CC-200

COMPUTATION CENTER
Massachusetts Institute of Technology
Cambridge 39, Massachusetts

September 18, 1962

TO: All Fortran Users

FROM: Jessica D. Hellwig

TITLE: Subroutines in Fortran

Introduction

This memorandum discusses at some length the five Fortran subroutine types. Included are descriptions of the essential differences between them, their purpose, definition, and proper usage. Special problems associated with the use of subprograms are discussed in detail.

This is essentially a coordination and expansion of material gathered from the Fortran manuals published by IBM.

General Description

A subroutine may be considered any pre-defined sequence of instructions which performs a specific operation and which may be executed repeatedly, at various points within a program.

Fortran provides two kinds of subroutines. One kind is the set of functions, of which there are four types. The other kind are known as Subroutine subprograms. The functions bear a close resemblance in many ways to "functions" in the mathematical sense; Subroutine subprograms may also represent mathematical functions but they lack certain resemblances and play in fact a much more general role.

In mathematics, a function is a quantity whose value depends upon the values of variables (arguments of the function) in a fixed relationship. The function

$$f(x) = Ax^2 + Bx \qquad (1)$$

implies a series of operations to be carried out on x, regardless of the value of x; however f(x) itself is a quantity, having a different value for each value of x. Thus one may write

$$g(x,y) = \frac{f(x) - f(y)}{x - y} \quad , \qquad (2)$$

illustrating the status of f(x) as quantity, as well as the interchangeability of arguments. Lastly one might define a variable z, such that

$$z = g(x,y)\ f(x) \quad . \qquad (3)$$

In Fortran, the various functions have much the same status. Equation (1) is in effect the definition of the function; this could be translated in Fortran (as one type of function) as follows:

```
FIRSTF(X) = A*X**2 + B*X
```

Equation (2) illustrates the use of the function, or its appearance as a variable, in an arithmetic statement (which is incidentally another function definition). In Fortran this might be written

```
SECØNDF(X,Y) = (FIRSTF(X) - FIRSTF(Y))/ (X-Y)
```

while equation (3) would become

```
Z = SECØNDF(X,Y)* FIRSTF(X)
```

It becomes clear then that a function in Fortran has the following properties:

1) Its definition is distinct from its use, or call. It is called within an arithmetic statement where the function value is an operand in the statement.

2) Hence, the name of a function is the name of its value. During execution the instruction sequence that comprises

the subroutine is carried out, and the result replaces the function name in the arithmetic statement.

3) The arguments given in the definition of a function are replaced, when the function is referenced in an arithmetic statement, by the arguments required at that time. Hence the arguments in the definition are in effect dummy arguments.

The only major restriction on Fortran functions which is not generally applied to mathmatical functions is that Fortran functions are always single-valued.

In Fortran, functions, then, are subroutines which represent single-valued functions of single or multiple arguments, called within an arithmetic statement, where the name of the subroutine is the name of the function value.

A Subroutine subprogram is by nature a more general type of subroutine. It may, in fact, be just like a function in its definition, in the status of its arguments, and in its mathematical significance. It may be quite different, however, in all these respects, and is <u>always</u> different in the following:

1) A Subroutine subprogram is called by a separate special statement which initiates the execution of the instruction sequence which it comprises.

2) Hence the <u>name</u> of a Subroutine subprogram corresponds not to a value but to the instruction sequence itself.

In addition, a Subroutine subprogram may produce multiple outputs. Its "arguments" may be not only arguments in the strict mathematical sense but may be input of any sort from the calling program to the Subroutine subprogram; in particular, the argument list may include names to be assigned values computed by the sub-

routine.

Functions - Definition

There are four types of functions in Fortran. The first two are used, but not in general defined, by the Fortran programmer. The second two are used and usually defined by the programmer.

1. Built-in Functions. These exist in the Fortran processor. They differ from all other subprogram types in being open; that is, the instruction sequence is generated by Fortran and incorporated into the program, during compilation, at each occurrence in the source program of the function name.

2. Library Functions. These exist in a program library, either as a card deck or on a library tape. They are originally coded in machine language according to definite conventions.

3. Arithmetic Statement Functions. These are defined by an arithmetic statement in the calling program. Following their definition they are used, or called, by other arithmetic statements in the same program.

4. Function Subprograms (also called Fortran Functions). These are independent subprograms defined by separate compilation. The first statement in the definition program has the form

FUNCTIØN name(arg_1, arg_2, ... arg_n)

which identifies the subprogram as being of function form and implies the identification of the value of the function with the specified name.

Subroutine Subprograms --Definition

These are all of one type. They are independent subprograms defined by separate compilation. The first statement of the

definition program has the form

SUBRØUTINE name(arg_1, arg_2, ... arg_n)

which identifies the subprogram as being of Subroutine subprogram form and may imply the identification of one or more of the argument names with the value(s) computed by the subroutine. The execution of the subroutine is effected by a statement in the calling program of the form

CALL name(arg_1, arg_2, ... arg_n)

These five types of subroutines may be differentiated on the basis of their form (restrictive or not) and also on the basis of the generality of their application (i.e. distinguishing functions frequently used by most programs from special-purpose routines). One may expect that those most commonly used should be most readily available, or simplest to use.

1. The built-in functions are a special case, being open routines. In fact, they look like subroutines from the vantage point of the source program but not from that of the object program. Since the generated coding is reproduced in the object program each time the function is referenced, such routines must be limited to functions which are simple in terms of machine-language coding. There is a decrease in execution time in the object program, though at the expense of memory space. Since these routines are implicit in the Fortran Processor (occupying valuable space and time in it) they must be restricted to functions which are of the most general utility and which become an extension of the Fortran language itself, rather than being germane to its applications. For such basic functions the open type combines the simplicity of subroutine usage in the source program and the

efficiency of machine-language straight line coding in the object program.

2. Library functions, as the name implies, are of sufficiently general utility to be permanently maintained by the computer installation. The most commonly used of these are maintained on a special tape which is automatically available to every program running in the system. Others are maintained as binary decks also readily available to the user. Functions made available in this form are likely to include those most often needed by the user of a particular installation -- their generality, in other words, is peculiar to the principal applications of one computer.

Library functions, since they are pre-written and used frequently, are coded in machine language mainly in the interests of economy. They are subprograms with a unique convention for linkage which demands machine-language coding.

3. Arithmetic statement functions are the first and simplest of the subroutines written by the user. They are restricted to those functions which may be expressed in a single Fortran arithmetic statement. They satisfy, however, the basic requirements of a subroutine -- i.e., repeated availability without redefinition, and interchangeability of arguments. The limitations on their definition are counterbalanced by a distinct advantage over subprograms: since they are internal to the calling program, all the variables of the calling program are available to the subroutine as arguments and variables, without the problems of transmission introduced by the use of subprograms.

4. Function subprograms are used for the computation of functions which may not be defined by a single arithmetic

statement. They may, therefore, be mathematically as complicated as necessary. (We may note here that at one end of the scale are subroutine types which are highly restrictive but which are used for functions of the most general nature; on the other end are subroutine types of the most generalized form, used for more special applications.)

Function subprograms are principally used for mathematical operations. This is implicit in the major restriction placed on them, that they compute a single quantity which takes its place in the arithmetic statement which referenced the subroutine. It is possible to extend the application of these programs so that, for example, they may compute many other quantities as well, or may perform additional non-arithmetic operations. Their most usual application, however, and the one for which they are intended, is the computation of a single-valued arithmetic function.

5. Subroutine subprograms, finally, are the most general form of subroutines available. They can be used to do anything a Function subprogram does, but are even less restrictive. Probably their most fitting applications are for the computation of multi-valued functions (although this can also be done indirectly by means of a Function subprogram) and especially for the execution of non-arithmetic operations (e.g., input and output).

Functions - Naming and Calling

1. Built-in, Library, and Arithmetic Statement functions are all named and called in the same manner.

The name consists of 4-7* alphanumeric characters with

---

* Double precision and complex Library and Built-in functions: 4-6 characters.

these restrictions: the last character must be F; the first character must be alphabetic, and is X if and only if the value of the function is of fixed point mode.

N.B. The mode of a function and of its arguments is determined by the function definition. Hence a change in the mode configuration would require a new definition and a new name.*

Examples:

Built-in

```
I = (M+N/2)*XMØDF(M,N)
X = DELTAX + MAX1F(YA,YB,YC,10.0)
```

Library

```
B = A + CØSF(A)
M = XDETRMF(NN,N,AMATR,D)
```

Arithmetic Statement

```
(Definition)   TESTF(ARG1,ARG2) = (A/ARG2)*(ARG1-ARG2)
               ...
               ...
(Use)          A = 3.0*DELTA
               GAMMA = ALFA*TESTF(ALFA,BETA)
               ...
```

2. Function Subprograms

The name consists of 1-6 alphanumeric characters with these restrictions: the last character must not be F unless the entire name is less than 4 characters; the first character must be alphabetic and must govern the mode of the function according to the rules for naming variables. This name is used to call the function in the same manner as the other functions. No F is

---

* There are no fixed-point double-precision or complex function names. The names of double-precision and complex Library and Built-in functions are listed with a prefix D or I. This prefix must not be used in function references.

attached to the function name when it is used in a statement.

Examples:

```
(Definition)    FUNCTIØN      FUNK(A,B,C)
                ...
                ...
                FUNK = P+Q*B*C
                RETURN
                ...
                END
(Use)           A = FUNK(DELTA,X,25.) + ALPHA
```

Subroutine Subprograms - Naming and Calling

The name consists of 1-6 alphanumeric characters with these restrictions: the first character is alphabetic; the last character must not be F unless the entire name is less than 4 characters. Since the name of the subroutine is not the name of a value, the name does not affect the mode of any of its values.

The names of values computed by the subroutine may be designated in the argument list. These govern the mode of the values they represent. There must be correspondence in the mode of each argument in the subroutine definition with the respective arguments in every call for the subroutine.

Examples:

```
(Definition)    SUBRØUTINE      LØVE(ABLE,BAKER,NAN)
                ...
                ...
                NAN = ...
                RETURN
```

```
                END
(Use)           X = X + DELX
                CALL LØVE(X,Y,M)
                Y = Y **M
```

Double-Precision and Complex Functions and Subroutines

An arithmetic statement referring to a double-precision or complex function is prefixed with a D or I in column 1. A D or I prefixing a CALL statement signifies that all floating point arguments in the statement are double-precision or complex, respectively.

A double-precision or complex Arithmetic Statement function definition must of course have D or I in column 1.

Example:

```
D     AREAF(R) = SQRTF (PI*R**2)
```

Use of Function and Subroutine Names as Arguments

The argument list of a Function subprogram or a Subroutine subprogram may include the names of Library functions, other Function subprograms, or other Subroutine subprograms (but not Built-in or Arithmetic Statement functions). The following rules apply:

1. The name of a Function subprogram, Subroutine subprogram, or Library function used in an argument list must appear also in a pseudo-statement, in the calling program, of the form

```
F     NAME1, NAME2, ....NAMEn          (F in column 1)
```

where in the case of a Library function the terminal F has been dropped.

2. In the case of a Library function the terminal F is dropped also in the argument list.

3. As with any other argument, a dummy function or subroutine name appears in the argument list of the subroutine definition.

4. In the case of a double-precision or complex Built-in or Library function, the name appearing on the F card is prefixed with a D or I.

Examples:

```
          SUBRØUTINE     PRIME(DUMMY,X,A)
          ...
          ...
          A = DUMMYF(X)
          ...
```

This would allow the selection of any library function when calling PRIME, as

```
          CALL PRIME(SIN,ALPHA,Q)
          ...
          CALL PRIME(CØS,ALPHA,P)
          ...
     F    SIN,CØS
```

Again,

```
          FUNCTIØN  PETER(DUMMY,X,A)
          ...
          ...
          PETER = DUMMY(Y)
          ...
```

allows the calling program to specify various Function subprograms when calling PETER, as

```
      X = Y + PETER(FIRST, ARG, A1)
      A = PETER(SECØND, ARG, A2)
      ...
F     FIRST, SECØND
```

Finally,

```
      SUBRØUTINE    PAUL(DUMMY, X, A)
      ...
      CALL DUMMY(X)
      ...
```

allows the calling program to specify subroutine subprograms when calling PAUL, as

```
      CALL PAUL(SUBR, EX, EA)
      ...
F     SUBR
```

Note that the definitions of Arithmetic Statement functions, as well as of Function subprograms and Subroutine subprograms, may involve other functions. An Arithmetic Statement function definition may involve other Arithmetic Statement functions which have been previously defined in the same program.

Rules for Dimensioning Subprogram Arguments

The arguments in a SUBRØUTINE or FUNCTIØN statement (the dummy arguments of the subprogram definition) may not be subscripted. An array may be an argument; the array name (non-subscripted) appearing in a CALL statement or function subprogram reference must of course be dimensioned in the calling subprogram; the corresponding dummy argument of the subprogram definition must be dimensioned in the subprogram and the

dimensions must correspond in both calling and called programs.

In fact, the dimensions of the dummy array in a subprogram need not be exactly the same as those of the corresponding array in the calling program. Variation is permitted as follows:

The dimension of a one-dimensional array may vary;

The second dimension of a two-dimensional array may vary;

The third dimension of a three-dimensional array may vary;

in other words, for an n-dimensional array, the first (n-1) dimensions must correspond exactly between subprograms.

A single element of an array may be used in the argument list of a calling statement. It is of course subscripted, and corresponds to a single non-subscripted dummy variable, as:

Calling:	Called:
CALL SUB(A,B(I,J))	SUBRØUTINE SUB(X,Y)

On the other hand,

Calling:	Called:
DIMENSIØN B(15,10)	SUBRØUTINE SUB(X,Y)
...	
CALL SUB(A,B)	DIMENSIØN Y(15,10)

transmits the whole array B.

An exception to the above occurs in the case of double-precision or complex variables. Given a double-precision or complex array (say ALPHA, dimensioned (n,n)) where it is desired to use a single element ALPHA(I,J) as an argument to a subprogram, then the corresponding dummy argument in the subprogram must be given like dimensions (n,n) in a double-precision or complex

DIMENSIØN statement. Otherwise ALPHA(I,J) must be set equal to a single variable for use in the argument list. For example, the following illustrates both cases:

Calling:		Called:	
			SUBRØUTINE NAME(X,Y)
D	DIMENSIØN A(10),B(5,5)	D	DIMENSIØN Y(5,5)
D	C = A(5)	D	Y = X + (2., 1.)
D	CALL NAME (C, B(1,3))		RETURN

Dimensioning Y in the subroutine enables the FORTRAN processor properly to compute the address of the imaginary or least significant element of the pair represented by B(1,3). Reference is then made in the subprogram to the variable Y, which is in fact the first element, i.e. Y(1,1), of an implicit array. Use of subscripts (other than (1,1)) with Y should be employed only with great care and a thorough understanding of the ordering of arrays in storage.

Use of CØMMØN For Subroutine Linkage

Variables may be transmitted to a function or subroutine subprogram by the use of CØMMØN storage. Consider the arithmetic statement function

FIRSTF (X,Y) = (A*X+B)/Y .

The arguments X and Y are dummy arguments and will be replaced when the function is called; A and B, however, are variables which always remain the same (though not necessarily of the same value). This sort of arrangement is possible because the arithmetic statement function is, though closed, contained within the calling program.

In the case of a function or subroutine subprogram, however, where the subroutine is an independent program and all the programs are relocatable, variables like A,B which we may call "function constants", must be transmitted from the calling to the called subprogram. One could write

```
CALL   SUBR (X,Y,A,B)
CALL   SUBR (P,Q,A,B)
     etc.,
```

but it is also possible to use CØMMØN storage for the purpose.

The statement

```
CØMMØN   A,B,...
```

assigns the given variables to specific nonrelocatable locations in upper core. The first variable given is assigned to 77461_8, the second to 77460_8, and the rest in order proceeding downward. Successive CØMMØN statements within a program will continue to store variables below those already encountered. Each subprogram will again assign the variables listed in CØMMØN statements to upper core starting at 77461_8. Thus one may write:

Calling:

```
DIMENSIØN   T(5,5)
CØMMØN A,B,T,C
...
CALL   SUBR(X)
```

Called:

```
SUBRØUTINE   SUBR(D)
DIMENSIØN   DUMMY(5,5)
CØMMØN   P,Q,DUMMY,R
```

Because of the correspondence in the order of variables in the two CØMMØN statements, the pairs A and P, B and Q, and T and DUMMY will be identical.

For most purposes, the simplest and safest way to guarantee consistency among subprograms is to make CØMMØN

statements, and DIMENSIØN statements for those variables which appear in CØMMØN statements, identical in all subprograms and the main program. This may easily be done and there is of course no need, generally, to change variable names between subprograms. (The use in those examples of different names for arguments in subprogram definitions is primarily to stress their status as dummy variables.) As will be seen from what follows, departures from this procedure should be made with caution.

In the example above, should it be desired to call another subprogram which involves the variable C but not the others, it would be necessary to include in a CØMMØN statement in the subprogram, dummy variables of proper dimension to assign CØMMØN storage properly, thus:

```
      SUBRØUTINE   NAME(ARG)
      DIMENSIØN    DUMMY(5,5)                    (1)
      CØMMØN       P,Q,DUMMY,R
```

This is necessary to force correspondence between all CØMMØN assignments.

Additional care must be taken with double-precision or complex variables assigned to CØMMØN storage, where several subprograms refer to CØMMØN storage. Consider the example of a main program where A and B are complex variables and T is a complex array:

```
I     DIMENSIØN   T(2,4)
      CØMMØN      A,B,T
      ...
I     X = A**2+B**2
I     CALL   ABLE (X)
      ...
```

Since complex variables consist of two parts, they are stored like arrays. In this example CØMMØN storage would look like this:

```
77461   :   A (real part)
   60   :   A (imaginary part)
   57   :   B (real part)
   56   :   B (imaginary part)
   55   :   T(1,1)   (real part)
   54   :   T(2,1)     "    "
   53   :   T(1,2)
     . . . .
77445   :   T(1,1)   (imaginary part)
        etc.
```

The appearance of A and B in complex arithmetic statements, and of T in a complex dimension statement, guarantees that double storage will be assigned. Suppose now that the subprogram ABLE is to operate on the array T, but does not involve A or B. If we write

```
      SUBRØUTINE   ABLE(D)
  I   DIMENSIØN    DUMMY(2,4)
      CØMMØN   P,Q,DUMMY
      ...
```

then P and Q are for the purpose of "spacing down" to the location of the array in upper core. Since we have assumed, however, that P and Q never appear again in ABLE, there is within this independent subprogram no indication that P and Q are complex variables. Hence CØMMØN storage will look like this:

```
77461   :   P
77460   :   Q
```

```
77457   :   DUMMY (1,1)  (Real part)
77456   :   DUMMY (2,1)      "    "
        etc.
```

and the desired correspondence between the arrays T and DUMMY is not accomplished.

The simplest way to circumvent this problem is to include all complex (double-precision) variables assigned to CØMMØN in a complex (double-precision) DIMENSIØN statement. Thus in the example above, if we write

```
      SUBRØUTINE   ABLE(D)
  I   DIMENSIØN    DUMMY(2,4), P(1), Q(1)
      CØMMØN       P,Q, DUMMY
```

The dimensions of P and Q are not really changed but their status as complex variables is established, and each will be assigned two locations.

Clearly the use of CØMMØN assignments may be applicable not only to what we have called function constants, but to arguments as well. Arguments which are transmitted to a subprogram by being placed in CØMMØN storage are <u>implicit</u> arguments, as opposed to those <u>explicitly</u> mentioned in the subprogram argument list. Again, it is only necessary to create correspondence in upper core between each argument and its dummy correspondent in the subprogram.

It must be noted that a Function subprogram must have at least one argument. Thus implicit arguments may be used for all arguments of a Subroutine subprogram, but at least one explicit argument (a dummy, if so desired) must be given for a Function subprogram.

Relation Between CØMMØN and EQUIVALENCE Statements

The normal order of CØMMØN storage assignment, described above, is changed in the case where variables appear in both CØMMØN and EQUIVALENCE statements. Any CØMMØN variables appearing in EQUIVALENCE statements will be assigned upper core values above those not in EQUIVALENCE statements regardless of the order of the CØMMØN statement. For example,

```
CØMMØN  A,B,C,D
EQUIVALENCE  (C,G), (E,B)
```

will cause the following assignments:

77461_8	C and G
60	B and E
57	A
56	D

Note that the variables are assigned in the order in which they appear in the EQUIVALENCE, and then in the order of the CØMMØN statement. (Note also that a variable made equivalent to a CØMMØN variable is assigned to CØMMØN storage.)

Consequently, if equivalences are established involving variables in CØMMØN storage, similar equivalences should be set up in all subprograms which refer to CØMMØN in order to force proper correspondence between subprograms.

In the case of double-precision or complex variables, a D or I prefixing a CØMMØN or EQUIVALENCE statement is ignored, but the double nature of the variables is preserved when they are so defined in other statements within the program. Generally, single-precision variables should not be made equivalent to double-precision or complex variables, nor should they be given

the same location in CØMMØN storage.

Transmission of Variable Subscripts Between Subprograms

A variable subscript has a somewhat ambiguous status because of the way FORTRAN performs indexing. If a DØ loop (or an input list) has an index I which appears only as index and subscript, there is no storage location set aside for I: the indexing and the subscript reference are taken care of in another way. If I also appears as a variable, either inside or outside of the loop, then there is a storage location assigned to I, and this location is referred to when necessary. As a result, the storage location "I" may sometimes have a value different from the indexing value of I. For example:

```
      I = N/2
      A(I) = X
      DØ  1    I = 1,N
      B(I) = X*A(I)
1     CØNTINUE
```

When the loop is completed, i.e. when the index I is equal to N+1, the storage location "I" will still contain the value N/2, and any references to I as a variable or as a subscript will use this value until a new value is assigned or a new loop is initiated. Suppose we continue the above program, following statement 1 by:

```
      NU = I**2      (equivalent to NU = (N/2)**2)
      DØ  2    I = 1,N
      ...
2     L = 2*I-1      (equivalent to L = 1,3,... 2*N-1)
```

Following statement 2, the storage location of I (i.e., I in the

role of variable) will be equal to N+1, the current value of the index I. This is because I is used as a variable within the loop.

Because of the ambiguous nature of subscripts, care must be exercised in transmitting them between subprograms. (N.B.: A variable subscript which is not being used as an index is called, in the FORTRAN manual, a "relative constant". For further details about variable subscripts, see Fortran Reference Manual, pp 82-85.)

1. In an Argument List

Transmission of variable subscripts via an argument list is straightforward.

```
      I = N/2                          SUBRØUTINE   SUBR(A,N,Y)
      CALL   SUBR (X,I,B)              DIMENSIØN    Q(10)
                                       Y = Q(N)
```

The value of I will be transmitted properly. Other examples of correct linkage are:

```
      DØ  1    I = 1,10
      CALL   SUBR(X,I,B)
    1 ...
```

Where although I is not explicitly used as a variable its appearance in the argument list allows for proper transmission, and

```
      CALL   SUPER (I,Y)               SUBRØUTINE   SUPER (K,A)
                                       K = XFIXF(A**2-X**2)
                                       RETURN
```

where I will be properly transmitted from the calling to the called program and back again. Note however that if the CALL statement in this example were within the range of a DØ with index I, the change in I effected by SUPER would not hold, and

I would be reset, upon return, to the proper indexing value.

It should be observed that in this last example, the argument K is used both as an input and an output parameter. This is in general a dangerous procedure; any change in the value of input variables may cause difficulties, because it requires considerable understanding of FORTRAN operation to be sure of exactly how this change will be effected.

2. Transmission of Variable Subscripts via CØMMØN Statements

Consider the following example:

```
CØMMØN   K          (1)      SUBRØUTINE   ABC(X)        (6)
K = K + 1           (2)
A(K) = B(K)**2      (3)      CØMMØN   N                 (7)
CALL   ABC (A(K))   (4)      ...
TERM = B(K)         (5)      N = N+2                    (8)
```

When ABC is called, K has its proper value in CØMMØN storage and thus N has the same value. Statement (8) changes the value of K in CØMMØN storage. At execution of statement (5), however, the value of K in CØMMØN is not used to find the appropriate term in the array B; because of the way FORTRAN is set up the subscript K is determined by a different storage method, and on return from a subprogram which recalculates the variable K, the subscript K will not have been changed.

The statement

```
K = K                (4a),
```

inserted between (4) and (5) in the example, will cause the subscript K to be reset by the value of the variable in CØMMØN.

Now consider this example.

```
      CØMMØN   K              (1)      SUBROUTINE   ABC(X)      (4)
      ...                               CØMMØN   N               (5)
      DØ  1    K = 1,J        (2)
    1 CALL   ABC (A(K))       (3)
```

Throughout the loop on K, the subscript K will be advanced in accordance with the DØ specification; the value of K in CØMMØN storage, however, as transmitted to the subprogram ABC, will remain at whatever value it had before entering the loop. (If statement (2) is the first appearance of K following statement (1), moreover, K in CØMMØN storage will have had no value assigned to it at all.) A statement

```
K = K                (2a)
```

inserted within the range of the DØ, or any appropriate statement with K on the left of an = sign, will reset K in CØMMØN storage (i.e., K in its role as <u>variable</u>) in accordance with the values of the index.