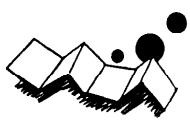
解　説



ALGOL 68 とその処理系 (1)†

川　合　慧††

1. はじめに

計算機用の高水準言語と称するものが世に現われてからかなりの時間がこれまでに経過した．それらの言語の多くは，場当り的な改良や機能の新設，あるいは特定の計算機上での効率改善などが主目的であったり，数学的な完璧さのみを追求した非実用的なものであったりする傾向があった．プログラム言語として必要な概念を整理し，人間・機械系としてのプログラミングに重点を置いた言語設計が行われるようになったのはそれほど古いことではない．そのような中にあって，IFIP が ALGOL 60 を作り，さらに ALGOL 68 の設計を行ったことは，単に処理系を使用したプログラミングを行う可能性ばかりでなく，プログラム言語の本質を明らかにするうえで貴重な活動であったということができる．

ALGOL 60 は，1960 年代の'実行能率至上主義'による評価基準に合わないところも多く，FORTRAN にくらべて使用される割合はずっと低かった．特に我が国においてその傾向は強い．ALGOL 68 は，この事情を受け継いでしまったように見受けられ，現在に至るまで実用的なプログラム言語としての王座を占めたことはない．しかし，プログラム言語に対する要求が高実行能率からプログラム自体の生産性へと移りつつある今日において，ALGOL 68 は以下の 2 点に関して重要な言語であると思われる．

1) プログラム言語としてかなり高い水準であるにもかかわらず，十分に効率的なコンパイルが可能であり，目的コードの実行効率も高くし得る．実用的なコンパイラが数多く開発されているのがその証拠である．

2) プログラム言語に含まれる概念がよく整理されており，言語自体の設計や開発を行う場合の拠所となり得る．

本解説では，日本において処理系の作成はおろか，言語の内容理解さえもほとんどされずにきた ALGOL 68 に関して，成立の経緯と一般的な内容の説明，言語の概要と実際のプログラミングについて述べる．

2. 成立の経緯

2.1 ALGOL-X

IFIP（国際情報処理連合）は，1958 年から 1962 年にかけて，電子計算機の機械語に直接翻訳できる簡潔なアルゴリズム表現のための言語として ALGOL 60 を設計した．ALGOL 60 の当初考えられていた適用分野は広範囲にわたる数値計算であったが，その後計算機の応用が非数値処理の分野にまで広がっていっても，ALGOL 60 の重要性は減らず，むしろ種々の高水準言語の原点として重要な役割を果してきたことは周知のとおりである．ALGOL 60 に関する知識は情報科学・情報処理の分野において必須のものであるといっても過言ではない．しかし，その後の計算機の発展によって言語のより高水準化の要求が高まり，非数値的アルゴリズム記述の必要性もあいまって，IFIP では，ALGOL 60 の後継言語を ALGOL-X と名づけ，作業グループ WG 2.1 がその設計作業を始めた．1963 年のころである．そして，1965 年 10 月の会合において提出された言語素案 3 案のうち，新しい言語記述法を使用した A. van Wijngaarden の案が採用された．ちなみにほかの 2 案の提出者は G. Seegmueller と N. Wirth である．また，同じ会合に提出された C. A. R. Hoare による 'Record Handling' という題の小論は，プログラム言語における重要概念のひとつであるデータ構造について論じたものであり，このあと設計作業が進められた ALGOL 68 と，N. Wirth が独自に開発することになる言語 PASCAL との両方に強い影響を与えた．

2.2 ALGOL 68 の成立

Wijngaarden のグループはこのあとも言語仕様を

† ALGOL 68 and its Compilers (I) by Satoru KAWAI (Faculty of Science, University of Tokyo).

†† 東京大学理学部

確定するための作業を続け，MR 88, MR 92 等と名づけられた中間報告書を発表していった．その目的は，作業の現状を公表するとともに，プログラム言語の専門家や使用者の意見および提案を，次に作成する版に取り入れることにあった．グループは Wijngaarden のほか，B. Mailloux, J. E. L. Peck, C. H. A. Koster から成っていた．そして，最終報告書である MR 100 が 1968 年 12 月に WG 2.1 において，さらに翌年 IFIP そのものによって承認された．

この初版 ALGOL 68[28] の特徴は，当時としてはかなり意欲的な言語仕様と，のちに W 文法と呼ばれるようになった二段階文法[29]を記述に使用[2]したことであった．言語仕様は ALGOL 60 とくらべてかなり複雑となり，ほかの言語においては平文で記述されている構文の意味を形式言語で表現する努力をした W 文法の複雑さも手伝って，「ALGOL 68 怪獣論」がささやかれたりした．報告書の厚さも 130 頁を越す部厚いものであった．MR 99 や MR 100 が発表されたころ，日本において島内らのグループが ALGOL-N を設計・発表したのもこの ALGOL 68 の大きさと複雑さに対する批判の意味からであった．PASCAL を設計した Wirth も同様な考えを持っていたと思われる．IFIP はこの事情に対処するため，報告書よりはずっと 'ユーザ向け' の ALGOL 68 手引書[17]を 1971 年に作成しているが，これもかなり細かい記述文書であり，全くの初心者を対象としたものではなかった．

2.3 改訂版の計画と作成

1969 年に承認された ALGOL 68 を実用的な言語として世に広めるために，IFIP の TC 2 は，ALGOL 68 の具体化に関する会議を翌年の 1970 年 7 月にミュンヘンで開催した[2),7),15]．この会議では ALGOL 68 の処理系の実現のためのさまざまな問題が討議されたが，それと共に，この言語の持つ問題点の指摘や拡張・変更の提案が数多くなされた．特に，この会議開催の時点で，英国 Royal Radar Establishment において処理系 ALGOL 68-R[7] がすでに実用的な計算業務に使用されていたことが注目された．

この会議の成果およびほかの数々の研究と使用経験とにより得られた文法と言語双方に対する改良提案[14]をもととして，前記のメンバに M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisker の 4 名を加えた作業グループが改訂作業を行った．そして，言語を改良し記述法に新しいくふうを取り入れた改訂版が 1973 年の WG 2.1 の会合で承認された．これが改訂 ALGOL 68[29] である．言語の全容は 1975 年に発表され，手引書も並行して改訂された[17] (1977 年)．今日，単に ALGOL 68 といえばこの改訂版を意味し，最初のものは旧版と呼ばれる．その後の WG 2.1 の活動の中心はより抽象レベルの高い設計言語へと移ったので，改訂 ALGOL 68 がこの水準のプログラム言語として IFIP が認めた最後のものとなった．

2.4 その後の動き

ALGOL 68 の言語自体は改訂報告書に記述されたものとして固定されており，その後の言語研究によって形成されてきた種々の概念，たとえば名前の有効範囲の制御，高水準同期制御などを取り込む変更は行われていない．現在まで継続して続けられている作業としては，部分言語の設定，入出力仕様の明確化，分割コンパイル処理などがある．

部分言語については，改訂作業以前[15]からその必要性が認められていた．部分言語は，もとの言語の機能のうちあまり使われないと思われるものを取り除いたもので，処理系の負担を軽くし，かつ，言語の習得と使用とを容易にするのが目的の言語[1]である．最終的に決められた部分言語[10]が持たない機能のうち主なものは，合併型，動的可変長配列，並列処理などである．

ALGOL 68 の入出力仕様の大半は，標準前置部と呼ばれる ALGOL 68 自身で書かれたプログラムによって規定されている．これらも改訂報告書に含まれているが，その後指摘された種々の不備を正す作業が続けられている．分割コンパイルについては，ALGOL 68 を巨大システム構築の手段とするために，モジュール化と分割処理方式の提案[16]がなされている．こちらの方はまだ最終的に確定はしていない．

2.5 改訂版における主な変更点

旧版と改訂版との差はそれほど大きなものではないが，言語の見かけ上の姿，および記述の方法が少し異なっている．旧版をご存知の読者のために，主な相違点を示しておく．

a) 制御構造 (条件，反復，場合分け) が正式に取り入れられ，形式が整備された．また，これらすべてが宣言の有効範囲を局所化する機能を与えられた．

b) 言語の拡張 (extension) として決められていたものの多くが正式の構文として取り入れられた．制御構造はその例．ほかに，たとえば

ref real x = **loc real**

のかわりであった

real x

などがその例．取り入れられなかったものもある．たとえば

mode a=**struct** (…)

や **mode** b=**union** (…)

の略記法であった

struct a=(…) や **union** b=(…)

および並列処理のための略記法など．

c） ふたつの値の型の一致を検査する演算子が削除され，かわりに合併型の値による場合わけ構文（要素型による）が追加された．

d） 構文評価のための環境を与える有効な宣言の集合を表わす構文変数と，生成規則の条件付適用を可能とした述語との使用により，平文で記述されていた文脈条件が不必要となった．

3. 言語記述の手法

3.1 BNF とその拡張

ALGOL 60 の文法記述に使われた BNF (Backus Naur Form) は，言語の構文記述に革命的進歩をもたらした．平文による長々とした，しかもしばしば曖昧さを含んだ構文記述では，今日における複雑なプログラム言語は規定できなかったであろう．文脈自由文法と対応する BNF は，処理系の設計に役立つこともあって，プログラム言語とその処理系の発展に大きな寄与をしている．その後いくつか現われた記法も，BNF 中によく現われるパターンや反復構造を簡潔に表現することが主な目的であった．次に示すのがその代表例であろう．

⟨empty⟩|⟨α⟩ ⟶ [⟨α⟩]

⟨empty⟩|⟨α⟩|⟨α⟩⟨α⟩|…

⟶ {⟨α⟩}

⟨α⟩|⟨α⟩⟨β⟩⟨α⟩|⟨α⟩⟨β⟩⟨α⟩⟨β⟩⟨α⟩|…

⟶ ⟨α⟩{⟨β⟩}$_{\infty}$

これらの記法は強力なものであり文法規則の簡潔でわかりやすい記述には役立ったが，しょせん‘字づら’の形だけを表わすものであり，構文要素の意味，あるいは各種属性による展開形式の制限などは，依然として平文で書かれねばならなかった．

3.2 2段階文法

プログラム言語に含まれている‘意味’や‘制限’の簡単な例として，式の型について考えてみよう．たとえば，次のような条件文の構文があったとする．

⟨if statement⟩ ::=

if ⟨expression⟩ **then** ⟨statement⟩{;}$_{\infty}$ **fi**

先頭にある式の値は，ふつうは論理型でなければならない．つまり，「**if** に続く式の型は論理型に限る」という一文が注釈として必要となる．この平文を文法に組み込むひとつの方法として，構文要素に式の型まで含めてしまうことが考えられる．上の例では，

⟨if statement⟩ ::=

if ⟨Boolean expression⟩ **then** … **fi**

となる．これを条件文の唯一の生成規則としておけば，たとえば，**if** の後に整数型の式が書いてあるテキストは‘この言語に含まれる文章（プログラム）ではない’と言い切ることができるわけである．別の例として代入文を取り上げよう．

⟨assignment⟩::=⟨variable⟩:=⟨expression⟩

これが普通の書き方であるが，このほかに両辺の型の一致に関する注釈文が必要となる．型を構文要素に含める方針でゆくと，以下のようになる．ここで，代入を許される値の型は下の3種しかないものとする．

⟨integer assignment⟩ ::=

⟨integer variable⟩:=⟨integer expression⟩

⟨real assignment⟩::=

⟨real variable⟩ :=⟨real expression⟩

⟨Boolean assignment⟩ ::=

⟨Boolean variable⟩ :=⟨Boolean expression⟩

この記述法は，平文による補助記述を要しないという意味で厳密なものではあるが，上の例でわかるとおり生成規則の数が非常に多くなるという欠点のほかに，プログラマが自分で定義した型についての規則をあらかじめ与えておくことができないという，重大な問題を含んでいる．

生成規則の数は，属性（上例では型）を示す‘構文変数’を設けることで減らすことができる．たとえば，型を示す構文変数 TYPE を用意すれば，上の代入の例は

⟨TYPE assignment⟩ ::=

⟨TYPE variable⟩ :=⟨TYPE expression⟩

となり，これとは別に TYPE に対して

TYPE :: integer; real; Boolean.

という生成規則が用意される．実際の代入文の構文は，3個の TYPE に integer, real, Boolean のいずれかひとつを‘一斉に代入する’(consistent substitution) ことで得られる．すなわち，最終的な生成規則が，一段上のメタな生成規則によって生成されることにな

る．この方式が2段階文法 (two level grammar) と呼ばれるものであり，ALGOL 68 を記述している文法の基礎となっている．

3.3 Wijngaarden 文法 (W 文法)

W文法自体はかなり複雑なものであるので[6),19)]，以下その概要を示すにとどめる．

a) 文法を構成する要素は次のとおり

notion…太い小文字と空白の列で，生成規則が与えられているもの．BNF の構文単位に相当する．

例；exponent part

-times ten to the power choice

metanotion…太い大文字の列で，(メタ) 生成規則が与えられているもの．前節で構文変数と呼んだものである．

例：**INTREAL SIZETY MODE**

hypernotion… 太い小文字と空白と metanotion の列．種々の生成規則に使われる．

例：**reference to INTREAL**
SIZETY integral

symbol…**symbol** で終る太い小文字と空白の列．生成規則による置換の最終段階は symbol の列になる．各 symbol を具体的な表現記号に置き換えればプログラムテキストが得られる．

例：**times ten to the power symbol**
letter e symbol
plus symbol

b) 生成規則の種類

production rule…通常の意味の生成規則．そのまま与えられることもあるが，大部分は下記の hyper-rule から導出される．

例：**exponent part:**
times ten to the power choice,
power of ten.

(**exponent part** が ':' に続くふたつの notion を並べたものに置き換えられることを示す．右辺における選択肢は ';' で区切って並べる．)．

times ten to the power choice:
times ten to the power symbol;
letter e symbol.

metaproduction rule…metanotion (構文変数) に対する生成規則．

例：**INTREAL ::**
SIZETY integral; SIZETY real.

(**INTREAL** が '::' に続く (';' で区切られた) ふたつの hypernotion のどちらにでも置き換えられることを示す．右辺の hypernotion の中に左辺の metanotion が現われる (再帰) ことも可能．)

SIZETY ::
long LONGSETY;
short SHORTSETY; EMPTY.
LONGSETY ::
long LONGSETY; EMPTY.
SHORTSETY ::
short SHORTSETY; EMPTY.

hyper-rule…hypernotion に対する生成規則で構文を記述する規則の雛形となるもの．ALGOL 68 の一般的な構文はこの hyper-rule の形式で与えられる．ふつうは metanotion を含んでおり，それを metaproduction rule で置換してゆくと production rule が '生成' される．置換は前述したとおり一斉に行う．

例：代入

REF to MODE NEST assignation:
REF to MODE NEST destination,
becomes token, MODE NEST source.

(**MODE** は ALGOL 68 で可能な型のひとつを，**NEST** は各時点で有効な宣言の集合を，また，**REF** はポインタを，それぞれ表わす metanotion．すなわちこの規則は，**MODE** 型の値の代入が <左辺> ≔ <右辺> (':=' が **becomes token**) の形式であり，左辺は変数 (**REF** が示す) に限り，両辺の評価を同じ環境 (宣言の集合，**NEST**) で行った場合に同じ型 (**MODE**) の値となることを要求している．)

この例でもわかるように，W文法の生成規則の左辺は文脈依存型となっている．一般に，形式言語において最も広い帰納的に可算なすべての言語に対して，それを生成する W 文法が存在することが知られている[24)]．

3.4 改訂版における記述法の改良

W文法は旧版，改訂版双方の記述に使用されたが，改訂版では以下のような記述法の改良がなされている．これらによって，必要な平文の量はさらに少なくなった．

a) 評価の環境を示す metanotion **NEST** が設けられた．**NEST** は，それに対する metaproduction rule を適当な回数適用すると，結局次のような形式に

なる.

new {DEC} {LAB} **new** {DEC} {LAB}…
…**new** {DEC} {LAB}

DEC は変数，定数，演算子の宣言を，**LAB** はラベルの宣言を示す．これらの宣言を含む構文を置換すると，それまでの **NEST** 末尾に **new** {DEC} {LAB} が新しくつけ加わる．つまり，**NEST** は一次元形式の名前表で，宣言局所化の目印として **new** が置かれているわけである．そして，文，式，変数名等，およそ評価の対象となるものの構文にはすべてこの **NEST** が含まれており，評価の環境を定めている．これによって，名前の使用とその宣言との対応，2重宣言の禁止などが全部生成規則に組み込まれ，旧版に平文の形で入っていた文脈条件 (context condition) が不要となった．

b) 生成規則の中に細かな条件を書く方法として，条件が成立すれば empty となり，成立しなければ'それ以後の置換が定義されない'(つまりプログラムテキストが生成され得ない) という，述語 (predicate) と呼ばれる構文が採用された．条件不成立の場合は，生成過程全体を袋小路 (blind alley) に追い込んでしまうわけである．述語は **where** あるいは **unless** で始まる構文単位であり，置換の結果

where false あるいは **unless true**

が出てくれば袋小路となる．述語に関する一般的な規則の例と用法の一例を示す．

where THING1 and THING2:
where THING1, where THING2.
(両方成立したときのみ成立．結果は empty.)
where THING1 or THING2:
where THING1; where THING2.
(どちらか一方が成立すればよい.)

このほかに，ふたつの notion **A, B** に対して

where (A) is (B), where (A) begins with (B)
where (A) contains (B)

などが定義されている．用法の一例として **NEST** における名前の使用と宣言との同定規則を示す (一部略).

```
PROP :: DEC; LAB.
PROPS :: PROP; PROPS PROP.
PROPSETY :: PROPS; EMPTY.
where PROP identified in
  NEST new PROPSETY:
    where PROP resides in PROPSETY,
      where PROP independent PROPSETY,
        where PROP identified in NEST.
```

4番目の(6行にわたる)規則は，使用された名前とその属性 (**PROP**) が，名前表 **NEST new PROPSETY** の中に存在するかどうかを検査するためのものである．第3行目で，もしそれが名前表の最新の部分 (**PROPSETY**) にあれば (**resides**)，成功の印として empty に置きかわることが示され，そうでなければ (**independent**) その外側 (**NEST**) における検査の結果が全体の結果となる (第6行目)．**resides** や **independent** の内容を与える規則も存在することは言うまでもない．

4. 言語の基本概念

4.1 値 と 型

ALGOL 60 における型の概念は，計算機内におけるビット列の解釈の種類を定式化したものであった．前出の Hoare の小論は，より高い抽象水準で考えた'値'の集合として型の概念をとらえたものであった．その基礎は，単純で現実の計算機上での実現性の高い基本となる型の定義と，種々の型を組み合わせてより複雑な構造の型を作ってゆく構造化手段である．ALGOL 68 もこの線を追求したわけであるが，後で述べる型の同値性に関してはほかの同類言語とはかなり異なった扱い (構造同値) をしている．

ALGOL 68 では型のことをモード (mode) と呼ぶが，単純な型として次のものを用意している．

整数 (**int,** integral) 実数 (**real**)
論理値 (**bool,** Boolean)
文字 (**char,** character)

この4個の単純型の値は，直接的な表記法が用意されている (1980, 1.23e-3, **false,** "Q" など)．整数と実数とに関しては，桁数や精度を示すための'大きさ'が指定できる．たとえば，半語で済む整数は **short int**，4倍精度実数は **long long real** というぐあいである．**long** や **short** を何個までつけられるかは処理系ごとに異なる．

4.2 データの構造化

構造化手段としては，構造 (structure)，配列 (row)，参照 (reference)，合併 (union) が用意されている．

a) 構造…種々の型の値を見出し名つきで並べたもの．個々の要素値は見出し名を指定することによって選択される．構造全体の型は，要素値の型と見出し

名の対の順序つき集合であらわされる．たとえば，整数値，文字値，実数値をこの順にひとつづつ，見出し名 *a*, *b*, *c* をそれぞれにつけて構成した構造値の型は

struct (**int** *a*, **char** *b*, **real** *c*)

となる．また，この型の値として *valstruct* と名づけられたものがある場合，要素値の参照は次のように行う．

a **of** *valstruct*, *b* **of** *valstruct*, *c* **of** *valstruct*

同種の構造データを扱うほかのプログラム言語（たとえば PASCAL や COBOL）とは，指定の順が逆になっていることに注意．(PASCAL での指定は *valstruct*. *a* 等とする)

ALGOL 68 には標準的な型として複素数型 **compl** が用意されているが，これは構造型

struct (**real** *re*, *im*)

と等価である．また，この型の値に対しては以下のような演算子が用意されている．

i 複素数値を作る	3.5 ***i*** 5.3
re im 実数（虚数）部分	**re** *z*1, **im** *z*2
abs 絶対値	**abs** (3 ***i*** 4)=5
arg 偏角	**arg** *z*3
conj 共役複素数	**conj** (2 ***i*** 3)=2 ***i***−3

b） 配列…同一の型の値を順に並べたもの．並べ方は何次元でもよいが，添字は整数型に限られる．要素値の並びのほかに，各次元の添字値の上下限の組を並べた記述子 (descriptor) と呼ばれるものも配列値の一部となっている．配列全体の型は，要素値の型と次元数とであらわされる．たとえば，4 次元の整数値配列と，2 次元の論理値配列の型はそれぞれ

[,,,**] int** **[**,**] bool**

となる．型には添字の上下限値が含まれておらず，したがって大きさ不明の配列値の受渡しが可能である．実行時に配列の添字値の範囲を知るには，記述子の内容を取り出す演算子 **upb**（上限値）と **lwb**（下限値）とを使う．たとえば，*a* という名前の **[**,**] real** 型の記述子は

((1 **lwb** *a*, 1 **upb** *a*), (2 **lwb** *a*, 2 **upb** *a*))

である．記述子の構成からもわかるとおり，配列値を要素とする配列型と，それらをまとめた（大きな）配列型とは異なる型である．たとえば

[,**][ ] real** と **[**,,**] real**

は異なる型である．

配列値の個々の要素値は添字づけ (subscripting) によって選ばれる．*a* を **[**,**] real** 型の値とし，記述子の値が ((0, 100), (1, 12)) であったとすると，

***a*[32, 4]**

という添字づけによってひとつの要素値（実数型）が選ばれる．これとは別に，添字の範囲を指定することによる部分配列の切り出し (trimming) の操作も定義されている．たとえば上例の *a* に対して

a[75, 3: 8] *a*[20: 84, 4]

などの指定によって部分 1 次元配列を，また

a[40: 65, 2: 9]

といった指定によって部分 2 次元配列を，それぞれ切り出すことができる．範囲の指定を略すと全範囲とみなされる．つまり，*a*[*i*,] (*a*[, *j*]) という指定で *a* の横（縦）ベクトルを指定できる．

システムプログラムなどでは，機械語の 1 語の中に論理値（1 ビット）や文字値（1 バイト）を詰め込むことがよく行われる．ALGOL 68 ではこれを考慮して，**bits** 型と **bytes** 型とを用意している．これらはそれぞれ

struct (**[ ] bool** *α*), **struct** (**[ ] char** *α*)

と同等に扱かわれる．ここで *α* はプログラマが指定できない見出し名で，結果として，要素値の添字づけによる参照は不可能となっている．そのかわりとして，これらの型の値に対しては特別な演算（要素の取り出し，左右シフト，整数や文字列との間の交換など）が用意されているほか，**bits** 型に対しては次の例のような直接的表記法も定義されている．

2*r*1010110 （2 進値に対応する **bits** 値）
8*r*3774 （8 進値 〃 〃 ）
16*r*f80*a* （16 進値 〃 〃 ）

c） 参照…指定された型の値を指し示すもので，ふつうポインタと呼ばれているものと対応する．ALGOL 68 では，この**参照値** (name と呼ばれる) と，それが指し示す入物（箱，locale．この中に値が入る）とが重要な役目を果している．参照の型は，参照されるものの型の前に **reference to** (**ref** と略す) がついたものとなる．たとえば次のとおり．

ref int, **ref [ ] ref real**

これらの型の値を図示すると，**図-1** のようになる．矢印であらわされているポインタ自体が値であることに注意すること．

構造や配列における要素の選択に対応する操作として，参照はがし (dereferencing) と呼ばれる操作が定義されている．内容は，参照値が参照している箱の中味を取り出すことで，平たくいえば，ポインタを 1 回

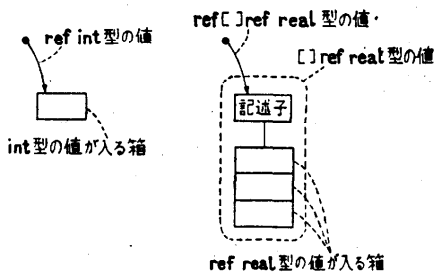


図-1 変数の概念図

たどることである．

参照値の定数としては，'何も参照していない' という値 **nil** がある．また参照値同志の等値は特別な比較構文，同一比較 (identity relation) によって調べられる．演算子は :=: (または **is**) と :≠: (または **isnt**) である．同一比較の構文では，比較すべきふたつの参照値の型がまず調べられ，片方の型が他方の型の前にいくつかの **ref** がついたものである場合には，余分な **ref** がはがされた後で参照値の比較が行われる．**図-2** にその例を示す．参照値 x と y の型は **ref ref int**，z の型は **ref int**．

i) x :=: y では両辺の値の型が一致しているのでそのまま比較され，同じ箱を指してはいないので結果は **false**.

ii) x :=: z では x の ref が1個はがされて x' となり，両辺の型が **ref int** と揃ったところで比較．結果は **true**.

d) 合併…複数個の型の合併をとったもの．この型には，要素となる型の値が全部含まれる．合併型は，要素となる型の（順序なし，つまりふつうの意味での）集合で表わされる．たとえば

union (int, real, [ ] char)

などと表わす．合併型の値を要素型の値として処理する目的のために，型選択構文 (conformity clause) が用意されている．上に示した型の値を valu とするとき

```
case valu in
  (int): {整数値としての処理}
  (real): {実数値としての処理}
  ([ ] char): {文字配列としての処理}
  esac
```

という形式で処理するのが型選択構文である．

4.3 参照値と変数

ALGOL 68 の特徴のひとつは，変数の定式化に参照の概念を使用したことである．ふつうのプログラム言語では，代入文の左辺にある変数名は参照値，すなわち変数の箱につけられた名前，右辺の式の中にある変数名は変数の箱の内容であると解釈する．この場所による変数名の意味の違いを，ALGOL 68 では参照値と自動型変換とによって記述したのである．

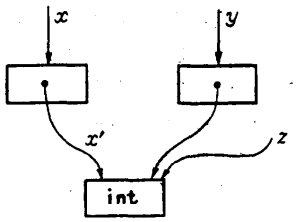


図-2 参照値の比較

MODE 型の変数(宣言の中を除く)を表わす hypernotion は次のとおりである（一部省略）．

MODE variable:
reference to MODE NEST applied identifier with TAG.

TAG は変数名を綴る文字列を表わす．これからもわかるとおり，変数名(**identifier with TAG**) は**MODE** 型の値を入れる箱への参照値 (**reference to MODE**) と結びつけられている．この参照値と箱との組が，ふつう '変数' と呼ばれているものと対応する．代入文の左辺に置かれるのはこの参照値であり，箱に入っている値ではない．一方，代入文の右辺（一般には式）の中で箱の中の値が必要とされる場合には，参照はがしの操作を自動的に行ってやればよい．この自動型変換 (coercion) の考えは，BLISS[30] や PASCAL などの '明示的' なポインタはがしとは対照的なものとなっている．

配列変数や構造変数の要素への代入についても，同様の配慮が必要である．たとえば代入

$a[8]:=100$

において，左辺に要求されているのもまた参照値である．そこで ALGOL 68 では，配列変数に添字づけあるいは切り出しを指定した形式 ($a[6]$ や $a[2:7]$) や構造変数の要素選択 (b **of** st など) によって得られるのは，中味の値そのものではなく，指定されたものへの参照値であるとしている(**図-3**)．こうすることによって，選ばれた要素や切り出された部分配列への値の代入が可能となる．

参照値の型は **reference to** で始まるが，指し示す箱の型が配列である場合，その上下限値が実行中に変

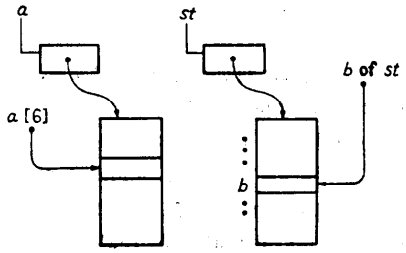


図-3 添字づけと選択の結果

化するのを許すことができる．こうするためには，参照値の型を reference to flexible row of…，略して

ref flex [ ]…

とする．これを使えば，'伸び縮みする配列変数' が定義できる．また，文字列を扱うための型 **string** が標準的に用意されているが，これは

flex [1 : 0] **char**

と等価である．また，この型に属する値の直接的な表現も用意されている．

"ABC" " " "String ot length 19"

4.4 宣　言

ALGOL 68 における宣言の種類は次の 4 種である．

モード宣言，名前宣言，演算宣言，優先度宣言

ここでは前 2 者について述べる．

名前宣言には変数宣言と同一宣言の 2 種があり，特に後者は，通常，定数宣言，手続き宣言，等価宣言などと呼ばれる宣言の機能をすべて含む宣言である．

変数宣言 (variable declaration) は，指定された型の変数を用意し，その参照値とプログラマが指定した綴りの名前とを結びつける役目をする．形式は，型に続けて名前のリストを置いたものであり，代入の形式で初期値を与えることもできる．

int count :=0, max, min ; [1 : 25] **real** *a* ;
ref [min : max×2]
　struct (**int** *si*, **bool** *sb*) table ;

型に含まれる式 (上例の min や max×2) や初期値は，その宣言が現われた時点で評価される．すなわち，動的配列などの宣言が可能である．

同一宣言 (identity declaration) は，型，名前，'=', 実体，をこの順に並べた形式を持ち，型の種類によってさまざまな宣言が可能である．まず，型の先頭に **ref** がなければ定数を宣言したことになる．

int *m*=100 ; **real** *pi*=3.1416 ;
struct ([1 : 3] **int** *set*, **bool** *f*) *entry*=
　((1, 5, 7), **true**) ;

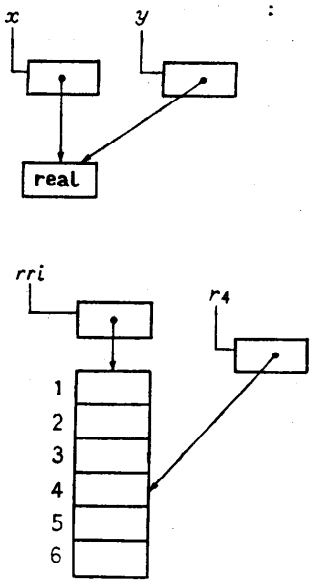


図-4 同一宣言の効果

real *area*=2×*pi*×(*set* **of** *entry*) [2] ;

型の先頭に **ref** がある場合，右辺に生成子があれば変数宣言と同じ効果を生じ，なければすでに作られているほかの変数との等価宣言となる．

ref real *x*=**loc real** ; (**real** *x* と等価)
ref [ ] **ref int** *rri*=**loc** [1 : 6] **ref int**
　([1 : 6] **ref int** *rri* と等価)
ref real *y*=*x* ; (*x* が指す箱を *y* も指す)
　ref int *r4*=*rri* [4] ;
　(*rri* [4] の箱を共有する変数 *r4* の宣言)

これらの宣言による効果を図-4 に示す．

後で述べる手続きの宣言も，この同一宣言の一種である

proc *yoneda*=(**int** *n*) **int** :
　if *n* **mod** 2=0
　　then *n* **over** 2 **else** 3×*n*+1 **fi** ;

この宣言の右辺は，整数型のパラメタ *n* を取り整数型の値を返す，手続き本体 (routine text) と呼ばれる手続き型の値として扱かわれる．

モード宣言は，任意の型に名前をつける宣言である．ただし型の名前は変数名のような綴り名 (identifier) ではなく指示名 (indicator) と呼ばれる別の種類のものとされ，**int** や **real** と同種の文字を使う．

mode table=**struct** (**string** *name*, **int** *count*),
　integer=**int**, **intreal**=**union** (**int**, **real**),

vector=[1 : *n*] **real**,

node=**struct** (**int** *value*, **ref node** *left*, *right*);

モード宣言の中にある式は，宣言の時点では評価されず，そのモード指示名が変数宣言などに使用された時点ではじめて評価される．上の例では **vector** がそのようなモードで，

vector *v*1;

と宣言された変数 *v*1 の長さは，宣言時における *n* の値で決められることになる．また，上例の **node** 型などのように，ポインタによる動的データ構造構築のための型を宣言することもできる．

モード宣言の場合，決定不可能な宣言，たとえば

mode a=**b**, **b**=**a**,

recursive=**struct** (**int** *i*, **recursive** *content*);

のようなものは，生成規則の適用によっては作られない，すなわち，(平文によることなく) 禁止されている．

4.5 手 続 き

ALGOL 68 では手続きもデータとして扱かわれる．しかし，容易に予想されるように，手続き値に対する操作は定義・呼出し(評価)・代入しか用意されていない．比較や合成もできない．すなわち，手続きは値とはいっても文面上のものしか扱かわれない．正確には，手続き本体とその場所における環境とを合わせたものが手続き値として扱かわれる．

手続き値の型は，(もしあれば) パラメタの型と結果の型から構成される．手続きは，'何も値がない' ことを示す型 **void** の値 **empty** も含めて，必ず何らかの値を返すものと定義される．したがって，ふつう関数と呼ばれているものも手続きに含まれる．手続き型の例をあげよう．

proc (**int**, **int**, **real**) **ref** [ ] **real**

この型の値は，整数2個と実数1個をパラメタとしてこの順にとり，実数配列への参照値を返す手続きである．パラメタや結果のない手続きの型は，それぞれ，たとえば

proc real, **proc** (**ref int**) **void**

などとなる．

手続きの本体は，もしあれば仮パラメタの型と名前を括弧で囲んだものに続けて結果の型と ':' を書き，その後に値を計算する部分を続ける．電気抵抗の並列合成値を計算する手続き本体の例を示す．

(**real** *a*, *b*) **real** : *a*×*b*/(*a*+*b*)

':' に続いている式の値が，この手続き本体を (仮パラメタ *a* と *b* の値を設定したうえで) 評価した結果となる．

手続きの宣言は，前節の同一宣言による．たとえば

proc (**real**, **real**) **real** *parplus*=

(**real** *a*, *b*) **real** : *a*×*b*/(*a*+*b*)

手続きの値の同一宣言に限り，左辺には **proc** と名前だけ書けばよいことになっているので，上の宣言は

proc *parplus*=(**real** *a*, *b*) **real** : *a*×*b*/(*a*+*b*)

となる．こうして宣言された名前 *parplus* の型は **ref** から始まっていないので，この宣言は '手続き定数' の宣言となる．この型の値を扱う変数の宣言は

proc (**real**, **real**) **real** *myplus*, *yourplus*;

とすればよい．次のような代入が可能である．

myplus:=*parplus*; (手続き値の代入)

yourplus:=(**real** *x*, *y*) **real** : *x*+*y*;

パラメタを持つ手続きは '呼び出される'．パラメタ機構は，同一宣言を使用した厳密な値渡しである．たとえば呼出し *parplus* (30.5, *r*4) においては，まず同一宣言

real *a*=30.5, *b*=*r*4;

が行われ，次に本体が評価される．参照渡しを行いたければ，パラメタの型を参照型 (**ref**…) にしておけばよい．たとえば，変数値を2倍する手続きを

proc *double*=(**ref real** *x*) **void** : *x*:=*x*×2

と宣言しておくと，実数変数 *p* に対する呼出し

double (*p*)

は次と等価となる．

(**ref real** *x*=*p*; *x*:=*x*×2)

前の例の *myplus* の型は **ref proc** (**real**, **real**) **real** であるが，呼出しの時点で自動的に参照はがしが行われるので

myplus (*r*13, 4.0)

といった書き方ができる．

パラメタを持たない手続きの呼出しは手続き名のみによる．ところが，前例の *myplus* への代入のように，手続き値そのものが要求されている文脈では呼出しが起きない．次の例で見てみよう．

proc *random*=**real** : …(標準的な乱数関数)

proc real *f*; **real** *x*;

f:=*random*;

x:=*random*;

明らかに，*f* への代入では手続き値そのものが代入されるが，*x* への代入では *random* が呼び出され，その結果が使われる．後者では，**proc real** 型の値から

real 型の値への自動変換，手続きはがし (deprocedur-ing) が施されたとみなされ，呼出し (call) とは区別されている．

4.6 演 算 子

多くのプログラム言語では，パラメタ数が1または2の関数のうち基本的なものを演算子の形式で提供している．四則演算や論理演算などがその例である．ALGOL 68 では，この手法をプログラマに開放した．演算子は，演算宣言 (operation declaration) によって，指示名 (indicator) か2文字以下の組記号として宣言され，優先度宣言 (priority declaration) によって，式の中における評価順位を決められる．前節の並列抵抗の例を演算子として宣言する例を示す．

```
op ×* = (real a, b) real : a×b/(a+b);
x :=3.5×*5.0×*13.0
```

演算子の優先順位は，標準的に以下のように決められている．数の大きいほど評価の優先度が高い(強い)．

2: **or**	6: +, −
3: **and**	7: ×, /, **over**, **mod**
4: =, ≠	8: ******, **lwb**, **upb**
5: <, >, <=, >=	

これらは2項演算子であり，単項演算子は一番強いものとされる．前出の演算子の強さを加算 (+，直列抵抗) より強い7にしたければ

```
prio ×* = 7
```

という優先度宣言をすればよく，次の式

```
3.0×*4.0+10.5×*6.7
```

は図-5の合成抵抗値をあらわすことにする．

演算子に関しては，パラメタの数と型とによって区別ができる限り，同じ演算子記号もしくは指示名に複数の意味を与えられるという，多義性 (overloading) が ALGOL 68 では許されている．四則演算の記号 (+−×/) はもともとこの性質を持つものとして扱かわれてきた．それを演算子に限ってプログラマに開放したわけである．多義性は，×* のようなユーザが定義した演算子のみならず，四則演算記号に代表されるごくふつうの演算子に対しても許される．

```
mode num=struct (bool active, real val);
op+=(num u, v) num:
        (active of u or active of v, val of u
        +val of v);
op*=(num u, v) num:
        (active of u and active of v,
        val of u×val of v);
```

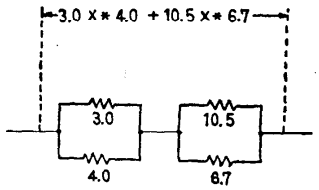


図-5 演算子の宣言

```
num p, q, r:=(true, 0.5), s:=(false, 1.5);
p:=r*s+s; q:=p+r;
```

(p の値は (**false**, 2.25), q の値は (**true**, 2.75) となる)

4.7 自動型変換 (coercion)

自動型変換とは，文脈によって要求されている型と実際に与えられた値の型とが異なる場合に，許される範囲内の自明な変換を自動的に行う機能である．これまでにも，変数の値を取り出す参照はがしや，パラメタなし手続きを呼び出す手続きはがしについて述べたが，ほかには次に示す4種類の変換がある．

値広げ (widening) 整数→実数→複素数
bits→[ ] **bool**, **bytes**→[ ] **char**

配列化 (rowing) **M** 型→[1 : 1] **M** 型
ref M 型→**ref** [1 : 1] **M** 型

合併化 (uniting) **M** 型→**union** (…, **M**, …) 型

値捨て (voiding) **M** 型→**void** 型

値捨ての変換は，得られた値がもう必要でない場合に起きる．また，配列化は'要素が1個の配列'に変換するだけであり，たとえば

```
[1:100] int ai; ai:=1
```

と書くと '1' が100個並んだ配列値を作ってくれるわけではない．

以上6種の変換を'明らかに必要'と思われる場合に自動的に行うために，自動型変換を施しうる文脈を以下のとおり5種類に類別してある．これらの類別や型変換の定義も，形式的な生成規則の中で記述されている．

strong…要求される型が文脈によって完全に決定できる場合．すべての変換を必要に応じて何回でも施すことが許される．例としては，同一宣言や代入の右辺，手続き呼出しの実パラメタ，変数の初期値，同一比較の片方など．

firm…式に使われる被演算数．参照はがしや手続きはがしを何回かやった後に1回だけ合併化が許されている (もちろん必要な場合にのみ)．

meek…参照はがしと手続きはがしが許される．配列の添字と呼び出される手続き名．

weak…手続きはがしと，**ref** を最低1個は残す参照はがし．添字づけされる配列名と選択を受ける構造名．

soft…手続きはがしのみが許される．代入の左辺と，同一比較の strong でない片方．

これらの文脈類別がこれだけで十分であるかは多少問題のあるところであるが，firm 以下の4類別を strong に対する'例外'と見なせば，比較的統一のよくとれた体系であるといえる．以下に，自動型変換の比較的に複雑な例を示す．*i*, *j*, *k* は外で宣言された整数値であるとしておく．

```
real x, y; [1: 1000] int a; int ijk;
```

(これらは変数であり，型はそれぞれ，**ref real**, **ref [ ] int**, **ref int** となる)

```
proc ind 1=(int n) int: abs n;
proc ind 2=(int n) int: n×n;
```

(*ind* 1 と *ind* 2 の型は **proc (int) int**)

```
proc select=ref real: if j>0 then x else
  y fi;
```

(*i* の値によって参照値 *x* と *y* のどちらかを返す手続き．型は **proc ref real**. 'fi' は条件文の閉じ括弧)

```
proc array=ref [ ] int: if j>0 then a [1:
  500] else a [501: 1000] fi;
```

(*j* の値によって配列 *a* の上半分あるいは下半分を参照値として返す手続き．いずれの場合も，返された配列の下限は1，上限は500となる)

```
proc index=proc (int) int:
        if k>0 then ind 1 else ind 2 fi;
ijk=i×j×k;
```

以上のような宣言のもとでの代入

```
select=array [index (ijk)]
```

は以下のように実行される．

左辺…代入の左辺の型は **ref** で始まることが要求されているが，*select* の型は **proc ref real** なので，soft 文脈で許されている手続きはがしが実行される．すなわち *select* が呼び出され，参照値 *x* または *y* (**ref real** 型) を得る．

配列…添字づけが許されるのは配列値または配列変数であり，*array* の型が **proc ref [ ] int** なので，weak 文脈で許されている手続きはがしが実行される．すなわち *array* が呼び出され，参照値 (**ref [ ] int** 型) を得る．

添字中の手続き…*index* の型は **proc proc (int) int** であり，パラメタつきで呼び出されることはない．そこで weak 文脈で許されている手続きはがしが実行され，パラメタつきで呼ばれる手続き (*ind* 1 か *ind* 2，型は **proc (int) int**) を得る．

パラメタ…*ind* 1 あるいは *ind* 2 の実パラメタは **int** 型でなければならない．そこで *ijk* (**ref int** 型) に対し strong 文脈で許されている参照はがしが行われ，**int** 型の値を得る．

添字づけ…代入の右辺の値は，**ref [ ] int** 型の値の indexing であり，結果は **ref int** 型となる．

代入…左辺の型が **ref real** であるので，右辺の値はまず参照はがしを受けて **int** 型となり，つぎに値広げを受けて **real** 型とされる．その値が代入されるわけである．

参 考 文 献

ALGOL 68 に関しては，文法，言語構造，各種の概念，プログラミング，応用，などの広い分野に膨大な数の文献が存在する．ここでは，文献リストのいたずらな長大化を防ぐために，tutorial 的なものと，本解説に直接関係を持ち入手しやすいもののみを示した．

1) Ammereal, L.: An Implementation of an ALGOL 68 Sublanguage, in Proc. of International Computing Symposium (E. Gelenbe and D. Potier ed.), North-Holland (1975).
2) Bährs, A. A., Ershov, A. P. and Rar, A. F.: On description of syntax of ALGOL 68 and its national variants, in ALGOL 68 Implementation (J. E. L. Peck ed.), North-Holland (1971).
3) Birrell, A. D.: Storage Management for ALGOL 68, Sigplan Notices, Vol. 12, No. 6, pp. 82–94 (1977).
4) Bourne, S. R., Birrell, A. D. and Walker, I.: ALGOL 68 C Reference Manual, University of Cambridge Computing Service (1974).
5) Braid, I. C. and Hillyard, R. C.: Geometric Modelling in ALGOL 68, Sigplan Notices, Vol. 12, No. 6, pp. 168–174 (1977).
6) Cleaveland, J. C. and Uzgalis, R. C.: Grammars for Programming Languages, Elsevier, North-Holland (1977).
7) Currie, I. F., Bond, S. G. and Morison, J. D.: ALGOL 68-R, in ALGOL 68 Implementation (J. E. L. Peck ed.), North-Holland (1971).
8) Gardner, P. J.: A transportation of ALGOL 68 C, Sigplan Notices, Vol. 12, No. 6, pp. 95–101 (1977).
9) Hamlet, R.: Ignorance of ALGOL 68 Con-

sidered Harmful, Sigplan Notices, Vol. 12, No. 4, pp. 51-56 (1977).
10) Hibbard, P. G.: A sublanguage of ALGOL 68, Sigplan Notices, Vol. 12, No. 5, pp. 71-79 (1977).
11) Hunter, R. B., Kingslake, R. and McGettrick, A. D.: Some ALGOL 68 Compilers, Algol Bulletin No. 41, pp. 71-73 (1977).
12) Kawai, S.: Lattice Structure Segmentation of ALGOL-like Programs, Software-Practice and Experience, Vol. 9, pp. 485-498 (1979).
13) Kawai, S. and Ishihata, K.: A Transportation of Multiphase Compiler, J. of Information Processing, Vol. 2, pp. 143-145 (1979).
14) Lindsey, C. H.: ALGOL 68 with Fewer Tears, Computer J., Vol. 15, pp. 176-188 (1972).
15) Lindsey, C. H.: Some ALGOL 68 sublanguages, in ALGOL 68 Implementation (J. E. L. Peck ed.), North-Holland (1971).
16) Lindsey, C. H. and Boom, H. J.: A Modules and Separate Compilation facility for ALGOL 68, Algol Bulletin No. 43, pp. 19-53 (1978).
17) Lindsey, C. H. and van der Meulen, S. G.: Informal Introduction to ALGOL 68, North-Holland (1971), and revised edition (1977).
18) McGettrick, A. D.: Algol 68 a first and second cource, Cambridge University Press (1978).
19) Meersman, R. and Rozenberg, G.: Two-level meta-Controlled Substitution Grammars, Acta Informatica, Vol. 10, pp. 323-339 (1978).
20) Needham, R. M.: The CAP project - an interim evaluation, Proc. of 6th ACM Symposium on Operating Systems Principles, pp. 17-22 (1977).
21) 沖野教郎, 久保 洋: 形状モデリングと CAD/CAM, 情報処理, Vol. 21, No. 7 pp. 725-733 (1980).
22) Pagan, F. G.: A Practical Guide to Algol 68, John Wiley (1976).
23) Peck, J. E. L.: Two-level grammars in action, in Proc. of IFIP Congress 74, North-Holland (1974).
24) Sintzoff, M.: Existence of a Van Wijngaarden syntax for every recursive enumerable set, Annales de la Societe Scientifique de Bruxelles, Vol. 81, pp. 115-118 (1967).
25) Tanenbaum, A. S.: A comparison of PASCAL and ALGOL 68, Computer J. Vol. 21, pp, 316-323 (1978).
26) Tanenbaum, A. S.: A Tutorial on ALGOL 68, Computing Surveys, Vol. 8, pp. 155-190 (1976).
27) Valentine, S. H.: Comparative Notes on ALGOL 68 and PL/I, Computer J. Vol. 17, pp. 325-331 (1974).
28) van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L. and Koster, C. H. A.: Report on the algorithmic language ALGOL 68, Numerische Mathematik, Vol. 14, pp. 79-218 (1969).
29) van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T. and Fisker, R. G.: Revised report on the algorithmic language Algol 68, Acta Informatica, Vol. 5 pp. 1-236 (1975).
30) Wulf, W., Russell, D. and Habermann, A.: BLISS: A Language for Systems Programming, CACM, Vol. 14, pp. 780-790 (1971).
31) 米田信夫: 新算法言語 ALGOL 68, 数理科学, Vol. 7, No. 6 (1969)—Vol. 8, No. 7 (1970) ダイヤモンド社.

(昭和55年8月4日受付)