5. DECLARATIONS

Declarations introduce a measure of security into the language by associating identifiers with variables, arrays, switches, procedures and classes used in the programs.

A declaration determines the type and structure of a quantity.

Declarations must appear in a block head, and upon exit from that block (unless it is the outermost block of a class body), they lose their significance.

declaration

{ type-declaration
| ARRAY-declaration
| SWITCH-declaration
| PROCEDURE-declaration
| CLASS-declaration
| EXTERNAL-declaration }

Common to each of the succeeding sections, which discuss these declarations, is the definition of an identifier-list.

identifier-list

identifier [, identifier]...

in which the identifiers must all be different.

Examples:

valid	AXOLOTL	
	A, B, MARY, B_14	
invalid	A B	no separating comma
	A, END, Q	END is a key word and is not allowed as an identifier.
	B1,B2, B1	the same identifier appears twice
	BIORTHOGONAL, BIORTHOGONALISATION	the identifiers are not distinct (only the first twelve characters are significant).

5.1 TYPE DECLARATIONS

Type declarations associate each identifier in the associated identifier list with a variable of a given type. This restricts the range of values that may be assigned to that variable.

type-declaration

```
{ [LONG]REAL                  }
{ [SHORT]INTEGER              }
{ CHARACTER                   } identifier-list
{ BOOLEAN                     }
{ REF(CLASS-identifier)       }
{ TEXT                        }
```

Examples:

valid: value-type

```
REAL R1, R2
LONG REAL P1
INTEGER I, J, K
SHORT INTEGER SH1
CHARACTER A, B, C, D
BOOLEAN B1, B2, B3
```

reference-type

```
REF(POINT) P, Q
TEXT MY_STRING, YOUR_STRING
```

invalid:

```
SHORT REAL ONE          no such type
REF LINE_1, LINE_2      no such type - the qualification
                                is omitted.
```

INTEGER and SHORT INTEGER variables may only assume whole numbers.

REAL and LONG REAL variables may only assume numbers.

N.B. An arithmetic expression of any arithmetic type (SHORT INTEGER, INTEGER, REAL, LONG REAL) may be assigned to an arithmetic variable. If the types do not correspond, the expression is first converted to the type of the variable and then transferred, subject to its being in range. If it is not in range, then a run time error will result.

BOOLEAN variables may only assume the values TRUE and FALSE.

CHARACTER variables may only assume values from the data character set.

TEXT variables may only assume strings of characters from the data character set or NOTEXT.

REF variables may only assume the value NONE or references to objects belonging to their qualifying class or its subclasses.

Each variable declared in a type declaration has an initial value (given in the table below). Thereafter the value of a variable is the one last assigned to it, or if no assignment has yet been made, the initial value.

Type	Initial value	Assignable range
INTEGER	0	Whole number in the range -2^{31} through $2^{31}-1$.
SHORT INTEGER	0	Whole number in the range -2^{15} through $2^{15}-1$.
REAL	0.0	Number in the range $\pm(0 \rightarrow 10^{75}$ approx.) to 7 decimal places.
LONG REAL	0.0	Number in the range $\pm(0 \rightarrow 10^{75}$ approx.) to 16 decimal places.
BOOLEAN	FALSE	TRUE, FALSE.
CHARACTER	CHAR(0)	CHAR(0), CHAR(1), ... CHAR(255).
REF(CLASS-identifier)	NONE	NONE or any object of the qualifying class or included in the qualifying class.
TEXT	NOTEXT	NOTEXT or any string of characters from the data character set of length 0 through $(2^{15}-20)$ characters.

5.2 ARRAY DECLARATIONS

An array is a structure of many components (subscripted variables) all of the same type. Each component has the same identifier (the array identifier), and they are distinguished one from another by subscripts. Arrays invite the user to group like data under one identifier.

Arrays are declared with a certain shape. They can have 1 through 8 dimensions (which is the number of subscripts necessary to specify a certain component), and each dimension has a fixed range specified by giving an upper and lower bound.

Pictorial representations of one, two and three dimensional arrays are now given.

One dimensional array

```
INTEGER ARRAY NUMBER (4:9);
```

NUMBER(4)
NUMBER(5)
NUMBER(6)
NUMBER(7)
NUMBER(8)
NUMBER(9)

Dimensions 1:
Lower subscript bound = 4
Upper subscript bound = 9
Declares 6 subscripted variables each of type INTEGER and initialised to 0.

Example:

Find the sum of the components of the array NUMBER.

```
SUM := 0;
FOR I := 4 STEP 1 UNTIL 9 DO
    SUM := SUM + NUMBER(I);
```

Two dimensional array

```
REF(POINT) ARRAY A(0:3, 2:4);
```

A(0, 2)	A(0, 3)	A(0, 4)
A(1, 2)	A(1, 3)	A(1, 4)
A(2, 2)	A(2, 3)	A(2, 4)
A(3, 2)	A(3, 3)	A(3, 4)

Dimensions 2:
Lower subscript bound 1 = 0
Upper subscript bound 1 = 3
Lower subscript bound 2 = 2
Upper subscript bound 2 = 4

Declares 12 subscripted variables each of type REF(POINT) and each initialised to NONE.

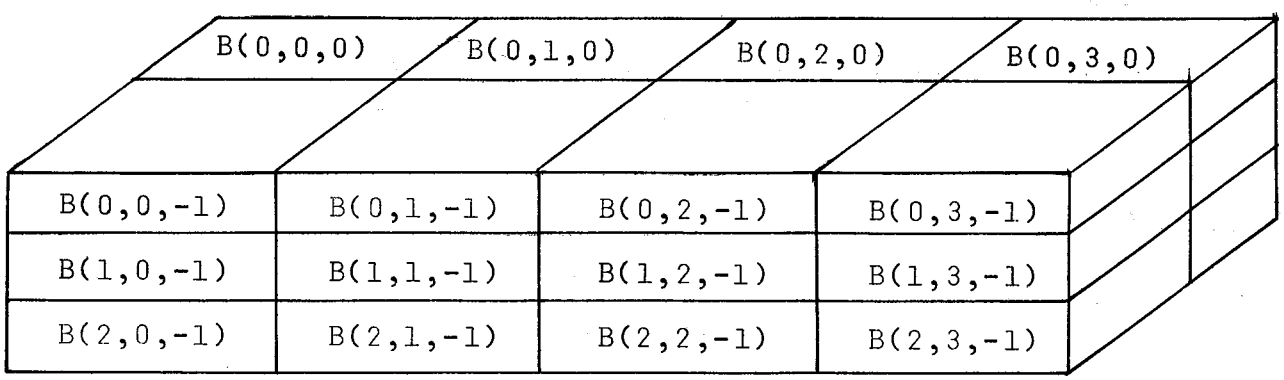
Example:

Scan the array and count how many subscripted variables are currently referencing NONE.

```
COUNT := 0;
FOR I := 0 STEP 1 UNTIL 3 DO
    FOR J := 2 STEP 1 UNTIL 4 DO
        IF A(I, J) == NONE
          THEN COUNT := COUNT + 1;
```

Three dimensional array

```
BOOLEAN ARRAY B(0:2, 0:3, -1:0);
```



(Each component is represented by a cube).

Dimensions 3:
Lower subscript bound 1 = 0
Upper subscript bound 1 = 2
Lower subscript bound 2 = 0
Upper subscript bound 2 = 3
Lower subscript bound 3 = -1
Upper subscript bound 3 = 0

Scan through the array and set the values of each subscripted variable to TRUE.

```
FOR I := 0 STEP 1 UNTIL 2 DO
    FOR J := 0 STEP 1 UNTIL 3 DO
        FOR K := -1 STEP 1 UNTIL 0 DO
            B(I,J,K) := TRUE;
```

ARRAY-declaration

[type] ARRAY ARRAY-segment [,ARRAY-segment]...

ARRAY-segment

identifier-list (lower-bound:upper-bound
[,lower-bound:upper-bound]...)

Examples:

```
ARRAY-segment        A(1:10)
                     B,C(1:15, 0:N**2)

ARRAY-declaration    ARRAY ALFA (0:N)
                     REF(LINE) ARRAY L,M(0:P.X)
                     TEXT ARRAY R,S(1:5), T(1:4,ININT:IF X=0
                                          THEN 10 ELSE 100)
```

Each identifier in an array declaration is associated with an array of a given type (if no type is given, the type is taken to be REAL). To enable arrays of the same type, but with different shapes (number of dimensions and subscript ranges) to be declared in the same place, an array declaration contains one or more ARRAY-segments which are identifier-lists followed by their common shape. The number of dimensions is the number of upper-bound lower-bound pairs and the range of each subscript is specified directly by the upper-bound/lower-bound pairs taken in order.

Note that although the lower- and upper-bounds may be arithmetic expressions their value is the rounded integer as evaluated on entry to the block. To be valid, the value of each lower bound must be less than or equal to the value of its associated upper bound. Neither bound may refer to a quantity declared in the same block head. The value of each subscripted variable is initialised according to the type of the array.

5.3 SWITCH DECLARATIONS

A switch is declared with a list of designational-expressions which are accessed by an index. The length of the list, N, gives the number of switch elements. The value of each element is the current value of the designational-expression (a label to a statement, or another switch value).

SWITCH-declaration

```
SWITCH SWITCH-identifier := designational-expression
                                [,designational-expression]...
```

Examples:

```
SWITCH SW := L1, L2, T(N), M1
SWITCH T := M1, IF N<0 THEN L1 ELSE L3, M1
```

A SWITCH declaration contains a list of designational expressions each of which is given a positive index (starting from 1) by counting the items in the list from left to right. The value of the switch-designator (see section 6.6) corresponding to a given index is the current value of the designational expression having the index in the corresponding list. (An expression in a switch list is re-evaluated every time it is referred to).

5.4 PROCEDURE DECLARATION

A procedure declaration defines an action cluster and gives it a formal name. The action cluster is the body of the PROCEDURE. To increase the generality of the concept, there are facilities for transmitting parameters and returning a function value at run time. Thus a procedure is able to interact with its calling environment by bindings established by the calling mechanism.

By means of the procedure concept, special instances of declared action clusters become meaningful units within the SIMULA program.

PROCEDURE-declaration

{proper-procedure-declaration
function-declaration}

proper-procedure-declaration

PROCEDURE PROCEDURE-heading PROCEDURE-body

function-declaration

type PROCEDURE PROCEDURE-heading PROCEDURE-body

PROCEDURE-heading

PROCEDURE-identifier{; formal-parameter-part mode-part spec-part}

formal-parameter-part

(identifier-list);

mode-part

{VALUE identifier-list;
 NAME identifier-list;}...

spec-part

{{type
 [type] ARRAY
 PROCEDURE
 type PROCEDURE
 LABEL
 SWITCH} identifier-list;} ...

PROCEDURE-body

statement

Examples:

proper-procedure-declaration

```
PROCEDURE SWAP(A,B); NAME A, B;
                     REAL A, B;
BEGIN  REAL X;
       X := A;
       A := B; B := X;
END ***SWAP***

PROCEDURE OUTCOLUMN(A,N); REAL ARRAY A; INTEGER N;
BEGIN  INTEGER J;
       FOR J := 1 STEP 1 UNTIL N DO
           BEGIN  OUTFIX (A(J), 5, 12);
                  OUTIMAGE;
           END;
END ***OUTCOLUMN***

PROCEDURE TREETRAVERSE(N); REF(NODE)N;
INSPECT N DO
BEGIN  TREETRAVERSE(LEFTBRANCH);
       DUMP;
       TREETRAVERSE(RIGHTBRANCH);
END ***RECURSIVE TRAVERSE***
```

A proper-procedure is used as a statement in its own right.

```
INIMAGE

OUTTEXT("PROPER-PROCEDURE")

IF B THEN SWAP(TEMP1,TEMP2)
```

function-declaration

```
REF(POINT) PROCEDURE ADD(Q); REF(POINT)Q;
   IF Q =/= NONE THEN ADD := NEW POINT(X+Q.X,Y+Q.Y)

REAL PROCEDURE NORM(A,N); REAL ARRAY A; INTEGER N;
BEGIN  REAL T; INTEGER I;
       FOR I := 1 STEP 1 UNTIL N DO
           T := T + A(I)**2;
       NORM := SQRT(T)
END ***NORM***

INTEGER PROCEDURE FACTORIAL(N); INTEGER N;
IF N < 0 THEN ERROR ELSE
   IF N < 2 THEN FACTORIAL := 1
            ELSE FACTORIAL := N*FACTORIAL(N-1)
```

A function returns a value of the type indicated in its declaration, and may be used wherever a value of that type is legal. (It may also be used as a statement in which case the function value is ignored);

```
P :- R.ADD(S)

X := NORM(MATRIX, 10)

IF NORM (MATRIX, 10) < &-6 THEN
   OUTTEXT ("ELEMENTS␣ALL␣ZERO")
```

A PROCEDURE-declaration defines a procedure associated with a PROCEDURE-identifier. The chief component of a PROCEDURE declaration is the PROCEDURE-body which may be activated through a PROCEDURE-statement or a function-designator.

If the procedure has formal-parameters, then their identifiers are listed in the formal-parameter part. No formal-parameter identifier may appear more than once, and specifications of all formal-parameters must be supplied. Whenever the procedure is activated by a PROCEDURE-statement or function-designator, the formal-parameters are either assigned the values of the actual-parameters (call by value, call by reference) or else replaced by the actual-parameters (call by name).

Identifiers in the PROCEDURE-body which are not formal-parameters are local if declared within that body, otherwise they are non-local. A PROCEDURE-body acts like a block regardless of its format. Consequently, the scope of any label to a statement within the body or to the body itself can never be extended beyond the PROCEDURE-body. In addition, if the identifier of a formal-parameter is redeclared within the PROCEDURE-body (or used as a label), it is given local significance and the corresponding actual-parameter becomes inaccessible.

When a PROCEDURE-declaration is given a type, it designates a function. The type of this function is the type of the PROCEDURE and for any activation its value is the last value assigned to an occurrence of the PROCEDURE-identifier within the PROCEDURE-body. If no such assignment is made, then the result of the function-designator takes a default value which is identical to the initial value of a declared variable of that type.

Within the PROCEDURE-body of a function-procedure, an assignment may be made to an occurrence of the PROCEDURE-identifier. Every other occurrence of a PROCEDURE-identifier denotes another activation of the procedure (recursion).

PARAMETER TRANSMISSION

N.B. There are three possible modes for parameter transmission

call by value
call by reference
call by name

All three are allowable for parameters to procedures, but only call by value and call by reference are valid for parameters to classes. Since call by value and call by reference are common to procedure and class declarations, these sections apply to class declarations as well.

If no mode is specified in the mode part, then the parameter is transmitted by the appropriate default mode which is call by value for value type parameters, and call by reference for other kinds of parameters.

The available transmission modes for legal parameters to procedures are shown in the following table:

PROCEDURE PARAMETERS

Parameter	Transmission		
	by value	by reference	by name
value-type	D	X	0
object-reference	X	D	0
TEXT	0	D	0
value-type ARRAY	0	D	0
reference-type ARRAY	X	D	0
PROCEDURE	X	D	0
type PROCEDURE	X	D	0
LABEL	X	D	0
SWITCH	X	D	0

D : default mode
0 : optional
X : illegal

Call by value

A formal-parameter called by value designates a local copy of the value obtained by evaluating the actual parameter. The evaluation takes place at the time of procedure entry. Throughout the lifespan of the procedure or class body, the formal-parameter acts like a local variable (or value type array) and its contents may be accessed and changed by local assignments. In the case of an object, the contents may be accessed and changed from without by remote accessing.

Value specification is redundant for a parameter of value type. There is no call by value option for object references, reference-type ARRAYS, PROCEDURES, type-PROCEDURES, LABELS and SWITCHES.

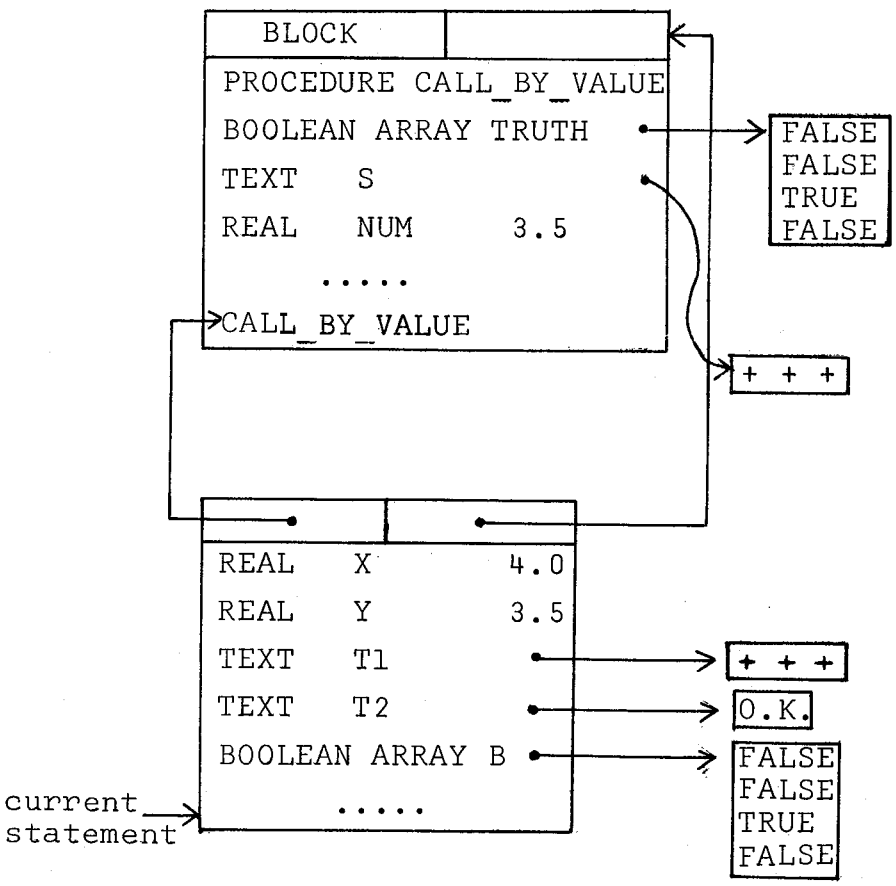
Example:

Given the program:

```
BEGIN  PROCEDURE CALL_BY_VALUE(X,Y,T1,T2,B);
       VALUE T1, T2, B;
       REAL X, Y; TEXT T1, T2; BOOLEAN ARRAY B;
       BEGIN ..................... END;

       BOOLEAN ARRAY TRUTH(0:3);
       TEXT S;
       REAL NUM;
       NUM := 3.5; S :- COPY("+++");
       TRUTH(2) := TRUE;
       CALL_BY_VALUE(4,NUM,S,"O.K",TRUTH);
       ...
END
```

then a snapshot on procedure entry shows the initialised values for the formal-parameters.


BLOCK
PROCEDURE CALL_BY_VALUE
BOOLEAN ARRAY TRUTH
TEXT S
REAL NUM 3.5
.....
CALL_BY_VALUE
FALSE
FALSE
TRUE
FALSE
+ + +
REAL X 4.0
REAL Y 3.5
TEXT T1
TEXT T2
BOOLEAN ARRAY B
.....
+ + +
O.K.
FALSE
FALSE
TRUE
FALSE
current
statement

Call by reference

A formal-parameter called by reference designates a local copy of the reference obtained by evaluating the corresponding actual-parameter. The evaluation takes place at the time of the procedure entry or object generation. Throughout the lifespan of the PROCEDURE- or CLASS-body, the formal-parameter acts like a local quantity.

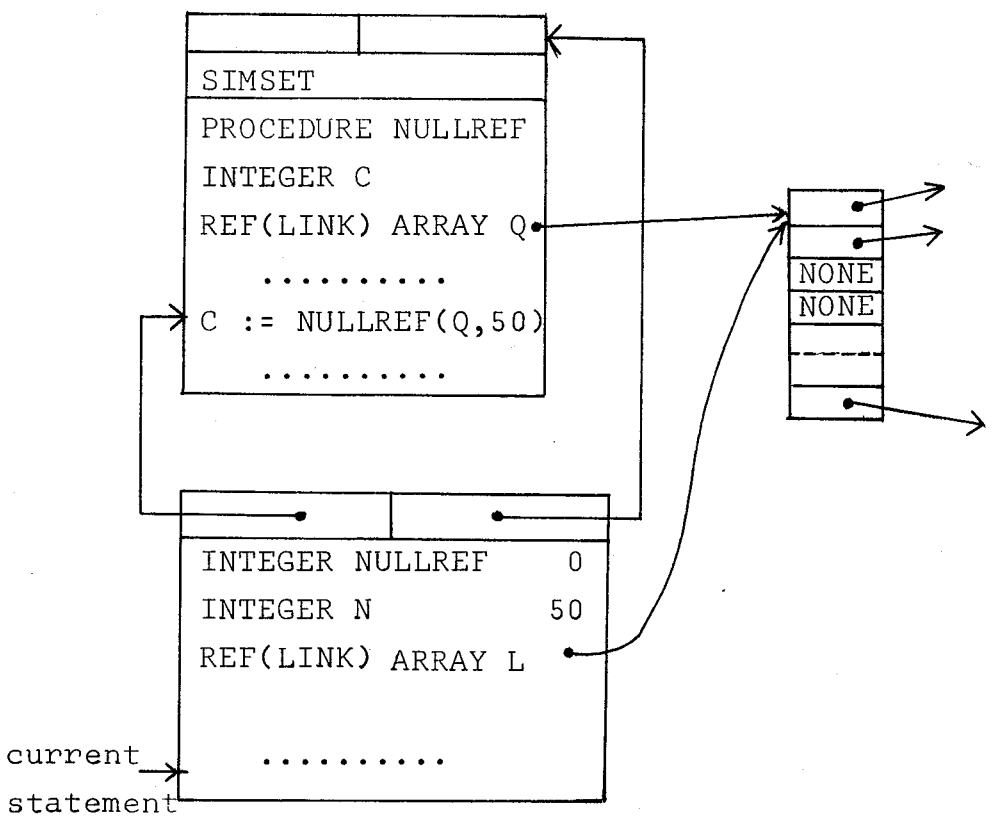
If the formal-parameter is a reference-type variable, its contents may be changed by assignments within the body (or externally by remote accessing in the case of objects).

An ARRAY, PROCEDURE, LABEL or SWITCH parameter called by reference remains fixed and references the same entity throughout its scope. Of course, the contents of an ARRAY called by reference may well be changed through assignments to its components.

For an object-reference type parameter (which may be a variable, PROCEDURE or an ARRAY), the qualification of the matching actual-parameter may coincide with, or be inner to, the qualification of the formal-parameter.

```
SIMSET BEGIN  INTEGER PROCEDURE NULLREF(L,N); REF(LINK) ARRAY L;
                                                INTEGER N;
              BEGIN  INTEGER J, COUNT;
                     FOR J := 1 STEP 1 UNTIL N DO
                         IF L(J) == NONE THEN COUNT := COUNT + 1;
                     NULLREF := COUNT;
              END;
              INTEGER C;
              REF(LINK) ARRAY Q(1:50);
              ......
              C := NULLREF(Q,50);
              ......
```

On the procedure call, a snapshot is



Call by name

Call by name is an optional transmission mode available for parameters to procedures, but not to classes. It represents a textual replacement in that the formal-parameter may be considered replaced throughout the PROCEDURE-body by the corresponding actual-parameter.

Whereas call by value and call by reference operate on variables local to the PROCEDURE-body itself, call by name operates on non-local quantities and can alter global quantities. It is therefore especially useful for the controlled alteration of several variables external to the PROCEDURE-body.

The following rules apply:

1) the type of a name parameter is that prescribed by the corresponding formal specification.

2) if the type of the actual-parameter does not coincide with that of the formal specification, then an evaluation of the expression is followed by an assignment of the value or reference obtained to a fictitious variable of the latter type. This assignment is subject to the rules of section 7.2. The value or reference obtained by the evaluation is the contents of the fictitious variable.

Section 7.2 defines the meaning of an assignment to a variable which is a formal-parameter called by name, or is a subscripted variable whose array identifier is a formal-parameter called by name, if the type of the actual parameter does not coincide with that of the formal specification.

Assignment to a PROCEDURE-identifier which is a formal parameter is illegal, regardless of its transmission mode.

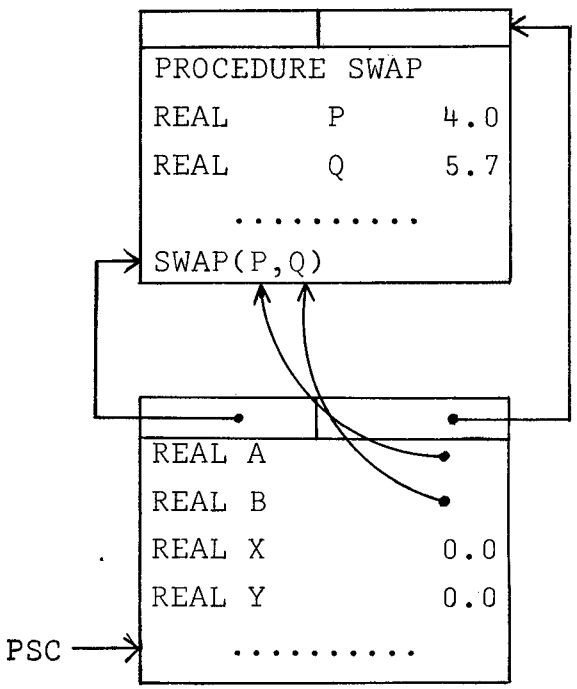
Notice that each dynamic occurrence of a formal-parameter called by name, regardless of its kind, may invoke the execution of a non-trivial expression, e.g. if its actual-parameter is a remote-identifier.

Example:

```
BEGIN  PROCEDURE SWAP(A,B); NAME A, B; REAL A, B;

       BEGIN  REAL X;
              X := A;
              A := B; B := X;
       END ***SWAP***;
       REAL P, Q;
       P := 4.0; Q := 5.7;
       SWAP(P,Q);
       ..........
END
```

A snapshot at the procedure call is:



Program Sequence Control (PSC) references the current statement

No local copies are made. Every occurrence of A or B in the PROCEDURE-body means a re-evaluation of the actual-parameter. Notice that the actual-parameters are evaluated in the context of the procedure call.

5.5 CLASS DECLARATIONS

CLASS declarations define concepts. CLASS instances are called objects and many objects of a given class may exist simultaneously to be examined and manipulated by the program. Objects are generalisations of blocks and procedures, as the environment of either a block instance or a procedure instance can only observe the result of its actions.

CLASS-declaration

[CLASS-identifier]main-part

main-part

CLASS CLASS-identifier { ; / parameter-part } [virtual-part]CLASS-body

parameter-part

(identifier-list); [VALUE identifier-list;]... CLASS-spec-part

CLASS-spec-part

{ { [LONG]REAL / [SHORT]INTEGER / CHARACTER / BOOLEAN / REF(CLASS-identifier) / TEXT / [type]ARRAY } identifier-list; } ...

virtual-part

```
          ⎧⎧ [LONG] REAL                ⎫                     ⎫
          ⎪⎪ [SHORT] INTEGER            ⎪                     ⎪
          ⎪⎪ CHARACTER                  ⎪                     ⎪
          ⎪⎪ BOOLEAN                    ⎪                     ⎪
VIRTUAL:  ⎪⎪ REF(CLASS-identifier)      ⎪   identifier-list;  ⎪ ...
          ⎨⎨ TEXT                       ⎬                     ⎬
          ⎪⎪ [type] ARRAY               ⎪                     ⎪
          ⎪⎪ PROCEDURE                  ⎪                     ⎪
          ⎪⎪ type  PROCEDURE            ⎪                     ⎪
          ⎪⎪ LABEL                      ⎪                     ⎪
          ⎩⎩ SWITCH                     ⎭                     ⎭
```

CLASS-body

```
{ statement  }
{ split-body }
```

split-body

```
BEGIN [declaration;] ... [statement;] ... INNER
                             [; statement] ... END
```

The available transmission modes for legal parameters to classes are shown in the following table:

CLASS PARAMETERS

Parameter	Transmission	
	call by value	call by reference
value-type	D	X
reference-type	X	D
TEXT	O	D
value-type ARRAY	O	D
reference-type ARRAY	X	D

D : default mode
O : optional
X : illegal

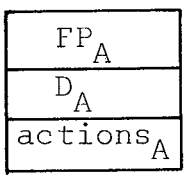
The transmission modes "call by value" and "call by reference" are explained in section 5.4.

The discussion of CLASS-declarations begins by considering two selected examples of increasing scope.

Example 1: Empty prefix and virtual part

The declaration of a class A can have the form:

```
CLASS A(FP_A); SP_A;
BEGIN  D_A;
       actions_A;
END
```



CLASS A declaration A object

A: CLASS-identifier

FP_A: list of the formal parameters of A

SP_A: list of specifications of each of the formal parameters of A

D_A: declarations of the CLASS-body of A

$actions_A$: actions of the CLASS-body of A

The CLASS-body of A is a block (one form of statement).

The quantities passed as parameters or declared in the outermost block of the class body are called the attributes of class A and are attributes of every object of the class. Attributes specified as formal-parameters may only be simple variables or arrays. Declared attributes may be simple variables, arrays, proper-procedures, function-procedures and classes.

The expression

NEW A(...)

creates an object of the class A (the order and number of the actual-parameters must correspond with the order and number of the formal-parameters) and commences execution of its actions. The execution continues until the final END of the class body is encountered, when the execution is terminated. However this may be interrupted in three ways - by a GOTO-statement (which leads out of the object), or by calls on the system procedures "detach", "resume" or "call". A GOTO exit will leave the object in the terminated state. "Detach" suspends the actions of the class body and names it an independent component of the program. Its actions may be continued later by a call on "resume".

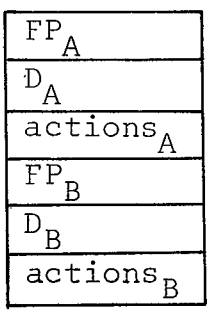
Example 2: Non-empty prefix and empty virtual part

A CLASS-declaration may contain a prefix which is the identifier of another class declared either in the same block, or in the prefix to the same block. The prefixed class is a subclass of the prefix.

e.g.

```
A CLASS B(FP_B); SP_B;
  BEGIN  D_B;
         actions_B;
  END
```

CLASS B declaration



B object

A B-object is a compound object which has all the attributes and actions of the prefix A and the main-part of class B. A B-object is generated by

```
NEW B(AP_A, AP_B)
```

with a list of actual-parameters corresponding in number and order to those of an A-object and those of the main-part of B. On generation, the actions of the A-part are executed first and then those of the B-part. The actions of the A-part have access to the attributes of A only, the actions of the B-part to those of B and of A.

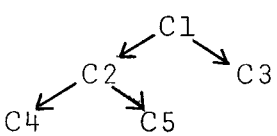
CLASS HIERARCHIES

In general, a hierarchy of classes may be introduced.
e.g.

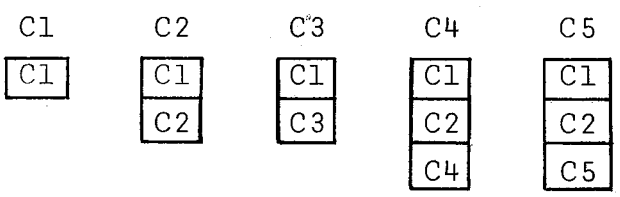
declarations

```
   CLASS C1.....;
C1 CLASS C2.....;
C1 CLASS C3.....;
C2 CLASS C4.....;
C2 CLASS C5.....;
```

hierarchy



objects

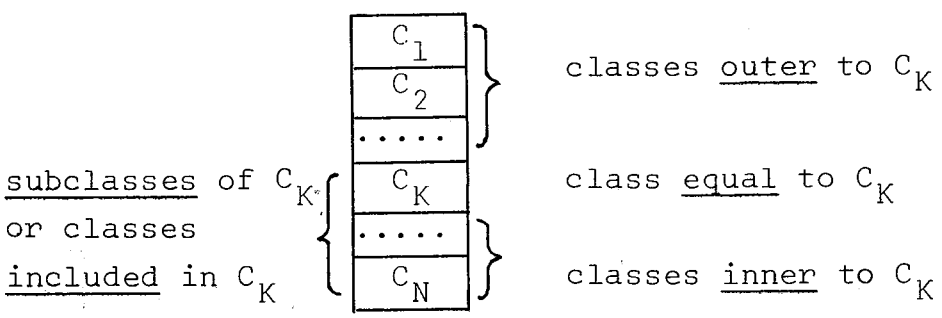


The prefix sequence of a class is the sequence of classes in its prefix chain

e.g. in the above figure, the prefix sequence of

C1 is empty
C2 is C1
C5 is C1 C2

If the prefix sequence of a class C_N is C_1, C_2,....., C_{N-1}, then a C_N object may be depicted by:



An object of a prefixed class is a compound object which is the union of the formal-parameters, declarations and actions of the classes in its prefix sequence together with the structure defined in its own main-part.

The mechanism of concatenation may be extended to the case of prefixed-blocks, e.g.

```
SIMULATION  BEGIN
                ...
            END
```

The execution of a prefixed block begins by executing the actions of the initial operations of the class and then those of the block body. The attributes of the class are accessible inside the block body and this block is thus given a built-in environment in which to operate.

INNER

The order of the execution of actions in an object may be altered from their textual order by use of INNER.

Consider

```
CLASS A ...;
BEGIN  ....; S1; INNER; S2  END;

A CLASS B ...;
  BEGIN .....; S3  END
```

In an A object INNER acts as a dummy statement, and the actions executed are

```
S1; S2
```

In a B object, INNER forces execution of S3 before those of S2, thus

```
S1; S3; S2
```

VIRTUAL QUANTITIES

The virtual mechanism is a method of extending the rules of accessibility, whilst still retaining security. The virtual mechanism is useful in two circumstances.

1) to redefine attributes
2) to give accessibility to attributes declared at inner levels

1) attribute redefinition

Usually when a class hierarchy is constructed, an attribute has a fixed meaning no matter how many prefix levels are added to the class containing its declaration.

Sometimes, however, we may wish to redefine the attribute throughout its scope.

Example: Consider the description of a CLASS ROW for manipulating vectors.

```
CLASS ROW(A,N); REAL ARRAY A; INTEGER N;
BEGIN  REAL PROCEDURE NORM;
       BEGIN  REAL T; INTEGER I;
              FOR I := 1 STEP 1 UNTIL N DO
                  T := T + A(I)*A(I);
                  NORM := SQRT(T);
       END ***NORM***;
       PROCEDURE NORMALISE;
       BEGIN  REAL T; INTEGER I;
              T := NORM;
              IF T ¬= 0 THEN
                 BEGIN  T := 1/T;
                        FOR I := 1 STEP 1 UNTIL N DO
                            A(I) := A(I)*T;
                 END;
       END ***NORMALISE***;
       IF N < 1 THEN ERROR;
END ***ROW***
```

A new ROW object has four attributes:

1) a REAL ARRAY A running from 1 through N
2) a specified upper bound N
3) a REAL PROCEDURE NORM which computes the square root of the sum of the square of its components. (NORM gives the "magnitude" of the ARRAY A).
4) a PROCEDURE NORMALISE which divides through each array element by the current value of NORM.

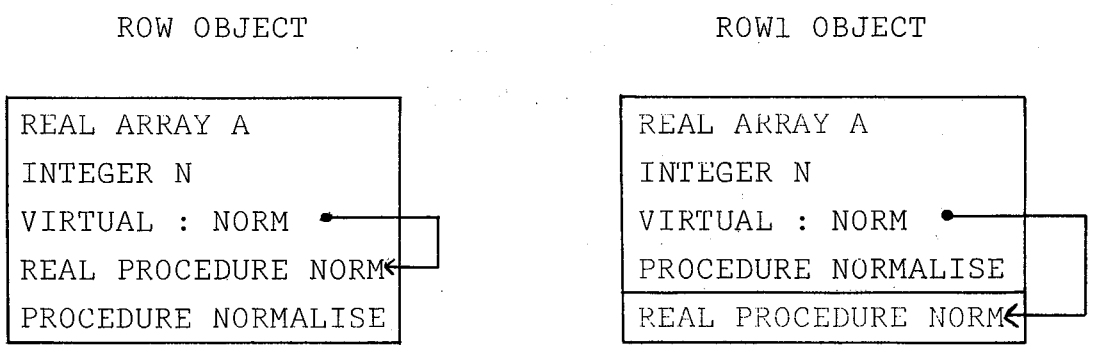
If the user wishes to use a different NORM then the class has to be rewritten.

However by altering the class outline to

```
CLASS ROW(A,N); REAL ARRAY A; INTEGER N;
                VIRTUAL : REAL PROCEDURE NORM;
BEGIN  REAL PROCEDURE NORM...;
       PROCEDURE NORMALISE...;
       IF N < 1 THEN ERROR;
END ***ROW***
```

it becomes possible to alter the definition of NORM in a subclass of ROW, and yet have the new meaning available at the ROW level where it is needed in any call on NORMALISE e.g.

```
ROW CLASS ROW1;
    BEGIN  REAL PROCEDURE NORM;
           BEGIN  REAL S,T; INTEGER I;
                  FOR J := 1 STEP 1 UNTIL N DO
                      BEGIN  T := ABS(A(I));
                             IF T > S THEN S := T;
                      END;
                      NORM := S;
           END ***NORM*** ;
    END ***ROW1***
```



In a ROW1 object, the NORM attribute declared at the ROW level is deleted and is never available. Through the VIRTUAL mechanism, the only valid NORM attribute is that declared at the ROW1 level.

2) Inner accessibility

This is illustrated by an example.

In processing a linked structure which contains objects of various classes, (say A, B) the links are provided by their common part CLASS NODE.

```
CLASS NODE; BEGIN  REF(NODE)NEXT;  END;
NODE CLASS A ....;
NODE CLASS B ....;
```

If the structure is linked by following REF(NODE) NEXT references, then, in the normal way, the information in the structure would be dumped by

```
X :- FIRST;
WHILE X =/= NONE DO
BEGIN  IF X IS A THEN X QUA A.DUMP
                 ELSE X QUA B.DUMP;
       X :- X.NEXT;
END;
```

which is tedious to write. By writing the classes by

```
CLASS NODE; VIRTUAL : PROCEDURE DUMP;
BEGIN  REF(NODE) NEXT;
       ...
END ***NODE*** ;
NODE CLASS A;
     BEGIN  PROCEDURE DUMP ...;
     END;
NODE CLASS B;
     BEGIN  PROCEDURE DUMP ...;
     END;
```

we can write instead

```
X :- FIRST;
WHILE X =/= NONE DO
BEGIN  X.DUMP;
       X :- X.NEXT;
END;
```

A virtual quantity is either unmatched or else matched with an attribute (with the same identifier) declared at the same or an inner prefix level. The matching attribute must be of the same type and the same kind (variable, array, procedure, label or switch) as that specified in the virtual part, except in the case of object-reference-function-procedures. The type REF(C) may be matched with the type REF(C) or type REF(D), where D is a subclass of C. A VIRTUAL proper-PROCEDURE may be matched with any type of procedure. In both these cases, the type of the match is the type of the matching declaration.

A virtual quantity of a given object can have at most one matching attribute. If matching declarations are given at more than one prefix level, then the one at the innermost prefix level is taken. The match is valid at all prefix levels of the object equal or inner to that of the specification.

5.6 EXTERNAL DECLARATIONS

External declarations will be a part of future system development.

6. EXPRESSIONS

Expressions are rules for computing values. Expressions may occur on the right hand sides of assignment statements and as actual parameters.

expression

{ arithmetic-expression
condition
CHARACTER-expression
designational-expression
object-expression
TEXT-expression
TEXT-value }

IF-clause

IF condition THEN

The constituents of expressions are operators and operands. The operands are constants, variables or function designators. No two operands may occur side by side - they must be separated by at least one operator. A discussion of variables and function designators precedes the discussion of expressions.

6.1 VARIABLES

The value of a variable may be used in expressions for forming values and may be changed by assignments to that variable. The value of a variable is the value of the last assignment to it, or its initial value if no assignment has yet been made.

variable

{simple-variable / subscripted-variable / remote-variable}

simple-variable

identifier

subscripted-variable

{identifier / remote-variable} (subscript-list)

subscript-list

arithmetic-expression [, arithmetic-expression]...

remote-variable

{simple-text-expression . / simple-object-expression .} identifier

Examples:

simple-variable:	EPS
	A15
	A16_C
subscripted-variable:	SON(2)
	ELEMENT(I,J)
	THIS SQUARE.LONG_SIDE(2)
remote-variable:	ANNA.FATHER
	APOSTLE(6).LEADER
	C.P.X
	LINE(2).SIDE(3)

A variable is a designation for a single value.

simple-variable

The type of the value of a particular simple variable is given in the declaration for that variable.

remote-variable

An attribute of an object is identified by the following information:

1) a reference to the object
2) a class which includes that of the object
3) an attribute identifier defined in that class or any class belonging to its prefix sequence.

Let X be a simple object expression qualified by class C, and A an attribute identifier.

The X.A is valid if X =/= NONE and class C has at least one attribute identified by A. If there is only one attribute with identifier A contained in class C or in its prefix chain, then it is designated by X.A. If class C contains more than one attribute with identifier A, then X.A designates the attribute with identifier A at the innermost prefix level.

N.B. The main part of any class declaration can contain at most one attribute with that identifier.

Example:

```
    REF(C1)X1; REF(C2)X2; REF(C3)X3; REF(C4)X4

    CLASS C1(A); REAL A;
    BEGIN ..................... END;
C4  CLASS C2;
    BEGIN  TEXT A; ... END;
C2  CLASS C3;
    BEGIN  ..................... END;
C3  CLASS C4;
    BEGIN  BOOLEAN A; ........ END;
X1 :- X2 :- X3 :- X4 :- NEW C4(6.0);
```

remote-variable	type
X1.A, X2 QUA C1.A, X3 QUA C1.A, X4 QUA C1.A	REAL
X1 QUA C2.A, X2.A, X3 QUA C2.A, X4 QUA C2.A	TEXT
X1 QUA C3.A, X2 QUA C3.A, X3.A, X4 QUA C3.A	TEXT
X1 QUA C4.A, X2 QUA C4.A, X3 QUA C4.A, X3 QUA C4.A, X4.A	BOOLEAN

It is not possible to access either label or switch identifiers remotely.

If class C or a class in its prefix chain contains a class attribute, then accessing of any attribute in class C is only possible through a connection statement.

Example:

```
     CLASS C1; BEGIN  REAL D;  END;
  C1 CLASS C2;
     BEGIN  CLASS F.....;
            BOOLEAN Q;
     END;
     REF(C2)X;
```

Then X.F, X.Q, X.D
are all illegal attempts at remote accessing.

But it is legal to write

```
INSPECT X DO
BEGIN  ...F...
       ...Q...
       ...D...
END
```

or, for the REAL attribute of C1, X QUA C1.D.

A simple-text-expression is itself a compound structure in the sense that it has attributes accessible through the dot notation. TEXT-value assignments may be made through the use of the system-defined procedure SUB, STRIP and MAIN to sub-fields (T.SUB, T.STRIP) and super-fields (T.MAIN) of the text object referenced by T.

e.g. T.SUB(1,24) := "CHANGE␣IN␣FIRST␣24␣CHARS"
T.MAIN.STRIP := NOTEXT

Subscripted variables

A subscripted variable is an array component. The type of a subscripted variable is the type of the array. Each arithmetic expression in the subscript list is called a subscript and the complete list of subscripts is enclosed in parentheses ( ). Each subscript acts as a variable of type INTEGER and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable.

No attempt is made to ensure that each subscript lies within the corresponding subscript bounds in an array of two or more dimensions. Such arrays are transformed into one dimensional table and the subscripts computed to give access to the appropriate entry. Provided this rule gives an entry within the bounds of the table, then the subscripts will be accepted. Otherwise, a runtime error will result.

6.2 FUNCTION DESIGNATORS

A function designator denotes the value obtained by evaluating the associated procedure body when supplied with the associated actual parameter list (if any).

function-designator

{identifier / remote-identifier} actual-parameter-list

actual-parameter-list

(actual-parameter[, actual-parameter]...)

actual-parameter

{expression / ARRAY-identifier / SWITCH-identifier / PROCEDURE-identifier / LABEL-identifier}

Examples:

```
ININT
INTEXT(20)
SIN(A+B)
H.FIRST
CURRENT.NEXTEV.SUC
POINT(5).MODULUS
```

Function-designators denote single arithmetic, BOOLEAN, TEXT or CHARACTER values or object- or TEXT-references. These values are obtained by applying a set of rules defined by the procedure declaration to a set of actual parameters. The actual parameter list must correspond in number and order to the formal parameter list in the corresponding procedure declaration.

Certain identifiers, expressed as procedures, are pre-defined in the SIMULA system. A list is given as Appendix B. Calls to standard procedures conform to the syntax of calls to declared procedures and are equivalent to normal procedure calls. The identifier of a system defined procedure is not reserved, and may be declared to have another meaning at any level. (This is not recommended practice as it will obviously make the program more difficult to read and understand). The identifier then assumes the new meaning throughout the scope of the block in which it was declared.

6.3 ARITHMETIC EXPRESSIONS

An arithmetic expression is a rule for computing a number.

arithmetic-expression

```
{simple-arithmetic-expression                          }
{IF-clause simple-arithmetic-expression                }
{          ELSE arithmetic-expression                  }
```

simple-arithmetic-expression

```
[±] arithmetic-primary [arithmetic-operator arithmetic-
                                              primary] ...
```

arithmetic-primary

```
{arithmetic-constant                  }
{arithmetic-variable                  }
{arithmetic-function-designator       }
{(arithmetic-expression)              }
```

arithmetic-operator

```
{+ | - | * | ** | / | //}
```

Examples:

arithmetic-primary

```
15.7
#4
X
SIN(23*PI/180)
(IF X>0 THEN X ELSE -X)
```

arithmetic-expression

```
X**2 + Y**2
X + A(J)**2
EXP(2 + (IF B THEN 3 ELSE 4))
IF X > 0 THEN 1 ELSE IF X < 0 THEN -1 ELSE 0
-3*(-5)
```

Examples of incorrect arithmetic expressions:

`(A + B) C)`	left parenthesis missing
`-3*-5`	two operators together
`(A + B) & - 0.5`	exponent is not a primary
`X + (IF Y < 0 THEN 2)`	ELSE alternative is missing
`X + IF Y < 0 THEN 2 ELSE 3`	the IF expression must be enclosed in parentheses

The value of a simple-arithmetic-expression is the value obtained by executing the arithmetic operations on the actual numeric values of the primaries of the expression. The actual numeric value of a primary is obvious in the case of numbers.

For variables, it is the current value (assigned last in the dynamic sense); and for function-designators it is the value arising from the computing rules defining the function-procedure when applied to the current values of the procedure parameters in the expression. For arithmetic-expressions enclosed in parentheses, the value must, through a recursive analysis, be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic-expressions, which include IF-clauses, a simple-arithmetic-expression is selected on the basis of the actual values of the BOOLEAN conditions.

If the form of the arithmetic expression is

IF condition THEN simple-arithmetic-expression
ELSE arithmetic-expression

the selection of the appropriate value is made according to the rules:

1. The condition in the IF-clause is evaluated.

2. If the result of 1 is TRUE, then the value of the expression is the value of the simple arithmetic expression following the IF-clause. Otherwise it is the value of the arithmetic expression following the ELSE.

   e.g. IF X < 0 THEN 4 ELSE 17

   if X < 0 then the value is 4
   if X = 0 or X > 0 then the value is 17.

Arithmetic operators and types

Apart from the CONDITIONS of IF-clauses, the constituents of arithmetic expressions are of types SHORT INTEGER, INTEGER, REAL or LONG REAL.

The following hierarchy may be associated with these types.

type number	type	shortened notation
1	SHORT INTEGER	SI
2	INTEGER	I
3	REAL	R
4	LONG REAL	LR

The types are ranked according to their associated type number. Thus LONG REAL is higher than INTEGER, INTEGER is lower than REAL.

The meanings of the basic operators are:

+	addition
-	subtraction
*	multiplication
**	exponentiation
/	division
//	integer division

The types of arithmetic expressions are given by the following rules:

1) The operators + and - may be used as unary operators

```
+ operand
- operand
```

The type of the resulting expression is the type of the operand.

2) The type of the arithmetic expression

```
operand1 {+|-} operand2
```

is the higher of the types of operand1 and operand2. This is displayed in the table:

type of operand1	type of operand2			
	SI	I	R	LR
SI	SI	I	R	LR
I	I	I	R	LR
R	R	R	R	LR
LR	LR	LR	LR	LR

3) / denotes division, and

```
operand1 / operand2
```

always results in an arithmetic expression of type REAL or LONG REAL. If either of operand1 or operand2 are of type LONG REAL then so is the resulting expression, otherwise it is of type REAL.

This is displayed in the table:

operand1 / operand2

type of operand1	type of operand2			
	SI	I	R	LR
SI	R	R	R	LR
I	R	R	R	LR
R	R	R	R	LR
LR	LR	LR	LR	LR

A run time error will always result if the value of operand2 is zero.

4) // denotes integer division. The operator is defined only between two operands of type INTEGER and gives the INTEGER result

A // B = SIGN (A*B) * ENTIER (ABS(A/B))

If, in

operand1 // operand2

the types of operand1 and operand2 are not INTEGER, then they are rounded to type INTEGER and then the operation, as defined above, is carried out.

Examples:

```
10 // 5      =  2
 9 // 5      =  1
-9 // 5      = -1
 9 //-5      = -1
```

5) The operation

operand1**operand2

denotes exponentiation, where operand1 is the base and operand2 the exponent.

e.g.	2**3	means	8 (=2*2*2)
	2**(3**4)	means	$2^{(3)^4}$
	2**3**4	means	$(2^3)^4$

The type is real unless one of operand1 or operand2 is of type LONG REAL when it is LONG REAL.

6) When the type of an arithmetic expression can not be determined at compile-time, it is considered to be the highest possible type in the hierarchy of the alternatives.

If in

IF X > 0 THEN P.R ELSE S

the result of an expression is of one arithmetic type and the expression is assigned to a variable of another arithmetic type, the result is transformed to the appropriate type.

Arithmetic operator precedence

The evaluation of an arithmetic-expression can be considered as taking place in a number of scans through the expression, each scan going from left to right:

1) The values of the constants, variables and function designators are evaluated.
2) Any subexpression between a left parenthesis and its matching right parenthesis is evaluated (according to rules 2, 3, 4, 5) and is used as a primary in subsequent calculations.
3) All exponentiations are evaluated.
4) All multiplications, divisions and integer divisions are evaluated.
5) All additions and subtractions are evaluated.

The operator precedences may be summarised by

1. ( )
2. **
3. * / //
4. + -

Arithmetics of REAL and LONG REAL quantities

Quantities of type REAL and LONG REAL are defined to a finite accuracy. Thus any operation performed on such quantities is subject to a possible deviation. The control of the possible consequences should be carried out by a numerical analyst.

6.4 CONDITIONS

Conditions (or BOOLEAN-expressions) return the values TRUE or FALSE. Their prime use is in conditional expressions and conditional statements.

condition

```
{BOOLEAN-primary [BOOLEAN-operator  BOOLEAN-primary]...}
{IF-clause BOOLEAN-primary ELSE condition               }
```

BOOLEAN-primary

```
                  {BOOLEAN-variable          }
                  {BOOLEAN-function-designator}
                  {TRUE                      }
[¬]               {FALSE                     }
                  {relation                  }
                  {(condition)               }
```

BOOLEAN-operator

```
{AND|OR|EQV|IMP}
```

relation

```
{simple-value-expression     {=  }  simple-value-expression     }
{                            {>  }                              }
{                            {>= }                              }
{                            {<  }                              }
{                            {<= }                              }
{                            {=  }                              }
{simple-reference-expression {==}  simple-reference-expression }
{                            {=/=}                              }
{simple-reference-expression {IS}  CLASS-identifier            }
{                            {IN}                               }
```

Note the equivalent modes of writing:

¬	or	NOT
=		EQ
¬=		NE
<		LT
<=		LE
>		GT
>=		GE

Examples:

```
TRUE
SON AND HOLY_GHOST
X = 0 OR ¬ (Y > C)
¬ (X IN HEAD)
T == NOTEXT
FATHER =/= NONE
C = 'A'
```

BOOLEAN OPERATORS

The meanings of the logical operators

NOT	not
AND	and
OR	or
IMP	implication
EQV	equivalent

are given in the table:

P	T	T	F	F	
Q	T	F	T	F	
¬P	F	F	T	T	IF P THEN FALSE ELSE TRUE
P AND Q	T	F	F	F	IF P THEN Q ELSE FALSE
P OR Q	T	T	T	F	IF P THEN TRUE ELSE Q
P IMP Q	T	F	T	T	IF P THEN Q ELSE TRUE
P EQV Q	T	F	F	T	IF P THEN Q ELSE Q

Precedence of operators

The sequence of operations within one condition is from left to right with the following additional rules:

first:	arithmetic expressions (+, -, **, *, /, //) according to section 6.3
second:	relations (<, <=, =, ¬=, >, >=, ==, =/=, IS, IN)
third:	¬
fourth:	AND
fifth:	OR
sixth:	IMP
seventh:	EQV

The use of parentheses is interpreted as in section 6.3.

Relations

1) IS, IN

The operators IS and IN test the class membership of a referenced object:

X IS C — is TRUE only if X is a reference to an object belonging to the class C, otherwise the value is FALSE

X IN C — is TRUE only if X is a reference to an object belonging to either the class C or to a subclass of C, otherwise the value is FALSE.

Both X IN C and X IS C are FALSE if X == NONE.

2) Reference comparators (==, =/=)

These operators compare either two simple text references or two simple object expressions. Two object (text) references X and Y are identical if they refer to the same object (text value instance) or if both are NONE (NOTEXT). In these cases, the relation X == Y is TRUE. Otherwise it is FALSE.

The relation X =/= Y has the value of ¬(X==Y).

If X and Y are two text references, then X =/= Y and X = Y may both be TRUE if X and Y refer to physically distinct character sequences which are equal.

3) Value comparators (=, ¬=, >, >=, <, <=)

The relations take the value TRUE if the corresponding relation is satisfied for the actual values of the expressions involved, otherwise FALSE.

Examples:

5	=	3	is	FALSE
'␣'	=	'+'	is	FALSE
X	<	0	is	FALSE if the current value of X is positive or zero, otherwise it is TRUE

If the values of two arithmetic expressions are compared and they are of different types, then the value lower down in the hierarchy is converted to the type of the higher before comparison is made.

Character values may be compared with respect to the collating sequence. A relation

```
X REL Y
```

where X and Y are character values, and REL is a relational operator has the same truth value as

```
RANK(X) REL RANK(Y)
```

Example:

```
'+' < '='      is TRUE        RANK('+') = 78
                              RANK('=') = 126
```

Two text values are equal if they are both empty or if they are both instances of the same character sequence. Otherwise they are unequal. A text value T ranks lower than a text value U if they are unequal in length and one of the following conditions is fulfilled (otherwise it ranks higher):

1) T is empty
2) U is equal to T followed by one or more characters
3) If the first i-1 characters of T and U are the same, and the ith character of T ranks lower than the ith character of U.

Examples:

```
NOTEXT = ""          TRUE
"0"  <  "9"          TRUE
"ABCDE"="ABCDEF"     FALSE
"+12" =/= "=12"      TRUE
"ABC"  =  "ABC␣"     FALSE
```

6.5 CHARACTER EXPRESSIONS

CHARACTER-expression

```
{simple-character-expression                          }
{IF-clause simple-character-expression                }
{   ELSE character-expression                         }
```

simple-character-expression

```
{CHARACTER-constant                 }
{CHARACTER-variable                 }
{CHARACTER-function-designator      }
{(CHARACTER-expression)             }
```

CHARACTER-constant

```
'{any character from the data character set}'
```

Example:

```
''''
'A'
IF X < 0 THEN '␣' ELSE 'Y'
```

The set of internal characters is ordered according to the collating sequence (Appendix A). The collating sequence defines a one-one mapping between internal characters and integers expressed by the function procedures:

```
INTEGER PROCEDURE RANK (C); CHARACTER C;
```

whose value is in the range 0 through 255 and

```
CHARACTER PROCEDURE CHAR (N); INTEGER N;
```

The parameter value must be in the range 0 through 255 otherwise a run time error results.

Two character subsets are defined by the standard procedures

```
BOOLEAN PROCEDURE DIGIT (C); CHARACTER C;
```

which is TRUE only if C is a digit, and

```
BOOLEAN PROCEDURE LETTER (C); CHARACTER C;
```

which is TRUE only if C is an upper case letter.

Example:

The following program scans an input file on SYSIN until "end of file" is met and records the number of occurrences of the digits 0 - 9.

```
BEGIN  INTEGER ARRAY INCIDENCES (RANK('0') : RANK('9'));
       CHARACTER C; INTEGER R;
       INSPECT SYSIN DO
       WHILE ¬ ENDFILE DO
       BEGIN  C := INCHAR;
              IF DIGIT (C) THEN
                 BEGIN  R := RANK(C);
                        INCIDENCES(R) := INCIDENCES(R) + 1;
                 END;
       END;
       ...
END
```

6.6 DESIGNATIONAL EXPRESSIONS

A designational expression is a rule for obtaining the label of a statement.

designational-expression

```
{simple-designational-expression                    }
{IF-clause simple-designational-expression          }
{   ELSE designational-expression                   }
```

simple-designational-expression

```
{LABEL-identifier                  }
{SWITCH-identifier (subscript)     }
{(designational-expression)        }
```

Examples:

valid
```
LAB
CHOOSE (I)
SELECT (IF X < 0 THEN 2 ELSE 3)
IF B < 0 THEN LAB_1 ELSE SLAB(1)
```

invalid `X.LAB` — remote SWITCH- or LABEL-identifiers are not allowed

The principle of evaluation is analogous to that of arithmetic-expressions. In the general case, the conditions in the IF-clauses will select a simple-designational-expression. If this is a label the desired result is found. A switch designator, on the other hand, refers to the corresponding switch declaration and by use of the actual numerical value of its subscript (an arithmetic-expression) selects one of the designational-expressions listed in the switch declaration by counting from left to right. Since this value itself may be a switch designator, the definition is recursive.

The evaluation of the subscript is analogous to that of subscripted variables.

6.7 OBJECT EXPRESSIONS

An object-expression is a rule for computing a reference to an object or NONE. The type of the expression is REF (qualification).

object-expression

```
{simple-object-expression                    }
{IF-clause simple-object-expression          }
{    ELSE object-expression                  }
```

simple-object-expression

```
{NONE                                                    }
{variable                                                }
{function-designator                                     }
{NEW CLASS-identifier[actual-parameter-part]             }
{THIS CLASS-identifier                                   }
{simple-object-expression QUA CLASS-identifier           }
{(object-expression)                                     }
```

Examples:

```
X
X.Y
SIDE (3)
THIS HEAD.SUC
NEW INFILE("CARDS")
THIS LINK QUA SUB_CLASS.ATTRIB
IF SUC IN LINK THEN SUC ELSE NONE
IF X =/= NONE
   THEN (IF X.F =/= NONE THEN X.F.F ELSE NONE)
   ELSE NONE
```

Qualification

The qualification of an object-expression is defined by the following rules:

1) The expression NONE is qualified by a fictitious class which is inner to all declared classes.

2) A variable, array or function designator is qualified as stated in the declaration (or specification, see below) of the variable or array or procedure in question.

3) An object-generator, local-object or qualified-object is qualified by the class of the identifier following the symbol NEW, THIS or QUA respectively.

4) A conditional object-expression is qualified by the class at the innermost prefix level which includes the qualification of both alternatives. If there is no such class, the expression is illegal.

   In

   ```
   IF B THEN NONE ELSE S
   ```

   the qualification is that of S (by rule 1).

5) A formal parameter of object-reference type is qualified according to its specification regardless of the qualification of the corresponding actual parameter.

6) The qualification of a function-designator whose procedure identifier is that of a virtual quantity, depends on the access level. The qualification is that of the matching declaration, if any, occurring at the highest prefix level equal to or less than the access level, or if none, that of the virtual specification.

Object generators (NEW A....)

An object-generator invokes the generation and execution of an object belonging to the identified class. The object is a new instance of the corresponding (concatenated) class body. The evaluation of an object generator consists of the following actions.

1) The object is generated and the actual parameters of the object generator are evaluated. The parameter values and/or references are transmitted. (For parameter transmission modes, see section 5.4).

2) Control enters the body of the object through its initial BEGIN whereby it becomes operating in the "attached" state.

The evaluation of the object generator is completed:

Case a: Whenever the basic procedure "detach" is executed "on behalf of" the generated object (see PART 3, section 1), or

Case b: upon exit through the final end of the object.

The value of an object-generator is the object generated as the result of its evaluation. The state of the object after the evaluation is either "detached" (case a) or "terminated" (case b).

Local objects

A local-object "THIS C" is a meaningful expression within

1) the class body of C or that of any subclass of C, or
2) a connection block whose qualification is C or a subclass of C.

The value of a local-object in a given context is the object which is, or is connected by, the smallest textually enclosing block instance, in which the local object is a meaningful expression. If there is no such block, the local object is illegal. For an instance of a PROCEDURE- or CLASS-body, "textually enclosing" means containing its declaration.

Instantaneous qualification

Let X represent any simple reference expression, and let C and D be class identifiers such that D is the qualification of X. The qualified object "X QUA C" is then a legal object expression, provided that C includes D or is a subclass of D. Otherwise, i.e. if C and D belong to disjoint prefix sequences, the qualified object is illegal.

The evaluation of X QUA C at run time gives an error if the value of X is NONE, or an object belonging to a class not included in C. Otherwise the value of X QUA C is that of X. (Note the qualification of X is D, whilst that of X QUA C is C).

6.8 TEXT EXPRESSIONS AND TEXT VALUES

A TEXT-expression is a rule for computing a reference to a TEXT object. A TEXT-value is a rule for computing the value of a text (i.e. string of characters).

TEXT-expression

```
{ simple-text-expression                                              }
{ IF-clause simple-text-expression ELSE TEXT-expression               }
```

simple-text-expression

```
{ NOTEXT                  }
{ TEXT-variable           }
{ TEXT-function-designator}
{ (TEXT-expression)       }
```

TEXT-value

```
{ simple-text-expression }
{ TEXT-constant          }
```

Examples:

valid

TEXT-expression

```
NOTEXT
IF T.LENGTH < 5 THEN NOTEXT ELSE BLANKS(80)
(RUNE,T)
T.SUB(1,12)
```

TEXT-value

```
"ANOTHER␣CONSTANT"
NOTEXT
```

invalid

```
IF T.LENGTH < 5 THEN "MAX␣FOUR"
                ELSE "FIVE"
```

TEXT-constants may not be constituents of conditional expressions.

7. BLOCKS AND STATEMENTS

A program is structured into blocks. A block consists of a block head which defines properties of the block, and a compound tail which defines the actions of the block. A block may be prefixed which means that the block is built into a predefined environment.

There is no way for a block environment to interact with an inner block (i.e. examine its current state). The environment can only observe the result of its actions. A block may only be entered through its initial BEGIN and attempts to do so by GOTO statements are thus illegal.

7.1 BLOCKS AND COMPOUND STATEMENTS

program

{block / compound-statement}

block

{prefixed-block / main-block}

prefixed-block

CLASS-identifier [actual-parameter-list]{main-block / compound-statement}

main-block

BEGIN {declaration;}...[statement;]... [statement]END

compound-statement

BEGIN [statement;]...[statement]END

A block automatically introduces a new level of nomenclature. Any identifier may be declared or appear as a label in the block and is then said to be local to it. An entity represented inside a block has no existence outside it and any entity represented by this identifier outside the block cannot be directly accessed by use of it (see the explanation of the binding rule in section 3).

When a block is prefixed, the identifiers declared in the corresponding CLASS are made available. Nevertheless, an identifier in the CLASS may be redefined in the main-block or compound-statement.

The execution of a block is as follows:

step 1: if the block is prefixed then the actual parameters if any are evaluated.

2: if the declarations of the block contain array bounds then these are evaluated. (They may make reference to parameters of the prefix).

3: Execution of the statements body begins with the first statement of the prefix, if any, otherwise with the first statement of the main block. After execution of the block body (unless it is a GOTO statement) a block exit occurs and the statement textually following the entire block is executed.

A CLASS identifier possibly followed by an actual parameter list can prefix a main-block or compound-statement. This results in concatenating the object of the stated class with that main-block or compound-statement, which means that the capabilities of the stated class and its including classes are available within that main-block or compound statement.

When an instance of a prefixed block is generated, the formal parameters of the class are initialised as indicated by the actual parameters of the block prefix. A virtual quantity is identified by the quantity defined by a matching declaration in the block head of the main-block or compound-statement, or by the matching definition at the innermost prefix level of the prefix sequence. The operation rule of the concatenated object is defined by principles similar to those given in section 5.5.

A prefixed-block plays a particular role in the quasi-parallel sequencing of SIMULA: it defines a quasi-parallel system, a system whose components have interactions independent from the program outside the prefixed block, see PART 3.

The following restrictions must be observed when prefixing blocks:

An object in which reference is made to the object itself through use of THIS is an illegal block prefix.

The CLASS-identifier of a prefix must refer to a class local to the smallest block enclosing the prefixed block. If that CLASS-identifier is that of a system class, it refers to a fictitious declaration of that system class occurring in the block head of the smallest enclosing block.

A compound-statement is a means of grouping several statements together to act as one unit, as in

```
FOR I := 1 STEP 1 UNTIL 10 DO
    BEGIN  SUM := SUM + A(I);
           SUMSQ := SUMSQ + A(I)*A(I);
    END
```

where the controlled-statement is a compound-statement,

or of legalising a statement structure

```
IF X > 0 THEN BEGIN  IF Y > 0 THEN QUADRANT := 1;
              END
```

The structure is illegal without the BEGIN - END pair.

7.2 STATEMENTS

A statement is a unit of action in a program. Sequences of statements may be grouped together to form compound statements or blocks.

Special notice must be taken of conditional statements which effectively prevent a certain ambiguity from arising. For consider

```
IF condition1 THEN IF condition2 THEN S2 ELSE S3
```

which has two interpretations

1) IF condition1 THEN IF condition2 THEN S2 ELSE S3

2) IF condition1 THEN IF condition2 THEN S2 ELSE S3

This ambiguity is resolved by not allowing a conditional statement to follow the THEN symbol. Now the two meanings are resolved by writing

```
1) IF condition1 THEN
      BEGIN  IF condition2 THEN S2 ELSE S3  END

2) IF condition1 THEN BEGIN  IF condition2 THEN S2  END
                     ELSE S3
```

The same sort of ambiguity can arise with a WHILE-statement, a FOR-statement, or a connection statement following a THEN.

```
IF condition1 THEN
    WHILE J < 10 DO
       IF condition2 THEN S1 ELSE S2

IF condition3 THEN
   INSPECT X DO
   IF condition4 THEN S3 ELSE S4
```

Here the syntax forces the ELSE-branch to belong to the nearest THEN, but means that whereas we can write

IF condition THEN { WHILE-statement / FOR-statement / connection-statement } ,

we can not write

IF condition THEN { WHILE-statement / FOR-statement / connection statement } ELSE S

In this case, BEGIN - END are necessary round the WHILE-, FOR-, or connection-statement.

statement

{open
closed
IF-statement}

open

[label:]... {FOR-statement
WHILE-statement
connection-statement}

closed

[label:]... {block
compound-statement
activation-statement
PROCEDURE-statement
dummy-statement
GOTO-statement
assignment-statement
object-generator}

label

LABEL-identifier

ASSIGNMENTS

Assignment-statements assign the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may occur only within the body of the function procedure defining the value of that function. The process of assignment takes place in three steps:

1) any subscript expressions in the left variables are evaluated in sequence from left to right
2) the expression is evaluated
3) the value of the expression is assigned to the left part variables from right to left (see example below under Types).

assignment-statement

{value-assignment / reference-assignment}

value-assignment

{variable := / PROCEDURE-identifier := / TEXT-function-designator :=} ... value-expression

Examples:

```
X := 3.45
C := D(I) := '+'
P.X := R.X + S.X
T := "MESSAGE"
T.SUB(1,5) := S := "12345"
Y := IF X > 0 THEN X ELSE -X
```

TEXT-value-assignment

Consider the TEXT-value-assignment

```
R := T
```

let the length of R be Lr and the length of T be Lt. The assignment is legal if Lr >= Lt, otherwise a run time error results.

Lr = Lt — the text value of T is copied into R

Lr > Lt — The text value of T is copied into R left justified and the remaining Lr-Lt positions are filled with blanks

Assignments to overlapping texts are unpredictable.

e.g. with

```
        T.SUB(10,10) := T.SUB(15,10)
    or  T.SUB(15,10) := T.SUB(10,10)
```

The value assignment

```
T := NOTEXT
```

sets the value of T to blanks.

Types

The type of the expression and all the left part variables or PROCEDURE-identifiers must be identical except in the case of arithmetic assignments. In this case, the assignment

```
v1 := v2 := ..... vn := expression
```

is equivalent to

```
vn := expression
vn-1 := vn
  .....
v1 := v2
```

So that if X is REAL and I INTEGER,

```
X := I := 3.57
```

is equivalent to

```
I := 4;
X := 4.0
```

and

```
I := X := 3.57
```

is equivalent to

```
X := 3.57;
I := 4
```

GOTO-STATEMENTS

A GOTO statement interrupts the normal sequence of operations. The value of a designational expression indicates the value of the label of the next statement to be executed.

GOTO-statement

```
GOTO designational-expression
```

Examples:

```
GOTO L
GOTO S(4)
GOTO S(IF N=0 THEN 2 ELSE 6)
GOTO IF N=0 THEN L1 ELSE L2
```

A GOTO statement may not lead into

- a connection statement
- a block which is not already active
- a FOR-statement
- a WHILE-statement

A GOTO statement leaving an attached object leaves the object in the terminated state.

When the value of a designational expression is a switch with an undefined value (the index is out of range), the GOTO statement is equivalent to the dummy statement
e.g.

```
;GOTO S(-1);          is equivalent to
; ;
```

Example:

The program below prints out the first three verses of "The Twelve Days of Christmas". The program logic is built around a SWITCH.

```
BEGIN  SWITCH CASE := LINE1, LINE2, LINE3;
       INTEGER VERSE;
       TEXT ARRAY T(1:3);
       T(1) :- COPY("FIRST");
       T(2) :- COPY("SECOND");
       T(3) :- COPY("THIRD");

IF_WE_USE_FOR_HERE_THE_LABELS_BECOME_INVISIBLE:
       VERSE := VERSE + 1;
       OUTTEXT("ON␣THE␣"); OUTTEXT(T(VERSE));
       OUTTEXT("␣DAY␣OF␣CHRISTMAS"); OUTIMAGE;
       OUTTEXT("MY␣TRUE␣LOVE␣SENT␣TO␣ME"); OUTIMAGE;
       GOTO CASE(VERSE);
LINE3: OUTTEXT("THREE␣FRENCH␣HENS");
       OUTTEXT("TWO␣TURTLE␣DOVES,␣AND");
       OUTTEXT("A␣PARTRIDGE␣IN␣A␣PEAR␣TREE");
       EJECT(LINE + 2);
       IF VERSE < 3 THEN
          GOTO IF_WE_USE_FOR_HERE_THE_LABELS_BECOME_INVISIBLE;
END
```

OUTPUT:

```
ON THE FIRST DAY OF CHRISTMAS
MY TRUE LOVE SENT TO ME
A PARTRIDGE IN A PEAR TREE

ON THE SECOND DAY OF CHRISTMAS
MY TRUE LOVE SENT TO ME
TWO TURTLE DOVES, AND
A PARTRIDGE IN A PEAR TREE

ON THE THIRD DAY OF CHRISTMAS
MY TRUE LOVE SENT TO ME
THREE FRENCH HENS
TWO TURTLE DOVES, AND
A PARTRIDGE IN A PEAR TREE
```

DUMMY STATEMENTS

A dummy-statement executes no actions. Its main use is to place a label before an END

dummy statement

Examples:

```
IF X > 0 THEN ELSE X := -X;

BEGIN ..... LAB : END;
```

WHILE STATEMENTS

The WHILE-statement is used when a statement is to be executed while a condition is TRUE.

WHILE-statement

WHILE condition DO statement

Examples:

```
WHILE ¬ LASTITEM DO
BEGIN  X := ININT;
       COUNT := COUNT + 1;
       SUM := SUM + X;
END;

WHILE X =/= NONE DO
BEGIN  IF X IN POINT THEN NOP := NOP + 1 ELSE
          IF X IN LINE THEN NOL := NOL + 1 ELSE
             ERROR("FALSE␣ENTRY");
       X :- X.SUC;
END;
```

ACTIVATION STATEMENTS

Activation statements are only valid inside a SIMULATION block. They are fully described in part 3, section 3.

FOR STATEMENTS

FOR-statements are control statements which cause controlled-statements to be executed zero or more times.

There are three basic types of FOR statement element.

1) the controlled-statement is executed for a list of values (usually irregular).

```
FOR I := 2, 13, 17 DO controlled-statement
```

2) the controlled-statement is executed a known number of times

```
FOR I := 1 STEP 1 UNTIL 10 DO
    X(I) := I*I
```

3) the controlled-statement is executed until a condition is met

```
FOR X :- X.SUC WHILE X =/= NONE DO
    X QUA LINK.OUT
```

To increase the generality of the concept, these elements themselves are allowed to form a list

```
FOR I := 1, 2, 4 STEP 1 UNTIL 10, I*I WHILE I < 200 DO
  controlled statement
```

FOR-statement

[label:]... FOR controlled-variable for-right-part DO
controlled statement

for-right-part

{:= value-element [,value-element]
:- object-element [,object-element]}

value-element

{value-expression
arithmetic-expression STEP arithmetic-expression
UNTIL arithmetic-expression
value-expression WHILE condition}

object-element

{object-expression
object-expression WHILE condition}

Each execution of the controlled statement is preceded by an assignment to the controlled-variable. Assignments may change the value of this controlled variable during execution of the controlled-statement.

for list elements

The for list elements are considered in the order in which they are written. When one for list element is exhausted, control proceeds to the next, until the last for list element in the list has been exhausted. Execution then continues after the controlled statement.

The effect of each type of for list element is defined below using the following notation:

C: controlled variable
V: value expression
O: object expression
A: arithmetic expression
B: Boolean expression
S: controlled statement

The effect of the occurrence of expressions in for list elements may be established by textual replacement in the definitions.

α, β, σ are different identifiers which are not used elsewhere in the program. σ identifies a non-local simple variable of the same type as A_2.

1. V

```
    C := V;
    S;
    next for list element
```

2. A_1 step A_2 until A_3

```
    C := A1;
    σ := A2;
α : if  σ*(C-A3) > 0 then goto β;
    S;
   σ := A2;
    C := C + σ;
    goto α;
β : next for list element
```

3. V while B

```
α : C := V;
    if ¬ B then goto β;
    S;
    goto α;
β : next for list element
```

4. 0

```
    C :- 0;
    S;
    next for list element
```

5. 0 while B

```
α : C :- 0;
    if ¬ B then goto β;
    S;
    goto α;
β : next for list element
```

The controlled variable

The controlled-variable is a simple-variable which is not a formal-parameter called by name, a PROCEDURE-identifier, a remote-identifier nor a TEXT-identifier.

To be valid, all for list elements in a for statement must be semantically and syntactically valid.

In particular each implied assignment is subject to the rules of section 7.2.1.3.

The value of the controlled variable upon exit

Upon exit from the for statement, the controlled variable will have the value given to it by the last (explicit or implicit) assignment operation.

Labels local to the controlled statement

The controlled statement always acts as if it were a block. Hence, labels on or defined within the controlled statement may not be accessed from without the controlled statement.

PROCEDURE STATEMENTS

A procedure-statement calls for the execution of the PROCEDURE-body. Before execution, the formal-parameters of the procedure are replaced by the actual-parameters.

PROCEDURE-statement

[simple-object-expression.] PROCEDURE-identifier [(expression
[, expression]...)]

Examples:

```
INTO(H)
OUTTEXT("***0***")
SYSIN.INIMAGE
```

The procedure statement must have the same number of actual parameters in the same order as the formal-parameters of the procedure heading.

Restrictions

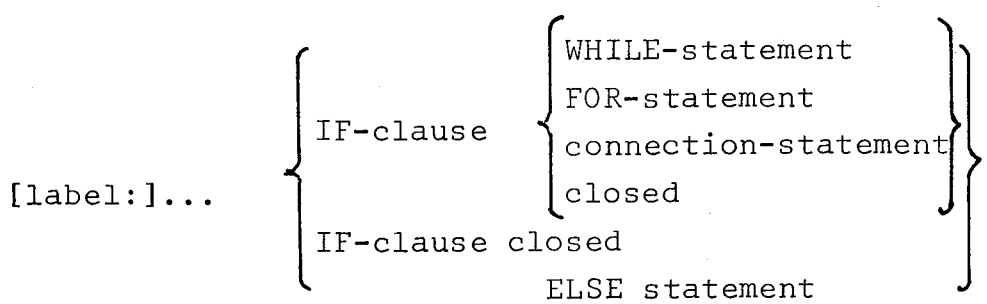
1) An actual-parameter corresponding to a formal-parameter called by NAME which is assigned to within the PROCEDURE-body must be a variable.

2) If the formal-parameter is an ARRAY (PROCEDURE), then the number of dimensions (actual-parameters) used within the PROCEDURE-body must correspond to the number of dimensions (actual-parameters) of the actual ARRAY (PROCEDURE).

CONDITIONAL STATEMENTS

Conditional statements cause certain statements to be executed or skipped depending on the current values of certain conditions. They provide an important structural framework in the language.

IF-statement



Examples:

```
L:  IF X > 0 THEN X := -X;

    IF X = 0 AND Y = 0 THEN
L:      BEGIN  OUTTEXT("ORIGIN");
               OUTIMAGE;
        END;

        X :- T;
        WHILE X =/= NONE DO
        BEGIN  IF X QUA PERSON.MALE THEN MAN := MAN + 1
                                    ELSE WOMAN := WOMAN + 1;
               X :- X.SUC;
        END;
```

The statement sequence

```
IF condition THEN S1; S2
```

is equivalent to

```
(evaluate condition) S1; S2     if the condition is TRUE
(evaluate condition) S2         if the condition is FALSE.
```

The statement sequence

```
IF condition THEN S1 ELSE S2; T
```

is equivalent to

```
(evaluate condition) S1; T      if the condition is TRUE, and
(evaluate condition) S2; T      if the condition is FALSE.
```

A GOTO statement may lead directly into a conditional statement e.g

```
IF B THEN  BEGIN  OUTTEXT("TRUE");
           L:     OUTIMAGE;
           END
     ELSE  BEGIN  OUTTEXT("FALSE");
                  GOTO L;
           END
```

CONNECTION STATEMENTS

Connection-statements are a form of remote accessing which are mainly used as a user convenience if there are to be many accesses to a particular object.

e.g.

we may replace

```
SYSBIN.INIMAGE;
X := SYSBIN.INREAL;
Y := SYSBIN.INREAL;
C := SYSBIN.INCHAR;
```

by

```
INSPECT SYSBIN DO
BEGIN  INIMAGE;
       X := INREAL;
       Y := INREAL;
       C := INCHAR;
END
```

Connection statement

```
INSPECT object-reference DO { statement
                            { [WHEN CLASS-identifier DO statement]...
                            {                    [OTHERWISE statement]
```

Examples:

```
INSPECT SYSOUT DO
BEGIN  OUTTEXT("TITLE");
       OUTIMAGE;
       EJECT(LINE+10);
END;

INSPECT X WHEN A DO OUTTEXT("X IN A")
          WHEN B DO OUTTEXT("X IN B")
          OTHERWISE OUTTEXT("ERROR");

INSPECT X DO
  INSPECT Y DO P := Q
            OTHERWISE OUTTEXT("Y==NONE");
```

To avoid ambiguity, an OTHERWISE refers back to the nearest INSPECT.

The remote accessing of objects of classes may be accomplished by the dot notation or by connection. In most cases the methods are interchangeable if the object contains a class attribute at a certain level, then attributes at that level and levels inner to it can only be accessed by connection.

To access an attribute of an object we have to know:

1) a reference to the object
2) the qualification of the reference
3) the attribute identifier.

The two variations are now explained:

1) a) INSPECT X DO S1 OTHERWISE S2
   b) INSPECT X DO S1

   If X =/= NONE then the statement S1 is executed. During execution of this statement the reference variable of X is evaluated once and stored. Access is now gained to the denoted object at the qualification level of X. All attributes of the qualifying class are now available by the occurrence of their identifiers.

   If X == NONE, then the statement S1 is skipped and the statement S2 is executed in case a), and the whole connection statement is skipped in case b).

   In an otherwise branch, no connection holds.

   e.g.

```
            CLASS A;
            BEGIN  REAL X; ..... END;
          A CLASS B
            BEGIN  BOOLEAN Y; .. END;
            REF(A)Q; REF(B)Z;

INSPECT Q DO                    INSPECT Q QUA B DO
BEGIN  X := 2;                  BEGIN  X := 2;
       THIS B.Y := FALSE;              Y := FALSE;
       Z :- NONE;                      Z :- NONE;
END                             END
```

2) INSPECT X WHEN A DO S1
WHEN B DO S2
.....
OTHERWISE T

This discriminates on class membership at run time. The when clauses are considered in turn. If X is an object belonging to a class equal to or inner to the one identified in the clause then the associated statement is executed and the rest of the connection-statement is skipped. The OTHERWISE-clause (if present) is only executed of X == NONE or all preceding WHEN clauses are skipped.

OBJECT GENERATORS

Object-generators were discussed in section 6.5. They may also stand as statements in their own right, in which case their reference value is not assigned on exit. This does not mean that they cannot be referenced as the following example shows:

```
LINK CLASS B(H); REF(HEAD)H;
BEGIN  PTR :- THIS B;
       INTO(H);
END
```

After execution of the statement

```
NEW B(HD);
```

then the generated object is referenced by PTR and HD.LAST (if H =/= NONE).