講　　座

# ALGOL N について

## (VI) Standard declarations (つづき)

筧　　捷　彦*

### 6.4 Complex

```
(SD 4.0.1) let  complex  operate
                before all left after right
(SD 4.0.2) let  arg, Re, Im, i  operate
                before all left after right
(SD 4.1.1) let  complex  represent
  procedure( ) structure(real: real, imaginary:
    real)
  : structure(real: real, imaginary: real)
(SD 4.2.1) let  i  represent
  procedure( ) (complex)
  : structure (real: 0.0, imaginary: 1.0)
(SD 4.2.2) let  ( )i  represent
  procedure ( real a ) (complex):
    structure (real: 0.0, imaginary: copy a)
(SD 4.2.3) let  i( )  represent
  procedure( real a ) (complex):
    structure (real: 0.0, imaginary: copy a)
(SD 4.3.1) let  Re( )  represent
  procedure( (complex)a ) real: (a[real: ])
(SD 4.3.2) let  Im( )  represent
  procedure( (complex)a ) real: (a[imaginary: ])
(SD 4.4.1) let  ( )⁻  represent
  procedure( (complex)a ) (complex):
    structure(real: copy  Re  a,
              imaginary:-Im  a)
(SD 4.5.1) let  +( )  represent
  procedure( (complex)a ) (complex) : (copy a)
(SD 4.5.2) let  ( )+( )  represent
  procedure( (complex)a, (complex)b) (complex):
  (  (Re a+Re b)+i(Im a+Im b)  ) ,
  procedure( (complex)a, real b ) (complex):
  (  (Re a+b)+i Im a  ) ,
  procedure( real a, (complex)b ) (complex):
  (  (a+Re b)+i(real[mode a]0+Im b)  )
(SD 4.6.1) let  −( )  represent
  procedure( (complex)a ) (complex):
    (  −Re a+i(−Im a)  )
(SD 4.6.2) let  ( )−( )  represent
  procedure( (complex)a, (complex)b ) (complex):
  (  a+(−b)  ) ,
  procedure( (complex)a, real b ) (complex):
  (  a+(−b)  ) ,
  procedure( real a, (complex)b ) (complex):
  (  a+(−b)  )
(SD 4.7.1) let  ( )×( )  represent
  procedure( (complex)a, (complex)b )(complex):
  (  (Re a×Re b+Im a×Im b )
    +i( Im a×Re b+Re a×Im b )  ) ,
  procedure( (complex)a, real b ) (complex):
  (  Re a×b+i( Im a×b )  ) ,
  procedure( real a, (complex)b ) (complex):
  (  a×Re b+i( real[mode a]1×Im b )  )
(SD 4.8.1) let  ( )/( )  represent
  procedure( (complex)a, (complex)b )(complex):
    (  (a×b⁻)/Re(b×b⁻)  ) ,
  procedure( (complex)a, real b ) (complex):
  (  ( Rea / b )+i( Ima / b )  ) ,
  procedure( real a, (complex)b ) (complex):
    (  ( a×b⁻ )/Re( b×b⁻ )  )
(SD 4.9.1) let  ( )⁻¹  represent
  procedure( (complex)a ) (complex):
    (  a⁻/Re(a×a⁻)  )
(SD 4.10.1) let  ( )↑( )  represent
  procedure( (complex)a, (complex)b )(complex):
    (  exp( (log a)×b )  ) ,
  procedure( (complex)a, real b ) (complex):
```

* 東京大学工学部計算工学科

```
  (   exp( (log a)×b )   )
 procedure( real a, (complex)b )(complex):
  (   exp( (log a)×b )   )
```

{ (SD 4.4.1) から (SD 4.10.1) で定義された各演算の結果は, *parameter a* の (各 *real*:, *imaginary*: *-part* の) *mode* となる. }

```
(SD 4.11.1) let   abs( )   represent
 procedure( (complex)a )real:
  (   √( Re(a×a⁻) )   )
(SD 4.12.1) let   arg( )   represent
 procedure( (complex)a )real:
 (if   Re a=0
     then if  Im a=0  then  0.0
          else if  Im a>0  then π/2
                           else π×3/2
     else
 begin let  x  be  tan⁻¹(Im a/Re a);
       if   Re a>0
       then if  Im a≥0  then x else π×2−x
       else π+x
 end)
(SD 4.13.1) let   exp( )   represent
 procedure( (complex)a )(complex):
 (   exp(Re a)×( cos(Im a)+i sin(Im a) )   )
(SD 4.14.1) let   log( )   represent
 procedure( (complex)a )(complex):
  (   log abs a+i arg a   )
```

{ **arg** *a*, **log** *a* は, 偏角として $[0, 2\pi)$ をとる. }

### 6.5 List Processing

```
(SD 5.0.1) let   ,   operate   before left
                               after , right
(SD 5.0.2) let   .   operate
           before then, else, ,, ←, :=, ≡, ≠,
             =, ≠ left
           after . right
(SD 5.0.3) let   car, cdr, list   operate
           befor all left   after all right
(SD 5.0.4) let   step   operate
             befor , left
(SD 5.0.5) let   unitl   operate
             after right
(SD 5.0.6) let   while   operate
           before , left   after right
(SD 5.0.7) let   for   operate
           before all left
(SD 5.0.8) let   :=   operate
(SD 5.0.9) let   do   operate
           after all right
(SD 5.0.10) let   case   operate
           before all left
(SD 5.0.11) let   of   operate
           after all right
(SD 5.1.1) * let   comma   be
 structure(head: enproc nil, tail: enproc nil)
(SD 5.1.2) * let   step   be
 structure(from: enproc real, step: enproc real,
           until: enproc real)
(SD 5.1.3) * let   while   be
 structure(value: enproc real, while: enproc
   bits)
(SD 5.1.4) * let   simple   be
 structure(head: enproc real, tail: enproc nil)
(SD 5.1.5) * let   terminal   be
 structure(head: enproc real, tail: enproc real)
(SD 5.2.1) T, T′∈T とする.
 let   ( ),( )   represent
  procedure( T a, T′ b) reference:
  (enref structure(head: enproc a,
                   tail: enproc b) )
(SD 5.3.1) let   ( ).( )   represent
  procedure( reference a, reference b )
    reference:
  ( enref structure(car: a, cdr: b) )
(SD 5.3.2) let   car ( )   represent
  procedure( reference a )reference:
 ( (deref a as structure(car: nil, cdr: nil) )
                                  [car: ])
(SD 5.3.3) let   cdr ( )   represent
  procedure( reference a )reference:
  ( (deref a as structure(car: nil, cdr: nil))
                                  [cdr: ]   )
(SD 5.3.4) let   list ( )   represent
  procedure( reference a )reference:
  begin
    if   deref a match comma
  then begin let  s  be  deref a as comma;
             s[head: ] ( ). list s[tail:] ( )
       end
```

```
    else a.nil
    end
```

{ $a_1, a_2, \cdots, a_n$ が **reference**-*type* のとき，
**list** $a_1, a_2, \cdots, a_n$ は $a_1.\ a_2.\cdots.\ a_n.$ **nil** である． }

```
(SD 5.4.1) let ( )step( )until( ) represent
    procedure(real, real, real)reference : (a, b, c)
    (enref structure(from: enproc a, step:
                     enproc b, until: enproc c) )
(SD 5.4.2) let ( )while( ) represent
    procedure(real, Boolean)reference : (a, b)
    ( enref structure(value: enproc a, while:
                      enproc b)                 )
(SD 5.4.3) let for( ):=( )do( ) represent
    procedure( real a, real b, effect name c):
    begin a:=b; c end
    procedure( real name a, reference b,
               effect name c ):
(       if deref b match terminal
    then begin
         let s be deref b as terminal;
             a:=s[head:]( ); c;
             a:=s[tail:]( ); c          end
    else if deref b match simple
    then begin let s be deref b as simple;
             a:=s[head:]( ); c;
             for a:=s[tail:]( ) do c    end
    else if deref b match step
    then begine
         let s be deref b as step;
             let u, v be real;
             a:=s[from:]( );
         l: u:=s[step:]( ); v:=⌊until:]( );
             if sign (u)×(v−a) < 0
               then go to m;
             c; a:=a+s[step:]( ); go to l;
       m: end
    else if deref b match while
    then begin
         let s be deref b as while;
             a:=s[value:]( );
         l: if s[while:]( ) then
             begin c; go to l end
       end
    else if deref b match comma
    then begin
         let s be deref b as comma;
             for a:=s[head:]( ) do c;
             for a:=s[tail:]( ) do c
         end
    else dummy)
```

(SD 5.5.1) $T \in$ **T** とする．

```
  let ( )else( ) represent
    procedure( reference a, T name b )
      structure(then: reference, else: enproc T):
      sturcture(then: a, else: enproc b)
```

(SD 5.5.2) $T \in$ **T** とする．

```
  let case( )of( ) represent
    procedure( integer a, (T else T)b )T:
    (  if 1=a then b  ) ,
    procedure( integer a, (reference else T)b )
      T:
    begin let i be entier a;
          let x be b[then:];
          let y be b[else:];
          let z be sturucture (head:
                    enoproc T,
                  tail: enproc T);
      if i≤0 then y( )
    else if deref x match comma
    then begin
         let s be deref x as comma;
           if i=1 then s[head:]( )
           else case i−1 of s[tail:]( )
             else y( )
         end
    else if deref x match z
    then begin
         let s be deref x as z;
           if i=1 then s[head:]( )
           else if i=2 then s[tail:]( )
           else y( )
           end
    else T
    end
```

{ $c_1, c_2, \cdots, c_n$ と $d$ が *type* $T$ であり，$a$ の値が $i$ であったとすると，**case** $a$ **of** $c_1, c_2, \cdots, c_n$ **else** $d$ は，$1 \leq i \leq n$ のとき $c_i$，そうでないとき $d$ を意味する． }

【報告第2版では，この節に **collateral** $a_1, a_2, \cdots, a_n$ という *formula* が定義されている．これは **ALGOL 68** での collateral clause に相当するものであり，$a_1, a_2, \cdots, a_n$ を同時に *elaborate* することを意味する．報告第2版の記述のままでは，*standard declaration* として明確に取り入れることには困難がありこの講座では省略した．現在，こうした parallel な *elaboration* をも取り入れた第3版の検討中である．】

### 6.6 Mode

```
(SD 6.0.1) let    constant, scale, precision,
                  exact, varying, mode
                  operate
           before left after all right
(SD 6.1.1) T∈T とする.
  let    constant( )    represent
    procedure( T a )(procedure(T)T):
      ( procedure(T)T : (x)(copy a) )
(SD 6.2.1) let    scale( )    represent
    procedure( real a )(procedure((real)real):
      ( procedure(real)real : (x)(round(x/a)×a) )
(SD 6.2.2) let    ( )scale( )    represent
    procedure( real a, real b)(procedure(real)
      real):
      ( procdure(real)real : (x)
      (if    abs x≤a    then    (scale b) (x)
                        else    copy a      ) )
(SD 6.3.1) let    precision( )    represent
  procedure( real a )(procedure(real)real):
    ( procedure(real)real :(x)
      ( code( p1:x, p2:a ) real:
      core  let Q←parameter;
            let W←w(Q[p1:]);
            let R←w(Q[p2:]);
            let V←some real value such that
                  |V−W|<(1/2)(|V|+|W|)×R;
            g(Q)→Q';
            t(Q')←real;
            h(Q')←H0[real];
            w(Q')←V;
            ⇒Q'
      end of core
(SD 6.4.1) let    exact( )    represent
  procedure( bits a )(procedure(bits)bits):
    ( procedure(bits)bits : (x)
      (    (x conc (size a)*0) up to size
        a    )                          )
  procedure( string a ) (procedure(string)
    string):
    ( procedure(string)string : (x)
      ((x conc (size a)* filler) up to size a) )
(SD 6.5.1) T∈{bits, string} とする.
  let    varying( )    represent
    procedure( T a ) (procedure(T)T):
      ( procedure(T)T: (x)
      (if    size x > size a
          then    x up to size a    else
            x    )                      )
(SD 6.6.1) T∈{real, bits, string} とする.
  let    mode( )    represent
    procedure( T a )(procedure(T)T):
      ( procedure(T)T: (x)
      begin let y be copy a;y:=x; y end )
```

【報告第2版では，**bits, string, integer** 間の *type*-conversion 用 *procedure* が *standard declaration* として収録されている．これらは，入出力の *standard declaration* として考えるのが，適当であるので，ここでは省略した．

入出力については，未だ明確な記述がなされてはいないが，方針としては，

1. 任意 **type** の *value* が外界と出入りしうる．
2. 特に **string**-*type* での入出力を中心に考える．
3. **string**-*type* と他の *type* 間との conversion 用には，**ALGOL 60** 同様の formatted edit (indit) を用意する．

こととして，これも第3版には収録する予定である．】

(昭和47年7月13日受付)