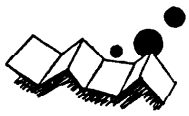
解　説



ALGOL 68 とその処理系（2）†

川　合　　慧††

5. ALGOL 68 の概要

5.1 式言語とユニット

一般的にいって，プログラムにおいては値の計算が中心的な作業となる．値の計算法を指定する構文は，ふつう，式と呼ばれる．このほかに，まとまった計算の順序などを制御する構文として，文と呼ばれるものが設定されている言語も多い．代入文，分岐文，条件文などといわれるものがその例である．ALGOL 68 では，これらの‘文’も何らかの値を結果とする式として扱い，得られた結果をさらに別の式の要素として使用できる方式をとっている．この方式をとる言語を一般に式言語[30]という．

文と式を総称したものを，ALGOL 68 ではユニット (unit) と呼ぶ．以下に，各種のユニットを示す．

a）名前とリテラル

値の直接表示(リテラル——1980, **true**, "A" など)はそれだけでユニットとなる．また，変数名，定数名，手続き名なども，宣言によってその名前と結びつけられている値を返すユニットである．たとえば，宣言

real *x*, [1 : 100] **int** *m*, **proc** *p*=**real**: *x*×2

のもとでは，名前 *x*, *m*, *p* はそれぞれ型 **ref real**, **ref** [ ] **int**, および **proc real** 型の値を返すユニットである．

b）配列の添字づけと切出し (slice)

添字づけによって配列のひとつの要素が，また切出しによって部分配列が，それぞれ指定される．部分配列の下限は，特に指定しない限り 1 となる．

[ ] **int** *chan*=(1, 3, 4, 6, 8, 10, 12);

chan [3 : 6] ... 値 (4, 6, 8, 10), 下限 1, 上限 4.

chan [2 : 3 **at** 7] ... 値 (3, 4), 下限 7, 上限 8.

この例は配列定数からの切出しなので，結果の型も定数 ([ ] **int**) となる．配列変数に対する添字づけと切出しの結果は，要素または部分配列を示す変数となる．

[13 : 65] **int** *uhf*;

uhf [42] ... 結果は **ref int** 型の参照値.

uhf [20 : 45] ... **ref** [ ] **int** 型の参照値.

c）手続き呼出し (call)

ALGOL 68 における手続きは，‘何もない’という値 **empty** (**void** 型) を含めて，何らかの値を返すことになっている．結果の値は，手続きの本体を評価して得られる値である．

proc *sumsq*=(**real** *a*, *b*) **real**: *a*×*a*+*b*×*b*;

sumsq (3. 14×6, 8. 49)

パラメタなしの手続きは，名前を指定するだけでは呼び出されない．必要がある場合には，手続きはがしの自動型変換によって評価される．

d）構造要素の選択 (selection)

構造値あるいは構造変数の値の要素を指定する．結果は要素の値，あるいは要素への参照値

mode person=**struct** (**string** *name*, **int** *age*);

person *slave*; ...**person** 型の変数.

person *hero*= ("Taro", 23); ...**person** 型の定数

name **of** *slave*...*slave* の *name* 領域への参照値. *age* **of** *hero*...*hero* の *age* 領域の値，すなわち 23.

e）生成子 (generator)

局所生成子 (**loc**) と大域生成子 (**heap**) とは，指定された型の値を収容する箱を確保し，それへの参照値を結果とするユニットである．

loc real 結果は **ref real** 型の値.

heap [ ] **int** 結果は **ref** [ ] **int** 型の値.

f）式 (formula)

単項および 2 項演算子で形成されるふつうの意味での式．先に単項演算子が作用し，後は優先度の高いものから順に演算を行う．優先順位は 1 から 9 まであり，数が大きいほど強い．代表的な演算子の優先度は

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

†† 東京大学理学部

or (2), **and** (3), = (4), < (5), +− (6), ×/ (7) であり，式，

$a<10$ **or** $b+c\times d<e-f=g+h$ **and** $i>j$

は次のように評価される．

($a<10$) **or** ((((($b+(c\times d)$)<($e-f$))=($g+h$)) **and** ($i>j$))

演算子としてはプログラマ定義のものも使用できる．また，優先度は，標準的な演算子も含めて，優先度宣言によって指定・変更することができる．たとえば，$b1, b2, b3, b4$ を論理値とするとき，宣言

prio or=4, ==2;

のもとでは次のような解釈が行われる．

$b1$ **or** $b2=b3$ **and** $b4$

は ($b1$ **or** $b2$)=($b3$ **and** $b4$) となる．

g) 手続き本体 (routine text)

評価される部分（ユニット）の前に（もしあればパラメタ指定と）結果として返す値の型を置いたものが，手続き本体であり，これもユニットとして扱われる．

(int k**)int**: (k+3)**2

型は **proc** で始まるもので，上例では **proc(int)int** となる．

h) 同一比較 (identity relation)

2個の参照値が同一であれば **true**，異なれば **false** を返す構文．自動的参照はがしの存在により，ふつうの演算子とは別格になっている．同一比較子 =:(**is**)と :≠:(**isnt**) とが使われる．比較は単純に行われるのではなく，適当な自動型変換によって両辺の型が同じ参照型になるときは，型変換が起きる．その場合，どちらか片側に対しては手続きはがししか許されず (soft)，もう片方にはすべての変換が許される (strong)．どちらがどうなるかは，双方の型を調べて決められる．このような機能を一般に釣合わせ (balancing) と呼ぶ．ここでは簡単な例を示すにとどめる．

real x; **ref real** $x1$; $x1:=x$;

$x=:x1$ 結果は **true**．右辺の **ref** がひとつはがされる．

i) 代入 (assignation)

代入は，代入記号の右辺の値が左辺の変数の箱に書き込まれるという副作用を持つユニットである．結果は左辺の参照値であり，代入された値ではない．左辺には手続きはがしの，右辺にはすべての種類の自動型変換が施されうる．たとえば

compl z;

proc int p=**int**: **round** (100×*random*)

という宣言を考える．(p の型は **ref proc int** である．) この時 $z:=p$ という代入によって

参照はがし (**ref proc int** → **proc int**)

手続きはがし (**proc int** → **int**)

値広げ2回 (**int** → **real** → **compl**)

が自動的にこの順に施される．

j) そ の 他

jump (goto 文)，skip (どんな型の値にもなり得るもの)，nihil (**nil**)，format (入出力用書式)，それに次節で述べる構造化構文 (clause) などがユニットとなっている．

5.2 構造化構文

前節で述べたユニットと，各種の宣言とが組み合わされて，構文が構成されてゆくわけであるが，その基本となるのが，順次節 (serial clause) と呼ばれる構文である．その形式は次のとおり．

ユニットと宣言とを ';' または **exit** で区切って並べたもので，最後がユニットで終る．

順次節は，その中に含まれている宣言を有効としたうえで，要素となるユニット（および宣言）を順番に評価するために使われる．宣言の有効範囲はそれを含む順次節の中に限られる．順次節の評価は，末尾のユニットあるいは完了符号 **exit** の直前のユニットの評価の終了，およびその順次節から飛び出す jump によって終了する．最後に評価されたユニットの値とその型が，順次節全体の値とその型となる．

宣言は，末尾でなければどこにあってもよいし，ユニットと混在していてもよい．これにより，各種の'動的な'宣言が可能となる．順次節自体はユニットとはならない．順次節の例を示す．

int i, j; read ((i, j)); [i : j] **int** a;

int k; **mode list**=[1 : k] **real**;

$k:=j-i+1$; **list** $l1$; $k:=k\times 2$; **list** $l2$;

…sequence of units…

この順次節を主体として構成される構造化構文について次に述べる．

a) 包囲構文 (closed clause)

順次節を **begin** と **end**，または '(' と ')' で囲んだもの．ユニットとして扱われる．要素である順次節の値と型が，包囲構文の値と型となる．内部にある宣言はその順次節にだけ局所化される．

begin int i, j; read((i, j));

int $k=i\times j$; print(k); k **end**

(全体の値は *k* の値，型は **int**)

'(' と ')' は **begin** と **end** と全く同格に扱われる．算術式で使われる括弧も，実はこの意味を持っている．すなわち，部分式を括弧で囲むことはその部分をユニットとして扱うことを意味する．したがって，次のような'一般的な'式も許される．

i:=20−(**int** *j*; read(*j*); *j*+100)×150

b) 選択構文 (choice clause)

選択肢を持つ構文は，選択の方法によって3種類が用意されている．

b-1) 論理値選択構文 (IF 文)．型式は次のとおり．

if E **then** S{**elif** E **then** S}[**else** S]**fi**

Eは論理値を結果とする順次節，Sは一般の順次節．Eが順に評価され，初めて値 **true** となったEに続くSのみが評価される．すべてのEが **false** となった場合，もし **else** 部があればそのSが評価され，なければ（それまでに評価されたいくつかのEを除いて）何も評価されない．個々のSと，それぞれのEから構文の終りまでが宣言の局所化の範囲となる．記号 **if, then, else, elif, fi** は，一斉に記号(,|,|,|:,) に置きかえることができる．下の例を参照のこと．

```
if real error=abs (f(x)−g(x));
  error<1e−10
    then print (x)
elif real rel=abs (error/f(x));
  rel<1e−6
    then print ((x, error))
    else print ((x, error, rel))
fi
```

b-2) 整数値選択構文 (CASE 文)．形式は次のとおり．

case E **in** U,U{,U}
{**ouse** E **in** U,U{,U}}[**out** S]**esac**

Eは整数値を結果とする順次節，Sは一般の順次節，Uはユニット．先頭のEが評価された結果を*n*とすると，それに続く **in** と **ouse** または **out** に囲まれたユニットのうち*n*番目のものが評価される．*n* が非正値またはユニットの個数よりも大きい場合は，**ouse** 部があれば今と同じことがくり返され，**ouse** 部がなくて **out** 部があればSが評価され，両方なければ(それまでに評価されたいくつかのEを除いて) 何も評価されない．**out** 部の S と，それぞれの E から構文の終りまでが宣言の局所化の範囲となる．記号 **case, in, out, ouse, esac** は，一斉に記号 (,|,|,|:,) に置きかえることができる．

```
int d; read(d);
print ((d,(d|"st","nd","rd"|"th"),
      " day of the week is ",
      case d in
        "SUNDAY","MONDAY",
        "TUESDAY","WEDNESDAY",
        "THURSDAY","FRIDAY",
        "SATURDAY"
      out "undefined"
      esac))
```

b-3) 型選択構文 (conformity clause)．形式は次のとおり．

case E **in** W,W{,W}
{**ouse** E **in** W,W{,W}}[**out** S]**esac**

Eは合併型の値を結果とする順次節．Sは一般の順次節．各 W は次の形式をとる．

〈型〉〈名前〉:〈ユニット〉

Eの値の型（合併型の要素型のひとつ）によって，Wのどれかが選ばれ，評価される．ほかは整数値選択構文と同じ．各 W の中では，〈名前〉は E の値（型は〈型〉）を表わすものとして使用できる．〈名前〉はなくてもよい．

```
union (int, real) uvar;...
case uvar in
  (int i): print (("integer", i)),
  (real): print ("real output not prepared")
esac
```

b-4) 釣合せ (balancing)．選択構文はユニットの一種であるので，次の例のように式の中でも使用できる．

```
real a, x, y; int i, j, k;
a:=x×(i<0|y|j)
```

選択構文 (*i*<0|*y*|*j*) において，*y* が実数型だということから演算子 '×' の型が **proc (real, real) real** であると決まる．この場合，整数 *j* の値は値広げの自動型変換を施され実数に変えられる．このように，選択構文全体の値の型が文脈から決定できる場合（この例では実数型），すべての選択肢の型をそろえるような型変換が自動的に施される．これを釣合せという．釣合せは以下の文脈で起りうる．

(i) 順次節の末尾のユニットと **exit** の前のユニット．

```
real x, y, z;
```

```
z:=x×(if y<0 then goto tail fi; read(y);
        if y<0 then goto tail fi; x×y exit
    tail: 100)
```

exit の前の $x \times y$ と，末尾の 100 との間で釣合せが起きる．

(ii) 選択構文の選択肢．CASE 文での例をあげる．

```
a:=x×case i in x, y, j out k esac
```

(iii) 同一比較 (identity relation) の両辺．これについては前に述べた．

c) 反復構文 (loop clause)

ALGOL 68 の反復構文は一種類であるが，種々の省略が用意されている．完全な形式は次のとおり．

```
for 〈名前〉 from Uf by Ub to Ut
    while E do S od
```

U_f, U_b, U_t は整数型のユニット，E は論理値を結果とする順次節，S は一般の順次節．〈名前〉は E と S の中で整数値定数として参照できる綴り名．反復の意味はほぼ下のようにあらわされる．

```
begin int f=Uf, int b=Ub, t=Ut;
  loop: if (b>0 and f≤t) or
           (b<0 and f≥t) or b=0
        then int 〈名前〉=f;
             if E then S; f:=f+b;
                  goto loop fi
        fi
end
```

'from U_f', 'by U_b', 'while E' は省略するとそれぞれ 'from 1', 'by 1', 'while true' であるとみなされる．'to U_t' を略すと最終値のテストが行われない．'for 〈名前〉' を略すと参照できる名前がなくなる．**do** と **od** に囲まれた順次節，および **while** から **od** までの間が宣言の局所化の範囲となる．次の例を参照のこと．

```
for i from −100 by 2 to 100 while a[i]>0
  do int p=a[a[i]];
       a[i]:=a[p]+a[p+1] od;
for i to 1000 do print(i) od;
to 66 do print (newline) od
```

5.3 標準前置部 (Standard Prelude)

ALGOL 68 の言語としての中核部は以上述べたものでほぼ尽くされており，それほど大きなものではない．これに含まれず，かつ一般的なプログラム言語が持っている機能（たとえば入出力や標準手続き）の多くは，標準前置部と呼ばれる宣言の集合の中で記述されている．そして，ユーザの書くプログラムは，これらの宣言の有効範囲の中で評価されるものとされる．宣言のほとんどは ALGOL 68 自身で記述されており，種々の機能の厳密な定義を与えていると同時に，プログラミングの例題の役割をも果している．

標準前置部の中の宣言の例を示す．

```
int intlength=〈許される long の最大数+1〉;
int maxint=〈最大整数値〉;
real smallreal=〈実数値の精度〉;
int bitswidth=〈bits 値に含まれるビット数〉;
int byteswidth=〈bytes 値に含まれる文字数〉;
op abs=(char a) int: 〈文字 a の対応整数値〉;
op repr=(int a) char: 〈abs c=a となる文字 c〉;
mode compl=struct (real re, im);
prio or=2, and=3, ==4, ≠=4, <=5, ≤=5,
  >=5, ≥=5, +=6, −=6, ×=7, /=7, over
  =7, mod=7, **=8, lwb=8, upb=8, i=9;
op or=(bool a, b) bool: (a|true|b);
op<=(int a, b)bool: 〈a が b より小さい〉;
op≤=(int a, b) bool: not (b<a);
op==(int a, b) bool: a≤b and b≤a;
op−=(int a, b) int: 〈a から b を引いた値〉;
op−=(int a) int: 0−a;
  op+=(int a, b) int: a−−b;
  op entier=(real a) int:
    begin int j:=0;
          while j<a do j:=j+1 od;
          while a<j do j:=j−1 od; j
    end;
op+:==(ref int a, int b) ref int: a:=a+b;
op−:==(ref int a, int b) ref int: a:=a−b;
real pi=〈円周率の近似値〉;
proc nextrandom=(ref int a) real:
begin a:=〈[0, maxint] 内の一様疑似乱数列の a
        の '次の' 値〉;
    〈疑似乱数の性質を保存する [0, maxint] から [0, 1) への写像を a に施した値〉
end
```

これらの例でもわかるように，標準前置部に含まれる宣言には，言語の外で定義しなければならないものと ALGOL 68 で書かれたものとがある．後者の部類に入る宣言（たとえば=, +, **entier** の宣言）をどのよ

うに実現するかは処理系の作成者にまかされており，機械語でコードしてもよいが，標準前置部で示されたものと同じ意味を持っていなければならない．

6. 入 出 力

6.1 入出力の基本概念

ALGOL 60 の欠点のひとつは入出力の方式を全く規定していないことである．いくら‘アルゴリズム記述言語’であるとはいえ，入出力を全然取り扱わなかったのは問題であった．ALGOL 68 ではこの点が考慮され，プログラムが外界と通信する手段である入出力の記述にも力が入れられた．入出力は transput と呼ばれる．

プログラムは**チャネル** (channel) と呼ばれる通信路によって入出力を行う．外界においてチャネルにつながっている入出力の対象は本 (book) と呼ばれ，文字あるいはバイナリ情報を要素とする3次元配列とみなされる．チャネルのプログラム側にはファイル (file) という型の変数を用意する．ファイル型は，本の中で行っている入出力の現在位置，コード変換表，本へのポインタなどから成る構造型である．これらの要素へのアクセスは，専用に準備された手続き群を使用して行う．以上の関係を**図-6** に示す．

入出力を開始するには，まず，ファイルと本とをチャネルを介して結合する標準手続きを呼ぶ．

file *myin*;
open (*myin*, "NEWDATA", *standinchannel*)

"NEWDATA" は本の名前で，ふつうは OS におけるデータセット名が使われる．*standinchannel* は，ファイル *myin* を順次入力用に使うことを示すチャネルである．ほかには，順次出力用の *standoutchannel*，ランダム参照可能な *standbackchannel* が標準的に定義されている．

入出力が可能なのは，参照値，手続き値，合併値の3個の型以外の基本型，およびそれらのみを要素とする構造値と配列値である．実際の入出力は，標準手続き *put* と *get* とによって行う．

int *i*, [1 : 10] **real** *x*,
string *message*:="GOOD";
get (*myin*, *i*); *put* (*myout*, (*x*[2 : 5], *message*))

put の例で示したとおり，複数個の入出力の指定は，それらを配列値の形式にして行う．入出力のリストには，次のように書式指定を含めることもできる．

put(*myout*, (*newpage*, *x*, *space*, *i*, *newline*))

newpage は新しいページへ，*newline* は新しい行へ，*space* は次の文字位置へ，それぞれ進むことを示す．ほかに *space* の逆指定，*backspace* がある．これらの書式指定は **proc (ref file) void** 型の手続きであり，*put* や *get* から呼び出されて働く．たとえば

put (*myout*, (*x*, *newline*))

は

put (*myout*, *x*); *newline* (*myout*)

と等価である．上に示した型であれば，ユーザが作成した手続きであっても書式指定として書くことができる．次の例を参照のこと．

proc *next*=(**ref file** *f*) **void**:
put (*f*, (*newline*, *line number*(*f*))
put (*myout*, ("HEAD", *next*, *x*, *y*, *next*, *z*, *newpage*))

line number は現在行の番号を知るための標準手続きである．

標準前置部には，ファイルに関する次のような部分が含まれている．

file *standin*, *standout standback*;
open (*standin*" ", *standinchannel*);
open (*standout*," ",*standoutchannel*);
open (*standback*," ",*standbackchannel*)

これらは，標準的に用意された3個のファイルを，使用システムにおける標準的な本（" " で示される）と結合しておくためである．加えて，ファイルとして *standin* と *standout* とを仮定した入出力手続き，*read* と *print* とが用意されているので，標準入出力のみを行うプログラムでは，ファイルの宣言や *open* はまったく必要がない．

print (("Type-in number", newline));
print (((**real** *x*; *read* (*x*); *sqrt* (*x*)),
"is the root"))

本はページと行という構造で配列化されているが，

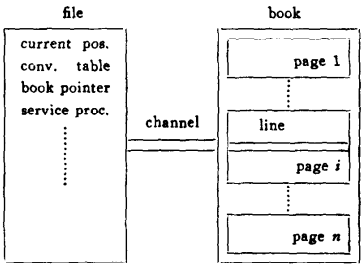


図-6 ファイルと入出力

本やページ，行の終りで入出力要求があると，ファイルに含まれている手続きが自動的に呼ばれる仕組みとなっている．これらの手続きは論理値を結果とするが，それが **false** である場合にはシステムで定義された標準的な処理が行われる．**true** の場合は，その手続きが有効な処理をしたものとみなし，システム側では何もやらない．行末処理を例にとると，ファイルを開いた時点では，行末処理手続きとしては標準的な値

(**ref file** *f*) **bool**: **false**

が登録されており，システムの標準処理 (*newline* を呼ぶこと) が常に実行される状態になっている．これを変更するには，標準手続き *on line end* を用いる．

proc *nextline*=(**ref file** *f*) **bool**:
(*newline* (*f*); *put* (*f*, *line number* (*f*));
true);
on line end (*standout*, *nextline*)

これ以降は，行末の標準的な処理内容は *nextline* のそれに変更されたことになる．同様なほかの処理手続きとして，*on logical file end* (入力本の終り)，*on physical file end* (出力本の終り)，*on page end*, *on value error* (入力した値が不適切な場合の処理) がある．

6.2 書式なし入出力 (formatless transput)

前節で述べたものは，書式なし入出力と呼ばれる．整数，実数，複素数は標準的な形式で入出力される．標準出力幅は標準前置部中の名前，*intwidth*, *realwidth*, *expwidth* の値として与えられる．入力の場合，必要な文字列が得られるまで空白と行末，ページ末は読みとばされる (標準的な場合)．

文字はそのままの形で入出力される．論理値を出力すると，標準前置部で宣言されている文字型定数 *flip* と *flop* の値 (通常は "T" と "F") が，**true** と **false** に応じて出力される．入力の場合はこの逆となる．

文字列の出力は文字配列の出力と同じであるが，入力の場合は，現在位置から「文字列の区切り」の前までがまとめて読み込まれる．行末は区切りとなるが，そのほかに区切り文字の集合を文字列の形式で指定しておくことができる．たとえば，標準入力から，空白，コンマ，斜線のいずれかで区切られた名前を文字列として読むには

maketerm (*standin*, ", /")

とすればよい．
そのほか，**bits** 型や **bytes** 型の入出力も用意されている．

ファイルに関する標準手続きとしては，これまでに述べたもののほかに，ページ (行，文字) 位置を知るための *page* (*line*, *char*) *number*，ランダム読み書き可能な場合に現在位置を任意に設定する *set* および本の先頭に戻す *reset*，現在行の中の文字位置を変更する *set char number* などがある．

書式なし入出力は簡単ではあるが，数値の変換形式が固定されているので，特に作表などの場合に困ることも多い．そこで，出力に関した変換をプログラマが容易に行えるように，次の手続きが用意されている．これらは，変換結果を文字列として返す．

whole (*v*, *w*)　*v* は整数か実数．結果は整数を表わす長さ |*w*| の文字列．ゼロサプレスが行われる．負号は常につけられるが，正号は *w*<0 の場合空白となる．*w*=0 の場合は符号を含めて必要最小限の長さとなる．

whole (1023, 7)="␣␣+1023"
whole (1023, −7)="␣␣␣1023"
whole (1023, 0)="1023"

fixed (*v*, *w*, *a*)　*v* は整数か実数．結果は固定小数点表示の長さ |*w*| の文字列．小数点以下の桁数が *a*．*w* の正，0，負は *whole* と同様の意味．

fixed (53.8951, 8, 3)="␣+53.895"
fixed (53.8951, −8, 2)="␣␣␣53.90"

float (*v*, *w*, *a*, *e*)　*v* は整数か実数．結果は浮動小数点表示の長さ |*w*| の文字列．*a* は小数点以下，*e* は符号を含めた指数部の桁数をそれぞれ示す．

float (0.02718, 11, 3, 2)="␣␣+2.718$_{10}$−2"

6.3 書式つき入出力 (formatted transput)

入出力する文字列を，実際に読み書きする値とは別に用意した書式によって制御するという方式は，COBOL, FORTRAN, PL/I などで採用されているが，ALGOL 68 の書式も同種のものである．その概要を示す．

書式 (format) は型として扱われ，書式値，書式変数，書式の代入などが使用できる．書式値の直接表示の例を示す．

$3*a*, 3*z*−*d*", "3*d*. 2*de*+2*d*"!?", 2(4*z*+*d*" : ")$

書式つき入出力は，前節の *put*・*get*・*print*・*read* に対応した標準手続き群 *putf*・*getf*・*printf*・*readf* によって行う．たとえば

printf((〈上例の書式〉, "///",1.234567, 198, −20)

によって標準出力には次の文字列が出力される．

"///␣␣␣␣1, 234.57$_{10}$−03!?␣␣

+198: ␣␣␣−20: "

書式は，コンマで区切られたいくつかのピクチャ (picture) から成り，各ピクチャは挿入 (insertion) と枠 (frame) から成る．挿入できるものには，例にも示されている文字列 (", ", "!?", ": ") のほかに，次の6種がある．これらには整数値の反復回数を指定することができる．

x 手続き *space* を呼ぶ　*l* *newline* を呼ぶ
y *backspace* を呼ぶ　*p* *newpage* を呼ぶ
q 空白を書く（出力）か読み飛ばす（入力）
k *set char number* を呼ぶ．すなわち，文字の位置が直接指定される．

枠は，変換された文字列を受け入れる指定である．*d* は数字1桁，. は小数点，*e* は指数部の開始を示す．また，+は "+" か "−" を，− は空白か "−" を示し，その直前に置かれた *Rz* はゼロサプレス *R* 桁を表わす．そのほかには，文字を表わす *a*，論理値を表わす *b*，**bits** 値を表わす *mr* (m=2, 4, 8 or 16) などが用意されている．特殊な機能としては，**format** 型の値を結果とする構文を書式中で使えること，反復数の実行時設定ができること，および下例に示すような，選択の指定が可能であること，などがある．

```
i:=2;
printf(($c("first", "second", "third")$, i))
```

枠 *c* の働きにより，"second" が印刷される．また，

```
readf(($c("first", "second", "third")$, i))
```

によって，入力文字列に対応する整数値 (1, 2 または 3) が *i* に代入される．

7. ALGOL 68 によるプログラミング

プログラム言語は，いかにその意味が明確に定義され，またいかに明解な概念構成を持っていたとしても，実用的に使用されない限り死語と化してしまうことになる．この章では，欧米，特にヨーロッパでは盛んに使用されているにもかかわらず[18),22)]，日本においてはその存在さえも知らない人が多いという ALGOL 68 について[9)]，その特徴的な諸点に関係するプログラミングの様子を，筆者の経験を踏まえて解説する．

7.1 式言語と制御構造

式言語の特徴は，結果を返す構文をほかの構文の一部として直接書けることである．多くの場合，こう書くことによってプログラムテキストが短くなり，実行効率も上昇する．ALGOL 68 では選択構文も値を返す単位（ユニット）であり，テキスト短縮とコードの効率化に役立っている．次の例をみてみよう．

```
(i<j|x|y)=(l<m|a|b)[k:=k+1]
```

語記号を使って書くと

```
if i<j then x else y fi:=
    if l<m then a else b fi [k:=k+1]
```

もし選択構文が値を返せないとすると，次のように書かざるを得ない（副作用はないものとする）．

```
k:=k+1;
if i<j then if l<m then x:=a[k]
                   else x:=b[k] fi
       else if l<m then y:=a[k]
                   else y:=b[k] fi
fi
```

このどちらが'読みやすい'かには議論の余地があるが，少なくとも文面の長さには大きな差が存在する．参照値が扱える（上の第1例の左辺など）のも文面短縮に役立っている．

整数値選択構文 (CASE 文) は，当然のことながら，順序数の扱いに適しており，そうでない場合には論理値選択構文を使うことになる．次の2例がそれを示している．

```
int i; read (i);
print ((i, (i|"st", "nd", "rd"|"th"),
    "planet of the solar system is",
    (i|"Mercury", "Venus", "the Earth",
      "Mars", "Jupiter", "Saturn", "Uranus",
      "Neptune",
      "Pluto"|(i>0|"not discovered"
                |"not defined")),
    ". ", newline))
```

出力例: 2nd planet of the solar system is Venus.
12th planet of the solar system is not discovered.

```
if int m=cryptogram mod 1980;
   m=0 then no problem
elif m=1 then easy
elif m=10 then private code
elif m=100 then consult expert
           else undefined code
fi
```

ALGOL 68 の反復構文は一種類であり，種々の省略が許されている．制御変数そのものの省略は，定数回の反復を表わすのによく利用される．

```
to maxline-currentline do print (newline) od
```

制御変数の初期値の標準値は1である．これは，配列の添字の下限としばしば対応する値であり，下限を0とするプログラミングにおいては注意を要する点である．

[1 : 100] **int** *a*; **for** *i* **to** 100 **do** *a*[*i*]:=0 **od**;

この例では変化値の標準値が1であることも利用されている．最終値の省略は条件部(**while**)と組み合わされることが多い．

int *k*:=0; **for** *i* **while** *k*<1000 **do** *k*:=*k*+*i* **od**

条件部だけが使用される形式もよく使われる．その場合，条件部が論理値を返す順次節であることを利用すれば，次のような‘途中ぬけ出し’が記述できる．

```
real s:=0;
while real t; read (t); print (t);
    t>0
  do s:=s+t; print ((s, newline)) od
```

7.2 局所化の範囲と宣言

ALGOL 68 の宣言局所化は，ごく一般的なブロック構造によっているが，ブロックを構成する構文の種類をふやしたこと，および，順次節の末尾を除く任意の場所で宣言を評価できるようにしたことが特徴となっている．これは，宣言の有効範囲を可能な限り小さくし，かつ使用する直前に宣言を置くことによって，テキストのモジュール化と読みやすさを高めるのがその目的である．テキスト上で，使用された名前に対応する宣言の見出しやすさは，findability とよばれ，これを高めるためには宣言を一カ所にまとめるのがよいとする議論もある．PASCAL ではこの立場をとっており[25]，宣言は各手続きの先頭でしか許さない．しかし手続きが大きくなってくると，さまざまのレベルの変数や定数の寄せ集まりとなり，かなり読みにくくなる．これを避けるためには手続きを細分化せねばならず，結局，小さなブロックを多用するのとそれほど変りはなくなるようである．宣言局所化の例を示す．

```
if real n; read (n); n≥0
  then int i:=0;
       while i**3<n do i:=i+1 od;
       print (i−1)
fi
```

7.3 定　数

型の先頭に **ref** がついていない値は定数とみなされるが，これの使用が ALGOL 68 のプログラミングにひとつの特色を与えている．定数の宣言は同一宣言で行われるが，これは静的なものである必要はなく，実行時に評価することもできる．したがって，プログラムの実行全体にわたって値が不変であるというふつうの意味での定数に加えて，ある範囲の中に限られた定数というものも宣言することができる．たとえば次のとおり．

```
begin real n; read (n);
    real n2=n×n, n3=n×n×n, ns=sqrt (n);
      ⋮
end
```

この種の宣言は単純型のみならず構造を持つ型についても適用できる．

```
[1 : 10, 1 : 100] int ii;
        ⋮
[ ] int subii=ii [3, 20 : 80]
```

subii は，この宣言が評価された時点における *ii* の部分一次元配列の内容を値とする定数配列となる．以後 *subii* への参照は一次元配列のそれとなり，*ii* への参照よりも効率よく行われる．すなわちこの種の同一宣言の利用は，不注意な書きかえから値を守るという安全性の面と，プログラマ指定による高能率化の実現という2通りの効用があるわけである．

7.4 型の同値性

ALGOL 68 における型の扱いは，完全な構造同値(structural equivalence)の立場をとっており，プログラムテキスト上での名前や宣言位置による区別を主とする名前同値(name equivalence)とは異なっている．後者の立場では，部分配列の型などにいくらかの問題を含んではいるものの，全体としては，「プログラマが異なる名前をつけて(あるいは異なる場所で)宣言した型は異なるものとして扱う」という方針が貫かれている．またこの方式の方が処理系も簡単になる．構造同値の考え方は，型の構造とその最下水準に位置する単純型の配置そのものがその型の意味を決めるというもので，値の意味を厳密に規定しようとする昨今のプログラム言語の流れから考えると，一段階低水準であるともいえよう．ただし，かなり大規模なプログラムにおいても，ALGOL 68 の方式で充分実用になっていることも確かである．

ALGOL 68 において，同じ構造を持つ複数個の型を識別する唯一の手段は，名前が型の一部に含まれている構造型を使用することである．たとえば，以下の4個の型は全く別のものとして扱われる．

struct (**int** *a*, **real** *b*)　**struct** (**int** *a*, **real** *c*)
struct (**int** *b*, **real** *a*)　**struct** (**int** *x*, **real** *y*)

これを利用すれば，次のようなプログラミングが可能となる．

```
mode distance=struct (real d),
     time=struct (real t),
     speed=struct (real s);
op ×=(speed a, time b) distance:
        (distance x;
        d of x := s of a×t of b; x);
op /=(distance a, time b) speed:
        (speed x; s of x:=d of a/t of b; x)
op −=(time a,b) time:
        (time x; t of x:=t of a−t of b; x)
distance total;
time start,finish, time;
speed speed;
```

上に宣言した3個の型はすべて相異なるものとなり，

```
total:=start; speed:=finish×time;
```

といった文は禁止される．かわりに，以下のようなプログラミングとなる．

```
d of total:=42.195;
read ((start, finish));
speed:=total/(finish−start)
```

3行目の "/" と "−" とはここで定義した演算子である．

7.5 演算子と関数

前節の例のように，プログラマが自分用の演算子を定義でき，しかもそれが既存の記号と同じであってもよい (overloading) ことは，特に高水準の応用プログラムにおいて有用な機能である．演算子を定義して使用する局面を大別すると次のふたつになる．

1) 四則演算や論理演算が数学的に定義されている集合，たとえば有理数(分数)，ベクトルや行列，記号列としての数式や論理式などを扱う場合．この場合には四則や論理演算の記号が使われる．3次元ベクトルの例を示す．

```
mode vec=struct (real x,y,z);
op x=(vec u) real: x of u,
   y=(vec u) real: y of u,
   z=(vec u) real: z of u;
op +=(vec u) vec : u,
   −=(vec u) vec: (−xu, −yu, −zu);
   +=(vec u,v) vec:
          (xu+xv, yu+yv, zu+zv);
   −=(vec u,v) vec:
          (xu−xv, yu−yv, zu−zv);
op ×=(vec u, real r) vec:
          (xu×r, yu×r, zu×r);
op ×=(real r, vec u) vec: u×r;
op ×=(vec u,v) vec: (yu×zv−zu×yv,
      zu×xv−xu×zv, xu×yv−yu×xv);
op /×=(vec u,v) real:
      xu×xv+yu×yv+zu×zv;
op perpen=(vec u,v) bool: u/×v≦smallreal
```

これらの宣言により，ベクトルをふつうの数学的記法に沿って書き表わし，演算を施すことができる．

2) その他一般の場合．指示名による演算子定義がなされることが多いが，'つけ加え' たり '取り去っ' たりする演算には "+" や "−" が使われることも多い．

```
op gcd=(int a,b) int:
      if b=0 then abs a
            else b gcd (a mod b) fi;
op min=([ ] real a) real:
       (real x:=maxreal;
        for i from lwb a to upb a
         do if a[i]<x then x:=a[i] fi od;
        x);
op yoneda=(int a) int:
          (a mod 2=0|a|a×3+1) over 2;
read (i); while i>1 do i:=yoneda i od
```

演算子は '結果' を返すものであるが，ALGOL 68では結果として返す値の型には全く何の制限もない．これは，大域生成子と動的データ構造を扱うプログラムで特に有用な性質である．すなわち，静的に宣言されたデータのみによって処理を行う場合には，'大きな' データのやりとりは効率を下げるだけでありがた味は少なく，参照渡しで行うふつうの方法でほぼ充分である．しかし，データが動的に増減する処理では事情は全く異なる．

7.6 参 照 値

ALGOL 68 のプログラミングでは，名前（ポインタ）は **ref** という形で直接的に取り扱われるほかに，参照はがしという自動型変換の対象ともなっている．この変換は，処理の対象が参照値そのものではなく参照されているほうの値であることが '文脈によって明らかな' 場合に行われるもので，プログラム言語に(ごく原始的ではあるが)知恵を盛り込む試みのひとつである．ほかの自動型変換についても同様である．次

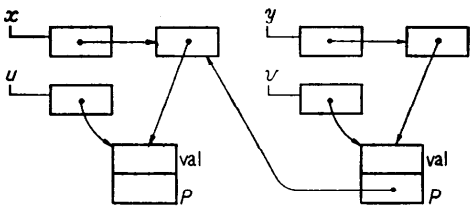


図-7 参照値によるデータ構造

の例でみてみよう.

mode *a*=**ref** *b*;
mode *b*=**struct** (**int** *val*, **ref** *a p*);
a x, *y*; *b u*, *v*;
x:=*u*; *y*:=*v*; *p* **of** *v*:=*x*;

ここまでで図-7のような構造が作られている. ここで

val **of** *p* **of** *y*:=100

とすると変数 *u* の *val* 領域に 100 が代入される. 参照はがしをプログラマが陽に指定しなければならない言語, たとえば PASCAL では同様な指定を

y. *p*. *val*

ではなく, 参照はがしの演算記号 '↑' を使って

y↑. *p*↑↑. *val*

とする必要がある.

参照値をこのように特別扱いした結果として, 参照値自体に対する演算 (代入, 比較) には多少の注意が必要となった. 上の例でも, 代入 '*x*:=*u*' によって変数 *x* に代入されるのは *u* の値ではなく *u* の名前 (*u* の値へのポインタ) である. また, 構造型の変数の要素を指定したもの, たとえば上の例でいえば *p* **of** *u* が見出し名 (*p*) で指定された領域の内容ではなく, その領域への参照値であるのにも注意しなければならない. したがって, たとえば内容そのものが **nil** であるかどうかを調べるには, 陽な型指定を使って

p **of** *u*=:**ref a** (**nil**)

とする. また, もう一段先の内容 (図-7 参照) を調べるには, *p* **of** *u* の型 (**ref ref a**=**ref ref ref b**) から参照を 2 回はがす指定をすればよい.

p **of** *u*=:**ref b** (**nil**) (または **a** (**nil**))

7.7 生 成 子

生成子のうち, 大域生成子 (**heap**) によって確保された箱はプログラム終了までの寿命を持つが, 局所生成子 (**loc**) による箱の寿命は, テキスト上でその生成子を含み, モード, 変数, 同一, および演算子の宣言のどれかひとつ以上を含む局所化範囲の終了までと決められている. この, '宣言を含んだら局所化範囲を閉じる' 方式は ALGOL 60 よりの継承である. 反復構文の制御名 (**for** の後の名前) は宣言されたことにならないので, 次のような三角行列の構成が可能である.

[1 : *n*] **ref** [ ] **real** *w*;
for *i* **to** *n* **do** *w*[*i*]=**loc** [1 : *i*] **real od**

局所生成子によって確保される箱の寿命が限られていることから, 参照値の代入に関してひとつの問題が生じる. すなわち, '存在しない箱への参照値' が使用される可能性が出てくるのである. 例を見よう.

ref real *xx*, *yy*; **real** *y*;
begin real *x*:=3.0;
　　xx:=*x*;
　　yy:=**loc real**:=4.0
end;
y:=*xx*; *print* (*y*); *y*:=*yy*; *print* (*y*)

begin と **end** で囲まれた部分には変数宣言 (**real** *x*) が含まれているので, **loc** で確保された箱と変数 *x* の箱の寿命は両方とも **end** までである. したがって, その後では変数 *xx* と *yy* の値は定義されないことになり, 出力命令で 3.0 と 4.0 は出力され得ない. これを防ぐために, 代入の規則として

'代入される値の有効範囲は変数の有効
範囲と同じかそれより広いこと'

という一項が設けられているが, これをプログラムテキストだけから検査するのは原理的に不可能であり, プログラマが注意するほかはない. 実際問題としては, 参照値を局所化範囲の外へ持ち出す場合には大域生成子 **heap** が使われるので, 上記の制限が問題となることはほとんどない.

8. ALGOL 68 C とその処理系

筆者の知る限り, ALGOL 68 の処理系が計算機メーカによってサポートされたことはない. それに, 言語自体の仕様が, 研究者個人で短期間に処理系を作成できるほど簡単なものでもない. これらが, ALGOL 68 の文化圏から我が国が取り残された大きな理由であろう. 欧州における事情は, これとは全く異なる. 長年にわたる ALGOL 60 の使用・研究の経験を踏まえて, ALGOL 68 の旧版・改訂版それぞれに対する処理系開発が各国で盛んに行われてきた. 文献の 3), 5), 8) を収めた Sigplan Notices は, 英国で行われた ALGOL 68 会議の内容を記録したものであるが, その席上で処理系に関する情報収集が行われた[11]. それ

によれば，英・米・仏・独・オランダ・ベルギー・チェコなどの大学や研究所において 20 以上の処理系作りが行われており，その大部分はすでに稼動し実用に供せられている．これらの処理系にはそれぞれ特徴があり，実際に動く計算機も IBM 360/370, PDP 10, PDP 11, ICL 1900, CYBER 70/72, CII 10070, UNIVAC 11×× など数多い．本章では，これら処理系の中で，処理方式に特徴があり，多くの移植例を持ち，実際の応用分野でも成果を上げている ALGOL 68 C について述べる．

8.1 ALGOL 68 C

ALGOL 68 C[4] は，英国ケンブリッジ大学で開発された ALGOL 68 の一方言，およびその処理系である．言語の設計と処理系の開発の作業は 1973 年ごろから開始された．この年は ALGOL 68 の改訂版[29]が WG 2.1 において承認された年である．したがって，開発作業は旧版[28]の内容に改訂の動きを盛り込んだものから出発して，後に改訂版の内容にあわせた変更が行われた．開発者は，同大学の計算機研究所および計算サービス部に属していた S. R. Bourne, A. D. Birrell, I. Walker の 3 人である．

開発に当っては，以下に述べる項目が示すような実用性に重点が置かれた．

1) 教育用の小さなプログラム（数百〜2, 3 千行程度）のみならず，巨大な応用プログラムにも適用可能なこと．このために，処理効率を重視して言語を制限したり，プログラムの大きさに上限を設けたりすることはされていない．

2) 多機種の上で働くようなくふうをすること．これは，コンパイル過程を機械独立な部分とほかの部分とに分割することで，かなりの程度実現されている．

3) コンパイラ自体の保守性が良いこと．このため，コンパイラは ALGOL 68 C 自身で記述され，かつ，1)で述べた巨大応用プログラムとして扱われている．

実用性を重んじた開発の効果は，いくつかの大規模応用プログラムへの適用と，多数の大学・研究機関への移植となって現われている．ケンブリッジ大学における応用プログラムの例としては，R. Needham[20] が開発している OS マシン，CAP 計算機の制御プログラム（約 1 万行）と，I. C. Braid[5),21] が行っている CAD プロジェクトのプログラム（約 5 万行）とがあげられる．どちらの場合も ALGOL 68 が持つデータや演算の定義機能が充分に活用されているとともに，巨大プログラム作成を目的とした分割コンパイルの機能の利用により，10 人程度の人数による共同開発が進められてきている．

移植に関しては，英国内の諸大学[8] (Essex, Edinbough など) にとどまらず，UCLA やカーネギーメロンなどの米国の大学にも広まっている．我が国においても，東京大学の大型計算機センター[13]と電気通信大学に移植され，利用されている．

8.2 言語の特徴とコンパイル方式

ALGOL 68 C が元の ALGOL 68 と異なる主な点は次のとおりである．

＊分割コンパイルを可能とする構文が導入されている．

＊実行の流れを制御する条件として，論理式のほかに述語とよばれる条件付 and と or とが使用できる．

＊新しい値を代入する以前の変数の（古い）値を一時的に保持する構文が新設されている．

＊並列実行と，そのための同期処理手段であるセマフォとが実現されていない．

＊書式つき入出力手続きがない．

＊ヒープのゴミ集め機能がない．

並列処理・書式・ゴミ集めの各機能を持たないにもかかわらず大規模プログラムの開発に使用されていることは，前節で述べたとおりである．なお，現在，これらの機能を加えた第 2 版作成の作業が進行中である．

コンパイルの方式として特徴的なことは，分割コンパイル可能なこと，および移植を意識した中間言語 (ZCODE) の採用である．分割コンパイル[12]は，あるプログラムのコンパイルを，あらかじめ処理されている親プログラムの環境のもとで行うことで実現される．論理的には，子プログラムの文面を親プログラムの指定された箇所へ埋め込むことと同等である．プログラミングに際しては，手続き本体や処理の一部分を‘別コンパイル指定’で置きかえたものをまず作成し，次に各部分を作成する．親のプログラムをコンパイルすると，別コンパイル指定の箇所におけるコンパイル環境，たとえば有効な変数・定数・型などの宣言内容や目的コード作成に必要な数値などが，指定されたファイルへ出力される．子プログラムのコンパイル時にはこのファイルの内容が使われる．

プログラムのコンパイル手順を次に示す．

（1） 初期化用ファイルが読み込まれる．このファイルには，後で述べる ZCODE 機械の仕様，たとえば，汎用レジスタの数，整数値の表現に必要な記憶ユ

ニット数，手続き呼出しのためのスタック上のフレームの大きさなどが書かれている．

(2) 親プログラムが指定されている場合はその環境ファイルが，指定されていない場合は標準前置部の環境ファイルが，それぞれ読み込まれる．

(3) プログラム自身が読み込まれ，構文解析が行われる．この場合，そのプログラムと親のプログラムの中で定義されていない変数や型がある場合は，親の親，さらにその親という具合に環境ファイルがたどられる．解析の結果は木構造データとして主記憶中に作られる．

(4) 型の同値性の決定，演算子の同定，strong でない文脈における自動型変換および釣合せなどの処理が行われる．

(5) 初期化ファイルの指定によって定まる仮想機械，ZCODE 機械のアセンブリプログラムの形式で目的プログラムが出力される．

以上の部分が，ALGOL 68 C において‘コンパイラ’と呼ばれている部分の仕事である．コンパイラ自体は ALGOL 68 C で書かれており，移植に際してもその内容を変更する必要はない．

8.3 中間言語 ZCODE

コンパイル処理全体の中で，機械従属な部分，すなわち ZCODE アセンブリプログラムを目的となる機械の上で動くプログラムへ変換する部分は，トランスレータと呼ばれている．高水準言語の処理に用いられる中間言語としては，かなり単純なスタック機械のそれが使われることが多い．例としては BCPL の P コードや PASCAL-P などがある．ALGOL 68 C では，移植時に作成する必要があるトランスレータの負担を軽くするために，中間言語用の仮想機械，ZCODE 機械をかなり現実の計算機に近いものとして設定している．具体的には，演算と指標づけとに共用できる汎用レジスタ複数個と，大域および局所フレームポインタ，手続き呼出し時の環境ポインタを備えたものとなっている．細かい構成はコンパイラに対する初期化ファイルによって指定できるようになっており，たとえば，IBM 370 系，PDP-11 系，単一アキュムレータ機械などに似せることができる．これらの仕組みにより，コンパイラの方は 1 万行を越す規模となっているのに対し，トランスレータの方は 1,500～3,000 行程度のプログラムとなっている．

8.4 分割コンパイル

これまでの OS におけるプログラムの結合は，名前(文字列)とその値あるいは番地だけを手がかりとして行われてきた．これに対して，プログラムに書かれたさまざまな文脈も結合の手段とする方式がいくつか提案され，開発されてきている．ALGOL 68 C では 8.2 節で述べた環境ファイルの手法を使ってこれを実現している．環境ファイルの保持には，目的プログラムが占めるスペースの 2～4 割程度のスペースが必要であり，コンパイラがこれを読み込むのにも多少の時間がかかる．しかし，環境ファイルは(親プログラムから子プログラムへの)一種のオンライン仕様書であり，コンパイラによる自動的な仕様合わせの作業にはなくてはならないものである．ALGOL 68 C の方式は，環境ファイルの形式やその前処理の程度の差こそあれ，数多くの分割コンパイルシステムによって採用されるようになってきている．

9. おわりに

本解説では，ALGOL 68 の成立，言語記述の手法，概念構成，言語の概要，実際のプログラミング，処理系の例としての ALGOL 68 C などについて概説した．第 1, 2 章でも述べたとおり，怪獣論のみが先行した感のある我が国においては，処理系作りはおろか，言語の理解さえもあまり行われてこなかったのが実情である．その点を考慮して，本解説では言語の内容の記述とプログラミングとに特に重点を置いた．現実のプログラミングにおける実用性と，言語設計・開発のための基礎概念の集積という点から考えて，ALGOL 68 の重要性は今後とも増してゆくものと思われる．

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 Considered 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 course, 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, 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～Vol. 8, No. 7 ダイヤモンド社 (1969, 1970).

(昭和55年8月4日受付)