Section 8: Execution control

The execution sequence may be controlled by constructs containing blocks and by certain executable statements that are used to alter the execution sequence.

8.1 Executable constructs containing blocks

The following are executable constructs that contain blocks and may be used to control the execution sequence:

(1) IF Construct
(2) CASE Construct
(3) DO Construct

There is also a nonblock form of the DO construct.

A **block** is a sequence of executable constructs that is treated as a unit.

R801 *block* **is** [*execution-part-construct*] ...

Executable constructs may be used to control which blocks of a program are executed or how many times a block is executed. Blocks are always bounded by statements that are particular to the construct in which they are embedded; however, in some forms of the DO construct, a sequence of executable constructs without a terminating boundary statement shall obey all other rules governing blocks (8.1.1).

NOTE 8.1

A block need not contain any executable constructs. Execution of such a block has no effect.

Any of these constructs may be named. If a construct is named, the name shall be the first lexical token of the first statement of the construct and the last lexical token of the construct. In fixed source form, the name preceding the construct shall be placed after character position 6.

A statement **belongs** to the innermost construct in which it appears unless it contains a construct name, in which case it belongs to the named construct.

NOTE 8.2

An example of a construct containing a block is:

```
IF (A > 0.0) THEN
   B = SQRT (A)   ! These two statements
   C = LOG (A)    ! form a block.
END IF
```

8.1.1 Rules governing blocks

8.1.1.1 Executable constructs in blocks

If a block contains an executable construct, the executable construct shall be entirely within the block.

8.1.1.2 Control flow in blocks

Transfer of control to the interior of a block from outside the block is prohibited. Transfers within a block and transfers from the interior of a block to outside the block may occur.

NOTE 8.3

For example, if a statement inside the block has a statement label, a GO TO statement using that label is only allowed to appear in the same block.

Subroutine and function references (12.4.2, 12.4.3) may appear in a block.

8.1.1.3 Execution of a block

Execution of a block begins with the execution of the first executable construct in the block. Unless there is a transfer of control out of the block, the execution of the block is completed when the last executable construct in the sequence is executed. The action that takes place at the terminal boundary depends on the particular construct and on the block within that construct. It is usually a transfer of control.

8.1.2 IF construct

The **IF construct** selects for execution no more than one of its constituent blocks. The **IF statement** controls the execution of a single statement (8.1.2.4).

8.1.2.1 Form of the IF construct

R802 *if-construct* **is** *if-then-stmt*
    block
  [*else-if-stmt*
    block] ...
  [*else-stmt*
    block]
  end-if-stmt

R803 *if-then-stmt* **is** [*if-construct-name* :] IF (*scalar-logical-expr*) THEN

R804 *else-if-stmt* **is** ELSE IF (*scalar-logical-expr*) THEN [*if-construct-name*]

R805 *else-stmt* **is** ELSE [*if-construct-name*]

R806 *end-if-stmt* **is** END IF [*if-construct-name*]

Constraint: If the *if-then-stmt* of an *if-construct* is identified by an *if-construct-name*, the corresponding *end-if-stmt* shall specify the same *if-construct-name*. If the *if-then-stmt* of an *if-construct* is not identified by an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-name*. If an *else-if-stmt* or *else-stmt* is identified by an *if-construct-name*, the corresponding *if-then-stmt* shall specify the same *if-construct-name*.

8.1.2.2 Execution of an IF construct

At most one of the blocks in the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks in the construct will be executed. The scalar logical expressions are evaluated in the order of their appearance in the construct until a true value is found or an ELSE statement or END IF statement is encountered. If a true value or an ELSE statement is found, the block immediately following is executed and this completes the execution of the construct. The scalar logical expressions in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated expressions is true and there is no ELSE statement, the execution of the construct is completed without the execution of any block within the construct.

An ELSE IF statement or an ELSE statement shall not be a branch target statement. It is permissible to branch to an END IF statement only from within the IF construct. Execution of an END IF statement has no effect.

8.1.2.3 Examples of IF constructs

NOTE 8.4

```
IF (CVAR .EQ. 'RESET') THEN
   I = 0; J = 0; K = 0
END IF

PROOF_DONE: IF (PROP) THEN
   WRITE (3, '("QED")')
   STOP
ELSE
   PROP = NEXTPROP
END IF PROOF_DONE

IF (A .GT. 0) THEN
   B = C/A
   IF (B .GT. 0) THEN
      D = 1.0
   END IF
ELSE IF (C .GT. 0) THEN
   B = A/C
   D = -1.0
ELSE
   B = ABS (MAX (A, C))
   D = 0
END IF
```

8.1.2.4 IF statement

The IF statement controls a single action statement (R216).

R807 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*

Constraint: The *action-stmt* in the *if-stmt* shall not be an *if-stmt*, *end-program-stmt*, *end-function-stmt*, or *end-subroutine-stmt*.

Execution of an IF statement causes evaluation of the scalar logical expression. If the value of the expression is true, the action statement is executed. If the value is false, the action statement is not executed and execution continues as though a CONTINUE statement (8.3) were executed.

The execution of a function reference in the scalar logical expression may affect entities in the action statement.

NOTE 8.5

An example of an IF statement is:

```
IF (A > 0.0) A = LOG (A)
```

8.1.3 CASE construct

The **CASE construct** selects for execution at most one of its constituent blocks.

8.1.3.1 Form of the CASE construct

R808 *case-construct* **is** *select-case-stmt*
  [*case-stmt*
    block] ...
  end-select-stmt

R809 *select-case-stmt* **is** [*case-construct-name* :] SELECT CASE (*case-expr*)

R810 *case-stmt* **is** CASE *case-selector* [*case-construct-name*]

R811 *end-select-stmt* **is** END SELECT [*case-construct-name*]

Constraint: If the *select-case-stmt* of a *case-construct* is identified by a *case-construct-name*, the corresponding *end-select-stmt* shall specify the same *case-construct-name*. If the *select-case-stmt* of a *case-construct* is not identified by a *case-construct-name*, the corresponding *end-select-stmt* shall not specify a *case-construct-name*. If a *case-stmt* is identified by a *case-construct-name*, the corresponding *select-case-stmt* shall specify the same *case-construct-name*.

R812 *case-expr* **is** *scalar-int-expr*
or *scalar-char-expr*
or *scalar-logical-expr*

R813 *case-selector* **is** (*case-value-range-list*)
or DEFAULT

Constraint: No more than one of the selectors of one of the CASE statements shall be DEFAULT.

R814 *case-value-range* **is** *case-value*
or *case-value* :
or : *case-value*
or *case-value* : *case-value*

R815 *case-value* **is** *scalar-int-initialization-expr*
or *scalar-char-initialization-expr*
or *scalar-logical-initialization-expr*

Constraint: For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For character type, length differences are allowed, but the kind type parameters shall be the same.

Constraint: A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.

Constraint: For a given *case-construct*, the *case-value-range*s shall not overlap; that is, there shall be no possible value of the *case-expr* that matches more than one *case-value-range*.

8.1.3.2 Execution of a CASE construct

The execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting value is called the **case index**. For a case value range list, a match occurs if the case index matches any of the case value ranges in the list. For a case index with a value of *c*, a match is determined as follows:

(1) If the case value range contains a single value *v* without a colon, a match occurs for data type logical if the expression *c* .EQV. *v* is true, and a match occurs for data type integer or character if the expression *c* .EQ. *v* is true.

(2) If the case value range is of the form *low* : *high*, a match occurs if the expression *low* .LE. *c* .AND. *c* .LE. *high* is true.

(3) If the case value range is of the form *low* :, a match occurs if the expression *low* .LE. *c* is true.

(4) If the case value range is of the form : *high*, a match occurs if the expression *c* .LE. *high* is true.

(5) If no other selector matches and a DEFAULT selector is present, it matches the case index.

(6) If no other selector matches and the DEFAULT selector is absent, there is no match.

The block following the CASE statement containing the matching selector, if any, is executed. This completes execution of the construct.

At most one of the blocks of a CASE construct is executed.

A CASE statement shall not be a branch target statement. It is permissible to branch to an END SELECT statement only from within the CASE construct.

8.1.3.3 Examples of CASE constructs

NOTE 8.6

An integer signum function:

```
INTEGER FUNCTION SIGNUM (N)
SELECT CASE (N)
CASE (:-1)
   SIGNUM = -1
CASE (0)
   SIGNUM = 0
CASE (1:)
   SIGNUM = 1
END SELECT
END
```

NOTE 8.7

A code fragment to check for balanced parentheses:

```
CHARACTER (80) :: LINE
   ...
LEVEL=0
DO I = 1, 80
   CHECK_PARENS: SELECT CASE (LINE (I:I))
   CASE ('(')
      LEVEL = LEVEL + 1
   CASE (')')
      LEVEL = LEVEL - 1
      IF (LEVEL .LT. 0) THEN
         PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
         EXIT
      END IF
   CASE DEFAULT
      ! Ignore all other characters
   END SELECT CHECK_PARENS
END DO
IF (LEVEL .GT. 0) THEN
   PRINT *, 'MISSING RIGHT PARENTHESIS'
END IF
```

NOTE 8.8

The following three fragments are equivalent:

```
IF (SILLY .EQ. 1) THEN
   CALL THIS
ELSE
   CALL THAT
END IF
```

```
SELECT CASE (SILLY .EQ. 1)
CASE (.TRUE.)
   CALL THIS
CASE (.FALSE.)
   CALL THAT
END SELECT
```

NOTE 8.8 ***(Continued)***

```
SELECT CASE (SILLY)
CASE DEFAULT
   CALL THAT
CASE (1)
   CALL THIS
END SELECT
```

NOTE 8.9

A code fragment showing several selections of one block:

```
SELECT CASE (N)
CASE (1, 3:5, 8)  ! Selects 1, 3, 4, 5, 8
   CALL SUB
CASE DEFAULT
   CALL OTHER
END SELECT
```

8.1.4 DO construct

The DO construct specifies the repeated execution of a sequence of executable constructs. Such a repeated sequence is called a **loop**. The EXIT and CYCLE statements may be used to modify the execution of a loop.

The number of iterations of a loop may be determined at the beginning of execution of the DO construct, or may be left indefinite ("DO forever" or DO WHILE). In either case, an EXIT statement (8.1.4.4.4) anywhere in the DO construct may be executed to terminate the loop immediately. A particular iteration of the loop may be curtailed by executing a CYCLE statement (8.1.4.4.3).

8.1.4.1 Forms of the DO construct

The DO construct may be written in either a block form or a nonblock form.

R816 *do-construct* **is** *block-do-construct*
or *nonblock-do-construct*

8.1.4.1.1 Form of the block DO construct

R817 *block-do-construct* **is** *do-stmt*
do-block
end-do

R818 *do-stmt* **is** *label-do-stmt*
or *nonlabel-do-stmt*

R819 *label-do-stmt* **is** *do-construct-name* :] DO *label* [*loop-control*]

R820 *nonlabel-do-stmt* **is** [*do-construct-name* :] DO [*loop-control*]

R821 *loop-control* **is** [,] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■
■ [, *scalar-int-expr*]
or [,] WHILE (*scalar-logical-expr*)

R822 *do-variable* **is** *scalar-int-variable*

Constraint: The *do-variable* shall be a named scalar variable of type integer.

R823 *do-block* **is** *block*

R824 *end-do* **is** *end-do-stmt*
or *continue-stmt*

R825	*end-do-stmt*	**is**	END DO [*do-construct-name*]

Constraint: If the *do-stmt* of a *block-do-construct* is identified by a *do-construct-name*, the corresponding *end-do* shall be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a *block-do-construct* is not identified by a *do-construct-name*, the corresponding *end-do* shall not specify a *do-construct-name*.

Constraint: If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.

Constraint: If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the same *label*.

8.1.4.1.2 Form of the nonblock DO construct

R826	*nonblock-do-construct*	**is**	*action-term-do-construct*
		or	*outer-shared-do-construct*
R827	*action-term-do-construct*	**is**	*label-do-stmt*
			do-body
			do-term-action-stmt
R828	*do-body*	**is**	[*execution-part-construct*] ...
R829	*do-term-action-stmt*	**is**	*action-stmt*

Constraint: A *do-term-action-stmt* shall not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

Constraint: The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer to the same label.

R830	*outer-shared-do-construct*	**is**	*label-do-stmt*
			do-body
			shared-term-do-construct
R831	*shared-term-do-construct*	**is**	*outer-shared-do-construct*
		or	*inner-shared-do-construct*
R832	*inner-shared-do-construct*	**is**	*label-do-stmt*
			do-body
			do-term-shared-stmt
R833	*do-term-shared-stmt*	**is**	*action-stmt*

Constraint: A *do-term-shared-stmt* shall not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

Constraint: The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmt*s of the *shared-term-do-construct* shall refer to the same label.

The *do-term-action-stmt*, *do-term-shared-stmt*, or *shared-term-do-construct* following the *do-body* of a nonblock DO construct is called the **DO termination** of that construct.

Within a scoping unit, all DO constructs whose DO statements refer to the same label are nonblock DO constructs, and are said to share the statement identified by that label.

8.1.4.2 Range of the DO construct

The **range** of a block DO construct is the *do-block*, which shall satisfy the rules for blocks (8.1.1). In particular, transfer of control to the interior of such a block from outside the block is prohibited. It is permitted to branch to the *end-do* of a block DO construct only from within the range of that DO construct.

The range of a nonblock DO construct consists of the *do-body* and the following DO termination. The end of such a range is not bounded by a particular statement as for the other executable constructs (e.g., END IF); nevertheless, the range satisfies the rules for blocks (8.1.1). Transfer of control into the *do-body* or to the DO termination from outside the range is prohibited; in particular, it is permitted to branch to a *do-term-shared-stmt* only from within the range of the corresponding *inner-shared-do-construct*.

8.1.4.3 Active and inactive DO constructs

A DO construct is either **active** or **inactive**. Initially inactive, a DO construct becomes active only when its DO statement is executed.

Once active, the DO construct becomes inactive only when the construct it specifies is terminated (8.1.4.4.4). When an active DO construct becomes inactive, the *do-variable*, if any, retains its last defined value.

8.1.4.4 Execution of a DO construct

A DO construct specifies a loop, that is, a sequence of executable constructs that is executed repeatedly. There are three phases in the execution of a DO construct: initiation of the loop, execution of the loop range, and termination of the loop.

8.1.4.4.1 Loop initiation

When the DO statement is executed, the DO construct becomes active. If *loop-control* is

[,] $do\text{-}variable = scalar\text{-}int\text{-}expr_1$, $scalar\text{-}int\text{-}expr_2$ [, $scalar\text{-}int\text{-}expr_3$]

the following steps are performed in sequence:

(1) The initial parameter m_1, the terminal parameter m_2, and the incrementation parameter m_3 are of type integer with the same kind type parameter as the *do-variable*. Their values are established by evaluating $scalar\text{-}int\text{-}expr_1$, $scalar\text{-}int\text{-}expr_2$, and $scalar\text{-}int\text{-}expr_3$, respectively, including, if necessary, conversion to the kind type parameter of the *do-variable* according to the rules for numeric conversion (Table 7.10). If $scalar\text{-}int\text{-}expr_3$ does not appear, m_3 has the value 1. The value m_3 shall not be zero.

(2) The DO variable becomes defined with the value of the initial parameter m_1.

(3) The **iteration count** is established and is the value of the expression $(m_2 - m_1 + m_3)/m_3$, unless that value is negative, in which case the iteration count is 0.

NOTE 8.10

The iteration count is zero whenever:

$m_1 > m_2$ and $m_3 > 0$, or
$m_1 < m_2$ and $m_3 < 0$.

If *loop-control* is omitted, no iteration count is calculated. The effect is as if a large positive iteration count, impossible to decrement to zero, were established. If *loop-control* is [,] WHILE (*scalar-logical-expr*), the effect is as if *loop-control* were omitted and the following statement inserted as the first statement of the *do-block*:

IF (.NOT. (*scalar-logical-expr***)) EXIT**

At the completion of the execution of the DO statement, the execution cycle begins.

8.1.4.4.2 The execution cycle

The **execution cycle** of a DO construct consists of the following steps performed in sequence repeatedly until termination:

(1) The iteration count, if any, is tested. If it is zero, the loop terminates and the DO construct becomes inactive. If *loop-control* is [,] WHILE (*scalar-logical-expr*), the *scalar-logical-expr* is evaluated; if the value of this expression is false, the loop terminates and the DO construct becomes inactive. If, as a result, all of the DO constructs sharing the *do-term-shared-stmt* are inactive, the execution of all of these constructs is complete. However, if some of the DO constructs sharing the *do-term-shared-stmt* are active, execution continues with step (3) of the execution cycle of the active DO construct whose DO statement was most recently executed.

(2) If the iteration count is nonzero, the range of the loop is executed.

(3) The iteration count, if any, is decremented by one. The DO variable, if any, is incremented by the value of the incrementation parameter m_3.

Except for the incrementation of the DO variable that occurs in step (3), the DO variable shall neither be redefined nor become undefined while the DO construct is active.

8.1.4.4.3 CYCLE statement

Step (2) in the above execution cycle may be curtailed by executing a CYCLE statement from within the range of the loop.

R834 *cycle-stmt* **is** CYCLE [*do-construct-name*]

Constraint: If a *cycle-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.

A CYCLE statement belongs to a particular DO construct. If the CYCLE statement refers to a DO construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct in which it appears.

Execution of a CYCLE statement causes immediate progression to step (3) of the current execution cycle of the DO construct to which it belongs. If this construct is a nonblock DO construct, the *do-term-action-stmt* or *do-term-shared-stmt* is not executed.

In a block DO construct, a transfer of control to the *end-do* has the same effect as execution of a CYCLE statement belonging to that construct. In a nonblock DO construct, transfer of control to the *do-term-action-stmt* or *do-term-shared-stmt* causes that statement or construct itself to be executed. Unless a further transfer of control results, step (3) of the current execution cycle of the DO construct is then executed.

8.1.4.4.4 Loop termination

The EXIT statement provides one way of terminating a loop.

R835 *exit-stmt* **is** EXIT [*do-construct-name*]

Constraint: If an *exit-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.

An EXIT statement belongs to a particular DO construct. If the EXIT statement refers to a DO construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct in which it appears.

The loop terminates, and the DO construct becomes inactive, when any of the following occurs:

(1) Determination that the iteration count is zero or the *scalar-logical-expr* is false, when tested during step (1) of the above execution cycle

(2) Execution of an EXIT statement belonging to the DO construct

(3) Execution of an EXIT statement or a CYCLE statement that is within the range of the DO construct, but that belongs to an outer DO construct

(4) Transfer of control from a statement within the range of a DO construct to a statement that is neither the *end-do* nor within the range of the same DO construct

(5) Execution of a RETURN statement within the range of the DO construct

(6) Execution of a STOP statement anywhere in the program; or termination of the program for any other reason.

When a DO construct becomes inactive, the DO variable, if any, of the DO construct retains its last defined value.

8.1.4.5 Examples of DO constructs

NOTE 8.11

The following program fragment computes a tensor product of two arrays:

```
DO I = 1, M
   DO J = 1, N
      C (I, J) = SUM (A (I, J, :) * B (:, I, J))
   END DO
END DO
```

NOTE 8.12

The following program fragment contains a DO construct that uses the WHILE form of *loop-control*. The loop will continue to execute until an end-of-file or input/output error is encountered, at which point the DO statement terminates the loop. When a negative value of X is read, the program skips immediately to the next READ statement, bypassing most of the range of the loop.

```
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
DO WHILE (IOS .EQ. 0)
   IF (X .GE. 0.) THEN
      CALL SUBA (X)
      CALL SUBB (X)
           ...
      CALL SUBZ (X)
   ENDIF
   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
END DO
```

NOTE 8.13

The following example behaves exactly the same as the one in Note 8.12. However, the READ statement has been moved to the interior of the range, so that only one READ statement is needed. Also, a CYCLE statement has been used to avoid an extra level of IF nesting.

```
DO     ! A "DO WHILE + 1/2" loop
   READ (IUN, ’(1X, G14.7)’, IOSTAT = IOS) X
   IF (IOS .NE. 0) EXIT
   IF (X < 0.) CYCLE
   CALL SUBA (X)
   CALL SUBB (X)
      . . .
   CALL SUBZ (X)
END DO
```

NOTE 8.14

Additional examples of DO constructs are in C.5.3.

8.2 Branching

Branching is used to alter the normal execution sequence. A branch causes a transfer of control from one statement in a scoping unit to a labeled branch target statement in the same scoping unit. A **branch target statement** is an *action-stmt*, an *if-then-stmt*, an *end-if-stmt*, a *select-case-stmt*, an *end-select-stmt*, a *do-stmt*, an *end-do-stmt*, a *forall-construct-stmt*, a *do-term-action-stmt*, a *do-term-shared-stmt*, or a *where-construct-stmt*.

It is permissible to branch to an END SELECT statement only from within its CASE construct.

It is permissible to branch to an END IF statement only from within its IF construct.

It is permissible to branch to an *end-do-stmt* or a *do-term-action-stmt* only from within its DO construct. It is permissible to branch to a *do-term-shared-stmt* only from within its *inner-shared-do-construct*.

8.2.1 Statement labels

A **statement label** provides a means of referring to an individual statement. Only branch target statements (8.2), FORMAT statements, and DO terminations shall be referred to by the use of statement labels (3.2.4).

8.2.2 GO TO statement

R836 *goto-stmt* **is** GO TO *label*

Constraint: The *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the *goto-stmt*.

Execution of a GO TO statement causes a transfer of control so that the branch target statement identified by the label is executed next.

8.2.3 Computed GO TO statement

R837 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

Constraint: Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same scoping unit as the *computed-goto-stmt*.

The same statement label may appear more than once in a label list.

Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If this value is *i* such that 1<=*i*<=*n* where *n* is the number of labels in *label-list*, a transfer of control occurs so that the next statement executed is the one identified by the *i*th label in the list of labels. If *i* is less than 1 or greater than *n*, the execution sequence continues as though a CONTINUE statement were executed.

8.2.4 Arithmetic IF statement

R838 *arithmetic-if-stmt* **is** IF (*scalar-numeric-expr*) *label* , *label* , *label*

Constraint: Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.

Constraint: The *scalar-numeric-expr* shall not be of type complex.

The same label may appear more than once in one arithmetic IF statement.

Execution of an arithmetic IF statement causes evaluation of the numeric expression followed by a transfer of control. The branch target statement identified by the first label, the second label, or the third label is executed next as the value of the numeric expression is less than zero, equal to zero, or greater than zero, respectively.

8.3 CONTINUE statement

Execution of a CONTINUE statement has no effect.

R839 *continue-stmt* **is** CONTINUE

8.4 STOP statement

R840 *stop-stmt* **is** STOP [*stop-code*]

R841 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

Constraint: *scalar-char-constant* shall be of type default character.

Execution of a STOP statement causes termination of execution of the program. At the time of termination, the stop code, if any, is available in a processor-dependent manner. Leading zero digits in the stop code are not significant.