body>=<etc.>