S-FORTRAN LANGUAGE

REFERENCE GUIDE

MARCH 1980

1st edition: October 1974
2nd edition: January 1976
3rd edition: May 1977
4th edition: March 1980

TABLE OF CONTENTS

1. INTRODUCTION

The S-FORTRAN language developed by Caine, Farber & Gordon, Inc., is a powerful extension of the FORTRAN language to allow easy, efficient and reliable structured programming in a FORTRAN environment.

The language results from the adjunction of a carefully chosen set of control structures to existing FORTRAN. These extensions encompass all those that have been proposed in the literature and shown to be useful and safe in practice. These extensions are compatible with standard FORTRAN as well as with currently existing extended FORTRAN.

The language is simple to learn. The syntax as well as the semantics of the new control structures have been chosen to emphasize coherence and regularity. S-FORTRAN programs are easy to read and understand, accurately and without ambiguity.

The language is simple to remember. In practice, this is an important factor to guarantee that programs will never be misinterpreted. As a matter of fact, you will probably find that after your initial reading of this reference guide, you can read and write S-FORTRAN programs without having to refer to this manual but rarely.

Although the language is simple to learn and remember, it is nevertheless quite powerful. Any program that you can now write in FORTRAN can advantageously be written in S-FORTRAN and yield a more readable, more reliable, better documented program.

The S-FORTRAN language is implemented on a variety of machines using a production S-FORTRAN to FORTRAN translator. The purpose of the translator is to list the input programs automatically indented, scan for possible errors and print appropriate diagnostics, and produce equivalent FORTRAN programs that can subsequently be compiled and executed.

The FORTRAN code produced by the translator is target compiler dependent primarily to achieve efficiency. The translator output code generator is sufficiently good to obviate the tendency of some programmers to write locally tricky code in order to achieve some efficiency gain. S-FORTRAN programs can, therefore, remain clear and readable without having to sacrifice object code efficiency.

2. RELATION TO OTHER FORTRAN LANGUAGES

The syntax of S-FORTRAN cannot be fully defined without making reference to an underlying FORTRAN language. Being an extension of FORTRAN, the S-FORTRAN language can be considered an extension of ANSI standard FORTRAN or an extension of some extended FORTRAN language as implemented by current compilers. Thus, when we refer to a *logical condition* in an S-FORTRAN statement, the syntax of the logical condition is defined not at the S-FORTRAN level but, rather, at the level of the underlying FORTRAN.

This implies that the user of S-FORTRAN is free to write compiler dependent S-FORTRAN programs if he wishes to do so. Alternatively, he may restrict his programming to conform to standard FORTRAN and obtain portable S-FORTRAN programs.

3. BASIC LANGUAGE ELEMENTS

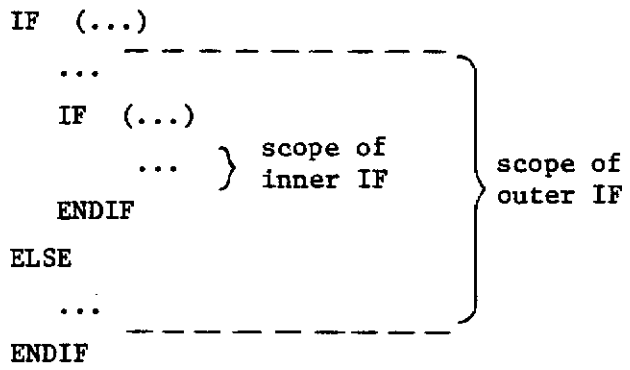
3.1 Definitions

An S-FORTRAN statement is made up of S-FORTRAN keywords used in conjunction with constants, variables, and expressions.

A construct is composed of one or more statements which must appear in a precise sequence. If a construct is composed of more than one statement, it necessarily has an opening statement and a closing statement, each of which contains the same characteristic keyword.

If the opening statement keyword is XXX, the closing statement is of the form END XXX (for instance, IF...ENDIF, DOWHILE...ENDDO WHILE). Any statement physically located between XXX and ENDXXX is said to be in the scope of XXX.

Example:



A group of statements is said to be well formed if any construct it contains is complete. Either this construct is a single statement or else both its opening and closing statements are within the well formed group.

Examples of well formed groups are:

```
{                                   empty well formed group
{ X = 3.
{ Y = SIN(X**2)
{ IF  (...)
{     CALL OUTPUT (BUFFER, LENGTH)
{ ENDIF
```

Conversely, the following example:

```
{ X = F(Y,L)
{ IF  (...)
{    IF  (...)
{        Y = X**2
{    ENDIF
```

is not well formed because the first IF is not closed by an ENDIF.

Well formed groups will be denoted by S_1, S_2, S_3, ...

An S-FORTRAN *label* is written as a FORTRAN label except that its range is restricted to satisfy

$$1 \leq label \leq 49999$$

Logical condition designates a condition acceptable in a logical IF in the associated FORTRAN language.

Similarly *arithmetic expression* designates an expression acceptable in an arithmetic IF in the associated FORTRAN language.

3.2 S-FORTRAN Statements

Source programs consist of FORTRAN and S-FORTRAN statements, freely intermixed. S-FORTRAN statements are composed of keywords used in conjunction with constants, variables, and expressions. Keywords are not reserved words.

S-FORTRAN statements are written one to a card within columns 7 through 72. As in FORTRAN, if a statement is too long it can be continued on up to 19 continuation cards which must have neither a blank nor a zero in column 6. For the first card of a statement, column 6 must be either blank or zero. As in FORTRAN, blanks are not significant and can be used freely to improve clarity if necessary. For instance

```
ENDIF
END IF
E N D   I F
```

are equivalent ways of writing the ENDIF keyword.

A numeric label may be placed in column 1 through 5 of the first card of a statement. Blanks and leading zeros are ignored. An S-FORTRAN *label* must be in the range

$$1 \leq label \leq 49999$$

if the program unit (program or subprogram) contains at least one S-FORTRAN construct.

Columns 73 through 80 can be used for any purpose since they are simply listed by the translator.

Comments to explain the program are indicated by the letter C in column 1. Comments may appear anywhere, including immediately before a continuation card.

4. S-FORTRAN CONSTRUCTS

In this section, we define the syntax and explain the operation of each S-FORTRAN construct. Every example listed forms a well formed group and could be used in place of S_1, S_2, S_3,... in any example. Notice that if two consecutive S-FORTRAN statements are written without any intervening S_i group or "...", then no S-FORTRAN or FORTRAN statement except possible comments can appear in between. For instance

```
IF  (logical condition 1)
    S1
ENDIF
```

or

```
IF  (logical condition 1)
    ...
ENDIF
```

means that between an IF and its associated ENDIF statement, any well formed group of statements may appear, including a null group.

However

```
DO  CASE  unsubscripted integer variable
    case  case specification 1
        ...
```

implies that no statement except possibly some comment statements may appear between DO CASE and its first CASE statement.

4.1 IF Construct

4.1.1 IF...ELSE...ENDIF

The simplest form of the IF construct is

```
IF  (logical condition 1)
   S1
ENDIF
```

where *logical condition 1* is a FORTRAN logical condition acceptable in a logical IF, and S_1 is any well formed group of statements. S_1 is executed if *logical condition 1* has the value .TRUE., otherwise it is bypassed.

A single ELSE statement can be added to specify that S_2 should be executed if *logical condition 1* is found to be equal to .FALSE.. The IF construct is then written as

```
IF  (logical condition 1)
   S1
ELSE
   S2
ENDIF
```

As before, S_1 and S_2 can be any well formed groups of statements. Obviously, S_1 could be a null well formed group as in

```
IF  (logical condition 1)
ELSE
    S2
ENDIF
```

however, it is better in this case to write

```
IF (.NOT. (logical condition 1))
   S2
ENDIF
```

and optionally to negate the *logical condition 1* if this improves readability.

Since the two forms of the IF construct are well formed groups, S_1 and/or S_2 may also contain some instances of IF...ENDIF or IF...ELSE...ENDIF. For instance, we may write:

```
IF  (logical condition 1)
    IF  (logical condition 2)
        S3
    ENDIF
ELSE
    IF  (logical condition 3)
        S4
    ELSE
        S5
    ENDIF
ENDIF
```

This "nesting" property can be carried out to any depth level since S_3, S_4 and S_5 are also well formed groups.

4.1.2 IF...ELSEIF...ELSE...ENDIF

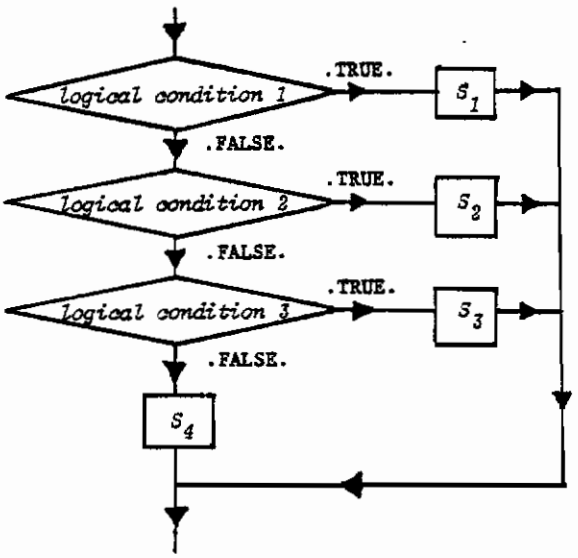
The complete form of the IF construct may include one or more ELSEIF statements that must appear before the ELSE statement, if one exists. The purpose of ELSEIF is to avoid deep indentation levels when many IF's would have to be nested. For instance

```
IF  (logical condition 1)
    S1
ELSEIF  (logical condition 2)
    S2
ELSEIF  (logical condition 3)
    S3
ELSE
    S4
ENDIF
```

where S_1, S_2, S_3, S_4 are well formed groups of statements, is strictly equivalent to

```
IF  (logical condition 1)
    S1
ELSE
    IF  (logical condition 2)
        S2
    ELSE
        IF  (logical condition 3)
            S3
        ELSE
            S4
        ENDIF
    ENDIF
ENDIF
```

Thus, the IF...ELSEIF...ELSEIF...ELSE...ENDIF construct corresponds to the often occurring pattern of a sequential sieve. Each test is carried out in turn. As soon as one succeeds, its associated well formed group is executed and we proceed with the first statement following ENDIF. If the sequential tests all fail and there is an ELSE statement, the well formed group associated with ELSE is executed.



Example:

```
 30      *        IF (CHAR .NE. BLANK)
 31    1 *           IF (CHAR .EQ. L PAREN)
 32    2                LEVEL = LEVEL + 1
 33    1 *           ELSEIF (CHAR .EQ. R PAREN)
 34    2                LEVEL = LEVEL - 1
 35    2 *              IF (LEVEL .LT. 0)
 36    3                   CALL ERROR (LEVEL)
 37    2 *              ENDIF
 38    1 *           ELSE
 39    2                LENGTH = LENGTH + 1
 40    1 *           ENDIF
 41      *        ENDIF
```

* indicates an S-FORTRAN statement

nesting level

Line number

4.2 Repetitive DO Constructs

There are four constructs which define and control the execution of loops in S-FORTRAN. Each repetitive DO construct comprises both an opening and a closing statement that define the scope of the loop.

4.2.1. DO FOR...ENDDO FOR

This construct is the structured equivalent of the FORTRAN DO loop. It is written as

DO FOR $I = I_1, I_2$

S_1

END DO FOR

or

DO FOR $I = I_1, I_2, I_3$

S_1

END DO FOR

I, I_1, I_2, I_3, follow the rules imposed by the target FORTRAN compiler. However, even if not necessary in FORTRAN, the following restrictions must be observed or the results are unpredictable:

1. The index variable, I, must be of type INTEGER.
2. The termination and increment expressions, I_2 and I_3, must not be caused to change within the loop or its extended range.

As in FORTRAN, the well formed group S_1 is always executed at least once. Therefore, appropriate tests may be necessary to bypass the loop if I_1, I_2, I_3 are incompatible.

```
43        C                                C     BUILD DIAGONAL MATRIX
44      *       IF (N .GT. 0)
                +-------------------------------------------------------
45    1 *       |DO FOR I = 1, N
                |  +----------------------------------------------------
46    2 *       |  |DO FOR J = 1, N
47    3 *       |  |   IF (I .EQ. J)
48    4         |  |      A(I,J) = 1
49    3 *       |  |   ELSE
50    4         |  |      A(I,J) = 0
51    3 *       |  |   ENDIF
52    2 *       |  |ENDDO FOR
                |  +----------------------------------------------------
53    1 *       |ENDDO FOR
                +-------------------------------------------------------
54      *       ENDIF
```

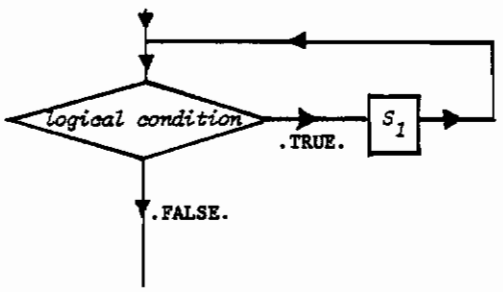
4.2.2 DO WHILE...ENDDO WHILE

DO WHILE together with ENDDO WHILE is used to specify that a group of statements must be repetitively executed as long as the associated *logical condition* remains equal to .TRUE.. The well formed group of statements is not executed at all if the *logical condition* is false upon reaching the DO WHILE statement. Whenever an iteration terminates by reaching the END DO WHILE statement, the *logical condition* is evaluated again. If it still yields a value .TRUE., a new iteration is started. Otherwise, the first statement following END DO WHILE is given control.

The format is

DO WHILE (*logical condition*)
  S_1
END DO WHILE

and the execution flow chart of the DO WHILE construct is:



Example:

```
56        C                                C     BUILD DIAGONAL MATRIX
58         I = 0
          +-------------------------------------------------------
59      *  |DO WHILE (I .LT. N)
60    1    |   I = I + 1
61    1    |   J = 0
           |  +----------------------------------------------------
62    1 *  |  |DO WHILE (J .LT. N)
63    2    |  |   J = J + 1
64    2 *  |  |   IF (I .EQ. J)
65    3    |  |      A(I,J) = 1
66    2 *  |  |   ELSE
67    3    |  |      A(I,J) = 0
68    2 *  |  |   ENDIF
69    1 *  |  |ENDDO WHILE
           |  +----------------------------------------------------
70      *  |ENDDO WHILE
          +-------------------------------------------------------
```

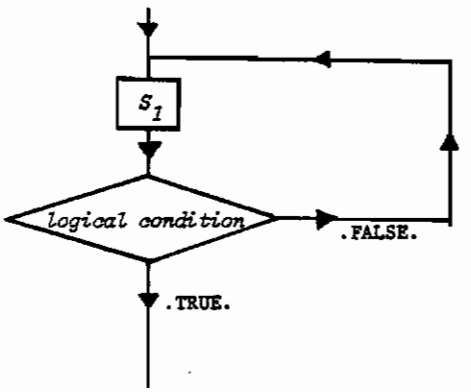
4.2.3 DO UNTIL...ENDDO UNTIL

DO UNTIL together with ENDDO UNTIL is used to specify that a group of statements must be repetitively executed as long as a given *logical condition* is still .FALSE.. Notice that contrary to the DO WHILE construct, the well formed group of statements is always executed at least once before the condition is first evaluated. Thus, DO UNTIL should be read as DO AT LEAST ONCE UNTIL the *logical condition* becomes true. If the *logical condition* is equal to .FALSE., a new iteration is started and so forth until *logical condition* finally becomes equal to .TRUE., at which point control is passed to the first statement following END DO UNTIL.

The format is

```
DO UNTIL  (logical condition)
  S1
END DO UNTIL
```

and the execution flow chart of the DO UNTIL construct is



Example:

```
72        C                                C     BUILD DIAGONAL MATRIX
73              I = 1
                +-------------------------------------------------------
74        *     |DO UNTIL (I .GT. N)
75    1         |   J = 1
                |  +----------------------------------------------------
76    1 *       |  |DO UNTIL (J .GT. N)
77    2 *       |  |   IF (I .EQ. J)
78    3         |  |      A(I,J) = 1
79    2 *       |  |   ELSE
80    3         |  |      A(I,J) = 0
81    2 *       |  |   ENDIF
82    2         |  |   J = J + 1
83    1 *       |  |ENDDO UNTIL
                |  +----------------------------------------------------
84    1         |   I = I + 1
85        *     |END DO UNTIL
                +-------------------------------------------------------
```

4.2.4 DO FOREVER...END DO FOREVER

This construct is used to set up an unconditional loop. It is written as

```
DO FOREVER
   S1
ENDDO FOREVER
```

and is strictly equivalent to

```
DO WHILE (.TRUE.)
   S1
END DO WHILE
```

However, the advantage of using a DO FOREVER...ENDDO FOREVER is to draw the attention of the reader to the fact that the loop is potentially an infinite loop. Therefore, precautions must be taken to terminate or exit from the loop at some point. The well formed group S_1 must contain one or more statements that will cause either directly or indirectly a return, stop, or some form of exit from the S_1 group. Exit from DO groups will be explained later in section 4.4.

Example:

```
 88     *        IF (OPTION .EQ. RD IN)
                 +-----------------------------------------------
 89   1 *        |DO FOREVER
 90   2          |   READ(IN UNIT, 500) CODE, TEXT
 91   2 *        |   IF (CODE .EQ. 0)
 92   3          |      PRINT 600
 93   3          |      STOP
              #--------------
 94   3   600   |      FORMAT(25H LAST CARD FOUND, STOP OK)
 95   2 *        |   ELSE
 96   3          |      CALL BUILD   (CODE, TEXT)
 97   2 *        |   ENDIF
 98   1 *        |ENDDO FOREVER
                 +-----------------------------------------------
 99     *        ELSE
100   1          CALL PRCS
101     *        ENDIF
```

4.3 Non-Repetitive DO Constructs

The next four constructs are called non-repetitive DO constructs because they are DO groups but not loops. They all comprise an opening and a closing statement.

4.3.1 DO...END DO

This construct is used to bracket a well formed group of statements. It is written as

```
DO
  S1
END DO
```

and has no effect by itself. However, the purpose of this construct will become clear in section 4.4 when exit mechanisms applicable to DO groups are discussed.

4.3.2 DO CASE construct

This construct is used to execute selectively one among several well formed groups of statements depending upon either the value of an *unsubscripted integer variable* or the sign of an *arithmetic expression*.

4.3.2.1 DO CASE...CASE...END DO CASE

This first form of the DO CASE construct is used to execute selectively one among several well formed groups of statements depending upon the value of an *unsubscripted integer variable*. Its format is

```
DO CASE unsubscripted integer variable
   CASE case specification 1
      S1
   ...
   CASE case specification n
      Sn
END DO CASE
```

Each CASE statement contains a *case specification* which defines one or more integer values. For the associated well formed groups S_i to be executed, the CASE variable *unsubscripted integer variable* must have a value equal to one of the values defined in *case specification i*. In particular, if *unsubscripted integer variable* happens to have a value which is not associated with any of the CASE statements, the whole DO CASE...ENDDO CASE group is bypassed.

There are three ways of associating integer values with a particular well formed group of statements.

a. by specifying the integer constant explicitly, for instance

```
CASE 2
```

or if there are several such values, by adding commas to separate the signed or unsigned integer constants, for instance

```
CASE -1,3,+4
```

b. by specifying the set of integers less than or greater than a signed or unsigned integer constnat, for instance

```
CASE  .LT.-10
```

or

```
CASE  .GT.  2048
```

or

```
CASE  0, .LT.-10, .GT. 10
```

(notice that '.LT.' and '.GT.' can be written '<' and '>' respectively when these characters are available).

c. by specifying the set of integers not contained in any of the type a or b specifications. This is written

```
CASE OTHER
```

and is interpreted as any integer value not otherwise specified in any of the CASE statements associated with this particular DO CASE construct.

Examples of correct *case specifications* are

```
CASE  2
CASE  -1, 3, 4
CASE  .LT. -10, .GT.+10, -2, +2
CASE  OTHER
```

whereas the following case specifications are illegal

```
CASE  2.0
CASE  K
```

Let us now consider a complete example of a DO CASE construct including several CASE statements

```
                +-------------------------------------
119     *       |DO CASE I2
120   1 *       |   CASE 1, 3
121   2         |      CALL PGA
122   1 *       |   CASE 2,4
123   2         |      CALL PGB(I2)
124   1 *       |   CASE .LT.1
125   2         |      PRINT 610
126   2         |      STOP
              #-----------
127   1 *       |   CASE .GT. 4
128   2 *       |      IF (F(I2) .LT. WIDTH)
129   3         |         F(I2) = F(I2) * COEFF
130   2 *       |      ENDIF
131   2         |      RETURN
              <-----------
132     *       |END DO CASE
                +-------------------------------------
```

It is interpreted as:

- if I2 = 1 or I2 = 3, call the subroutine PGA
- if I2 = 2 or I2 = 4, call the subroutine PGB
- if I2 < 1, print a message and stop
- if I2 > 4, execute the IF test and RETURN

It is important to notice that contrary to the IF...ELSEIF...ELSE...ENDIF construct where the order of the IF and ELSEIF statements is important, the order of the CASE groups is not. The above example could just as well have been written:

```
                +-----------------------------------------
104     *       |DO CASE I2
105   1 *       |   CASE .LT.1
106   2         |      PRINT 610
107   2         |      STOP
              #-----------
108   1 *       |   CASE 1, 3
109   2         |      CALL PGA
110   1 *       |   CASE 2,4
111   2         |      CALL PGB(I2)
112   1 *       |   CASE .GT. 4
113   2 *       |      IF (F(I2) .LT. WIDTH)
114   3         |         F(I2) = F(I2) * COEFF
115   2 *       |      ENDIF
116   2         |      RETURN
              <-----------
117     *       |END DO CASE
                +-----------------------------------------
```

The *case specifications* associated with all the CASE statements that are part of a DO CASE group are not independent. They must satisfy jointly the constraint that any integer value is specified at most once, either explicitly or implicitly. Thus, any two *case specifications* belonging to the same DO CASE construct must have a null intersection. For instance:

```
                +-----------------------------------------
135     *       |DO CASE I
136   1 *       |   CASE 1,3,7
137   2         |      S1
138   1 *       |   CASE 3,4
139   2         |      S2
140     *       |ENDDO CASE
                +-----------------------------------------

*** ERROR ***   DUPLICATE CASE SPECIFICATION IGNORED
************
```

is illegal because the same integer value 3 appears in two distinct CASE statements.

```
                    +-----------------------------------------
  142      *        |DO CASE   INDEX
  143    1 *        |   CASE .LT.   5
  144    2          |      CALL FF(I, INDEX)
  145    1 *        |   CASE .LT. 1

*** ERROR ***    DUPLICATE CASE .LT. IGNORED
*************
  146    2          |      INDEX = 1
  147      *        |ENDDO CASE
                    +-----------------------------------------
```

is illegal because two '.LT.' clauses are used on two CASE statements that belong to the same DO CASE construct. Their intersection is the set of integers less than or equal to 1 which is illegal since any two *case specifications* must always have an empty intersection.

Therefore, in order to write legal DO CASE constructs, the following rules should be observed:

- at most one '.LT.' clause, one '.GT.' clause and one 'OTHER' can appear in the CASE statements of a given DO CASE construct
- any integer value specified explicitly in a CASE statement must be distinct from any other value specified in the same or any other CASE statement belonging to the same DO CASE construct
- If the clause

  .LT. n

  is used, the integer n must be less than or equal to any integer appearing in any of the CASE statements of the same DO CASE construct

. If the clause

.GT. n

is used, the integer n must be greater than or equal to any integer appearing in any of the CASE statements of the same DO CASE construct.

4.3.2.2 DO CASE SIGN OF...CASE...END DO CASE

This second form of the DO CASE construct is used to execute selectively one of exactly three well formed groups depending upon whether the value of an *arithmetic expression* is positive, negative or equal to 0. It is written as:

```
DO CASE SIGN OF (arithmetic expression)
    CASE  .LT.  0
       S1
    CASE  0
       S2
    CASE  .GT.  0
       S3
ENDDO CASE
```

where *arithmetic expression* is an arithmetic expression of any type except complex and S_1, S_2, S_3 are well formed groups of statements. S_1 will be executed when *arithmetic expression* is strictly negative, S_2 when it is exactly equal to 0 and S_3 when it is strictly positive. When the characters exist, '.LT.' can also be written as '<' and '.GT.' as '>'.

The order in which the three CASE statements appear is immaterial. The example above could also have been written:

```
DO CASE SIGN OF (arithmetic expression)
    CASE  0
       S2
    CASE  .LT.  0
       S1
    CASE  .GT.  0
       S3
ENDDO CASE
```

In some instances, S_1, S_2, S_3 may be null. Nevertheless, the corresponding CASE statement must be present. For example:

DO CASE SIGN OF (*arithmetic expression*)

CASE 0

S_1

CASE > 0

S_2

CASE < 0

ENDDO CASE

Example:

```
149       C                                C     QUADRATIC EQUATION SOLVER
151             DISCR = B**2 - 4.*A*C
                +-------------------------------------------------------------
152         *   |DO CASE SIGN OF (DISCR)
153     1   *   |   CASE  .LT. 0
154     2       |      ROOT1 R = -B/(2.*A)
155     2       |      ROOT1 I = SQRT(-DISCR) / (2.*A)
156     2       |      ROOT2 R = ROOT1 R
157     2       |      ROOT2 I = - ROOT1 I
158     1   *   |   CASE  0
159     2       |      ROOT1 R = -B / (2.*A)
160     2       |      ROOT1 I = 0.
161     2       |      ROOT2 R = ROOT1 R
162     2       |      ROOT2 I = 0.
163     1   *   |   CASE  .GT. 0
164     2       |      ROOT1 R = (-B+SQRT(DISCR)) / (2.*A)
165     2       |      ROOT1 I = 0.
166     2       |      ROOT2 R = (-B-SQRT(DISCR)) / (2.*A)
167     2       |      ROOT2 I = 0.
168         *   |ENDDO CASE
                +-------------------------------------------------------------
```

4.3.3 DO LABEL...LABEL...END DO LABEL

Although not part of ANSI standard FORTRAN, abnormal returns from SUBROUTINE and FUNCTION calls as well as END and ERR exits in READ and WRITE statements are widely used in extended FORTRAN languages. The DO LABEL...LABEL...ENDDO LABEL construct is available in S-FORTRAN to handle these abnormal cases.

The DO LABEL construct has the general form:

```
statement ... label1 ... label2 ... label3 ...
DO LABEL
    LABEL label 1, label 2, ...
        S1
    LABEL label 3, ...
        S2
    ...
END DO LABEL
```

A DO LABEL group should immediately follow each statement containing the abnormal labels. Each *label i* designates an S-FORTRAN label and, optionally, one *label i* in any DO LABEL group can be replaced by the character *.

The interpretation of the DO LABEL group is as follows:

- if executing the abnormal statement causes a return via one of the abnormal labels, the well formed group associated with the abnormal label is executed and control then proceeds with the first executable statement following ENDDO LABEL
- if on the contrary, the abnormal statement returns normally, control transfers to the next statement in sequence, i.e. DO LABEL. At that point, if the DO LABEL group contains

a 'LABEL *' statement, the associated well formed group is executed and control then proceeds with the first executable statement following END DO LABEL. If no 'LABEL *' statement is specified as part of the DO LABEL group, the DO LABEL group is completely bypassed.

Example:

```
170                READ (5, 500, END = 100, ERR = 200) CARD
                   +-----------------------------------------------
171      *         |DO LABEL
172    1 *         |   LABEL 100
173    2           |       PRINT 650
174    2           |       EOF = .TRUE.
175    1 *         |   LABEL 200
176    2           |       PRINT 651
177    2           |       CALL DIE
178    2           |       STOP
                   #-----------
179      *         |ENDDO LABEL
                   +-----------------------------------------------
```

In this example, PRINT 650... is executed if an end of file is found on logical unit 5. If an I/O error condition is detected, PRINT 651... is executed. If the read operation terminates normally, the DO LABEL group is simply bypassed.

Clearly, in a given program or subprogram, all labels appearing in LABEL statements must be distinct. It is also in order at this point to warn against an indiscriminate use of the DO LABEL construct which might seriously jeopardize program clarity.

4.4 DO Groups Exit Constructs

When writing structured code, it is sometimes necessary to exit prematurely from a DO group, repetitive or not. Although such a premature exit can be achieved by appropriately setting and testing logical switch variables, more readable programs are usually obtained using the UNDO and CYCLE statement.

4.4.1 UNDO statement

The UNDO statement is used to exit prematurely from a DO group, repetitive or not. It causes a transfer to the first statement immediately following the closing statement of the DO group to which it is applied.

4.4.1.1 UNDO, unconditional

The simplest form of the UNDO statement is written

```
UNDO
```

When this statement is encountered, the innermost DO group, repetitive (DO FOR, DO WHILE, DO UNTIL, DO FOREVER) or not (DO, DO CASE, DO CASE SIGN OF, DO LABEL) in which UNDO is nested is terminated and control proceeds with the first executable statement following the terminated DO group.

Example:

```
                +-----------------------------------------
182     *       |DO FOREVER
183   1         |   S1
184   1 *       |   IF (EOF)
185   2 *       |      UNDO
                <-------
186   1 *       |   ENDIF
187   1         |   S2
188     *       |ENDDO FOREVER
                +-----------------------------------------
189             S3
```

In this example, the infinite loop is terminated when UNDO is encountered. Notice that the S-FORTRAN processor places an arrow after the UNDO statement to stress the semantic action, namely that the loop is left at this point. Control proceeds with the first executable statement of S_3.

It sometimes becomes necessary to exit not from the innermost DO group but from an outer DO group. The UNDO statement is then written

UNDO *label*

where *label* is an S-FORTRAN label which must have appeared as the opening statement label of a repetitive or non repetitive DO construct whose scope includes the UNDO statement.

Example:

```
                +-----------------------------------------------
191      * 10   |DO FOR I = 1, N
192   1         |   S1
                |  +--------------------------------------------
193   1 * 20    |  |DO WHILE (T .GT. 0.)
194   2         |  |   ...
195   2 *       |  |   UNDO 10
                <--------
196   2         |  |   ...
197   2 *       |  |   UNDO
                |  <----
198   2         |  |   ...
199   2 *       |  |   UNDO 20
                |  <----
200   2         |  |   ...
201   1 *       |  |ENDDO WHILE
                |  +--------------------------------------------
202   1         |   S2
203     *       |ENDDO FOR
                +-----------------------------------------------
204             S3
```

4.4.1.2 UNDO...IF, conditional

As a notational convenience, the S-FORTRAN language allows either form of the UNDO statement to be made conditional by appending a logical test. The UNDO statement can, therefore, be written:

UNDO IF (*logical condition*)

or

UNDO *label* IF (*logical condition*)

These conditional UNDO statements are strictly equivalent to

```
IF (logical condition)
    UNDO
ENDIF
```

or

```
IF (logical condition)
    UNDO label
ENDIF
```

respectively.

Example: The first UNDO example shown in page 28 could also have been written:

```
                +-----------------------------------
206     *       |DO FOREVER
207   1         |   S1
208   1 *       |   UNDO IF (EOF)
                <-----
209   1         |   S2
210     *       |END DO FOREVER
                +-----------------------------------
211               S3
```

4.4.2 CYCLE statement

The CYCLE statement is used to skip any statements remaining to be executed in the current cycle of a repetitive DO group (DO FOR, DO WHILE, DO UNTIL, DO FOREVER). When a CYCLE statement is encountered, control proceeds to the closing statement END DO... of the designated repetitive DO group. Any subsequent actions take place as if the closing statement END DO... had been normally encountered as the next statement to be executed.

4.4.2.1 CYCLE, unconditional

The CYCLE statement is used to skip any statements remaining to be executed in the current cycle of a repetitive DO group (DO FOR, DO WHILE, DO UNTIL or DO FOREVER). Its format is

```
CYCLE
```

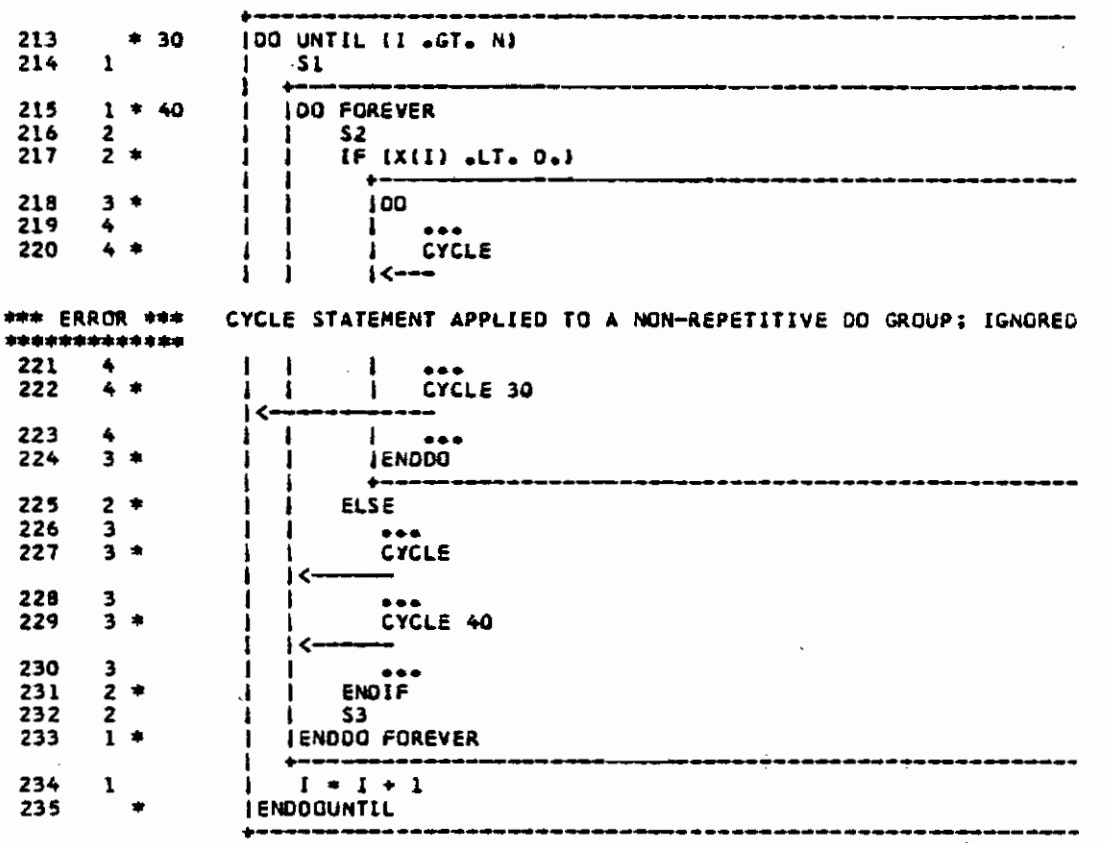
or

```
CYCLE label
```

The first form of the CYCLE statement is used to unconditionally transfer to the closing statement ENDDO... of the innermost DO group in which the CYCLE statement is located. It is an error to try applying CYCLE to a non repetitive DO group.

The second form is used when the target DO group is not the innermost one but one of the outer DO groups. The *label* must be an S-FORTRAN label associated with the opening statement DO... of a repetitive DO construct whose scope includes the CYCLE statement. If such a label does not exist or is associated with a non repetitive DO group, an error occurs. If *label* happens to be associated with the innermost repetitive DO group whose scope includes CYCLE, then the two forms of the CYCLE statement are equivalent.

Examples:


213 * 30 DO UNTIL (I .GT. N)
214 1 S1
215 1 * 40 DO FOREVER
216 2 S2
217 2 * IF (X(I) .LT. 0.)
218 3 * DO
219 4 •••
220 4 * CYCLE
*** ERROR ***
CYCLE STATEMENT APPLIED TO A NON-REPETITIVE DO GROUP; IGNORED
221 4 •••
222 4 * CYCLE 30
223 4 •••
224 3 * ENDDO
225 2 * ELSE
226 3 •••
227 3 * CYCLE
228 3 •••
229 3 * CYCLE 40
230 3 •••
231 2 * ENDIF
232 2 S3
233 1 * ENDDO FOREVER
234 1 I = I + 1
235 * ENDDOUNTIL

4.4.2.2 CYCLE...IF, conditional

As a notational convenience, either form of the CYCLE statement can also include a *logical condition*. In that case, CYCLE is written

CYCLE IF (*logical condition*)

or

CYCLE *label* IF (*logical condition*)

These conditional CYCLE statements are strictly equivalent to

```
IF (logical condition)
    CYCLE
END IF
```

or

```
IF (logical condition)
    CYCLE label
END IF
```

respectively.

4.5 Procedures

A very powerful mechanism of the S-FORTRAN language is the ability to define and execute procedures. A procedure is simply a well formed group of statements with a descriptive name attached to it. A procedure can be remotely executed from one or several places within the same program or subprogram by simply referencing its name. This allows one to write modular programs and develop them top down.

It is important to understand the differences between a subprogram and a procedure. A subprogram is an independently compiled unit, with an optional list of arguments and its own data space. On the contrary, procedures are not independently compiled but reside within the program or subprogram that references them. Procedures are not passed arguments and clearly share the same data space as the program or subprogram in which they reside.

In practice, procedures are used not in place of subprograms but whenever creating a subprogram just to enhance modularity seems like too much trouble. Procedures have the advantage of a very low overhead both from the standpoint of core occupation and execution time.

4.5.1 PROCEDURE...END PROCEDURE

A procedure is defined as

PROCEDURE (*procedure name*)
S_1
END PROCEDURE

Optionally, *procedure name* may be repeated on the END PROCEDURE statement as follows

END PROCEDURE (*procedure name*)

The name *procedure name* is a string of alphanumeric characters (A through Z, 0 through 9), the first one of which must be alphabetic. The length of procedure name must be at least 1 and at most 31 non blank characters. Any leading, trailing and embedded blanks are ignored. In other words, a *procedure name* is built like a standard FORTRAN identifier except that its length can go up to 31. Example of valid procedure names are:

PROCEDURE (INVERT MATRIX 1)

or equivalently

PROCEDURE (INVERTMATRIX1)

PROCEDURE (P2)
PROCEDURE (COMPUTE TRAJECTORY CORRECTION)

whereas the following procedure names are illegal:

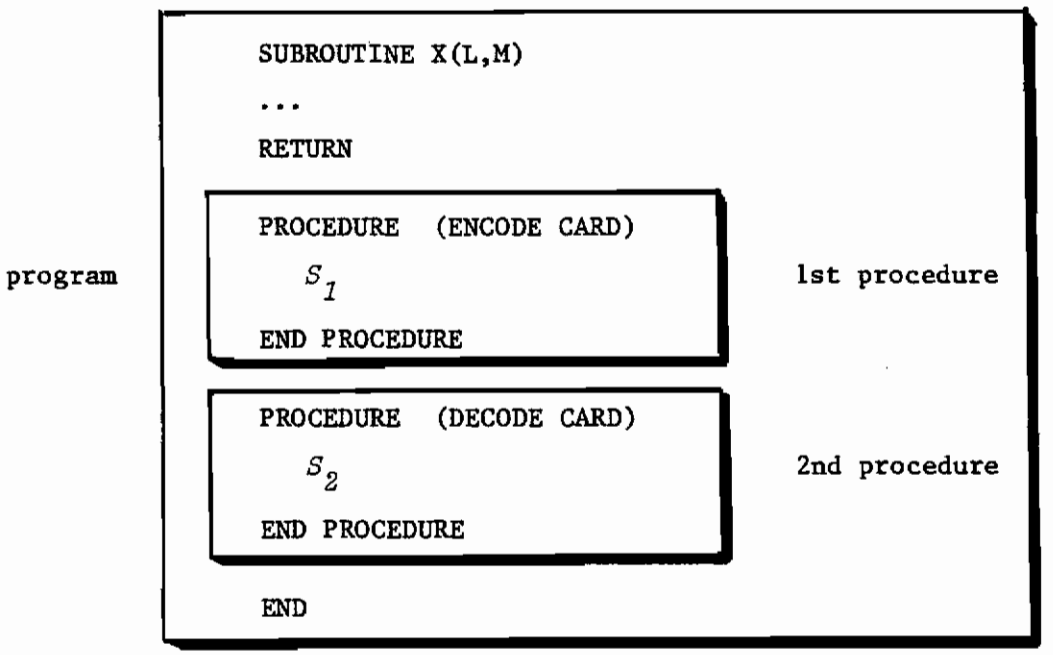
PROCEDURE ( )
is illegal since procedure name must contain at least one non blank character

PROCEDURE (EVALUATE A + B)
    is illegal since '+' is not alphanumeric
PROCEDURE (2)
    is illegal since the first character must be alphabetic
PROCEDURE (NEW VERSION OF THE DIFFERENTIAL EQUATION SOLVER)
    is illegal because the name contains 41 non blank characters which is larger than the maximum permitted length 31.

Each program or subprogram may contain any number of procedures. However, all procedures within a given program or subprogram must appear after every executable statement of the program body (i.e. the part of the program or subprogram that is not in any procedure). The only statements that can appear after the last END PROCEDURE statement are the END statement and some optional FORMAT statements.

A procedure may not contain another procedure or a portion of another procedure. Thus, the scope of each procedure must be totally disjoint from the scope of any other procedure.

A typical program layout might be the following:



4.5.2 EXECUTE statement

Statements contained within a procedure can only be executed by means of an EXECUTE statement. A procedure cannot be fallen into and executed. It must be invoked by a statement of the form

EXECUTE PROCEDURE (*procedure name*)

or in short

EXECUTE (*procedure name*)

When a subprogram is entered as a result of a CALL statement, any procedures it contains are considered to be inactive.

When a procedure is invoked by an EXECUTE statement, control transfers to the first executable statement following the PROCEDURE statement. The procedure is then said to be active.

The procedure remains active until the END PROCEDURE statement is encountered. At that point, control transfers to the first executable statement immediately following the invoking EXECUTE statement and the procedure now becomes inactive again.

A procedure should never reference itself either directly or indirectly. Thus, an EXECUTE statement should never be applied to a procedure which is currently active or an infinite loop may result.

Notice that a procedure may contain RETURN or STOP statements. If a RETURN statement is encountered, a return from the subprogram is performed and all procedures contained in the subprogram become inactive.

```
CFG, INC.      S-FORTRAN TRANSLATOR VERSION  1.3          RDCARD

LINE NST
 NUM LVL S INPUT S-FORTRAN STATEMENT

 682              SUBROUTINE RD CARD(CARD, SWITCH)
 683              INTEGER CARD(20)
 684              LOGICAL SWITCH
 685              ...
 686     *        IF (SWITCH)
 687   1 *           EXECUTE (READ A CARD)
 688     *        ELSE
 689   1 *           EXECUTE (ABORT TASK)
 690     *        ENDIF
 691              ...
 692              RETURN
                <-----
```

```
CFG, INC.      S-FORTRAN TRANSLATOR VERSION  1.3

LINE NST
 NUM LVL S INPUT S-FORTRAN STATEMENT

 694     *        PROCEDURE (READ A CARD)
 695   1             ...
 696   1             READ (INPUT, 500, ERR = 100) CARD
                    +-----------------------------------
 697   1 *          |DO LABEL
 698   2 *          |   LABEL 100
 699   3 *          |      EXECUTE (ABORT TASK)
 700   1 *          |ENDDO LABEL
                    +-----------------------------------
 701   1             ...
 702     *        END PROCEDURE
```

```
CFG, INC.      S-FORTRAN TRANSLATOR VERSION  1.

LINE NST
 NUM LVL S INPUT S-FORTRAN STATEMENT

 704     *        PROCEDURE (ABORT TASK)
 705   1             ...
 706     *        END PROCEDURE
 707            C                                  C
 708            500   FORMAT(2014)
 709                  END
```

4.5.3 EXIT statement

The execution of a procedure terminates when its END PROCEDURE statement is encountered. However, the execution of a procedure can be terminated prematurely using the EXIT statement. Its format is

```
EXIT
```

or

```
EXIT PROCEDURE
```

or

```
EXIT  (procedure name)
```

or

```
EXIT PROCEDURE  (procedure name)
```

EXIT always applies to the procedure in which it is located.

An EXIT statement outside the scope of a procedure is illegal. When *procedure name* is present, the name must match that of the enclosing procedure.

Example:

```
267     *       PROCEDURE (SEARCH FOR NEGATIVE)
268   1           ...
                  +-----------------------------------------
269   1 *         |DO FOR I = 1, M
270   2 *         |   IF (X(I) .LT. 0.)
271   3 *         |      EXIT
                <-----------
272   2 *         |   ENDIF
273   1 *         |ENDDO FOR
                  +-----------------------------------------
274   1           CALL ERROR
275     *       END PROCEDURE
```

INDEX