5

Control Statements

The preceding chapters cover only straightforward CL programming in which the statements are executed sequentially. Most programs (in any language) do not run that way. They perform decisions, execute loops, and so on. CL also has commands to control execution of the program.

The IF Command

First and foremost is the `IF` command. You use the `IF` command to perform a decision based on the result of a test. The nature of this test is always a logical expression.

Simple Logical Expressions

A simple logical expression has one comparison operator between two expressions of any other type. For example, if you say, “the option number is equal to 5,” you have a comparison operator (“is equal to”) and two other expressions (“the option number” and “5”). In CL, this could be coded as shown in Figure 5.1.

```
DCL &option     *DEC     2
IF (&option *EQ 5) CHGVAR &x 'A'
```

Figure 5.1: Expression with comparison operator

There are several points worth noting:

- The `IF` command has two parameters: `COND` (condition) and `THEN` (what to do if the condition is true). The example shown in Figure 5.1 has been given without keywords. With keywords, it would look like the example shown in Figure 5.2.

```
IF COND(&OPTION *EQ 5) THEN(CHGVAR VAR(&X) VALUE('A'))
```

Figure 5.2: Comparison expression without keywords

As you can see, `CHGVAR VAR(&X) VALUE('A')` is executed if the condition is true.

- The comparison operator in the example is `*EQ` (equal). Actually, `*EQ` is just one of the comparison operators you can use in a logical expression. See Table 5.1.
- The two expressions being compared must be of the same type. In the example, both have decimal values.
- The `COND` parameter must have a logical expression that can be evaluated to either `TRUE` or `FALSE`.

Table 5.1: Logical Operators

Comparison	Description
*EQ or =	Equal to
*NE or ¬=	Not equal to
*LT or <	Less than
*LE or <=	Less than or equal to
*NL or ¬<	Not less than
*GT or >	Greater than
*GE or >=	Greater than or equal to
*NG or ¬>	Not greater than

Because logical variables have only two possible values—`TRUE` or `FALSE`—you can take a shortcut when you test logical variables in conditions. Figure 5.3 shows an example.

```
/* Declare indicators 03 and 12 from a display file    */
/* These indicators turn on when F3 or F12 is pressed */
DCL &in03       *LGL     1
DCL &in12       *LGL     1

/* The two IFs that follow are equivalent */
/* Use either style in your programs.     */
IF (&in03 *EQ '1' *OR &in12 *EQ '1') ...

IF (&in03 *OR &in12) ...
```

Figure 5.3: Using a comparison operator with logical variables

To test whether or not `&IN03` is `FALSE`, you can code the expression as shown in Figure 5.4.

```
IF (&in03 *EQ '0') ...

/* Or the shortcut: */
IF (*NOT &in03) ...
```

Figure 5.4: An example of testing for FALSE

Complex Logical Expressions

The `COND` parameter can contain a complex logical expression that consists of several simple expressions connected with the logical operations `*AND`, `*OR`, or `*NOT`. Table 5.2 summarizes the logical value (T=`TRUE`, F=`FALSE`) of these logical operators. For example, a `TRUE` value and a `FALSE` value combined with an `*AND` operator yields a false value.

Table 5.2: Results of Complex Logical Expressions

*AND			*OR			*NOT	
First Value	Second Value	Result	First Value	Second Value	Result	Value	Result
T	T	T	T	T	T	T	F
T	F	F	T	F	T	F	T
F	T	F	F	T	T		
F	F	F	F	F	F		

For example, you can say, “Send a warning message to the system operator if the option number is 2 or 3.” In CL, this is coded as shown in Figure 5.5.

The COND parameter can contain extremely complicated logical expressions with many *ANDs, *ORs, and *NOTs, even using parentheses to group subexpressions when it becomes necessary to alter the natural order of evaluation. However, in all cases, the condition must evaluate to either TRUE or FALSE.

```
DCL &option     *DEC     2

IF (&option *EQ 2 *OR +
    &option *EQ 3     ) +
   SNDPGMMSG MSG('Warning') TOMSGQ(qsysopr)
```

*Figure 5.5: Example of complex logical expression using *OR*

Remember that:

- *NOT is evaluated first, whenever present.
- *AND is evaluated next, if present.
- *OR is evaluated last, if present.
- *NOT reverses the TRUE or FALSE value of the expression.
- *AND yields a TRUE result if both expressions are TRUE. In all other cases, it yields FALSE.
- *OR yields a FALSE result if both expressions are FALSE. In all other cases, it yields TRUE.

The DO and ENDDO Commands

An IF statement can be used to execute one command if the condition is met. With the DO and ENDDO commands, you can change the function of the IF command so that a group of statements is executed if the condition is met. This group of statements must be enclosed between a DO and an ENDDO command pair.

Single-Level DO Groups

Suppose you want to execute four commands when an option number is equal to three. You would need to code a CL routine like the one shown in Figure 5.6.

```
DCL &option       *CHAR    1

IF (&option *EQ '3') DO
   SNDPGMMSG MSGID(cpf9898) MSGF(qcpfmsg) +
             MSGDTA('You''ve taken option 3') +
             TOPGMQ(*EXT) MSGTYPE(*STATUS)
   CHGVAR &option ' '
   CALL abc (&this &that)
   CALL def (&the &other)
ENDDO
```

Figure 5.6: An example of single-level DO group

The four commands enclosed in the box (SNDPGMMSG, CHGVAR, CALL, and CALL) are executed if &OPTION equals '3'. Note that the DO is placed in the IF statement's THEN parameter, and the ENDDO is isolated.

Nesting DO Groups

Now that you know how to create a DO group, you should also know that you can code an IF statement inside the DO group. As shown in Figure 5.7, this second IF statement can open another DO group completely nested within the first one.

```
IF (&a *EQ '1') DO
   *
   *
   *
   IF (&b *EQ '2') DO
      *
      *
      *
   ENDDO
   *
   *
   *
ENDDO
```

Figure 5.7: An example of a nested DO group with indenting for easier reading

You can take nested DO groups to a maximum of 25 levels. The source code shown in Figure 5.7 takes advantage of CL's free-format nature to indent the code (thus revealing the levels of nested DOs). If you let the command prompter format the code for you, however, all commands are aligned the same. The result, as shown in Figure 5.8, is less readable code.

```
IF         COND(&A *EQ '1') THEN(DO)
*
*
*
IF         COND(&B *EQ '2') THEN(DO)
*
*
*
ENDDO
*
*
*
ENDDO
```

Figure 5.8: An example of a nested DO group without benefit of indenting

Nesting IF Commands

CL lets you nest `IF` commands up to 25 levels deep. You can code an `IF` statement within the `THEN` parameter of another `IF`, as shown in Figures 5.9 and 5.10.

```
IF (&a *EQ &b) IF (&c *EQ &d) CALL xyz
```

Figure 5.9: An example of nested IF commands (no keywords)

```
IF         COND(&A *EQ &B) THEN(IF COND(&C *EQ &D) +
                              THEN(CALL PGM(XYZ)))
```

Figure 5.10: Example of nested IF commands with keywords

In this example, the `CALL` command runs only if `&A` equals `&B` and `&C` equals `&D`. This particular case could be coded more clearly with the `*AND` logical operator in a single if statement, as shown in Figure 5.11.

```
IF (&a *EQ &b *AND &c *EQ &d) CALL xyz
```

*Figure 5.11: Alternative method of coding nested IFs using *AND*

There are cases, however, when nesting the conditions separately is perfectly valid and is the only way to code what you want. In this case, you should consider using the `DO` command for the outer `IF` statement, as shown in Figure 5.12.

```
   IF (&a *EQ &b) DO
      IF (&c *EQ &d) CALL xyz
      *
      *
      *
   ENDDO
```

Figure 5.12: An example of using the DO command for nested IF statements

The ELSE Command

The `ELSE` command works with the `IF` command. It provides instructions about what to do when the condition in the `IF` command tests `FALSE`. `ELSE` is optional.

For example, suppose that a user enters a “Y” or an “N” to a question presented by a display file (the variable name is `&ANSWER`). The CL program will use the response, but it first must be translated to `'*YES'` or `'*NO'`. The `IF` and `ELSE` pair shown in Figure 5.13 will take care of this problem.

```
   DCL &answer     *CHAR    1
   DCL &yesno      *CHAR    4

   IF (&answer *EQ 'Y') +
      CHGVAR &yesno '*YES'
   ELSE +
      CHGVAR &yesno '*NO'
```

Figure 5.13: An example of translating input variables to different values using IF and ELSE

The IF command evaluates the condition in the COND parameter. If true, it executes the command found in the THEN parameter (which is a CHGVAR command to assign '*YES' to variable &YESNO).

The ELSE command follows. If the condition previously tested is true, the ELSE command is skipped altogether. If the condition is not true, the system runs the command found in the CMD parameter (which is another CHGVAR command).

Using DO with ELSE

You also can use the DO/ENDDO pair with the ELSE command when you need to execute more than one command with an ELSE. See Figure 5.14.

```
IF (&a *EQ &b) DO
      *
      *
      *
   ENDDO
   ELSE DO
      *
      *
      *
   ENDDO
```

Figure 5.14: An example of using ELSE with DO/ENDDO

What you code inside each DO/ENDDO pair is up to you. Except to limit you to 25 levels of nested DO groups, CL places no restrictions on you. It is possible to have another IF. See the example shown in Figure 5.15.

```
 IF (&a *EQ &b) DO
   IF (&c *EQ &d) DO
      *
      *
      *
   ENDDO
   ELSE DO
      *
      *
      *
   ENDDO
ENDDO
ELSE DO
   IF (&e *EQ &f) DO
      *
      *
      *
   ENDDO
   ELSE DO
      *
      *
      *
   ENDDO
ENDDO
```

Figure 5.15: An example of IF commands within the ELSE construct

Note how indenting the code makes it much easier to follow the hierarchy of the various `IF`s and `ELSE`s.

The SELECT Command

The SELECT, WHEN, OTHERWISE, and ENDSELECT commands implement a case structure in CL procedures. Case structures are suited for situations in which only one of several alternatives is to be executed.

A SELECT group begins with the SELECT command and ends with the ENDSELECT command. Neither of these commands has parameters.

Each condition that must be tested is coded in the COND parameter of a WHEN command. The command to be executed is coded in the THEN parameter. You can see that WHEN is like IF in structure.

You may use the optional OTHERWISE command to execute a command when none of the conditions proves true. OTHERWISE accepts only a single parameter—CMD, which makes it similar in structure and function to ELSE.

In the example in Figure 5.16, a variable named &OPTION is tested for values of 1 through 4. If &OPTION has any of those values, one or two programs will be executed. If &OPTION has any other value, the SIGNOFF command will execute, ending the session.

```
DCL        VAR(&option) TYPE(*CHAR) LEN(1)

SELECT
   WHEN      COND(&option *EQ '1') THEN(CALL PGM(pgm1))
   WHEN      COND(&option *EQ '2') THEN(CALL PGM(pgm2))
   WHEN      COND(&option *EQ '3') THEN(CALL PGM(pgm3))
   WHEN      COND(&option *EQ '4') THEN(DO)
      CALL      PGM(pgm4)
      CALL      PGM(pgm5)
   ENDDO
   OTHERWISE CMD(SIGNOFF)
ENDSELECT
```

Figure 5.16: Case structures are implemented with SELECT and its associated commands.

The DOWHILE Command

Use the DOWHILE command to implement a top-tested loop. That is, the condition that controls the loop is tested before each iteration of the loop. If the condition is false when control reaches the DOWHILE command, the commands in the body of the loop will not be executed at all.

The DOWHILE takes one parameter—COND, which defines the condition that must be true for the loop to continue to execute. The end of the loop structure is indicated with the ENDDO command. The commands that make up the body of the loop follow DOWHILE and precede ENDDO.

In the example in Figure 5.17, three commands—two CALL commands and one CHGVAR—are governed by the DOWHILE command. If the &STATUS variable has a value of five zeros when control reaches the DOWHILE, the three inner commands will not execute. If &STATUS has a non-zero value, the loop will begin execution and continue until program PGM3 changes the value of &STATUS to a value of zeros.

```
DOWHILE   COND(&STATUS *NE '00000')
   CALL      PGM(pgm2)
   CALL      PGM(pgm3) PARM(&STATUS)
   CHGVAR    VAR(&COUNT) VALUE(&COUNT + 1)
ENDDO
```

Figure 5.17: DOWHILE defines a top-tested loop.

The DOUNTIL Command

The DOUNTIL command defines a bottom-tested loop. That is, the condition that controls the loop is tested after each iteration of the loop. The commands in the body of the loop will be executed at least once. In Figure 5.18, the loop continues to execute until program PGM3 returns a status value of five zeros.

```
DOUNTIL   COND(&STATUS *EQ '00000')
   CALL      PGM(pgm2)
   CALL      PGM(pgm3) PARM(&STATUS)
ENDDO
```

Figure 5.18: DOUNTIL defines a bottom-tested loop.

The DOFOR Command

Most programming languages have some form of the counted loop. This type of loop has a control variable, which is given an initial value and incremented or decremented with each iteration, until the control variable falls outside some acceptable range. The counted loop is defined with the DOFOR command.

DOFOR has three required parameters and one optional one. All four parameters require integer values. In the VAR parameter, provide the name of a signed or unsigned integer variable to be used for the control variable. The FROM parameter allows you to specify the value to which the control variable is to be initialized. In the TO parameter, specify the terminal value of the control variable. The last parameter, BY, is the quantity to be added to the control variable after each iteration. Specify a negative value for a descending loop.

The loop in Figure 5.19 executes four times. Variable &OFFSET assumes the following values: 3, 13, 23, and 33.

```
DCL   VAR(&OFFSET) TYPE(*INT) LEN(2)

DOFOR   VAR(&OFFSET) FROM(3) TO(33) BY(10)
 (CL commands)
ENDDO
```

Figure 5.19: DOFOR defines a counted loop.

The *DOSLTLVL Option

When DO and SELECT groups are nested, it can be difficult to match the beginning and ending commands. If you specify the *DOSLTLVL option on the CRTBNDCL (Create Bound CL Program), CRTCLMOD (Create CL Module), or CRTCLPGM (Create CL Program) commands, the CL compiler adds two columns—DO and SLT—to the compiler listing the nesting level.

The LEAVE and ITERATE Commands

The LEAVE and ITERATE commands provide further control over DOWHILE, DOUNTIL, and DOFOR looping structures. LEAVE causes an immediate exit from a loop. ITERATE passes control to the bottom of a loop.

You may specify an optional CMDLBL parameter with LEAVE and ITERATE. If you do not use the CMDLBL parameter, the LEAVE or ITERATE command applies to the innermost loop. To exit or continue an outer loop, provide a label for a DOWHILE, DOUNTIL, or DOFOR command, and refer to this label in the LEAVE or ITERATE command.

The example code segment in Figure 5.20 contains two loops, one within the other. The first two IF commands refer to the inner loop, referred to with label NEXT. The last IF refers to the outer loop, named PROMPT.

```
prompt: DOWHILE COND(*NOT &IN03)
        CHGVAR  VAR(&pos) VALUE(1)
next:   DOWHILE COND(&pos *LT 120)
        ...
        IF COND(&fname *EQ ' ') THEN(LEAVE)
        ...
        IF COND(&error *EQ '1') THEN(ITERATE)
        IF COND(&error *EQ '2') THEN(LEAVE CMDLBL(prompt))
        ...
        ENDDO
        ...
        ENDDO
```

Figure 5.20: LEAVE and ITERATE alter the normal behavior of loops.