講　　座

ALGOL N について

(I) 概　説

岩　村　　聯*

はじめに

プログラム言語 ALGOL N については，それと並んで開発された構文記述法とともに，情報処理第11巻6号で和田英一によって要約紹介が行なわれている．

これから数回にわたって，数人交代の執筆によって，ALGOL N の本格的な解説をする．構文の記述法，意味の記述法の新しい体系も解説の対象になる．各回は，だいたい，そこまでで一応完結した解説となるように仕組まれるはずである．

今回は，この言語と記述法の体系とを作った意図を説明し，またこの言語のごく大まかな内容的説明をする．後者においては，上に言及した記述法の体系には従わない．その体系の紹介は次回以降にゆずる．

意図の説明には ALGOL **68** との対比が有効であるが，ALGOL **68** のことは和田の要約（前出）や，情報処理第11巻第7号にのせられた米田信夫の講演の記録で紹介されているから，ここではそれについての全般的な説明は省略する．

ALGOL N を作ることは，情報処理学会の ALGOL 作業グループ AWG，ランゲージ記述グループ LDG の有志によって，1968年の夏に始められ，同年12月にはこれの第1版［京都大学数理解析研究所講究録 66 (1969年2月）と同内容］が米田によって IFIP の ALGOL 作業グループ WG 2.1 の会合に提出された．その後，これに関与する顔ぶれや組織は漸次に変わってきたが，改良育成の作業が継続して現在に至っている．

当初の目的は，そのころ WG 2.1 で検討されていた ALGOL **68** の案に対するわれわれの批判に，実際的な裏づけをすることであった．ALGOL **68**（の案）は，ALGOL **60** の後継者とはいうものの，それにくらべて飛躍的に強化されている．他面，それはたいへんに複雑難解であり，この点で ALGOL **60** の長所を失っている．この欠点は，プログラム言語の（実用を考慮した）強力さのために避けにくいものではなく，言語設計の方針や記述の方法などを再考して改善すべきものである．

そういう改善の可能性を例示するために，われわれが急いでまとめあげた体系が，ALGOL N の第1版である．この作業は，ALGOL **60** の発展として，強力なプログラム言語を簡単な理解しやすい形で提供する，という趣旨で進められた．プログラム言語の強力さとしては，ALGOL **68** にできることは何かの形で ALGOL N にもできるというのが，ひとつの目安になった．この目安に従って，取り扱うデータの種類は ALGOL **68** の場合とほぼ同じ線に落着し，変更はわずかであった．

この作業の方針のうち，めぼしいものとして次の諸点がある．

1) 目標とする言語は algorithmic language であることを本義として，体系をすっきりさせる．コンパイルの能率などを基本的には尊重するが，コンパイラ言語としての末梢的な工夫はしない．

2) syntax の記述を平明にする．ALGOL **68** の syntax 記述法は，ある種の理論から見て単純かつ強力であるが，実行に移したときに含意されるものは平明といいにくい．これは，すでに島内剛一が提出していた代案で置き換える．

3) semantics の記述を平明かつ厳密にする．この点では，いろいろな概念の構成法を整理する（結果としては論理派の数学者の流儀に近づいた）ほか，新しく *core language* と称する言語を使って，動的な部分が厳密に，見やすく記述されるようにする．

4) 記述を段階的にする．ALGOL **68** は，統一的なスタイルの syntax による *strict language* と，それに記述上の便宜的な変形（*extension* という）をつ

* 立教大学理学部

け加えた *extended language* の，2段階に分れている．ALGOL N では，この *extension* の考えを借用するとともに，いろいろな面でさらに段階を分けて，回帰的な定義のループを小さく単純にする．

5) ALGOL **60** における *type* の概念を発展させるのに，ALGOL N では **real** などという *type*（こんどは無限とおりある）による大まかな分類と，それの小分けとして **integer** などという *mode*（この用語は ALGOL **68** からの借用）による分類を設け，各 *mode* には上の *type* からその *mode* への写像である *projection* というものを付随させておく．たとえば，実数から整数への丸めは *projection* であり，整数はそのまま（*type* を変えずに）実数である．

〖上のように **integer** を **real** に従属させることについては，われわれの間にも多少の疑問が残っている．〗

6) 共通点のある概念や機能などはなるべく統合し，一般化して単純な形にまとめあげる．たとえば，普通の演算記号も **if** や **then** のようなものも，*mark* という広義の演算記号として統一的に扱い，*mark* による結合の結果を一般的に *formula* という．結合の優先順位や，*formula* の形式，意味などという個別的なことは，言語の規定から除外し，*declaration* によってなるべく自由に導入できるようにする．

7) もっと具体的な処置として，ALGOL **68** に見られる複雑な *coercion*（*mode* の自動変換）に相当することを，言語の規定の中には持ちこまない．ある程度の *coercion* に相当する機能は *formula* の活用によって再現することができるが，それは *declaration* の領分として，この言語の利用者あるいは *standard declaration* の作成者に選択をまかせればよい．

このような方針を実行に移す過程で，最初の“批判を裏づけるための例”という程度を上回わるいろいろな改革が行なわれた．たとえば，上の 6) や 7) にも一端が見られるように，多くのことが言語の直接の規定から除外され，(*standard*) *declaration* のほうに移されて，狭義の ALGOL N はごく基礎的な言語という形になった．もはや最初のように ALGOL **60**, **68** だけを意識するのでなく，*standard decleration* の設け方によって他のプログラム言語のような使い方もできるようにする，という考え方が加わったのである．

こうして ALGOL N を作りあげてみると，当初の意図をこえて，独自の存在理由をもつ体系になっていると思われたので，第1版の改良という作業が始められた．もともと第1版は，WG 2.1 のスケジュールにまにあわせた速成のもので，多くの不満足な点が残っていた．それらを一応改良した第2版は，英文アブストラクトの形で，すでに一部に配布されている（1970年夏）．

以下の紹介は第2版に準拠するが，一方，第3版への検討もいま行なわれているので，それについてのコメントが今後の連載のなかに見られるかも知れない．

§ 1. プログラムの外観

ALGOL N のプログラムは *basic symbol* の有限列として作られ，〈expression〉の形をしている．ここで〈experession〉というのは，ALGOL **60** の〈expression〉と〈statement〉とを統合，一般化したものである．

上記の“〈expression〉”のように，ALGOL **60** の syntax 記述に用いられたのと同様な形式のメタ言語変数を，今後の説明の中でも，*basic symbol* の有限列の形を示すのに用いる．*basic symbol* の有限列そのものを簡単に図形と呼ぶが，たとえば，“〈block〉の形の図形”の代わりに“〈block〉”と略称することもある．

また，たとえば〈block〉という形の図形をひとつ考えて，それをかりに B と名づけるとき，

“〈block〉 B”

と書いて形と名を同時に示すことがある．さらに進んで，たとえば

“D は〈identifier〉 V と〈expression〉 E を用いて作った“**let** V **be** E”という形をしている”

ということを

“D は“**let** 〈identifier〉 V**be** 〈expression〉 E”の形をしている”

とも略記する．なお，**let** も **be** も〈basic symbol〉であるが，このように ALGOL **60** から容易に類推されるようなことを，今回は詳しくは説明しない．

1. 〈expression〉と *quantity*

〈expression〉の形の図形は，それぞれ，1つの *quantity* を表わす．*quantity* は抽象的な対象であって，それの *type*, *mode*（または *projection*），*value* というもので定められる3とおりの属性をもっている．たとえば，*type* が **real** である *quantity* のことを，**real**-*type* の *quantity* という．

〖第1版では，各 *mode* に *projection* が付随することになっていた．第2版作成中に，*projection* のほうを主役にしようという考えが起こった．第3版まで

にまだ変化があるかも知れない．今回のあらい解説の程度では，この件について，第1版と第2版にそう違いはない．〗

〈expression〉 E が表わす *quantity* の *type*, *mode*, *value* を，それぞれ，E の *type*, *mode*, *value* という．プログラムの *elaboration* が進行するとき，E の *mode* または *value* の変更はありうるが，E の *type* は変わらない．"*elaboration*" は，実行時の処理（コンパイルも含めて）の総称である（ALGOL **68** からの借用）．

type, *mode*, *value* の間には特定の対応関係があって，ある *type* の *value* などといい表わされる．1つの *quantity* がもつ *type*, *mode*, *value* はこの対応関係によって対応するものである．

2. *Type, Value, Mode*

2.1 *type* には

effect, real, bits, string, reference

という5とおりの基本 *type* と，これから出発して順に合成される *type* とがある．基本タイプはいずれも〈basic symbol〉である．*type* の合成には

array structure procedure

という3とおりの〈basic symbol〉のどれかが使われ，それに応じて，結果の *type* は **array**-*style*, **structure**-*style*, **procedure**-*style* の3種類に分類される．

各 *type* とそれの *value* は次のとおりである．

1) **effect**-*type* の *value* は，**done** というただ1つの *value* だけである．

2) **real**-*type* の *value* は実数である．

3) **bits**-*type* の *value* はビットの有限列，すなわち各項が **1** または **0** になっている有限列である．**1** も **0** も〈basic symbel〉であって，**1** は *true* に，**0** は *false* に相当する．

4) **string**-*type* の *value* は〈character〉の有限列である．〈character〉は〈basic symbol〉の一種である．〈letter〉の a や〈digit〉の 0 に対応して，〈character〉の â や 0̂ があり，a や 0 とは区別される．

5) **reference**-*type* の *value* は *quantity* である．Q_1 が *reference-type* の *quantity* であり，それの *value* が Q_2 であるとき，Q_1 が Q_2 を *refer* するという．

6) T が *type* であるとき

array T

は **array**-*style* の *type* である．この *type* の *value* は

$$\{\langle v,\ Q_v\rangle,\ \langle v+1,\ Q_{v+1}\rangle,\ \ldots\ \langle u,\ Q_u\rangle\}$$

の形の集合である．ただし，これの各要素 $\langle i,\ Q_i\rangle$ は i と Q_i の順序対，i は整数，Q_i は T-*type* の *quantity* である．

7) **structure**-*style* の *type* を作るには

(**)**

の形の括弧と，〈selector〉とを要する．〈selector〉とは，〈letter〉または〈digit〉を1個以上ならべた後に

:

という〈delimiter〉をつけた形である．さて S_1, S_2, ..., S_n が相異なる n 個の〈selector〉であり，T_1, T_2, ..., T_n が n 個の *type* であるとき

structure $(S_1T_1,\ S_2T_2,\ \ldots,\ S_nT_n)$

は **structure**-*style* の *type* である．この *type* の *value* は

$$\{\langle S_1,\ Q_1\rangle,\ \langle S_2,\ Q_2\rangle,\ \ldots,\ \langle S_n,\ Q_n\rangle\}$$

の形の集合である．ただし各 Q_i は T_i-*type* の *quantity* である．

8) T_1, T_2, ..., T_n, T が $n+1$ 個の *type* であるとき

(**procedure** $(T_1,\ T_2,\ \ldots,\ T_n)\ T$)

は **procedure**-*style* の *type* である．この *type* の *value* は，T_1-*type*, T_2-*type*, ..., T_n-*type* の n 個の相異なる〈identifier〉をともなった T-*type* の〈expression〉E である．E を *procedure body* といい，それにともなう〈identifier〉を *formal parameter* という．

2.2 T-*type* の *value* の全体という集合に対して，それのある種の部分集合 M を T-*type* の *mode* という．M-*mode* の *value* とは，M の要素のことである．たとえば，整数の全体は **real**-*type* の *mode* であり，長さ5のビット列の全体は **bits**-*type* の *mode* である．

各 *mode* には，それの *projection* という1つの *procedure* が付随している．M を T-*type* の *mode* とすれば，M の *projection* は T-*type* の *value* の全体から M への写像であって，特に M-*mode* の各 *value* にはそれ自身を対応させる，すなわち M 上では恒等写像になっている．

ある *value* V がある *quantity* Q に *assign* されるときには，Q の（*mode* に付随する）*projection* が V に作用して，その結果の *value* が，それまでの Q

の *value* の代わりに置き換えられる.

Mode と *projection* が特に指定されるのは, **real**-*type* と **bits**-*type* と **string**-*type* の場合だけである. それ以外の場合については, その *type* の *value* の全体という *mode* だけがあって, それの *projection* は恒等写像にすぎない.

3. 〈expression〉

〈expression〉は次の 17 とおりに分類される.

〈identifier〉	〈go to statement〉
〈block〉	〈closed expression〉
〈code〉	〈effect notation〉
〈real notation〉	〈bits notation〉
〈string notation〉	〈reference notation〉
〈array notation〉	〈structure notation〉
〈procedure notation〉	〈array element〉
〈structure element〉	〈procedure call〉
〈formula〉	

このうちで, 第1行

〈identifier〉 〈go to statement〉

から第6行

〈array notation〉 〈structure notation〉

までの 12 とおりを〈primary〉という.

また, 最後の〈formula〉を除いた残り 16 とおりを〈secondary〉という.

3.1 〈identifier〉の形の図形を *variable* といい, *quantity* の名前として使う.

3.2 〈go to statement〉は

"**goto** L"

という形で, L は〈identifier〉である. この L を *label* という. 上の〈go to statement〉の意味は

"L:"

という *labelling* のあるところに control を移すことである.

3.3 〈block〉は

"**begin** D_1; D_2; ..., D_n;
L_1E_1; L_2E_2; ..., L_kE_k **end**"

という形で, $n \geqq 1$, $k \geqq 1$, また

1) 各 D_i は, ある *variable* か〈fomula〉か〈mark〉について, この〈block〉の内部で有効なある性質を宣言する〈declaration〉

2) 各 L_j は *labelling* をいくつかならべた (空かも知れない) 図形

3) 各 E_j は〈expression〉

である. E_1, E_2, ..., E_n は順に *elaborate* され, その最終結果の *quantity* が, 〈block〉の *elaboration* の結果となる.

3.4 〈closed expression〉は

"(E)"

の形, E は〈expression〉である. 括弧は普通のように結合の優先順位を示す.

3.5 〈code〉は

"**code** $(S_1E_1, \ldots, S_nE_n)$ T: X"

の形で, $n \geqq 0$, また

1) $S_1, \ldots, S_n$ は相異なる〈selector〉
2) $E_1, \ldots, E_n$ は〈expression〉
3) T は *primary*
4) X は〈code body〉

である. (ALGOL **60** の〈code〉のように無規定のものは, ALGOL N では〈code body〉である.) 上の〈code〉は,

structure $(S_1E_1, \ldots, S_nE_n)$

というパラメタをつけて X を *elaborate* した結果の *quantity* を表わす. T は, その *quantity* の *type* が T の *type* と同じであることを予告している. この T のように, それ自身は *elaborate* されないで *type* を表わすだけに用いられる〈identifier〉を *typifier* という.

3.6 *Notation* 類として, 〈... notation〉の形式の名をもつものが, 8とおりある. ある *notation* が *elaborate* されるごとに, 1つの *quantity* が新しく *generate* されて, その *notation* はこの新しい *quantity* を表わすことになる.

〈effect notation〉は "**effect**" であって, 何の実効も生じない. これは *typifier* であり, ときには *extension* で省略される.

3.7 〈real notation〉と〈bits notation〉と〈string notation〉はそれぞれ

"**real** 〈modifier〉〈real donor〉"
"**bits** 〈modifier〉〈bits donor〉"
"**string** 〈modifier〉〈string donor〉"

の形をしている.

〈modifier〉の部分は空であるか, または "**[ ]**" の形であるか, または "**[**〈expression〉**]**" の形である. とくに, *extension* として

"**real [ ]**" は "**integer**" に
"**bits [ ]**" は "**Boolean**" に
"**string [ ]**" は "**charactor**" に

書き換えてもよい.

〈...donor〉の部分は空であるか，または上記の3種類に応じてそれぞれ〈number〉か〈bits〉か〈string〉である．〈number〉はALGOL **60**の〈unsigned number〉である．〈bits〉は**0**と**1**(または一方だけ)の，空でない有限列の形である．〈string〉は〈character〉の有限列を‘ ’で包んだ形である．

〈...donor〉の部分が空でなく，〈modifier〉の部分が空である場合については，下の細目に記すような *extension* があって，たとえば“**real** 3.14”は“3.14”と書き換えてよい．

〈real notation〉は **real**-*type* の *quantity* を表わす．

3.7.1　“**real**”が表わす *quantity* の *mode* は，*real* という（この言語の処理系に依存する）*mode* である．この *quantity* の *value* は任意の，すなわちこの言語としては規定しない，実数である．これらのことは *quantity* が *generate* されたときの状態であって，後で変化しうる．以下も同様．

3.7.2　“**integer**”すなわち“**real** [ ]”が表わす *quantity* の *mode* は，*integer* という（この言語の処理系に依存する）*mode* である．この *quantity* の *value* は任意の整数である．

3.7.3　“**real** [〈expression〉 *H*]”が表わす *quantity* は，*H* を *projection* とする *mode* と，これの任意の *value* とをもつ *quantity* である．

H の標準的な形は次のとおり．ただし *x* は *real-type* の *value*，また **round** は *integer* への丸めである．

scale *a*　　ただし *a* は **real**-*type* で >0

内容は $x\rightarrow$ **round** $(x/a)\times a$

b **scale** *a*　　ただし *a*, *b* は **real**-*type* で >0

内容は，**abs** $x<b$ の場合には

$x\rightarrow$ **round** $(x/a)\times a$

であり，これ以外の場合には規定されない．

precision *a*　　ただし *a* は *real-type* で >0

これに応ずる *mode* は1つの適当な精度の浮動小数点表示で表わされる実数の集合であって，その適当な精度は1と $1+a$ とが区別される精度である．

constant *a*　ただし *a* は **real**-*type* のもの

内容は $x\rightarrow a$

3.7.4　〈number〉*J* に小数点または指数部があるとき，“**real** *J*”を“*J*”に書き換えてもよい．

3.7.5　〈number〉*J* に小数点も指数部もないとき，“**integer** *J*”を“*J*”に書き換えてもよい．

3.7.6　*J* が〈number〉であるとき

“**real** *J*”

“**integer** *J*”すなわち“**real** [ ] *J*”

“**real** [*H*] *J*”

はそれぞれ *J* がない場合と同じ *mode* の *quantity* を表わし，これに *J* の（普通の用法での）*value* が *assign* される．

3.8　〈bits notation〉は **bits**-*type* の *quantity* を表わす．

3.8.1　“**bits**”が表わす *quantity* の *mode* は，*bits* という（この言語の処理系に依存する）*mode* であり，この集合の要素はある一定の整数をこえない長さのビット列である．この *quantity* の *value* は任意の有限ビット列である．

3.8.2　“**Boolean**”すなわち“**bits** [ ]”が表わす *quantity* の *mode* は集合 {**0**, **1**} であり，この *quantity* の *value* は**0**または**1**である．この *mode* の *projection* は **exact 1** である．ただし **exact** については **3.8.3** 参照．

3.8.3　“**bits** [〈expression〉 *H*]”が表わす *quantity* は，*H* をそれの *projection* とする *mode* と，この *mode* に属する任意の *value* をもつ *quantity* である．

H の標準的な形は次のとおり，ただし *x* は *bits-type* の *value* である．

exact *a*　　ただし *a* は **bits**-*type* で長さ *n*.

内容は $x\rightarrow x$ の後部を切り捨てるか，後に **0** をつけるかした，長さ *n* のビット列.

varying *a*　　ただし *a* は **bits**-*type* で長さ *n*.

内容は $x\rightarrow x$ の長さが $>n$ ならば *x* の第 *n* ビットより後を切り捨てたもの，そうでなければ *x* そのもの.

(*n* が整数のとき

$n*1$ は $\underbrace{1\,1\,\ldots\,1}_{n}$ の意味であり，

$n*0$ は $\underbrace{0\,0\,\ldots\,0}_{n}$ の意味である.

これらを上の *a* として使うことができる.)

constant *a*　　ただし *a* は **bits**-*type* のもの.

内容は $x\rightarrow a$

3.8.4　*J* が〈bits〉であるとき，“**bits** *J*”を単に“*J*”と書いてもよい．このとき

“**bits** *J*”

“**Boolean** *J*”すなわち“**bits** [ ] *J*”

"bits [*H*] *J*"

はそれぞれ *J* がない場合と同じ *mode* の *quantity* を表わし，これに *J* の *value* が *assign* される．

3.9 〈string notation〉は **string**-*type* の *quantity* を表わす．

3.9.1 **"string"** が表わす *quantity* の *mode* は，*string* という（この言語の処理系に依存する）*mode* であり，この集合の要素はある一定の整数をこえない長さの〈string〉である．この *quantity* の *value* は任意の〈string〉である．

3.9.2 **"character"** すなわち **"string [ ]"** が表わす *quantity* の *mode* は長さ1の〈string〉の全体という集合であり，この *quantity* の *value* はこの *mode* に属する任意の〈string〉である．この *mode* の *projection* は **exact filler** である．

3.9.3 〈expression〉*H* に対し，**"string [*H*]"** が表わす *quantity* は，*H* を *projection* とする *mode* と，この *mode* に属する任意の *value* とをもつ *quantity* である．

H の標準的な形は次のとおり．ただし *x* は **string**-*type* の *value* である．

exact *a*　ただし *a* は *string-type* で長さ *n*．
内容は *x*→*x* の後部を切り捨てるか，後に ^ という〈character〉をつけた，長さ *n* の〈string〉．

varying *a*　ただし *a* は **string**-*type* で長さ *n*．
内容は *x*→*x* の長さが $>n$ ならば *a* の第 *n* 字より後を切り捨てたもの，そうでなければ *x* そのもの．

constant *a*　ただし *a* は **string**-*type* のもの．
内容は *x*→*a*

（*n* が整数で *c* が〈character〉であるとき，
*n**'*c*' は '$\underbrace{c\ c\ \ldots\ c}_{n}$' という長さ *n* の
〈string〉を意味する）．

3.9.4 *J* が〈string〉であるとき，**"string *J*"** を単に **"*J*"** と書いてもよい．このとき

"string *J*"
"character *J*" すなわち **"string [ ] *J*"**
"string [*H*] *J*"

はそれぞれ *J* がない場合と同じ *mode* の *quantity* を表わし，これに *J* の *value* が *assign* される．

3.10 〈reference notation〉は **"reference"** であって，**reference**-*type* の *quantity* を表わす．これは任意の **reference**-*value* をもち，何かある *quantity* を *refer* する．

3.11 〈array notation〉には **3.11.1**～**3.11.5** に示す諸種の形がある．

3.11.1 〈array notation〉の基本形は

"array [E_1: E_2] (F_1, F_2, ..., F_n)"

であって，$n \geqq 1$，また

1) E_1 と E_2 は *real-type* の〈expression〉
2) F_1, F_2, ... F_n はある同じ *type* T をもつ n 個の〈expression〉

である．この形の *notation* が表わす *quantity* Q の *type* は **array** T であり，*value* は次のようにして定められる．

v を E_1 の *value*，u を E_2 の *value* とする．
$v>u$ ならば，Q の *value* は空集合である．
$v \leqq u$ ならば，Q の *value* は集合

$$\{\langle v, Q_v\rangle, \langle v+1, Q_{v+1}\rangle, \ldots, \langle u, Q_u\rangle\}$$

である．ただし

$$Q_v,\ Q_{v+1},\ \ldots,\ Q_u$$

はそれぞれ

$$F_1,\ F_2,\ \ldots,\ F_{u-v+1}$$

が表わす *quantity* であるが，ここで $n<u-v+1$ ならば F_n を繰り返して F_{n+1}, F_{n+2}, ... とする．

3.11.2 **"array [E_1] (F_1, F_2, ..., F_n)"** は

"array [v: u] (F_1, F_2, ..., F_n)"

と同値である．ただし v は E_1 の *value* とし，u は $v+n-1$ とする．正しくは，v と u を *value* とする2つの〈real notation〉を上記 [v: u] の v と u の代わりに書くのである．

3.11.3 **"array [E_2] (F_1, F_2, ..., F_n)"** は

"array [1: E_2] (F_1, F_2, ..., F_n)"

と同値である．

3.11.4 **"array (F_1, F_2, ..., F_n)"**

"array [ ] (F_1, F_2, ..., F_n)"

はいずれも

"array [1: n] (F_1, F_2, ..., F_n)"

と同値である．

3.11.5 F が〈primary〉であって，〈mark〉以外の〈delimiter〉で終わっているとき，F を *primitive* という．〖この概念を追放しようという考えもあり，いま検討中である．〗　このとき

"array [E_1: E_2] F"

は

"array [E_1: E_2] (F)"

と同値である．このFがさらに

"**array** [E_1': E_2'] (F_1', F_2', ..., F_m')"

の形であれば，前記の〈array notation〉を

"**array** [E_1: E_2, E_1': E_2']
(F_1', F_2', ..., F_m')"

の形に書いてもよい．これは特別な形の2次元 *array* を簡単に書いたものであるが，3次元や4次元のものについても同様な記法が許される．

〖一般に"] **array** ["の代わりに", "とする *extension* がある．これを

"**array** [E_1: E_2] **array** [E_1': E_2']
(F_1', F_2', ..., F_m')"

に適用すると，**3.11.5** の最後に示した形ができる．〗

3.12　〈structure notation〉は

"**structure** (S_1E_1, S_2E_2, ..., S_nE_n)"

の形で，$n \geq 1$，また

1) 各 S_i は〈selector〉

2) 各 E_i は〈expression〉

である．この *notation* は次のような *quantity* Q を表わす．各 E_i の *type* を T_i とし，E_i が表わす *quantity* を Q_i とすれば，Q の *type* は

structure (S_1T_1, S_2T_2, ..., S_nT_n)

であり，Q の *value* は集合

{⟨S_1, Q_1⟩, ⟨S_2, Q_2⟩, ..., ⟨S_n, Q_n⟩}

である．

3.13　〈procedure notation〉

3.13.1　$n \geq 0$ で E_1, E_2, ..., E_n が n 個の〈expression〉，E が〈primary〉であるとき

"**procedure** (E_1, E_2, ..., E_n)E"

は〈procedure notation〉である．これが表わす *quantity* の *type* は

(*procedure* (T_1, T_2, ..., T_n)T)

であり，*value* はこの *type* の任意の *value* である．ただし各 T_i は E_i の *type*，T は E の *type* である．

3.13.2　E_1, E_2, ..., E_n, E が **3.13.1** と同様であって，さらに V_1, V_2, ..., V_n が n 個の相異なる〈identifier〉，また F が E と同じ *type* の〈primary〉であるとき

"**procedure** (E_1, E_2, ..., E_n)E
: (V_1, V_2, ..., V_n)F"

および

"**procedure** (E_1 **name** V_1, E_2 **name** V_2,
..., E_n **name** V_n) E: F"

はたがいに同値な〈procedure notation〉であり，これが表わす quantity の type は **3.13.1** と同様，*value* は T_1, T_2, ..., T_n という *type* の *formal parameter* V_1, V_2, ..., V_n をもった *procedure body* F である．

この *quantity* をさす *procedure* が F_1, F_2, ..., F_n という *actual parameter* で呼び出されるとき，*parameter* は ALGOL **60** の場合と同じ意味の *call-by-name* で処理される．

もしも *call-by-quantity*，すなわち

"*procedure body* を *elaborate* する前に，*actual parameter* F_i を *elaborate* して，その結果の *quantity* を *formal parameter* が表わすことにする".

というメカニズムが必要ならば，**2.13.2** の最初の形では"V_i"を"**quantity** V_i"に書き換え，第2の形では"E_i **name** V_i"を"E_i **quantity** V_i"に書き換える．特に E_i が *primitive* ならば，"E_i **quantity** V_i"を"E_iV_i"に書き換えてもよい．

もしも ALGOL **60** でいうような *call-by-value* が必要ならば，*formal parameter* V_i を *call-by-quantity* のものにして，*actual parameter* "F_i" を"**copy** F_i" または"**new** F_i" に書き換える．ここで **copy** を使うか **new** を使うかは，F_i が表わす *quantity* の *value* そのもの (*surface value* ともいう) が必要か，それをすこし変更した "*deep value*" というものが必要かによる．この変更が実際に行なわれるのは，問題の *quantity* の *type* が **array**-*style* か **structure**-*style* になっている場合である．これの説明は次回以降にゆずる．

3.14　〈array element〉は

"$E[F]$"

の形である．ただし E は *array* スタイルの *type* をもつ〈secondary〉，また F は **real**-*type* の〈expression〉とする．

E の *value* を

{⟨v, Q_v⟩, ⟨$v+1$, Q_{v+1}⟩, ..., ⟨u, Q_u⟩}

とし，F の *value* を整数に丸めて i とすれば，付帯条件 $v \leq i \leq u$ のもとで，上の〈array element〉は Q_i を表わす．

3.15　〈structure element〉は

"$E[S]$"

の形である．ただし E は

structure (S_1T_1, S_2T_2, ..., S_nT_n)

という *type* をもち，S は S_1, S_2, ..., S_n のどれ

かであるとする.

E の *value* が

$\{\langle S_1, Q_1\rangle, \langle S_2, Q_2\rangle, \ldots, \langle S_n, Q_n\rangle\}$

であって, S が S_i であれば, 上記の 〈structure element〉 は Q_i を表わす.

〈array element〉 の 〈array element〉 とでもいうべき $E[F_1][F_2]$ とか, 〈structure element〉 の 〈array element〉 とでもいうべき $E[S][F]$ などにおいて, "][" という組合せは "," で置き換えてもよく, もっと簡単な形

$E[F_1, F_2]$ $E[S, F]$

などを使うことができる.

3.16 〈procedure call〉 は

"$E(F_1, F_2, \ldots, F_n)$"

の形である. ただし

1) E は 〈secondary〉 でそれの *type* は (**procedure** $(T_1, T_2, \ldots, T_n)T$)
2) $F_1, F_2, \ldots, F_n$ はそれぞれ $T_1, T_2, \ldots, T_n$ という *type* の 〈expression〉
3) E の *value* は, T-*type* の *procedure body* E' に, それぞれ $T_1, T_2, \ldots, T_n$ を *type* とする *formal parameter* $V_1, V_2, \ldots, V_n$ がともなったもの

であるとする.

この 〈procedure call〉 が *elaborate* されるとき, *actual parameter* $F_1, F_2, \ldots, F_n$ は適当なしかた (**3.13.2** 参照) でそれぞれ $V_1, V_2, \ldots, V_n$ に結びつき, 次に E' が *elaborate* される. E' を *elaborate* した結果の *quantity* を $E(F_1, F_2, \ldots, F_n)$ が表わすことになる.

3.17 〈formula〉 は, 1個以上の 〈mark〉 で 〈expression〉 あるいは空な図形を結びつけた形である. このとき結びつけられる 〈expression〉 をすべて ( ) で置き換えた結果を 〈frame〉 という.

たとえば3個の 〈expression〉 F_1, F_2, F_3 と2個の 〈mark〉 M_1, M_2 から作られた

"$F_1M_1F_2M_2F_3$"

の形の 〈formula〉 があり, 各 F_i は *operand* と呼ばれる. この 〈formula〉 は,

$(\ )M_1(\ )M_2(\ )$

という 〈frame〉 と *operand* の *type* とに合った適当な 〈expression〉 E により,

$E(F_1, F_2, F_3)$

という 〈procedure call〉 と同値になる. 他の形の 〈formula〉 についても同様である.

4. 〈declaration〉

〈decleration〉 には

〈variable declaration〉
〈formula declaration〉
〈mark declaration〉

の3とおりがあり, いずれも **let** で始まる.

4.1 〈variable declaration〉 は

"**let** V **be** E"

という形であって, 次のことを宣言する.

1) 〈identifier〉 V が 〈block〉 の内部で *variable* として使われる.
2) V が表わす *quantity* は, 〈block〉 にはいるとき 〈expression〉 E を *elaborate* して得た結果の *quantity* である.

同じ 〈expression〉 E をもつ 〈variable declaration〉 の列

"**let** V_1 **be** E;
let V_2 **be** E;
...
let V_n **be** E;"

を

"**let** $V_1, V_2, \ldots, V_n$ **be** E;"

と書いてもよい. さらに, E が *primitive* ならば

"$E\,V_1, V_2, \ldots, V_n$;"

と書いてもよい.

4.2 〈formula declaration〉 は

"**let** H **represent** E"

の形である. 〈expression〉 E の *type* が

(**procedure** $(T_1, T_2 \ldots, T_n)T$)

であるとき, 上記の 〈declaration〉 は

"〈frame〉 H と, それぞれ $T_1, T_2, \ldots, T_n$ を *type* とする *operand* $F_1, F_2, \ldots, F_n$ とをもつ 〈formula〉 は, その 〈block〉 の中で, 〈procedure call〉

"$E(F_1, F_2, \ldots, F_n)$"

に同値である"

ということを宣言する.

同じ 〈frame〉 H をもつ 〈formula declaration〉 の列

"**let** H **represent** E_1;
let H **represent** E_2;
....
let H **represent** E_n;"

を

"let H **reprenspent** E_1, E_2, ..., E_n;"
に書き換えてもよい.

4.3 〈mark declaration〉は

"let M **operate before... left**
after... right"

の形か, この中で **"before... left"** の部分または **"after... right"** の部分(またはその両方)を削った形で, M は〈mark〉である. また **before** と **left** の間, **after** と **right** の間は, いくつか(0個でもよい)の〈mark〉を","で区切って並べた列か, **"all"** かである. 〈mark declaration〉は, 〈block〉の中で, 〈mark〉の *facing* という属性と *priority* という関係とを宣言する.

1つの〈mark〉の *facing* は *double-faced* か *left-faced* か *right-faced* か *non-faced* かである.

1) **"left** M **operate before... left**
after... right"

では, 〈mark〉M は *double-faced* と宣言される. この *facing* の M は, 〈frame〉の中にそれ1つだけで用いられる. たとえば ( ):=( ) の中の := や, +( ) の中の + などはそうである.

2) **"left** M **operate before... left"** では, 〈mark〉M は *left-faced* と宣言される. この *facing* の M は, 〈frame〉の中で〈mark〉の先頭として用いられる. たとえば **case** ( ) **of** ( ) の中の **case** や, ( ) **replacing first** ( ) **with** ( ) の中の **replacing** などはそうである.

3) **"left** M **operate after... right"**

では, 〈mark〉M は *right-faced* と宣言される. この *facing* の Mは, 〈frame〉の中で〈mark〉の末尾として用いられる. たとえば **for** ( ) **:=**( ) **do** ( ) の中の **do** などはそうである.

4) **"let** M **operate"**

では, 〈mark〉M は *non-faced* と宣言される. この *facing* の M は, 〈frame〉の中の〈mark〉のうちで中間のものとして用いられる. たとえば **for** ( ) **:=** ( ) **do** ( ) の中の := などはそうである.

5) 2つの〈**mark**〉M, N に対して,

"let M **operate... after...** N**... right"**

"let M **operate... after all right"**

"let N **operate before...** M**... left..."**

"let N **operate before all left..."**

のどれかの形の〈declaration〉があるとき, 順序対〈M, N〉が *reverse* に宣言されているという.

そのような〈declaration〉がないとき, 順序対は *natural* に宣言されているという.

Priority は〈mark〉による結合の先後を定める. 2つの〈mark〉M と〈mark〉N がこの順序で, 〈mark〉としては隣りあっているとき, もしも〈M, N〉が natural に宣言されていれば結合は左から右への順に処理され, もしも〈M, N〉が reverse に宣言されていれば結合は右から左への順に処理される.

〖*Facing* によって〈mark〉を分類してしまうと不便な点もある. 〈mark declaration〉をもっと自由なものにするための方法が考案されたので, 第3版はそれによることがほぼ確実である.〗 (**続く**)

(昭和46年3月4日受付)