数理解析研究所講究録
第66巻 1969年 66-74

6. Standard declarations

This section describes the ordered set SD of standard declarations supporting the use of the language. The description takes the form of orderly enumeration of a family of disjoint subsets in SD.† The description of each member of this family is headed by a reference numbering of the form (SDi.j.k). The ordering in SD is such that one element under ($SDi_1.j_1.k_1$) precedes another element under ($SDi_2.j_2.k_2$) if ($i_1.j_1.k_1$) precedes ($i_2.j_2.k_2$) in the lexical order.

Each member description begins either with a '{' (not a '{') or with an '∫'.

(a) If a member description begins with a '{', then the matching '}' terminates that description, and the member has a single element, which is the enclosed text.

(b) If a member description begins with an '∫', then the part enclosed by the first following '{' and the matching '}', with which the member description is terminated, gives an element of the member when asterisked symbols (T* for example) in that part are consistently replaced by certain objects as prescribed after the leading '∫' in terms of common mathematical notations (T*∈ **T** for example).

---

† In the present document, instead of the whole family covering SD, only some illustrating subfamily will be presented. Other members, including those for input/output operations, will be found elsewhere.

6.1. Basic operations

6.1.1. Copying and enproceduring operations

```
(SD1.1.1) {let copy operate before all left after copy right}

(SD1.1.2) {let enproc operate before all left after all right}

(SD1.1.3) ∫T*∈T {let copy() represent procedure(T*)T* by
    ((original) code (c:original, d:T* ) T* by
        (core(Q) let w(Q)={<c:,Q1>,<d:,Q2>};
            m(Q2)←m(Q1);  w(Q2)←w(Q1);⇒ Q2 end of core))}

(SD1.1.4) ∫T*∈T {let enproc() represent
   procedure(T*)procedure()T* by
          ((body) procedure()T* by
                          (() body))}
```

6.1.2. Simple assignment operations

```
(SD1.2.1) {let ←,:= operate before ←,:= left after all right}

(SD1.2.2) ∫T*∈{real, bits, string, reference}∪T_procedure, Δ*∈{←,:=}
   {let ()Δ*() represent procedure(T*,T*)effect by
       ((destination, source)
        code (d:destination, s:source) effect by
        (core(Q) let w(Q)={<d:,Q1>,<s:,Q2>};
           w(Q1)← p(m(Q1),w(Q2)); ⇒ Q0 end of core))}
```

6.1.3. Simple comparison operations

```
(SD1.3.1) {let =,≠ operate before   left after ←,:= right}

(SD1.3.2) {let <,≤,≥,> operate before =,≠ left
                              after ←,:= right}

(SD1.3.3) ∫T*∈T, Δ*∈{=,≠}; T*=real, Δ*∈{<,≤,≥,>}
   {let ()Δ*() represent procedure(T*,T*)bits by
      ((left,right) code (l:copy left, r:right, t:1 ) bits by
         (core(Q) let w(Q)={<l:,Q1>,<r:,Q2>,<t:,Q3>};
            if w(Q1)Δ*w(Q2) then ⇒Q3, else →next;
            w(Q3)← 0; ⇒Q3  end of core))}
```

6.1.4. Conditional operations

(SD1.4.1) {let if operate before all left}

(SD1.4.2) {let then operate}

(SD1.4.3) {let else, do operate after all right}

(SD1.4.4) ∫T*εT {let if()then()else() represent
procedure(bits, T*, T*) T* by
((condition, then, else)
code (c:condition, t:enproc then, e:enproc else)
enproc T* by
(core(Q) let $w(Q)=\{\langle c:,Q_1\rangle,\langle t:,Q_2\rangle,\langle e:,Q_3\rangle\}$;
if the bit string $w(Q_1)$ contains at least one 1
then $\Rightarrow Q_2$, else $\Rightarrow Q_3$ end of core) ())}

(SD1.4.5) {let if()do() represent procedure(bits, effect)effect
by((condition, statement) if condition then statement
else dummy)}

6.1.5. Basic arithmetic operations

(SD1.5.1) {let +,− operate before $=,\neq,<,\leq,\geq,>$ left
after ←,:= right}

(SD1.5.2) {let ×,/ operate before $=,\neq,<,\leq,\geq,>,+,-$ left
after ←,:= right}

(SD1.5.3) ∫Δ*ε{+,−,×,/} {let ()Δ*() represent
procedure(real,real)real by ((left,right)
code (a:copy left, b:right) real by
(core(Q) let $w(Q)=\{\langle a:,Q_1\rangle,\langle b:,Q_2\rangle\}$;
if the arithmetic operation meant by $w(Q_1)\Delta^* w(Q_2)$
can be performed then →next, else $\Rightarrow$L0;
let W_3 be the resulting value of that operation
(possibly with some implementation dependent deviation);
$w(Q_1)\leftarrow p(m(Q_1),W_3)$; $\Rightarrow Q_1$ end of core))}

(SD1.5.4) {let −() represent procedure(real)real by
((right) begin (right)b; (copy b)a; a:=0; a−b end)}

(SD1.5.5) {let /() represent procedure(real)real by
((right) begin (right)b; (copy b)a; a:=1; a/b end)}

6.1.6. Some enquiry operations

```
(SD1.6.1) {let mode, length, bd operate before all left
                                        after   right}

(SD1.6.2) {let mode() represent procedure(real)structure
   (fix:bits, min:real, step:real, max:real ) by ((real)
      begin (real)x; code (fix:1, min:copy x, step:real,
                              max:copy x) (mode real) by
        (core(Q) let w(Q)={<fix:,Q1>,<min:,Q2>,
                           <step:,Q3>,<max:,Q4>};
           let M stand for m(Q2);
           w(Q2)<- the minimum value in W_M;
           w(Q4)<- the maximum value in W_M;
           if M=real [ precision R ] with a real number R,
              then ->next, else ->Kfixed;
           w(Q3)<- p(m(Q3), R); w(Q1)<- 0; =>Q;
        Kfixed: let M=real [R1:R2:R3] with real numbers R1,
                                                        R2, R3;
           w(Q3)<- p(m(Q3), R2); =>Q  end of core) end)}

(SD1.6.3) ∫T*∈{bits,string} {let mode() represent
   procedure(T*)structure (exact:bits, length:real ) by ((some)
      code (s:some, t:(exact:1, length: ) ) (mode T*) by
      (core(Q) let w(Q)={<s:,Q1>,<t:,Q2>};        integer
         let w(Q2)={<exact:,Q3>,<length:,Q4>};
         let m(Q1)=T* [ Y I ] ,where I is an integer and Y
            is either exact or varying ;
         w(Q4)<- p(m(Q4), I);
         if Y=exact  then =>Q2, else ->next;
         w(Q3)<- 0;  =>Q2  end of core)                  )}

(SD1.6.4) ∫T*∈T {let mode() represent procedure(array T*)
   structure (lbd:real, ubd:real ) by  ((array)
      code (a:array, t:(lbd:1, ubd: ) ) (mode array T*) by
      (core(Q) let w(Q)={<a:,Q1>,<t:,Q2>};   integer
         let w(Q2)={<lbd:,Q3>,<ubd:,Q4>};
         let m(Q1)=array [I1:I2] T*  with integers I1,I2;
         w(Q3)<- p(m(Q3),I1); w(Q4)<- p(m(Q4),I2);
           =>Q2  end of core)       )}
```

6.1.6 continued

(SD1.6.5) ∫T*∈{<u>bits</u>,<u>string</u>} {<u>let</u> <u>length</u>() <u>represent</u>
<u>procedure</u>(T*)<u>real</u> <u>by</u> ((string) <u>code</u> (s:string,t: <u>)</u> <u>real</u> <u>by</u> → <u>integer</u>
(<u>core</u>(Q) let w(Q)={<s:,Q_1>,<t:,Q_2>};
let I be the integral length of w(Q_1);
w(Q_2)← p(m(Q_2),I); ⇒Q_2 <u>end of core</u>))}

(SD1.6.6) ∫T*∈T {<u>let</u> <u>bd</u>() <u>represent</u> <u>procedure</u>(<u>array</u> T*)<u>real</u>
<u>by</u> ((array) (<u>mode</u> array)[lbd:])}

(SD1.6.7) ∫T*∈T {<u>let</u> ()<u>bd</u> <u>represent</u> <u>procedure</u>(<u>array</u> T*)<u>real</u>
<u>by</u> ((array) (<u>mode</u> array)[ubd:])}

6.2. Extended operations

6.2.1. Repetitive operations

```
(SD2.1.1) {let while,until operate before   left after all right}

(SD2.1.2) {let succ,step operate before   left
   after ←,:=,=,≠,<,≤,≥,>,+,−,×,/ right}

(SD2.1.3) ∫T*εT {let ()succ() represent procedure(T*,T*)
   structure (init:enproc T*, succ:enproc T*) by
   ((a,s) (init:enproc a, succ:enproc s ) )}

(SD2.1.4) ∫T*εT {let ()while() represent procedure(T*succT*,bits)
   structure (init:enprocT*,succ:enprocT*,while:enproc bits ) by
   ((r,t) begin (     r)rl; (init:copy rl[init:],
             succ:copy rl[succ:], while:enproc t) end)}

(SD2.1.5) {let ()step() represent procedure(real,real)
   structure (init:enproc real, step:enproc real ) by
   ((a,b) (init:enproc a, step:enproc b ) )}

(SD2.1.6) {let step() represent procedure(real)(real step real)
   by ((b) 1 step b )}

(SD2.1.7) {let ()until() represent procedure(real step real,real)
   structure (init:enproc real, step:enproc real,
                                until:enproc real ) by
   ((progression,limit)
      begin (     progression)ab; (init:copy ab[init:],
             step:copy ab[step:], until:enproc limit) end)}

(SD2.1.8) {let ()until() represent procedure(real,real)
   (step real until real) by ((a,c) a step 1 until c )}

(SD2.1.9) {let until() represent procedure (real)(real until real)
   by ((c) 1 until c )}
```

6.2.1 continued

```
(SD2.1.10) {let from operate before all left}
                                      ( )
(SD2.1.11) ∫T*∈T {let ()from()do^represent
   procedure(T*, T*succT*while bits, effect) effect by
   ((cvar, domain, statement)
       begin (cvar)cv; (domain)dom; cv← dom[init:]();
       next: if dom[while:]() do
               begin statement; cv← dom[succ:](); go to next end
       end )}
(SD2.1.12)  {let ()from()do() represent
   procedure(real, until real, effect) effect by
   ((cvar, domain, statement)
       begin (cvar)cv; (copy cv)v1; (copy domain)dom;
           (copy dom[init:]() )a1; (copy dom[step:]())b1;
           (copy dom[until:]())c1;
           v1 from a1 succ v1+b1 while if b1>0 then v1≤c1
                   else if b1<0 then v1≥c1 else v1≠c1
             do begin cv:=v1; statement end
       end)}
```

6.2.2. Assign-and-hold operations

```
(SD2.2.1) {let the operate before all left after   right}

(SD2.2.2) ∫T*∈T {let the() represent
   procedure(T*) procedure(effect) T* by
   ((expression) procedure(effect) T* by ((dummy) expression))}

(SD2.2.3) ∫T*∈T, Δ*∈{←,:=}{let ()Δ*() represent
   procedure(the T*, T*) T* by ((the destination, source)
       begin (the destination(dummy))destination;
           destinationΔ*source; destination
       end)}
```

6.2.3. Reference handling operations

```
(SD2.3.1) {let ref, has type, as type operate before all left
                                               after   right }

(SD2.3.2) ∫T*∈T {let ref() represent procedure(T*)reference
   by ((referent) code (q:referent, r:reference ) reference
      by (core(Q) let w(Q)={<q:,Q1>,<r:,Q2>} ;
            w(Q2)← {Q1}; ⇒Q2 end of core))}

(SD2.3.3) ∫T*∈T {let ()has type() represent
   procedure(reference, T*)bits by ((ref,type)
      code (r:ref, t:type, b:0 ) bits by
      (core(Q) let w(Q)={<r:,Q1>,<t:,Q2>,<b:,Q3>};
         if w(Q1)= empty, then ⇒Q3,else →next;
         let w(Q1)={Q4};
         if t(Q2)=t(Q4) then →next, else ⇒Q3;
         w(Q3)← 1;  ⇒Q3  end of core))}

(SD2.3.4) ∫T*∈T, T*≠reference {let ()as type() represent
   procedure(reference, T*) T* by ((ref, type)
      code (r:ref, t:type) T* by
      (core(Q) let w(Q)={<r:,Q1>,<t:,Q2>};
         if w(Q1)=empty, then ⇒Q2, else →next:
         let w(Q1)={Q4};
         if t(Q2)=t(Q4) then ⇒Q4, else ⇒Q2 end of core))}
```