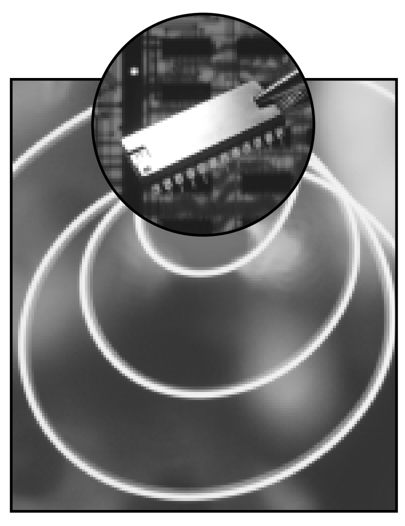


CHAPTER 4

REPORT OUTPUT AND VALIDATING INPUT

CHAPTER OUTLINE

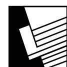
4.1 What Types of Reports Can Be Created?
4.2 What Types of Lines Can Be Included on a Report?
4.3 How Are Lines Spaced on a Page or Screen?
4.4 Example: Heading Lines on Printed Reports, No Data Input from Outside the Program, and Using External Subroutines
4.5 Example: Screen Output with Heading and Footing Lines
4.6 How Do We Include the Date on a Report?
4.7 Example: Exception Reports, Using an External Subroutine to Obtain the Current Date, Double-Spacing Detail Lines, and Printing the Page Number in a Footing Line
4.8 How Does Garbage Input Generate Output Errors?
4.9 What Types of Data Validation Can Be Done?
4.10 What Are the Differences Between Batch and On-Line Data Entry and Validation?
4.11 Example: Input Validation for a Batch Program
4.12 Example: Using an Interactive Program to Create and Validate an Input File
4.13 What Is a Menu-Driven Program?
4.14 How Do Event-Driven Programming Languages Enter and Validate Data?

CHAPTER OBJECTIVES

After completing this chapter you should be able to

1. Explain when it is appropriate to use detail, summary, exception, and query reports.
2. Describe the various types of lines that can be included on a report.
3. Explain how to space lines vertically on a page or screen.
4. Explain how to prevent scrolling on screen output.
5. Construct the logic for programs that produce paper and screen reports containing heading and/or footing lines.
6. Construct the logic for programs with single-spaced and double-spaced detail lines.
7. Construct the logic for programs that include the current date in a heading or footing line.
8. Explain and give examples of the types of errors that can occur during processing.
9. Explain how data-validation techniques prevent errors during processing and give examples of their appropriate uses.
10. Describe the difference between batch and on-line data entry and validation.
11. Construct the logic for programs that validate data already contained in a file.
12. Construct the logic for interactive programs that create and validate input files.
13. Describe how menu-driven programs are used to make programs user-friendly.
14. Describe how event-driven programming languages enter and validate data.

- In previous chapters we produced both paper and screen reports. In this chapter we explore reports in more detail with a focus on heading, footing, and detail lines.
- We begin with a discussion of the types of reports and report lines that can be generated. We examine vertical line spacing (single-spacing and double-spacing), as well as printing, or displaying, blank lines. Examples are used to explain external subroutines, including the current date on a report and spacing of detail lines.
- After studying report output, we change our focus to data input. We begin with a discussion of translation and execution errors that occur during computer processing. We discuss what data validation is and how data validation techniques can help prevent processing errors. Then we concentrate on batch and on-line input, as well as data validation in both modes. We conclude with a discussion of menu-driven and event-driven input and consider how programmers can prevent input errors in such programs.
- Before beginning the first section of the chapter, read the chapter outline and chapter objectives above. You should also read the summary and key terms and concepts at the end of the chapter. Mark those terms and concepts that you already know.



4.1 What Types of Reports Can Be Created?

Reports can be printed on paper or displayed on a screen. Regardless of the output media used, there are four basic types of reports: detail reports, summary reports, exception reports, and query reports.

Detail reports show information for every input record. Each report line may contain all or some fields from the input record. Examples of detail reports are:

- A customer list.
- A phone directory.
- Credit card statements showing all purchases and payments.
- A sales report listing all sales by salesperson and region.
- A class list.
- A report showing test scores for each student in a class.

Summary reports are very useful when detail information is not needed. Summary reports do *not* show one output line for each input record. Instead, the data are summarized. For example, regional sales managers might find detail reports listing all sales by salesperson and region to be useful. The vice president of sales is probably more interested in how regions, rather than individual salespersons, perform. Therefore, it is more useful for the vice president to have a report that summarizes the sales by region. Other examples of summary reports are:

- A report of monthly and yearly sales totals for various products.

- A report showing the total number of As, Bs, Cs, Ds, and Fs given in a class.
- A report listing the average amount charged last month by all customers.

Exception reports include some or all of the records from the input file. The information shown on the report is determined by criteria within the program or specified by the user. For example, instead of a detail report showing the number of items in stock for every item a retailer sells, an exception report might show only those items that need to be reordered. Other examples of exception reports are:

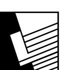
- A report listing students who are failing.
- A report listing customers who did not pay their bills last month.
- A list of salespersons who exceeded their yearly quotas.
- An error report showing input records that contain invalid data.

Query reports, typically generated from asking questions about the records contained in a database, allow the user to obtain an immediate answer to a question such as whether a customer's check was received. Other examples of query reports include:

- A report listing whether a student paid a parking fine.
- A report containing the number of parts ordered.
- A report showing Internet sites that meet particular search criteria.

This chapter focuses on detail and exception reports. We discuss summary and query reports in later chapters.

- Before continuing to the next section, answer the following question at the end of the chapter: Short Answer 1.



4.2 What Types of Lines Can Be Included on a Report?

Four types of lines can be written on a report: heading lines, footing lines, detail (data) lines, and total (summary) lines. So far we have printed only data, or detail, lines and have not printed any headings to describe the detail lines. It is important to include heading and/or footing lines in order to produce more meaningful reports. Total lines are useful for summarizing information on a report.

Heading (header) and **footing (footer)** lines identify the report and describe the information it contains. Heading lines appear at the beginning of the report and the top of each page. Footing lines are at the end of the report and the bottom of each page. There are four types of heading/footing lines:

- **Report heading/footing lines** appear only at the beginning (end) of the report, sometimes on a separate page or screen. Report heading lines include

information such as the name of the report, the date, the recipient of the report, copyright information, and so on. Report footing lines are often used to indicate that the report has ended or to show a final total.

- **Page (screen) heading/footing lines** appear on each page and contain information such as the name of the report, the page number, and the date. For screen reports, the footing line normally contains instructions for the user explaining how to continue displaying the report on the next screen.
- **Column heading lines** describe the fields that appear on the report.
- **Control heading lines** separate one group of data from another on a report.

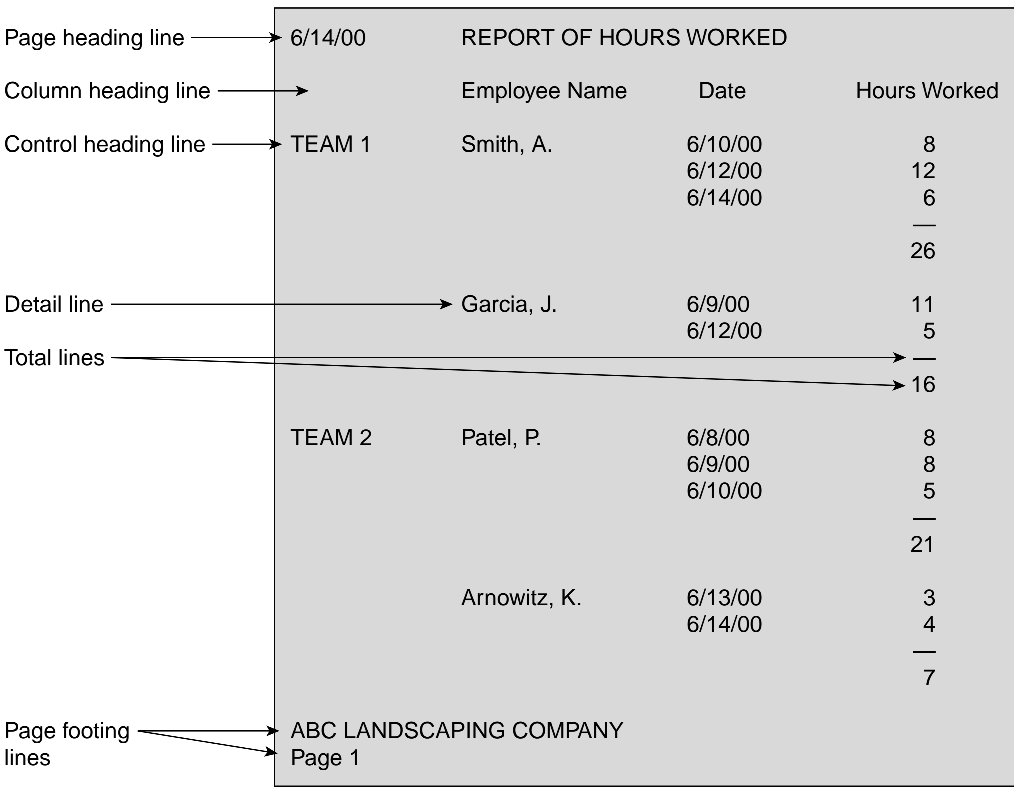
Detail (data) lines comprise the body of the report. A detail line holds information from the data contained on one or more input records.

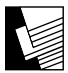
Total (summary) lines summarize information contained on a file or report.

The detail report shown in Figure 4–A demonstrates the use of some of the types of lines described above. It is important to keep in mind that reports may contain all *or* some types of lines.

- Before continuing to the next section, answer the following questions at the end of the chapter: True/False 1; Multiple Choice 1.

Figure 4–A
Report Lines





4.3 How Are Lines Spaced on a Page or Screen?

To print heading lines at the top of a page or screen, the computer needs to know how to find the top of the paper or screen. Assuming the printer is set up correctly, it knows where the top of the paper is. The program, however, must still tell the printer to start a new page. The computer uses **carriage control characters** to communicate with the printer. When a line is written, the program tells the operating system which carriage control character to send to the printer. This is accomplished differently depending on the programming language being used.

Similarly, for screen output, the program tells the computer where to place information on the screen. In this book, we will *not* use specific carriage control characters and programming language instructions to specify the location on the page or screen. Instead, we will use a more generic approach. For example, the instructions WRITE A LINE AT TOP OF PAGE or DISPLAY A LINE AT TOP OF SCREEN indicate that the line is to be written or displayed as the first line on a new page or screen.

The most common ways to display or print lines are to single- or double-space them. **Single-spaced lines** are displayed or printed one line after another with no blank lines between them. **Double-spaced lines** are displayed or printed with one blank line before each line with information. Because each programming language has different instructions for single- and double-spacing, in this book we use the following generic conventions:

- To single space: WRITE A LINE or DISPLAY A LINE.
- To double space: WRITE A LINE DOUBLE-SPACED or DISPLAY A LINE DOUBLE-SPACED.
- To tell the computer to move spaces to the output area and write or display the resulting blank line: WRITE A BLANK LINE or DISPLAY A BLANK LINE.

How does the computer know how many lines to put on a page? The printer will print one line at a time and continue printing line after line (even over perforations in continuous-form computer paper) unless the program tells it to do otherwise. For this reason, the programmer uses a **lines counter** to keep track of the number of lines written on a page. Then, after the desired number of lines is written, a program instruction tells the computer to write the next line at the top of a new page. The example shown in Section 4.4 illustrates this concept.

The number of lines that can fit on a page or screen depends on the size of the paper, the font, and the font size selected. A **font** describes how the letters, numbers, and other characters that you use will look. The **font size** is how large each character is. Figure 4–B shows some sample fonts and font

Figure 4–B

10 point Times New Roman font
12 point Arial font
20 point Arial font
28 point Times New Roman font

sizes. Font size is measured in the traditional printer's measure of points. One point is 1/72 of an inch; therefore, uppercase letters in a 72-point font would be one inch high. When using a standard 12-point font, approximately six lines of text fit in a vertical inch. This means for an 8½-by-11-inch piece of paper, 66 lines (not counting space for margins) will fit if you are using a 12-point font. If you have one-inch margins, then 54 lines will fit.

What happens when an output report is produced as a screen display rather than on paper? Typically, a screen display can hold fewer lines than a piece of paper. When the computer runs out of room on the screen, it moves the display up one line at a time. In so doing, the information at the top of the screen is erased to make room for new information at the bottom. This is called **scrolling.** Normally, scrolling occurs much faster than you can possibly read, so the screen appears as a fast-moving blur. To prevent scrolling and display output screen-by-screen (page-by-page), programs count the number of lines displayed on the screen. When the screen limit is reached, the program pauses to allow the user time to read the screen. The program then clears the screen and displays the next screen. Section 4.5 contains an example that illustrates this.

Printer and **display layout charts** are used by analysts to design printed output and screen displays. The columns and rows on the chart are numbered to show the spacing on the page or display. The number of columns and rows used by the analyst depends on the paper, special form, or screen format being designed.

The following conventions are followed on printer and display layout charts in this book:

- Xs show nonnumeric fields.
- 9s show numeric fields.
- Zs in a numeric field indicate **zero suppression** (that is, replacing leading zeros with spaces or blanks).
- All literals or constants such as heading lines, dollar signs, and decimal points are included exactly as they are to appear on the report.
- Dates shown as MM/DD/YY indicate a two-digit month followed by a slash (/) followed by a two-digit day followed by a slash (/) followed by a two-digit year.

The following is an example of a layout chart. Each page of the report contains one page heading line (line 1 on the chart). The month starts in

Layout Chart

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789	6 0123456789	7 0123456789
1	MM/DD/YY		MONTHLY PHONE BILLING REPORT					
2								
3	CUSTOMER		TELEPHONE	NUMBER	LENGTH	OPERATOR	CHARGE	
4	NAME		NUMBER	CALLED	OF	ASSISTED	FOR	
5					CALL		CALL	
6								
7	XXXXXXXXXXXXXXXXXXXX		XXX-XXXXXXX	XXX-XXXXXXXXX	Z,ZZ9	YES	$Z,ZZ9.99	
8	XXXXXXXXXXXXXXXXXXXX		XXX-XXXXXXX	XXX-XXXXXXXXX	Z,ZZ9	NO	$Z,ZZ9.99	

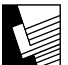
column 1, and MONTHLY PHONE BILLING REPORT starts in column 25. The second line of the page is blank. Lines 3 to 5 are used for column headings. Line 6 is blank. Two detail lines are shown (lines 7 and 8) to indicate that the detail lines are single-spaced. If the lines had been double-spaced, the chart would show a blank line between them. The first field in the detail line, CUSTOMER NAME, is nonnumeric and consists of at most 19 characters (there are 19 Xs). If the name is fewer than 19 characters, then it is left-justified, and the remaining space is filled with blank characters. The TELEPHONE NUMBER is the next field on the detail line. It contains three characters followed by a hyphen followed by seven more characters. The NUMBER CALLED column is the same format as the telephone number column. The LENGTH OF CALL column is shown using the format Z,ZZ9. Each of the Zs and the 9 will be replaced by a digit when the length is printed or displayed. Leading zeros are replaced by blanks. Since four digits are shown, the column will contain a number in the range 1 to 9,999. If the comma is not necessary, it is not used, and a blank is shown instead.

Sample Output in LENGTH OF CALL column

Input	Output Displayed
1156	1,156
0009	9

OPERATOR ASSISTED contains the constant YES or NO. Finally, the CHARGE FOR CALL column contains a dollar amount with suppression of leading zeros.

- Before continuing to the next section, answer the following questions at the end of the chapter: True/False 2, 3; Multiple Choice 2, 3, 4, 5, 6.



4.4 Example: **Heading Lines on Printed Reports, No Data Input from Outside the Program, and Using External Subroutines**

Usually it makes no sense to write a program when there is no input file, but occasionally you will come across such a program. The one shown in this example prints the squares, cubes, and square roots of the integers between 1 and 500.

Output consists of the printed report described in the following layout chart.

Layout Chart

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789
1	PAGE Z9				
2	INTEGER	SQUARE	CUBE	SQUARE	ROOT
3					
4	ZZ9	ZZZZZ9	ZZZZZZZZZZ9	ZZ.Z	Z9
5	ZZ9	ZZZZZ9	ZZZZZZZZZZ9	ZZ.Z	Z9

Each detail line contains the integer, the square of the integer, the cube of the integer, and the square root of the integer. Headings are as shown. At most, 25 detail lines are printed per page.

There is no input file used in this program.

The data dictionary for this example follows. Because all storage locations are defined as global, the scope is not included in the dictionary. The program uses no input file, therefore no input area is defined.

Data Dictionary

Output Area—INTEGER REPORT
Work Area

Name	Description	Type	Initial Value	Calculation
HEADING1	See layout chart—line 1			
"PAGE"		Constant		
PAGE-NUMBER		Numeric	0	1 is added for each page
HEADING2	See layout chart—line 2			
"INTEGER"		Constant		
"SQUARE"		Constant		
"CUBE"		Constant		
"SQUARE ROOT"		Constant		
INTEGER-COUNTER	Integer being "worked with"	Numeric	1	1 is added for each pass of the loop
SQUARE	Square of integer	Numeric		INTEGER-COUNTER * INTEGER-COUNTER
CUBE	Cube of integer	Numeric		SQUARE * INTEGER-COUNTER
SQUARE-ROOT	Square root of integer	Numeric		Square root of INTEGER-COUNTER
LINES-COUNTER	Number of detail, heading, and blank lines written on the page	Numeric	0	1 is added for each line written on the report
MAX-LINES	Maximum lines per page	Numeric	28	

The hierarchy chart, flowchart, and pseudocode are shown in Figure 4–C.

MAINLINE first does INITIALIZE. It then does CALCULATE until INTEGER-COUNTER is greater than 500. (INTEGER-COUNTER is the work area that tells the computer the integer that is being "worked on.") This is a counter-controlled loop. After completing the loop, files are closed, and the program terminates.

Look at INITIALIZE. The first instruction assigns initial values to work areas. The initial values for HEADING1 and HEADING2 only show

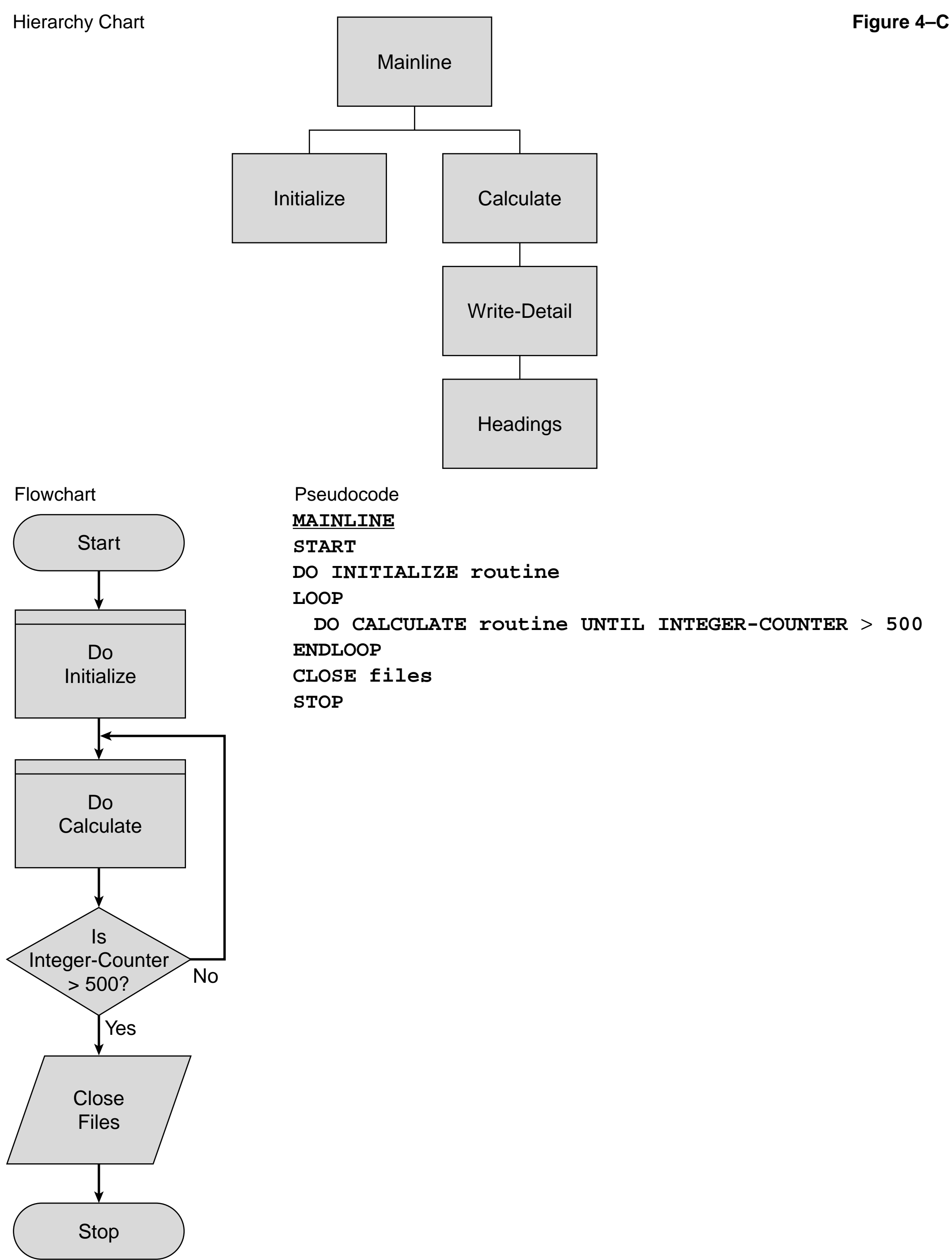


Figure 4–C

constants and variables (field names). They don't show the spaces between fields. Although not shown in the flowchart or pseudocode, the spaces must be included when you code the program. Besides heading lines, there are other storage locations defined in the work area (under OTHER). The first field is SQUARE. This location stores the square of the integer being "worked on." CUBE and SQUARE-ROOT are used to hold the cube and the square root of the integer. INTEGER-COUNTER is used by the counter-controlled

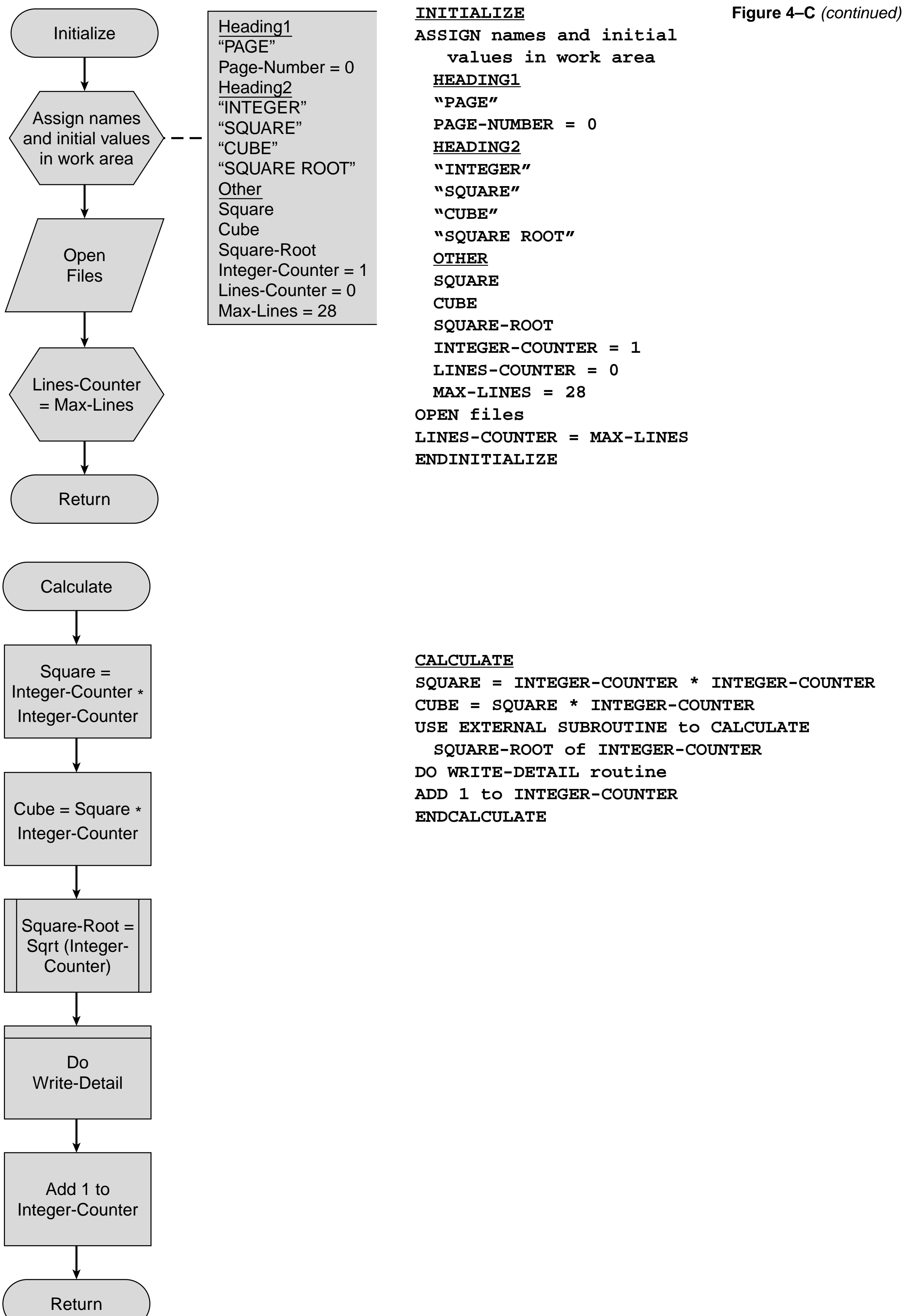


Figure 4–C *(continued)*

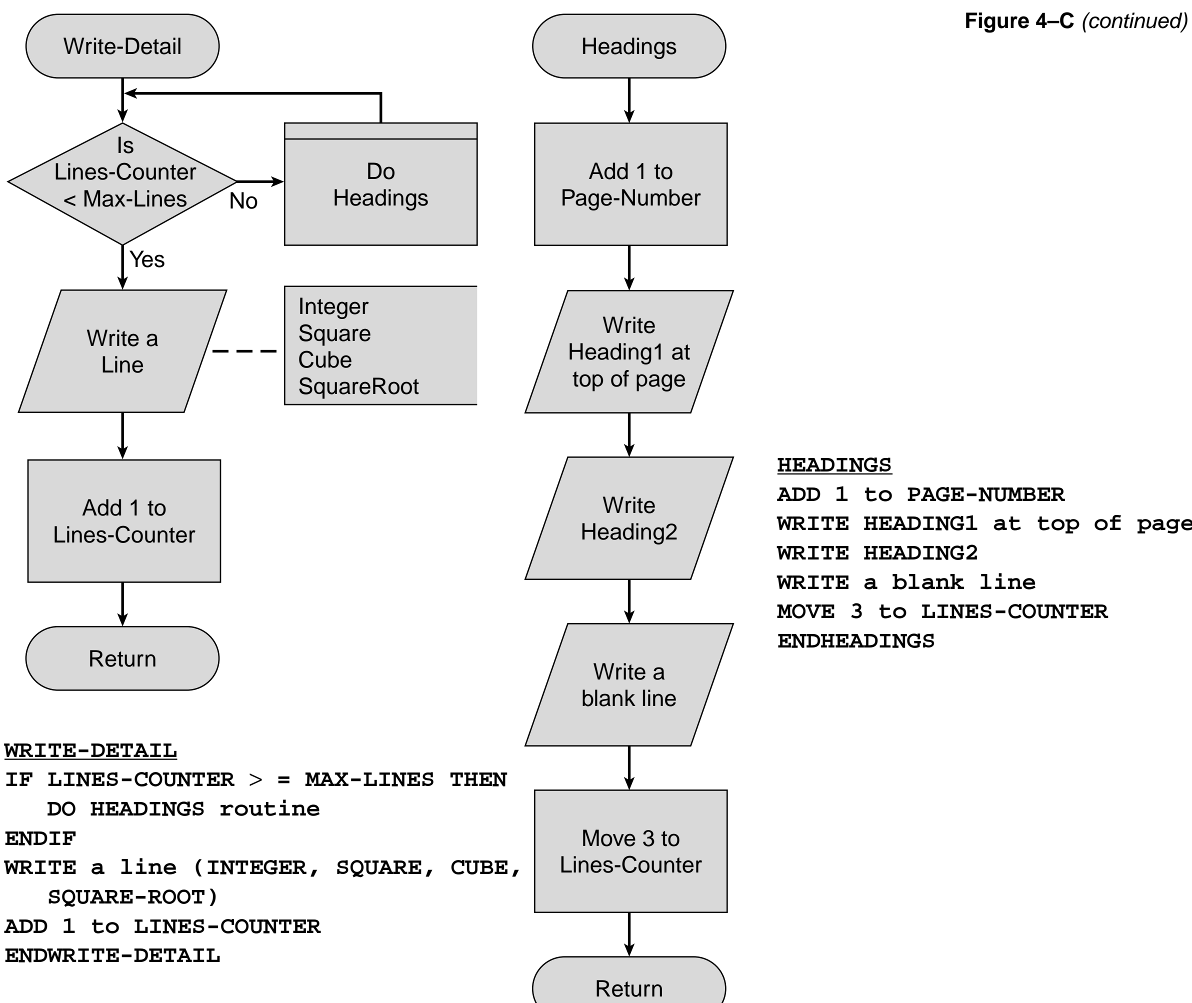


Figure 4–C *(continued)*

loop in the main routine. It is given an initial value of 1 in INITIALIZE and is incremented at the end of the CALCULATE routine. The variables LINES-COUNTER and MAX-LINES as well as the instruction LINES-COUNTER = MAX-LINES are explained later. Ignore them for now.

After completing INITIALIZE, processing returns to the MAINLINE routine, which does CALCULATE until INTEGER-COUNTER has a value greater than 500. The first thing that CALCULATE does is calculate the square of INTEGER-COUNTER (the square of X is X ∗ X). The next instruction calculates the cube of INTEGER-COUNTER (the cube of X is X ∗ X ∗ X or $X^2 * X$). The first time through the routine, INTEGER-COUNTER is equal to 1, so SQUARE becomes 1 ∗ 1 or 1, and CUBE also becomes 1 ∗ 1 or 1.

The next instruction uses an external subroutine to compute the square root (SQRT) of INTEGER-COUNTER. An **external subroutine** is a subroutine outside the program. It may be a routine or function included in the programming language, written by another programmer or purchased

from another source. Because it is coded outside the program, it is not shown in the hierarchy chart. The way in which an external subroutine is accessed by a program depends on the particular programming language used. In general, however you must provide the name of the routine being accessed (SQRT), the field name(s) used in the subroutine (INTEGER-COUNTER) and the field name(s) where the results are to be placed (SQUARE-ROOT).

After calculating the square, cube, and square root, CALCULATE calls WRITE-DETAIL to write a detail line on the report. Upon returning to the CALCULATE routine, 1 is added to INTEGER-COUNTER. Processing then returns to the MAINLINE routine. INTEGER-COUNTER now equals 2, so the program does the CALCULATE routine for the integer 2, and so on, until 501 is reached.

WRITE-DETAIL writes detail lines on the report and controls the number of lines written on each page. LINES-COUNTER is a work area used to keep track of the number of detail lines written on a page. Whenever LINES-COUNTER is greater than or equal to the maximum number of lines (**MAX-LINES**) to be printed on the page, HEADINGS is called to write heading lines at the top of the next page prior to writing the current detail line.

Notice in INITIALIZE that MAX-LINES is given a value of 28. This value is moved to LINES-COUNTER (LINES-COUNTER = MAX-LINES) at the end of the INITIALIZE routine even though no lines have been written on the report. This forces headings to be done prior to writing the first detail line on the report. In effect, we fool the computer into "thinking" that 28 lines have already been written on a previous page so that the headings will be done, even though in reality, no detail lines have yet been written. Some programmers write headings for the first page of the report in the INITIALIZE routine. Most do not for the following reason: In some cases (for example, when producing an exception report), there may not be any detail lines to write. If headings are done in the INITIALIZE routine, ending up with a report that contains heading lines and no detail lines is possible. In this example, heading lines are not written in the INITIALIZE routine. Instead, HEADINGS is called for the first time when there is a detail line to be written.

Now look at HEADINGS. HEADINGS adds 1 to PAGE-NUMBER, which is part of the first heading line. PAGE-NUMBER is initialized to zero, and 1 is added to it prior to printing HEADING1. If you initialize PAGE-NUMBER to 1, then the first page would show PAGE 2. Be careful! If you want to initialize PAGE-NUMBER to 1, then ADD 1 TO PAGE-NUMBER at the end of HEADINGS rather than at the beginning.

HEADINGS is called when LINES-COUNTER is greater than or equal to MAX-LINES (in this case 28). The last instruction in the HEADINGS routine moves 3 to LINES-COUNTER because HEADINGS writes three lines (HEADING1, HEADING2, and a blank line). These three lines are not available as detail lines. It may seem confusing why you must add 1 for the blank line, but remember that a line containing spaces uses just as much vertical space on the paper as a line containing other characters. In WRITE-DETAIL, one is added to LINES-COUNTER after a detail line is written. Therefore, after the first detail line is written, LINES-COUNTER is equal to 4. After the second detail line is written, LINES-COUNTER is equal to 5, and so on. After the 25th detail line is written, LINES-COUNTER equals 28 (3 heading lines and 25 detail lines have been written), and headings are written prior to writing the next detail line.

- Before continuing to the next section, answer the following questions at the end of the chapter: True/False 4; Multiple Choice 7.

4.5 Example: **Screen Output with Heading and Footing Lines**

In this example we produce a screen display showing a doctor's appointments for the day. Output is described in the following layout chart.

Layout Chart

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789
1		PATIENT		CHART	T	IME
2		NAME		NUMBER		
3						
4	XXXXXXXXX	XXXXXXXXXX	XXXX	XXXXX	XXX	XXXXX
5	XXXXXXXXX	XXXXXXXXXX	XXXX	XXXXX	XXX	XXXXX
.	XXXXXXXXX	XXXXXXXXXX	XXXX	XXXXX	XXX	XXXXX
.	XXXXXXXXX	XXXXXXXXXX	XXXX	XXXXX	XXX	XXXXX
.	XXXXXXXXX	XXXXXXXXXX	XXXX	XXXXX	XXX	XXXXX
15	XXXXXXXXX	XXXXXXXXXX	XXXX	XXXXX	XXX	XXXXX
16						
17	PRESS ANY	KEY TO CON	TINUE	(To be dis	played at t	he
18				bottom of	each scree	n when
				there are	more recor	ds to
				display)		

For each record input, the following fields are displayed: patient name, chart number, and appointment time. A maximum of 15 detail and heading lines appear on each screen. If more appointments need to be listed after displaying an entire screen, do the following: Hold the screen in its current position to allow the user to read it, and instruct the user to press any key at the keyboard to continue listing lines. Do not display additional lines until the user presses a key at the keyboard. When the "key press" event occurs, clear the screen. Then, start at the top of the screen to display more lines. When there are no more lines to display, leave a blank line and display the report footer message REPORT COMPLETE.

The data dictionary for this example is shown below. The reason for defining some areas as local is explained later.

Data Dictionary

Input Area—APPOINTMENT-DATA

Name	Description	Type	Scope
PATIENT-NAME	Patient's name	Nonnumeric	Global
CHART-NUMBER	Patient's chart number	Nonnumeric	Global
TIME	Time of appointment	Nonnumeric	Global

Output Area—SCREEN

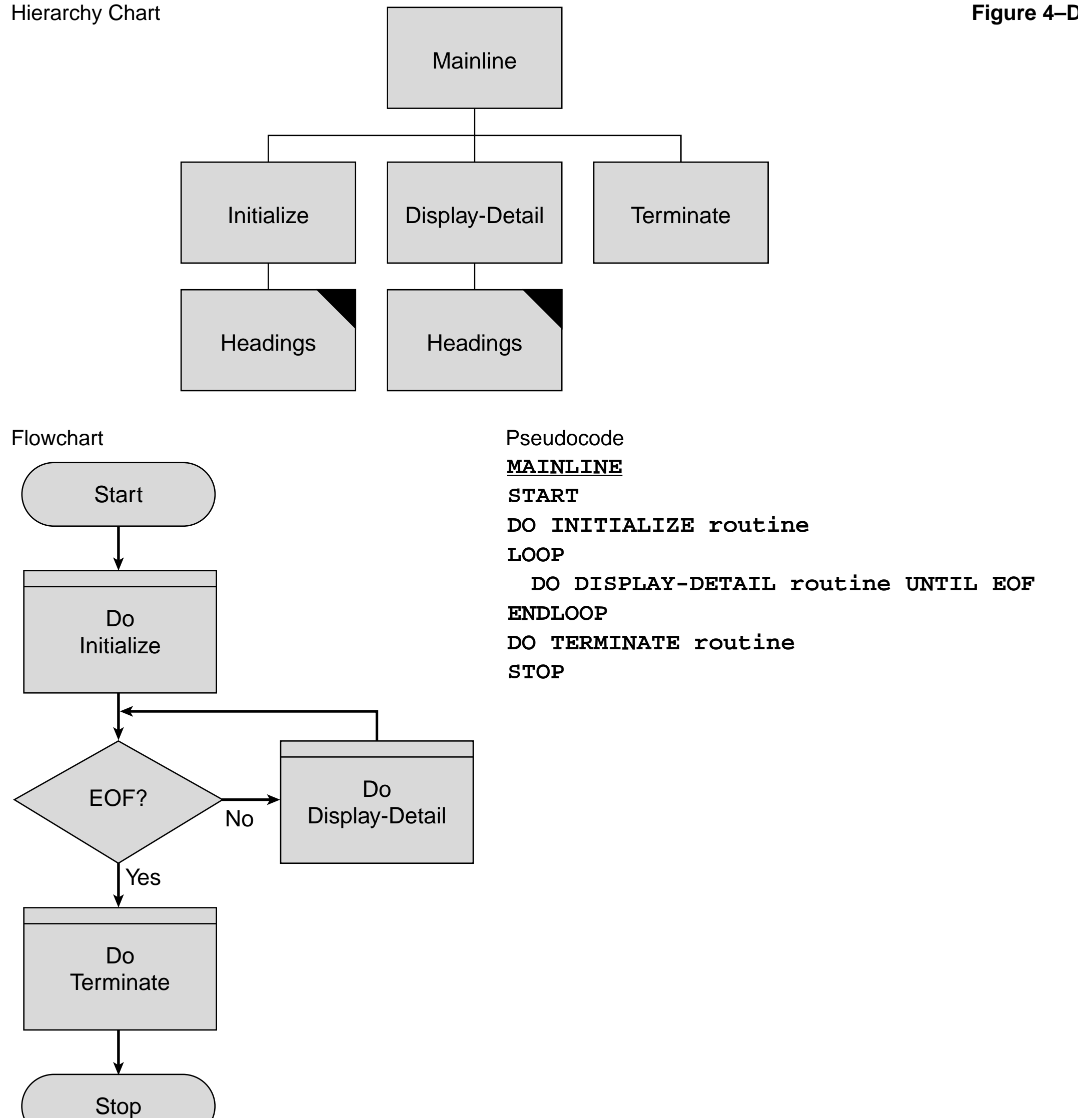
Work Area

Name	Description	Type	Initial Value	Calculation	Scope	Routines Accessing Location
HEADING1 "PATIENT" "CHART" "TIME"	See spacing chart—line 1	 Constant Constant Constant			Local	Headings
HEADING2 "NAME" "NUMBER"	See spacing chart—line 2	 Constant Constant			Local	Headings
FOOTING1 "PRESS ANY KEY TO CONTINUE"	Screen footing line	Constant			Local	Display-Detail
FOOTING2 "REPORT COMPLETE"	Report footing line	Constant			Local	Terminate
LINES-COUNTER	Number of detail lines displayed on the screen	Numeric	0	1 is added for each line displayed on the report	Global	
MAX-LINES	Maximum lines per screen (excluding footing lines)	Numeric	15		Global	

The hierarchy chart, flowchart, and pseudocode are shown in Figure 4–D.

MAINLINE does INITIALIZE, to assign global work area values, open files, do HEADINGS, and read the first input record. Notice that unlike the previous example, the heading and footing lines are not given values in INITIALIZE. Instead, HEADINGS uses them as constants rather than as storage locations that have previously been assigned values. Therefore, these values do not have to be defined globally or passed to the HEADINGS routine. Either method is correct. Another thing that is different in this example is that INITIALIZE calls HEADINGS. If the logic for this example was handled the same as that of the previous example, LINES-COUNTER would be assigned an initial value (MAX-LINES) in INITIALIZE, and HEADINGS would not be done until DISPLAY-DETAIL called it.

To understand why this example is handled differently, you need to look at the DISPLAY-DETAIL routine. In this example, when LINES-COUNTER equals MAX-LINES, a message is displayed at the bottom of the screen indicating that the user should press a key to continue. If LINES-



COUNTER were assigned the value stored in MAX-LINES as part of INITIALIZE, then the PRESS ANY KEY message would be displayed prior to displaying the first detail line. To prevent that, HEADINGS is called by INITIALIZE for the first headings. HEADINGS writes the headings and assigns a value to LINES-COUNTER.

Immediately before returning to the MAINLINE routine from INITIALIZE, the first record is read. MAINLINE checks for EOF. If it is not EOF, DISPLAY-DETAIL is called to display a detail line. Notice that this program does not do an error routine if the input file is empty. At the end of DISPLAY-DETAIL, the next input record is read, and processing returns to the main routine. If it is EOF, TERMINATE is called to display the report footing line and close files.

Now look at TERMINATE. You will notice that TERMINATE does not check LINES-COUNTER to determine whether there is room on the screen to display the line “REPORT COMPLETE.” When TERMINATE is called, at most

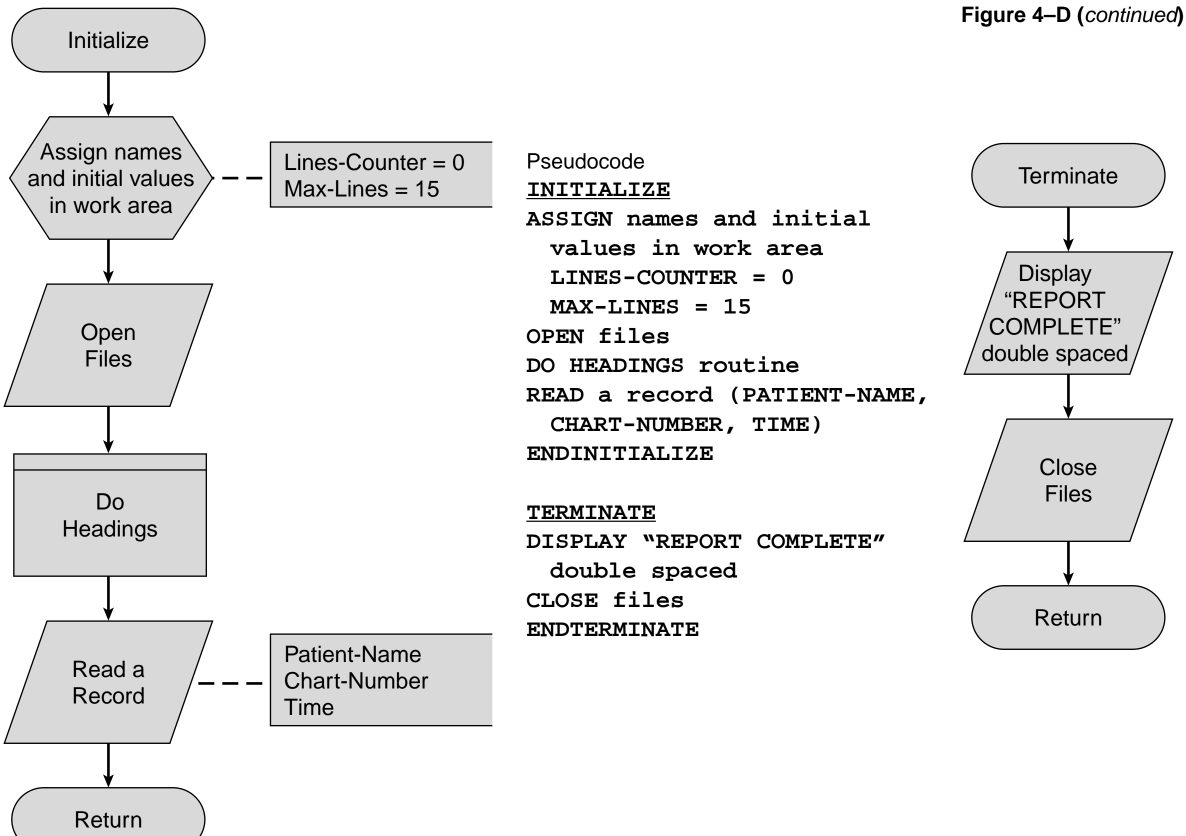


Figure 4–D (*continued*)

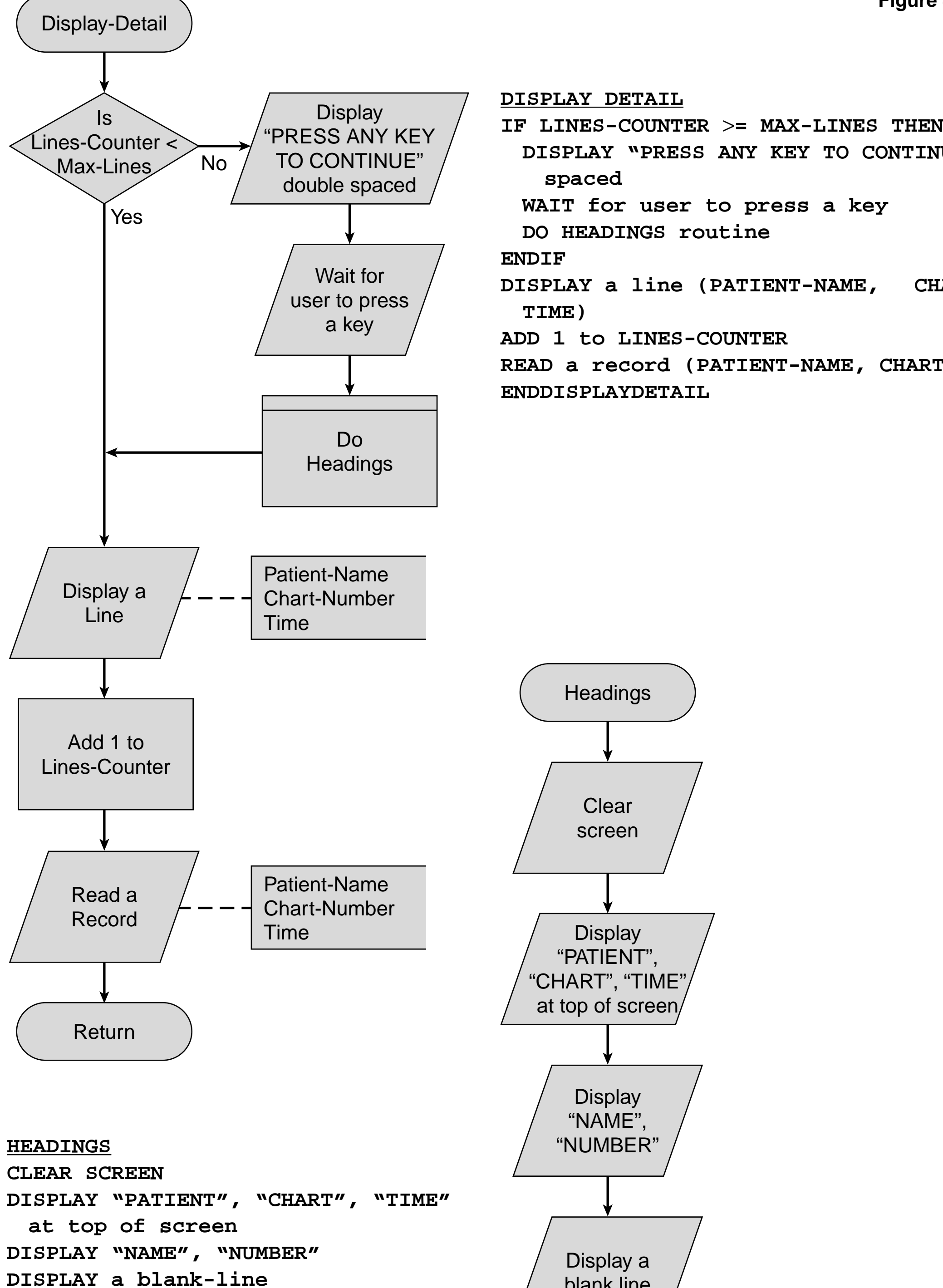
15 lines have been displayed on the current screen. Since there is physically room on the screen to display the footing line and the blank line that precedes it, it is not necessary to check LINES-COUNTER.

Let's go back and spend some more time examining DISPLAY-DETAIL, which is called to display each detail line. Upon entering the routine, the value in LINES-COUNTER is checked. If it is less than MAX-LINES, there is room on the screen for another line to be displayed. In this case the line is displayed, and 1 is added to LINES-COUNTER. If LINES-COUNTER is not less than MAX-LINES when the routine is entered, a message is displayed to the user indicating that a key should be pressed to continue. The program then waits for a response. This holds the screen in its current position to enable the user to read it. When the user presses a key, the program does HEADINGS prior to displaying the detail line.

It may seem strange to you that LINES-COUNTER is not incremented when footing lines are displayed. If you think about it, however, it makes perfect sense. After writing footing lines, no more lines are displayed on the current screen, HEADINGS is called, and LINES-COUNTER starts over again. Even if you increment LINES-COUNTER when you write the footings, you still call HEADINGS and start LINES-COUNTER over again. Therefore, incrementing LINES-COUNTER when you display footings is a waste of time.

The last routine to study is HEADINGS. The HEADINGS routine first clears the screen to remove whatever is currently displayed. HEADING1 is then displayed at the top of the screen. HEADING2 is displayed on the second line of the screen. The blank line between the headings and first detail line is displayed next. This blank line is displayed in the HEADINGS routine because it is the simplest place to do so. When displaying detail lines, we don't want

Figure 4–D (*continued*)




```
DISPLAY DETAIL
IF LINES-COUNTER >= MAX-LINES THEN
  DISPLAY "PRESS ANY KEY TO CONTINUE" double
    spaced
  WAIT for user to press a key
  DO HEADINGS routine
ENDIF
DISPLAY a line (PATIENT-NAME,  CHART-NUMBER,
  TIME)
ADD 1 to LINES-COUNTER
READ a record (PATIENT-NAME, CHART-NUMBER, TIME)
ENDDISPLAYDETAIL
```

```
HEADINGS
CLEAR SCREEN
DISPLAY "PATIENT", "CHART", "TIME"
  at top of screen
DISPLAY "NAME", "NUMBER"
DISPLAY a blank-line
MOVE 3 to LINES-COUNTER
ENDHEADINGS
```

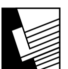
the processing to depend on whether we are displaying the first detail line on the screen (which needs a blank line before it), or whether we are displaying a subsequent line (which should be single spaced). By displaying the blank line between the heading and first detail line as part of the HEADINGS routine, all detail lines can be single-spaced, and we don't have to worry about which detail line is being displayed. In Section 4.7, you will see an example in which detail lines are double-spaced, and the blank line is written when the detail line is written, rather than in HEADINGS. This is done because a blank line is needed before *every* detail line.

Unlike paper output that requires the headings to be reprinted on each page, time and effort can be saved by erasing only that part of the screen that is changing. This technique is called **refreshing the screen.** In this program, for example, the programmer can leave the heading lines on the screen and erase only the detail and footing lines. To follow this approach, do the following:

1. Move the CLEAR SCREEN instruction in the HEADINGS routine to the INITIALIZE routine. This is done because the entire screen still must be cleared at the beginning of the program.

2. Replace the first four instructions in HEADINGS (CLEAR SCREEN and the three DISPLAY instructions) with an instruction to CLEAR SCREEN EXCEPT FOR LINES 1–3. This erases all lines except for heading lines from the screen.

- Before continuing to the next section, answer the following questions at the end of the chapter: Matching Part 1; Multiple Choice 8, 9, 10, 11, 12, 13.

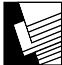


4.6 How Do We Include the Date on a Report?

Including the current date on a report is usually desirable. A program can access the current date from the computer's operating system. The method for doing this and the form in which the date is stored varies among languages and computer systems. If the date is stored in a format other than that desired by your program, it is a simple process to have your program change the format. Oftentimes, a date object is used to access and manipulate the date. In this book, an external subroutine is used to request the current date from the system, and it is assumed that the date is stored in the desired format. The example shown in the next section illustrates the process of using an external subroutine to obtain the current date from the computer system.

Sometimes the current date cannot be accessed from the system. Other times, even though the current date can be obtained from the system, the date needed on a report or file is not the current date. For example, in a payroll system, the program that produces paychecks may be run on Thursday, even though the checks are distributed on Friday. Friday's date would be the desired date on the checks. In a case such as this, the system date cannot be used, and it is necessary to input the desired date using a date record or modify the system date. We do not illustrate an example of this.

- Before continuing to the next section, answer the following questions at the end of the chapter: True/False 5; Short Answer 2.



4.7 Example: **Exception Reports, Using an External Subroutine to Obtain the Current Date, Double-Spacing Detail Lines, and Printing the Page Number in a Footing Line**

This example demonstrates the logic to produce an exception report for an airport manager. The report lists those flights that did not depart at their scheduled time. Output is described in the layout chart shown below.

Layout Chart

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789	6 0123456789	7 0123456789
1	MM/DD/YY			ON-TIME RE	PORT			
2								
3		AIRLINE		FLIGHT	SCHE	DULED	ACTUAL	
4								
5	XXXXX	XXXXXXXXXX	XXXX	XXXXX	XXX	XXXXX	XXXXXXXX	
6								
7	XXXXX	XXXXXXXXXX	XXXX	XXXXX	XXX	XXXXX	XXXXXXXX	
8								
.								
.								
50								
51	PAGE ZZ9							

The following fields are displayed: AIRLINE, FLIGHT, SCHEDULED DEPARTURE TIME, and ACTUAL DEPARTURE TIME. Not every input record is shown on the report. The program only includes records that have a scheduled departure time different from the actual departure time. A maximum of 49 detail (and blank) and heading lines appear on each page. Line 50 is always blank. Line 51 is used for the page number.

The data dictionary for this example is shown below. All storage locations are defined as global, therefore the scope column is not included in the data dictionary.

Data Dictionary

Input Area—SCHEDULE-DATA

Name	Description	Type
AIRLINE	Name of airline	Nonnumeric
FLIGHT	Flight number	Nonnumeric
SCHEDULED	Scheduled departure time	Nonnumeric
ACTUAL	Actual departure time	Nonnumeric

Output Area—ON-TIME REPORT

Work Area

Name	Description	Type	Initial Value	Calculation
HEADING1 DATE "ON-TIME REPORT"	See Spacing Chart—Line 1	 Nonnumeric (MM/DD/YY) Constant		
HEADING2 "AIRLINE" "FLIGHT" "SCHEDULED" "ACTUAL"	See Spacing Chart—Line 2	 Constant Constant Constant Constant		
FOOTING1 "PAGE" PAGE-NUMBER	Page Footing Line	 Constant Numeric	 0	 1 is added for each page
LINES-COUNTER	Number of detail lines written on the page	Numeric	0	1 is added for each line written on the report
MAX-LINES	Maximum lines per page (excluding footing lines)	Numeric	49	

The hierarchy chart, flowchart, and pseudocode are shown in Figure 4–E.

This program creates an exception report. Not all of the input records are included on the report. Deciding whether or not to include a record is fairly simple. The decision to include or not include a particular record is the first instruction in the PROCESS routine. If the ACTUAL time is not the same as the SCHEDULED time, then the flight is included on the report. If the record was for an on-time flight, it is *not* included. Instead, the program goes directly to the next instruction which reads a record.

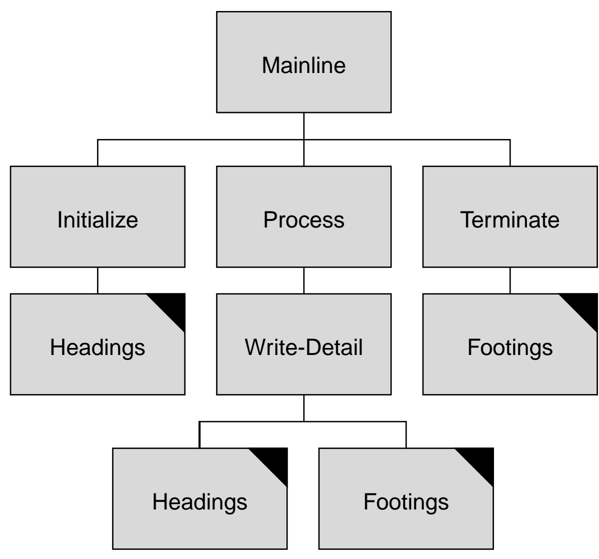
Now let's examine how the date is obtained from the system. It may surprise you that the date is obtained in the INITIALIZE routine rather than in HEADINGS (when it writes the date). There are two reasons for this:

1. The INITIALIZE routine is only done once, whereas the HEADINGS routine is done as many times as there are pages in the report. Once the date is obtained from the system and stored in DATE, it remains, there for the entire program unless another instruction stores a value in DATE. Therefore, obtaining the date in INITIALIZE is more efficient.

2. If your program is executed during third shift (for example, 11 P.M. to 7 A.M.), then it is possible that the program starts execution on one day and finishes the next day. If the date is obtained in HEADINGS, the date may change in the middle of the report. If the date is obtained in INITIALIZE, however, it will only be obtained once, so the date will be the same throughout the entire report.

Figure 4–E

Hierarchy Chart



Pseudocode

```
MAINLINE
START
DO INITIALIZE routine
IF EOF THEN WRITE "NO INPUT
  RECORDS" at top of page
ELSE LOOP
      DO PROCESS routine UNTIL EOF
      ENDLOOP
ENDIF
DO TERMINATE routine
STOP
```

Flowchart

Start
Do Initialize
EOF?
Yes
Write "No Input Records" at top of page
No
Do Process
EOF?
No
Yes
Do Terminate
Stop

```
INITIALIZE
ASSIGN names and initial
        values in work area
        HEADING1
        DATE
        "ON-TIME REPORT"
        HEADING2
        "AIRLINE"
        "FLIGHT"
        "SCHEDULED"
        "ACTUAL"
        FOOTING1
        "PAGE"
        PAGE-NUMBER = 0
        OTHER
        LINES-COUNTER = 0
        MAX-LINES = 49
OPEN FILES
GET DATE for HEADING1
DO HEADINGS routine
READ a record (AIRLINE,
  FLIGHT, SCHEDULED, ACTUAL)
ENDINITIALIZE
```

Initialize
Assign names and initial values in work area

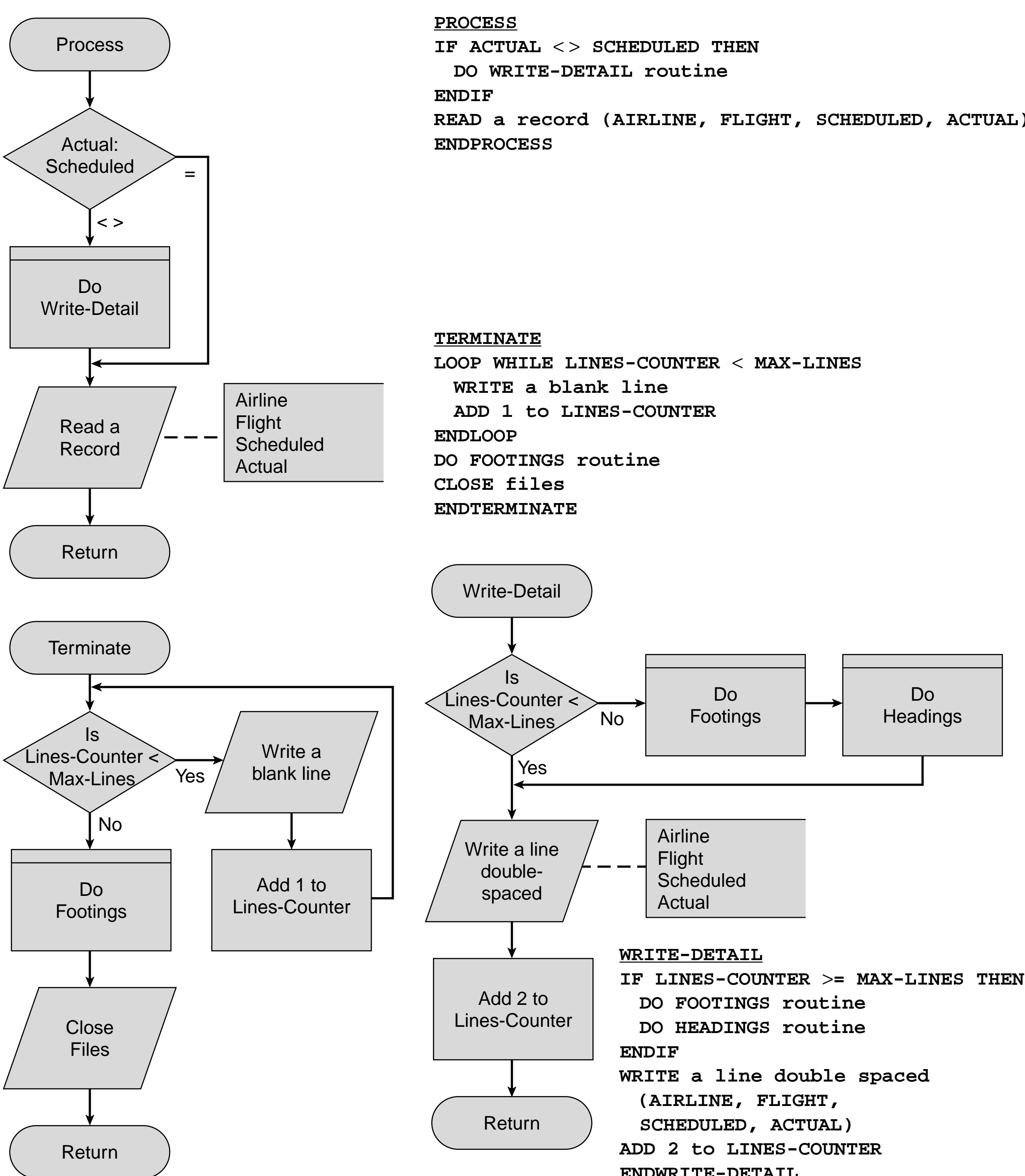
Heading1
Date
"ON-TIME REPORT"
Heading2
"AIRLINE"
"FLIGHT"
"SCHEDULED"
"ACTUAL"
Footing1
"PAGE"
Page-Number = 0
Other
Lines-Counter = 0
Max-Lines = 49

Open Files
Get Date for Heading1
Do Headings
Read a Record

Airline
Flight
Scheduled
Actual

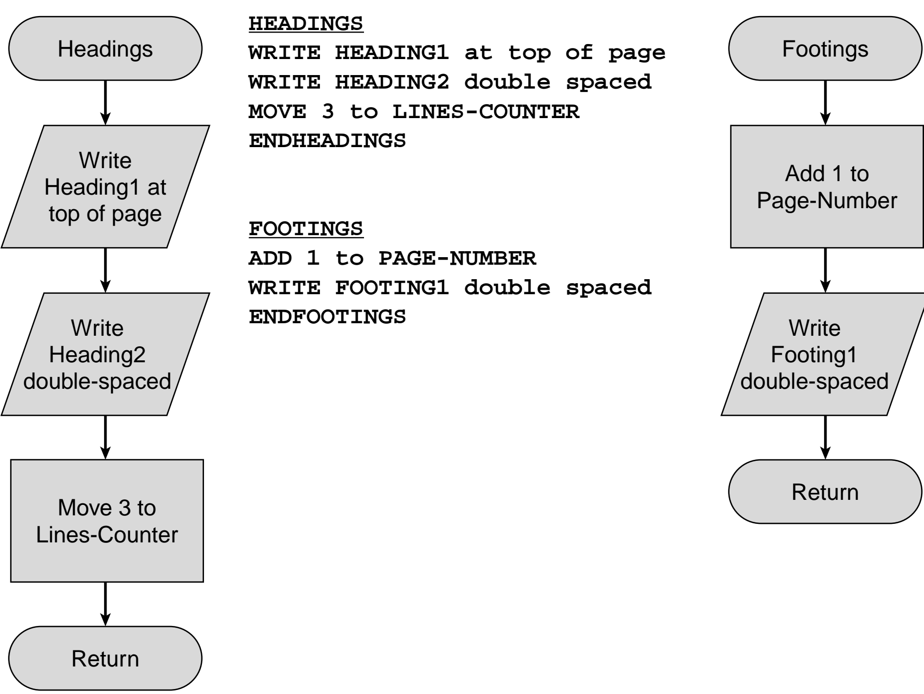
Return

Figure 4–E (*continued***)**



The next thing to consider is the vertical line spacing on the report. The report is double-spaced. That is, we want to print a blank line prior to each detail line. Two issues need to be considered. First, LINES-COUNTER needs to be incremented properly when lines are double-spaced. Notice that WRITE-DETAIL includes an instruction to ADD 2 TO LINES-COUNTER. One line is for the detail line and the other is for the blank line that precedes it. The second thing to consider is the blank line between the headings and first detail line. In the previous example, this blank line was written as part of

Figure 4–E (*continued*)



the HEADINGS routine. We do not do that in this example. This example differs from the previous one in that the detail lines there are single-spaced and the detail lines here are double-spaced. Because a blank line is always written prior to writing a detail line in this example, the blank line between the heading lines and the first detail line is written when the first detail line is written. In this example, if you wrote a blank line at the end of the HEADINGS routine, your output would have two blank lines prior to the first detail line of the page!

The last thing we need to explore in this example is how the footings are printed. Notice in WRITE-DETAIL that FOOTINGS is called prior to doing HEADINGS. Also notice that INITIALIZE does HEADINGS so that we don't write footing lines prior to printing the first detail line. Now, let's look at two questions regarding printing the footing line on the last page of the report.

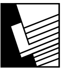
1. How does TERMINATE ensure that the footing line is printed on the 51st line of the page? When TERMINATE is called, LINES-COUNTER has a value between 5 and 49, depending on how many detail lines have been printed on the last page of the report. To handle this problem, TERMINATE writes blank lines until LINES-COUNTER equals MAX-LINES. At that point, FOOTINGS is called to write the footings on the final page of the report.

2. How does TERMINATE avoid writing a footing line if the input file was empty? TERMINATE is called to close files even if no lines were printed (the input file was empty). As a result, the page with the error message will have a footing line. If you choose, you can use an indicator (ERROR-INDICATOR) to make sure that no footing line is written on the page with the error message. Simply make the following changes to the logic:

- Assign a storage location for ERROR-INDICATOR and give it an initial value of NO.

- After writing the error message in MAINLINE, move YES to ERROR-INDICATOR.
- Check ERROR-INDICATOR at the start of the TERMINATE routine. If it is YES, do not print the footings prior to closing files.

- Before continuing to the next section, answer the following questions at the end of the chapter: Matching Part 2; True/False 6, 7, 8; Multiple Choice 14, 15, 16.



4.8 How Does Garbage Input Generate Output Errors?

The first part of this chapter concentrated on report output. Now we will shift our focus and spend some time considering input. Recall from Chapter 1 that before a computer program can be executed, it must be translated into machine language. During this translation process (which uses an assembler, compiler, or interpreter), **syntax errors** can occur. These errors result from violating rules of the programming language. When assemblers or compilers are used to do the translation, error messages are displayed or printed, and the program is not executed at all. When an interpreter is used, the program is executed until the first error is found. Examples of syntax errors include:

- Using a variable name that starts with a number instead of a letter (e.g., 1Amount).
- Branching to an instruction or calling a subroutine that does not exist.
- Misspelling an instruction (e.g., PRNT instead of PRINT).
- Using a variable name that has not been defined in languages that require the definition.

Even when a program translates successfully (without errors), the program may still have errors. Two types of **execution (logic) errors** can occur after the translation process while the program is being executed. These are

- Errors in logic that cause incorrect output to be produced during program execution. Examples include:

  1. Using an ADD rather than a SUBTRACT instruction when coding. Because it is capable of adding, the computer will not stop and print an error message. The output, however, will not be correct.

  2. Processing data in an incorrect sequence. This type of error results when an input file is not sorted prior to doing a program with control break processing (see Chapter 8).

- Errors in logic that cause a **program interrupt.** That is, the program stops executing, and an error is printed. For the unfortunate programmer or analyst who happens to be responsible for a program executed during the midnight shift, program interrupts often result in receiving a phone call in

the middle of the night and an unwanted trip to work to diagnose and correct the problem. The following are two examples of this type of error:

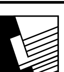
1. Attempting to do arithmetic on a field that does not contain a number. Such errors, called **data-exception errors,** result when an incorrect type of data is in a field.

2. Accessing a nonexistent entry in an array or table (see Chapter 6).

Translation errors and errors that cause a program interrupt may be unpleasant for the programmer, but usually they do not cause as much difficulty as a program that has undetected logic errors. Such programs may execute and end normally while producing incorrect output. Imagine a program that adds instead of subtracts deductions to gross pay to calculate the net pay! Carefully testing a program should eliminate all logic errors.

Even carefully testing a program, however, may not eliminate all sources of error. The output produced by a computer is only as accurate as the input it processes. The term **GIGO** (Garbage In, Garbage Out) describes the output produced by incorrect input. **Data-validation techniques** check for input errors prior to processing.

- Before continuing to the next section, answer the following questions at the end of the chapter: True/False 9, 10, 11, 12, 13, 14, 15; Multiple Choice 17; Short Answer 3.



4.9 What Types of Data Validation Can Be Done?

Many data-validation techniques are available. This section of the chapter includes discussion and examples for several of these techniques.

Data Type

Some input fields, such as those used in arithmetic, must contain only numeric data. Other input fields must contain alphabetic data. Validation for data type checks fields to ensure that they contain the proper data.

The way this checking is done varies from language to language. Because the development of logic in this book is *not* language-specific, checking for data type is done using instructions, such as

```
IF AMOUNT is not numeric THEN
  MOVE "AMOUNT IS NOT NUMERIC" to ERROR-MESSAGE
  DO WRITE-ERROR routine
ENDIF
```

Keep in mind that validation techniques can check whether a field is numeric, but they cannot check whether the correct number has been entered. For example, a program that validates a payroll file can check that the pay rate is numeric, but if the rate was intended to be 5.83 and it was input as 5.76, then no error will be found during validation. Thus, if a validation program or routine shows that no errors exist in an input file, it indicates only that those errors checked for were not found. There still may be other errors in the file!

Range of Values

Suppose you need to write a program to process test scores. Further suppose that a particular test has a maximum score of 100 points and a minimum score of 0 points. Validation for range would check the score to see if it is in the range 0 to 100. The logic for this range check can be shown in one of three ways:

Range of Values Example 1

```
IF SCORE < 0 THEN
   MOVE "SCORE IS INVALID" to ERROR-MESSAGE
   DO WRITE-ERROR routine
ELSE IF SCORE > 100 THEN
        MOVE "SCORE IS INVALID" to ERROR-MESSAGE
        DO WRITE-ERROR routine
     ENDIF
ENDIF
```

or

Range of Values Example 2

```
IF SCORE is not 0-100 THEN
   MOVE "SCORE IS INVALID" to ERROR-MESSAGE
   DO WRITE-ERROR routine
ENDIF
```

or

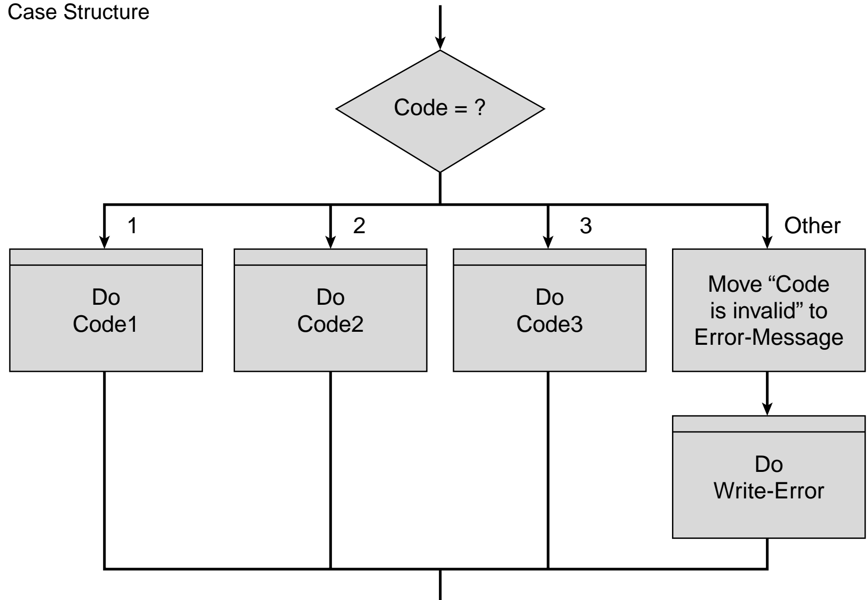
Range of Values Example 3

```
IF SCORE is < 0 or SCORE is > 100 THEN
   MOVE "SCORE IS INVALID" to ERROR-MESSAGE
   DO WRITE-ERROR routine
ENDIF
```

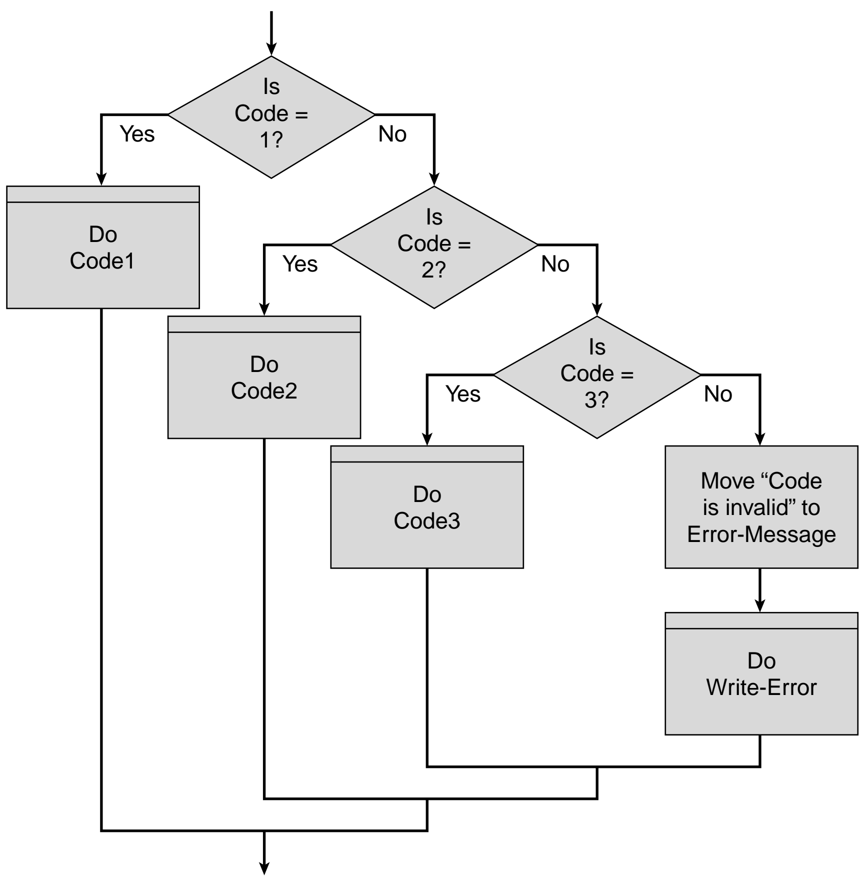
Keep in mind that a program can confirm that the range is 0 to 100, but it cannot confirm whether the correct number has been entered.

Exact Value or Values

Suppose commissions are calculated based on an input code of 1, 2, or 3. The program should validate the code entered to ensure that it is, in fact, a 1, 2, or 3. This type of logic can use either a case structure or a nested IF-THEN-ELSE as shown below.



Nested IF-THEN-ELSE



Sometimes codes or fields are entered as letters rather than as numbers. For example, an individual's gender is often coded as "M" for male and "F" for female. In such cases the value must be entered as uppercase by the user or the program must check for both uppercase and lowercase responses. This is required because an uppercase letter is not stored using the same bits as a lowercase letter in the computer. To the computer, "M" is not the same as "m"; that is, if the program compared "m" to "M," the values would be unequal. In the pseudocode segment below, comparisons are made to look for both uppercase and lowercase responses. If a valid response is lowercase, it is converted to uppercase by the program. Therefore, subsequent programs or routines can assume that the GENDER field is uppercase.

```
IF GENDER is equal to "m" THEN
   MOVE "M" to GENDER
ELSE IF GENDER is equal to "f" THEN
        MOVE "F" to GENDER
     ELSE IF GENDER is not equal to "M" THEN
             IF GENDER is not equal to "F" THEN
                MOVE "GENDER IS INVALID" to ERROR-MESSAGE
                DO WRITE-ERROR routine
             ENDIF
          ENDIF
     ENDIF
ENDIF
```

Keep in mind that a program can confirm that the value input is 1, 2, or 3 or whether it is "M" or "F," but it cannot confirm whether the correct value has been entered.

Sign

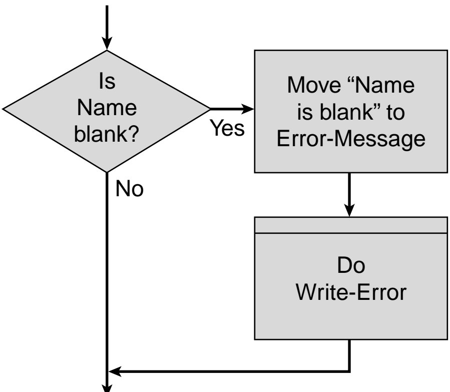
The test for valid sign ensures that a numeric value is either positive or negative. The following example validates that a purchase amount is greater than 0.

```
IF AMOUNT is less than or equal to zero THEN
  MOVE "AMOUNT IS INVALID" to ERROR-MESSAGE
  DO WRITE-ERROR routine
ENDIF
```

Remember that a program can confirm that the amount is greater than 0, but it cannot confirm whether the correct amount has been entered.

Data Presence

Validation for presence of data ensures that an input field contains a value other than spaces or blanks. This flowchart shows the logic for checking that the name input is not blank.



In this validation, as in others, a program can confirm that the name is not blank, but it cannot confirm whether the correct name has been entered.

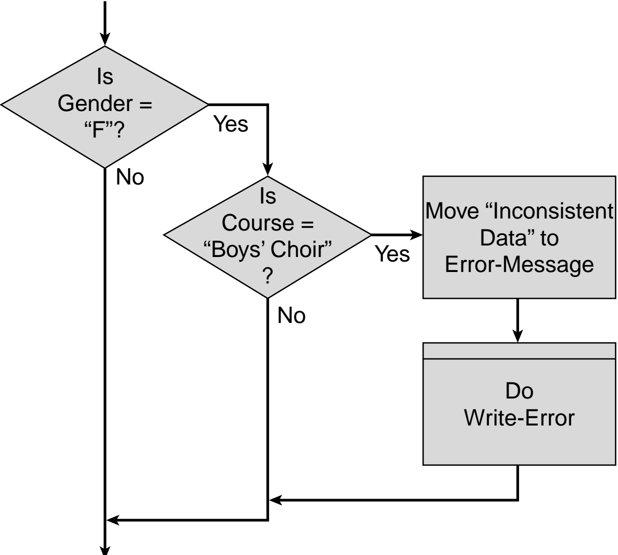
Data Consistency

Sometimes it is useful to check the consistency of one data value against another. The following are three examples:

1. Check that the date entered does not exceed the number of days possible in that month; for example, if the month entered is April, the day cannot exceed 30.

2. If the vehicle make is Ford, check that the vehicle model is actually one manufactured by Ford—Mustang, Taurus, and so on.

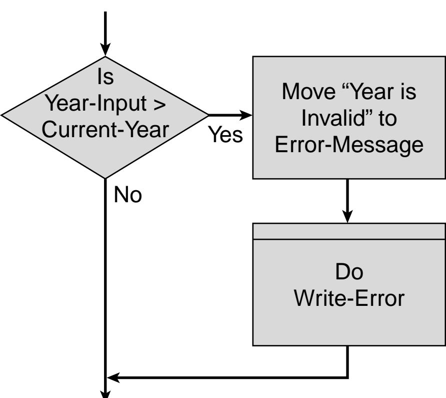
3. If the gender of a student is entered as "F" (for female), make sure that she does not register for boys' choir. The logic for this example is shown in the flowchart on the next page.



Data Reasonableness

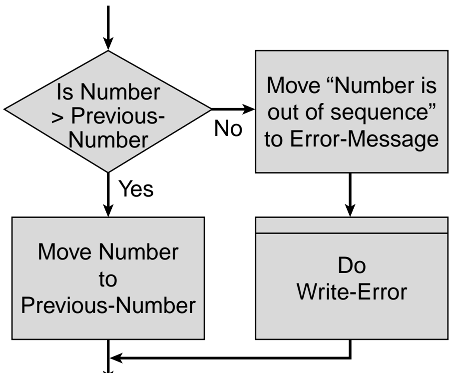
All the previously described techniques ensure that the input data are reasonable. Some examples of reasonableness checking do not fit in any of the previous categories, however. One example is checking a field that should contain a five-digit part number to confirm that it does, in fact, contain five digits.

Another example is illustrated in the following flowchart segment, which validates that the year that a payment was made is not greater than the current year.



Data Sequence

Sequence checking ensures that data are input in a certain order or that no values are missing. For example, it may be desirable to confirm that invoice numbers are entered in increasing order. The following flowchart segment illustrates sequence checking.



To do a sequence check, the program uses a work area to remember the last valid number entered. In the preceding flowchart, PREVIOUS-NUMBER is the name of the work area used. PREVIOUS-NUMBER must be given an initial value that is less than the first number in the input file. Typically, the initial value is zero or a value less than zero.

For each record input, the program compares the number on the input record (NUMBER) to PREVIOUS-NUMBER. If NUMBER is greater than PREVIOUS-NUMBER, the record is valid. When the record is valid, NUMBER is saved in PREVIOUS-NUMBER for the next comparison.

It makes no sense to compare NUMBER with an invalid invoice number. Therefore, if NUMBER is not valid, it is not saved in PREVIOUS-NUMBER. Because only valid numbers are moved to PREVIOUS-NUMBER, NUMBER is always compared to the value from the most recent valid input record.

Another example of sequence checking is the processing required to produce a bank statement that lists the numbers for "missing" checks. The following pseudocode segment shows how this type of validation could be done:

```
LOOP WHILE PREVIOUS-NUMBER is less than NUMBER
  MOVE PREVIOUS-NUMBER to ERROR-MESSAGE
  DO WRITE-ERROR routine
  ADD 1 to PREVIOUS-NUMBER
ENDLOOP
```

One problem with sequence checking is that a report showing sequence errors may not always be accurate. The inaccuracy occurs because a record shown as being OUT OF SEQUENCE may not actually be the out-of-sequence record. The following example clarifies this problem. Suppose the numbers on the input records are 1, 2, 3, 4, 8, 6, 9, 10. If you trace these data through the previous flowchart segment, you will see that the record with the number 6 will be correctly treated as out of sequence. However, suppose the numbers on the input records are 1, 2, 3, 99, 4, 5, 6, 7, 8. The record out of sequence is the one with the number 99. What appears on the report, though? The number 99 will be considered to be in sequence because it is larger than 3. Hence, 99 is moved to PREVIOUS-NUMBER. All remaining records (4, 5, 6, 7, 8) will be listed as errors.

Writing a program that accurately reflects which records are out of sequence is difficult, if not impossible. The person using the report should be warned that an out-of-sequence condition *will* be listed on the report, but it may be necessary to look at the input file to determine exactly which record or records have caused the condition.

Check-Digit Calculations

Check digits are sometimes used to validate numeric fields and to help guarantee that digits have not been transposed—for example, 123 entered instead of 132. When check digits are used, a mathematical formula is applied to the original value before it is input. The digit that results is appended to the original number. Then, when the value is input, the computer calculates the check digit to see if the check digit entered matches the check digit calculated. If they agree, the record is assumed to be valid.

To illustrate the process, let's use one method of generating a check digit as part of a customer number. Suppose the original number is 84913.

1. Multiply alternate digits by 2.

$$8 * 2 = 16$$

$$9 * 2 = 18$$

$$3 * 2 = 6$$

2. Add the digits that result from the multiplications (*Note:* 16 becomes the digits 1 and 6 and 18 becomes the digits 1 and 8).

$$1 + 6 + 1 + 8 + 6 = 22$$

3. Add the digits from the original number not included in step 1 to the result in step 2.

$$4 + 1 + 22 = 27$$

4. Use the 1s place of the result of step 3 (the value 7) as the check digit.

5. The check digit then becomes the last digit of the customer number 849137.

When the program validates the customer number, the computer uses the first five digits of the customer number entered and recomputes the check digit. If it is the same as the last digit of the customer number entered, it is assumed that no error was made when entering the number. This method cannot ensure that the correct customer number was entered, but it does find most errors in which the data-entry person reversed, or transposed, digits.

Table or File Lookup

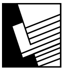
This technique involves looking up a value such as a customer number, student name, or part number in a table (array) or file to see if the value entered is valid. An example of this type of validation is illustrated in Chapter 6. Again, as in previously described validation methods, it is important to remember that a program can confirm that the value entered is in a table or file, but it cannot validate the value entered as the one intended by the person entering the data.

Visual Verification

Visual verification methods ask the user to look at the value entered and make sure it is correct. Prompts such as ARE YOU SURE?, IS THIS RECORD CORRECT?, or PRESS ENTER TO WRITE RECORD TO FILE force the data-entry

person to verify data visually and respond as to whether the program should write the data to a file, terminate the program, and so on. An illustration of this type of verification is shown later in this chapter. Programmers should not rely on visual verification to prevent program errors, however, if the data entry person is careful, this method does help ensure that the data has been entered correctly.

- Before continuing to the next section, answer the following question at the end of the chapter: Matching Part 4; True/False 16, 17; Multiple Choice 18, 19; Short Answer 4.



4.10 What Are the Differences between Batch and On-Line Data Entry and Validation?

There are two modes for processing data. How data are entered and validated depends on whether the system operates in batch or on-line mode. With **batch**, groups (batches) of data are collected. The data are processed at specific time intervals such as daily, weekly, or monthly. Payroll and billing are examples of applications that are typically done in batch mode. In **on-line** mode, data items are processed when they occur. Examples of on-line applications include services that sell tickets for sporting events or concerts and inventory systems that update inventory totals as soon as a sale is made.

In batch mode a **validation program** checks the input data. Sometimes the validation program is part of the data-entry process (see example in Section 4.12). At other times it is a separate program (see example in Section 4.11). Output generally consists of a list of errors and a file containing the valid records. If the input file has errors, there are two choices in how processing continues. These are explained in the following examples.

- Consider an order entry and processing system: Orders are processed daily. At the end of each day, the orders are entered into the computer, and a validation program is executed to confirm that the order records are as correct as possible. The system then processes the orders for all valid records. The records having errors are manually corrected and saved to run with the next day's batch.

- Consider a payroll system: Paychecks are generated weekly. The amount of each paycheck depends on employee time records. The time records are collected in a batch. At the end of the week, data from the time records are entered into the computer, and a validation program is executed to confirm that the records are as correct as possible. In the order entry and processing example, it was not a serious problem to make corrections and run orders the next day. Here it would be a very serious problem to tell an employee that he or she had to wait a week for a paycheck because the data on a time record were invalid! In this type of system, the actual payroll program is not run until all input records are valid.

A data clerk normally makes error corrections. The clerk uses the error report produced by the validation program to make the corrections to the input file. If it is possible for an input record to contain more than one error, all errors should be listed because, most likely, the clerk will correct only those errors listed on the report. This means that a program should *not* stop looking for errors once the first error is found.

In on-line mode, a single program both validates the data as they are entered and processes the validated data. Earliest on-line applications used **interactive programs** (programs that ask questions of and receive answers from the user) to enter and validate data. Section 4.13 and Section 4.14 discuss how menu-driven and event-driven programming languages validate input.

- Before continuing to the next section, answer the following questions the at end of the chapter: True/False 18, 19.

4.11 Example: **Input Validation for a Batch Program**

In this example, we validate the input for an accounts receivable system. As output, the program produces both an error report and a file containing the valid records. The file containing the valid records has the same format as the input file. The report (see layout chart below) contains at most 55 lines (including heading, detail, and blank lines) on each page. Page numbers are included. If multiple errors occur for a single input record, the record is listed only one time; however, all error messages are shown.

Layout Chart

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789	6 0123456789	7 0123456789
1				INVALID	RECORDS			PAGE ZZ9
2								
3			RECORD				ERROR	
4								
5	XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXX X	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
6	XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXX X	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
7					X	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
8								

The data dictionary for this example is shown below. Notice that validation rules are included in the input definitions. All storage locations are global so no scope is shown.

Data Dictionary

Input Area—ACCOUNT-IN-FILE

Name	Description	Type	Validation Rules
ACCT-NUM	Customer's account number	Numeric or Nonnumeric	Must be numeric and in increasing sequence (that is, the number of the current record is greater than the number on the previous record, and so on)
NAME	Customer name	Nonnumeric	None
PRODUCT	Product number	Numeric or Nonnumeric	Must be numeric
QUANTITY	Quantity ordered	Numeric or Nonnumeric	Must be numeric and greater than 0

Output Areas—REPORT and ACCOUNT-OUT FILE

Work Area

Name	Description	Type	Initial Value	Calculation
HEADING1	See spacing chart—line 1			
"INVALID RECORDS"		Constant		
"PAGE"		Constant		
PAGE-NUMBER		Numeric	0	1 is added for each page
HEADING2	See spacing chart—line 3			
"RECORD"		Constant		
"ERROR"		Constant		
ERROR-LINE	Error line (see spacing chart lines 5–7)			
BAD-RECORD	Record with error(s)	Nonnumeric		
ERROR-MESSAGE	Description of error	Nonnumeric		
LINES-COUNTER	Number of detail lines written on the page	Numeric	0	1 is added for each line written on the report
MAX-LINES	Maximum lines per page	Numeric	55	
RECORD-ERROR	Used to indicate whether the record has any errors	Nonnumeric		
FILE-ERROR	Used to indicate whether the file has any errors	Nonnumeric	NO	
PREVIOUS-NUM	Used to indicate the customer's account number from the previous valid record	Numeric	0	

The hierarchy chart, flowchart, and pseudocode are shown in Figure 4–F.

To comprehend the solution to this problem, you first must understand how the two indicators, FILE-ERROR and RECORD-ERROR, are used by the program. **FILE-ERROR** tracks whether any error messages are written on the report. **RECORD-ERROR** tracks whether any errors occur when checking the record currently being processed.

FILE-ERROR is given an initial value of "NO" in INITIALIZE. If its value is still "NO" in TERMINATE, it means that no error lines were written on the report during processing, and "NO ERRORS FOUND—ALL RECORDS CORRECT" is written on the report.

When an error occurs, the value stored in FILE-ERROR must be changed to "YES." Most beginning programmers would include an instruction in WRITE-ERROR to move "YES" to FILE-ERROR. This, however, is not the most efficient place to do so. Notice in this example that "YES" is moved to FILE-ERROR in the HEADINGS routine. The following explanation clarifies this.

Suppose that 100 errors occur. The WRITE-ERROR routine would be called 100 times, whereas the HEADINGS routine would only be done two times. By moving "YES" to FILE-ERROR in HEADINGS, the MOVE instruction

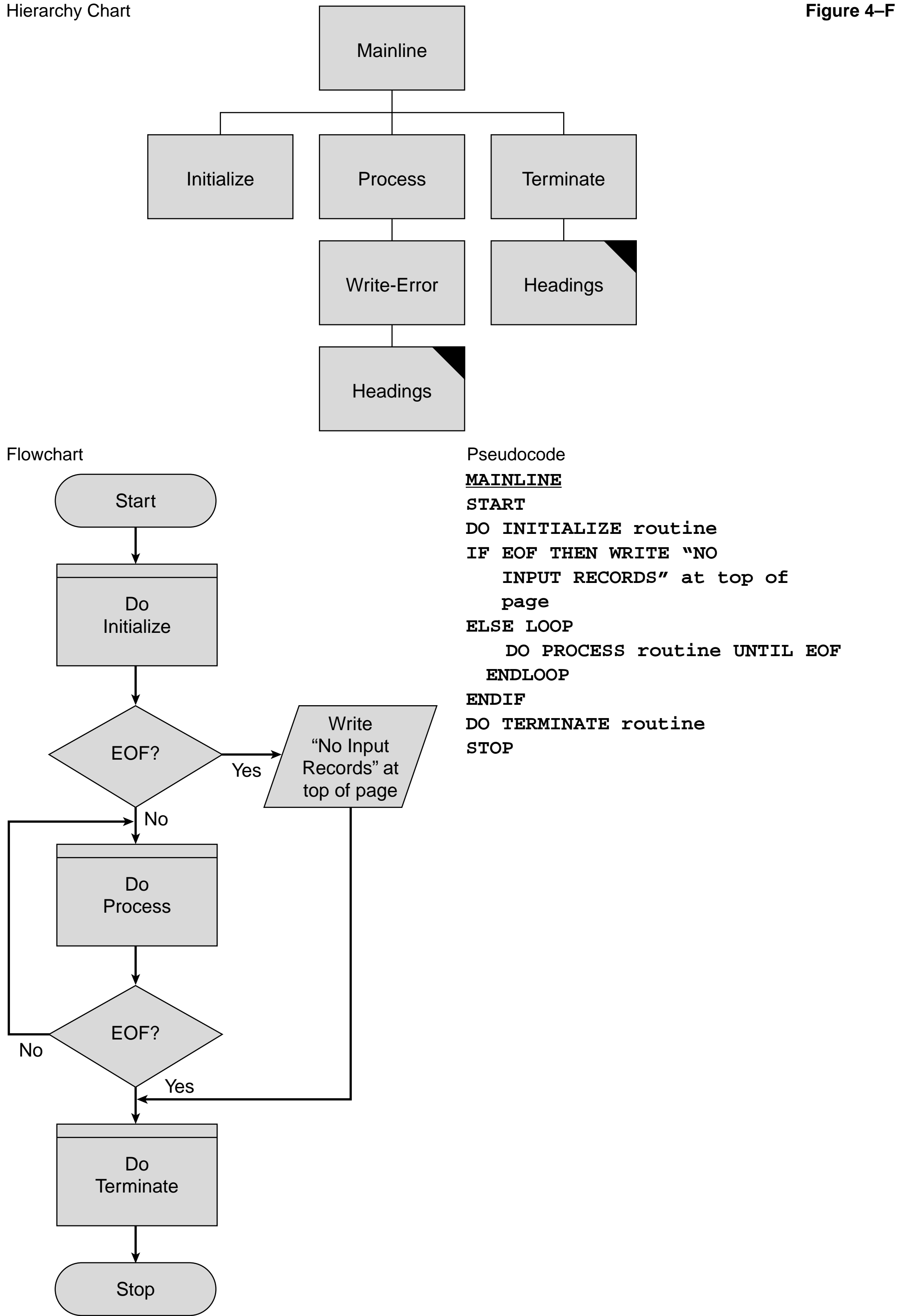


Figure 4–F

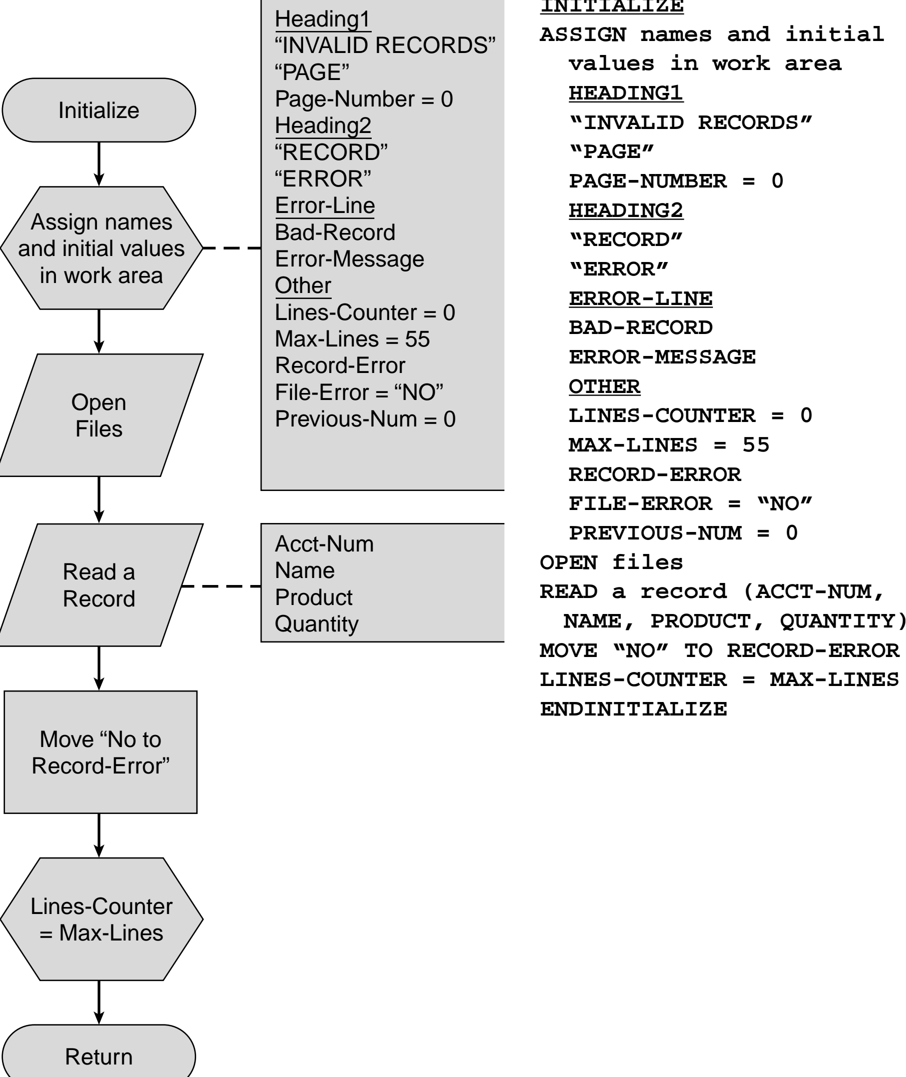


Figure 4–F (*continued*)

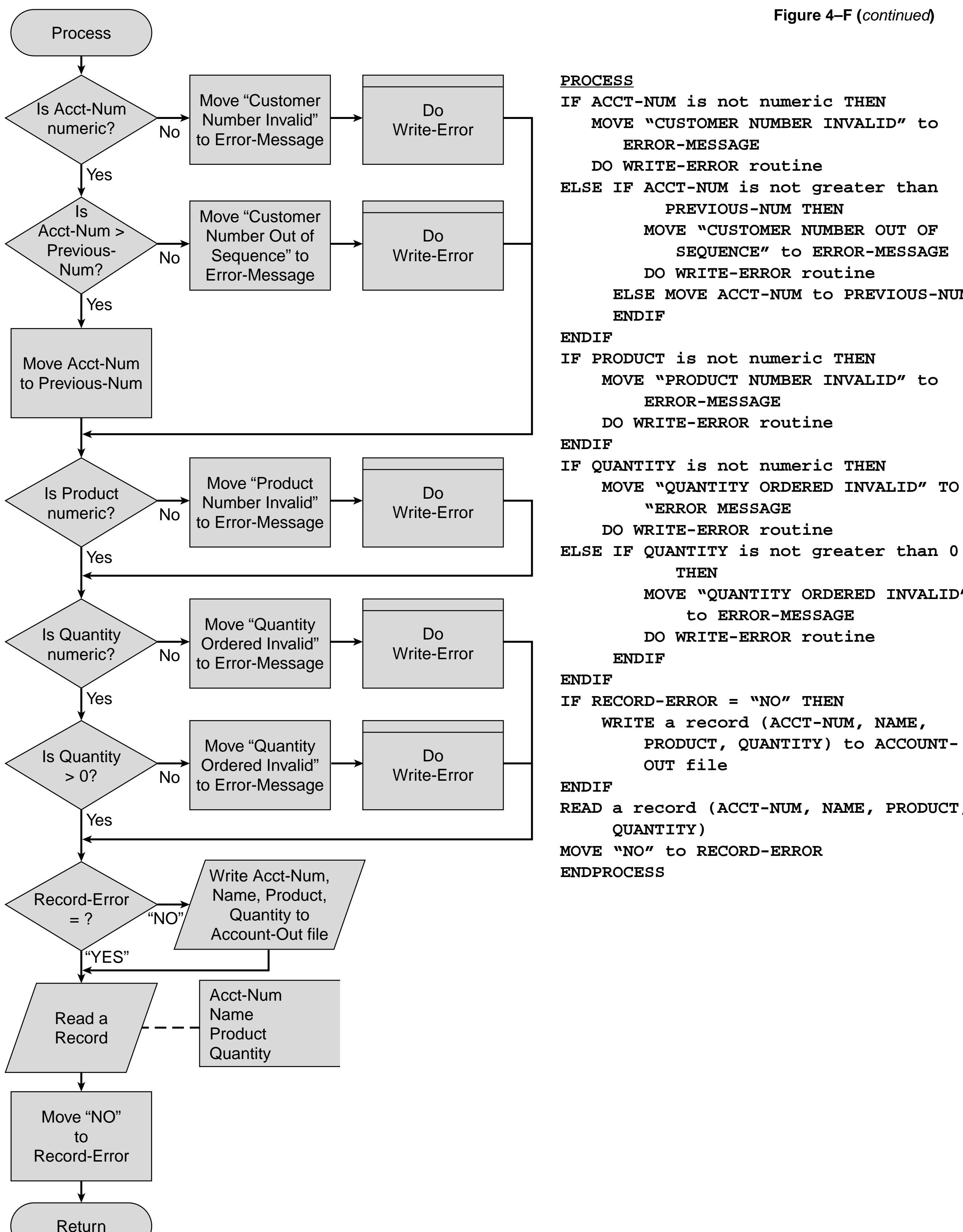
is executed 98 fewer times than if the MOVE was done in WRITE-ERROR. In TERMINATE, FILE-ERROR will be "NO" if the HEADINGS routine has not yet been done. HEADINGS is called only by WRITE-ERROR and TERMINATE. Therefore, if no error messages have been written, FILE-ERROR will still be "NO" when TERMINATE is called.

RECORD-ERROR tracks errors on individual records, not the entire input file. It is used for two reasons:

1. If the record currently being processed contains no errors, then the record must be written to the output file. This is done at the end of PROCESS by seeing whether RECORD-ERROR has a value of "NO" (that is, there were no errors when processing this record).

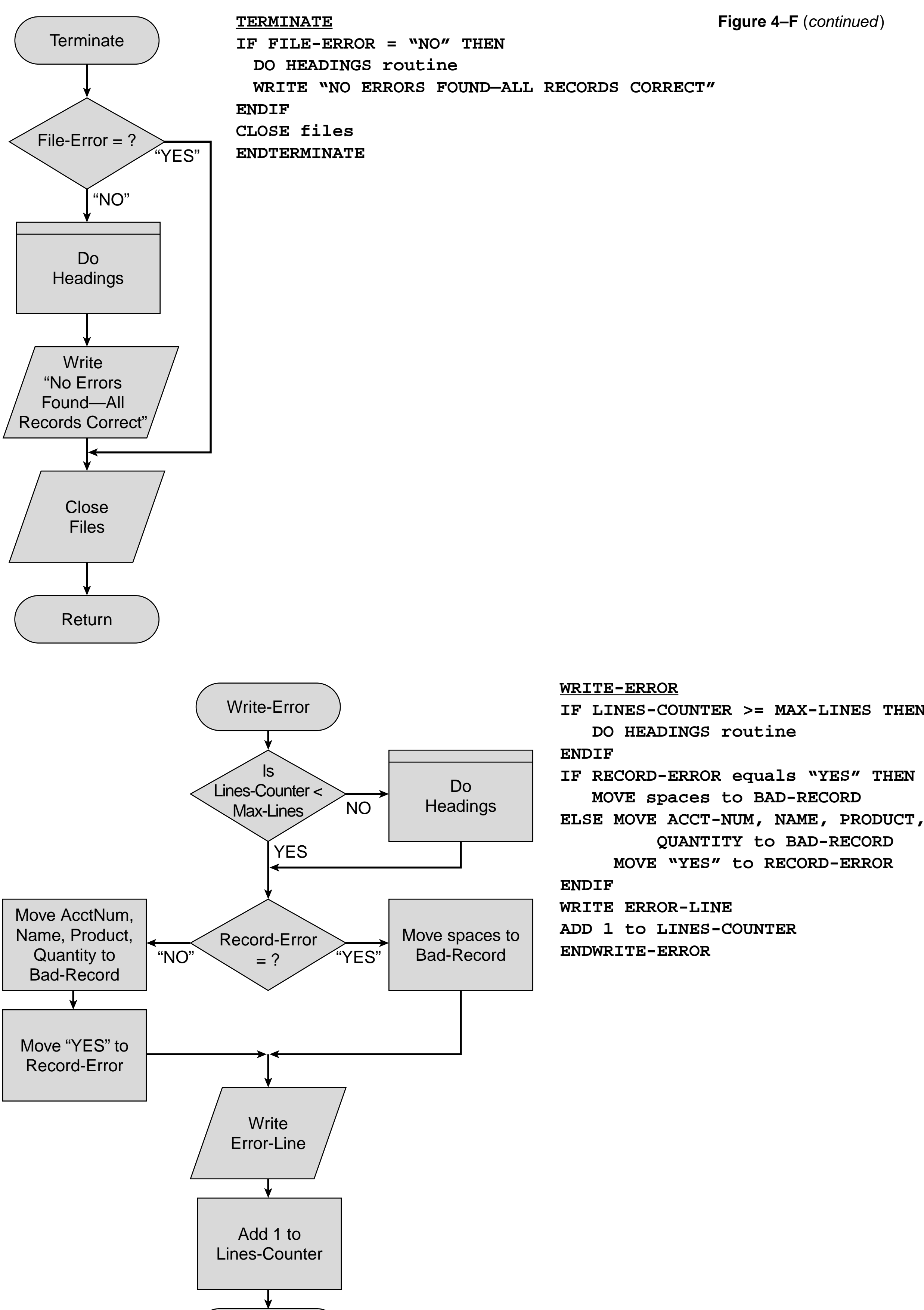
2. For every record input, the content of the record is listed only once on the error report, regardless of how many errors the record contains. Therefore, the WRITE-ERROR routine must know whether the error message being written is the first or a subsequent error for a particular input record. WRITE-ERROR

Figure 4–F (*continued*)



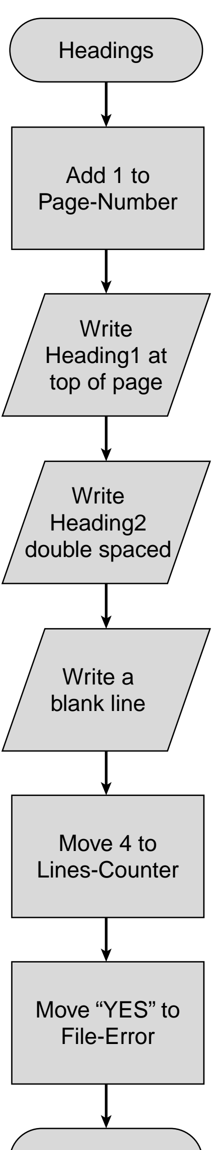

```
PROCESS
IF ACCT-NUM is not numeric THEN
   MOVE "CUSTOMER NUMBER INVALID" to
      ERROR-MESSAGE
   DO WRITE-ERROR routine
ELSE IF ACCT-NUM is not greater than
          PREVIOUS-NUM THEN
        MOVE "CUSTOMER NUMBER OUT OF
           SEQUENCE" to ERROR-MESSAGE
        DO WRITE-ERROR routine
     ELSE MOVE ACCT-NUM to PREVIOUS-NUM
     ENDIF
ENDIF
IF PRODUCT is not numeric THEN
    MOVE "PRODUCT NUMBER INVALID" to
        ERROR-MESSAGE
    DO WRITE-ERROR routine
ENDIF
IF QUANTITY is not numeric THEN
    MOVE "QUANTITY ORDERED INVALID" TO
        "ERROR MESSAGE
    DO WRITE-ERROR routine
ELSE IF QUANTITY is not greater than 0
           THEN
        MOVE "QUANTITY ORDERED INVALID"
            to ERROR-MESSAGE
        DO WRITE-ERROR routine
     ENDIF
ENDIF
IF RECORD-ERROR = "NO" THEN
    WRITE a record (ACCT-NUM, NAME,
        PRODUCT, QUANTITY) to ACCOUNT-
        OUT file
ENDIF
READ a record (ACCT-NUM, NAME, PRODUCT,
     QUANTITY)
MOVE "NO" to RECORD-ERROR
ENDPROCESS
```

Figure 4–F (*continued*)




```
TERMINATE
IF FILE-ERROR = "NO" THEN
  DO HEADINGS routine
  WRITE "NO ERRORS FOUND—ALL RECORDS CORRECT"
ENDIF
CLOSE files
ENDTERMINATE
```

```
WRITE-ERROR
IF LINES-COUNTER >= MAX-LINES THEN
   DO HEADINGS routine
ENDIF
IF RECORD-ERROR equals "YES" THEN
   MOVE spaces to BAD-RECORD
ELSE MOVE ACCT-NUM, NAME, PRODUCT,
         QUANTITY to BAD-RECORD
     MOVE "YES" to RECORD-ERROR
ENDIF
WRITE ERROR-LINE
ADD 1 to LINES-COUNTER
ENDWRITE-ERROR
```




```
HEADINGS
ADD 1 to PAGE-NUMBER
WRITE HEADING1 at top of page
WRITE HEADING2 double spaced
WRITE a blank line
MOVE 4 to LINES-COUNTER
MOVE "YES" to FILE-ERROR
ENDHEADINGS
```

Figure 4–F (*continued*)

checks the value in RECORD-ERROR. If the value is "NO," it is the first error for the input record, so the input record is written on the report. If the value is "YES" (that is, there have already been errors for this record), blanks (spaces) are shown on the report in the area that would otherwise contain the record.

Because every input record should have a fresh start, "NO" is moved to RECORD-ERROR whenever a record is read. Therefore, each time PROCESS is called, the value stored in RECORD-ERROR is "NO." The value stored in RECORD-ERROR is changed to "YES" whenever the WRITE-ERROR routine is called.

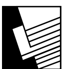
Let's review the use of these indicators. How does RECORD-ERROR differ from FILE-ERROR? Recall that FILE-ERROR determines if there are any errors on the entire input file. It is initialized to "NO" in INITIALIZE, and its value is changed to "YES" when an error occurs. Once the value is changed to "YES," it remains "YES" for the rest of the program. The value is checked in

TERMINATE to determine whether the message "NO ERRORS FOUND—ALL RECORDS CORRECT" should be written on the report.

RECORD-ERROR, on the other hand, indicates whether a particular record has errors. It is set to "NO" for each record read and changed to "YES" whenever an error occurs. It is checked in PROCESS to determine if the record should be written to the file of valid records and in the WRITE-ERROR routine to determine the format of the error line.

Now that you understand how the indicators are used, examine PROCESS in more depth. Be sure to note that ACCT-NUM is not saved in PREVIOUS-NUM for comparison purposes unless it is valid. We do not want to compare the next input record with a value that is nonnumeric or out of sequence. Also note that if the value entered for ACCT-NUM is not numeric, no check is made to see if it is greater than the previous number entered. Such a check would not make sense. Similarly, if QUANTITY is not numeric, no check is made to see if it is greater than 0.

- Before continuing to the next section, answer the following questions at the end of the chapter: Multiple Choice 20, 21, 22.



4.12 Example: **Using an Interactive Program to Create and Validate an Input File**

Sometimes the data required for processing is input interactively on-line. When using an interactive program to create a file, records can be validated as they are entered. No error report is required: Error messages are displayed on the screen, and invalid data are corrected immediately.

In this example we create an inventory file containing part numbers and the number of items added to or removed from inventory. The user of the program will enter the part number, number of items, and whether the item is an addition to or removal from inventory. Figure 4–G (page 172) shows the interactive procedure—the lines displayed, the possible responses, and the programming paths followed based on the responses. (*Note:* For ease of reading, all displayed lines are underlined in the figure. Underlines are *not* displayed.)

The data dictionary for this example follows. Notice that validation rules are included in the input definitions.

Data Dictionary

Input Area

Name	Description	Type	Scope	Validation Rules
RESPONSE			Global	
	Part number	Numeric or Nonnumeric		Must be between 0 and 100
	Number of items	Numeric or Nonnumeric		Must be a positive number
	Addition to or removal from inventory	Nonnumeric		Must be "A", "a", R", or "r"

Output Areas—DISPLAY-LINE and INVENTORY-FILE

Work Area

Name	Description	Type	Initial Value	Scope	Routines Accessing Location
LINE1 "INVENTORY FILE CREATE"	See interactive procedure	Constant		Local	INITIALIZE
LINE2 "ENTER PART NUMBER OR ZERO (0) TO TERMINATE THE PROGRAM"	See interactive procedure	Constant		Local	DISPLAY-LINE2
LINE3 "ENTER NUMBER OF ITEMS"	See interactive procedure	Constant		Local	DISPLAY-LINE3
LINE4 "IS THIS AN ADDITION TO (A) OR REMOVAL FROM (R) INVENTORY?"	See interactive procedure	Constant		Local	DISPLAY-LINE4
LINE5 "IS THE ABOVE RECORD CORRECT (Y OR N)?"	See interactive procedure	Constant		Local	WRITE-RECORD
LINE6 "PROGRAM TERMINATED"	See interactive procedure	Constant		Local	TERMINATE
ERROR1 "RESPONSE MUST BE BETWEEN 0 AND 100"	See interactive procedure	Constant		Local	DISPLAY-LINE2
ERROR2 "RESPONSE MUST BE A POSITIVE NUMBER"	See interactive procedure	Constant		Local	DISPLAY-LINE3
ERROR3 "RESPONSE MUST BE A OR R"	See interactive procedure	Constant		Local	DISPLAY-LINE4
ERROR4 "RESPONSE MUST BE Y OR N"	See interactive procedure	Constant		Local	WRITE-RECORD
DONE	Used to indicate whether processing is complete	Nonnumeric	NO	Global	
VALID-RESPONSE	Used to indicate whether the response is valid	Nonnumeric		Global	
PART	Part number	Numeric		Global	
NUMBER	Number of items	Numeric		Global	
A OR R	Addition to or removal from inventory	Nonnumeric		Global	

INVENTORY FILE CREATE
Clear the screen prior to displaying this line.

ENTER PART NUMBER OR ZERO (0) TO TERMINATE THE PROGRAM
Display a blank line prior to displaying this line.
A valid response is a number between 0 and 100.
Validate the response. If it is invalid, write the message RESPONSE MUST BE BETWEEN 0 AND 100 and repeat the request for part number.
If the response is zero, display a blank line followed by PROGRAM TERMINATED and stop the program.
If the response is greater than zero, save the response; and continue with the next line to be displayed.

ENTER NUMBER OF ITEMS
A valid response is a positive number.
If the response is not valid, display the message RESPONSE MUST BE A POSITIVE NUMBER, and repeat the request for number of items.
If the response is valid, save the response, and continue with the next line.

IS THIS AN ADDITION TO (A) OR REMOVAL FROM (R) INVENTORY?
Valid responses are A, a, R, or r.
If the response is not valid, write the message RESPONSE MUST BE A OR R, and repeat the request.
If the response is a valid uppercase (A or R) response, save the response. If it is lowercase (a or r), convert the response to uppercase before saving it.

IS THE ABOVE RECORD CORRECT (Y OR N)?
Display a blank line and the record created prior to this line.
Valid responses are Y, y, N, or n.
If the response is not valid, write the message RESPONSE MUST BE Y OR N, and repeat the request.
If the response is Y or y, write the record to the output file.
Clear the screen, and start again from the ENTER PART NUMBER . . . request.

Figure 4–G
Interactive Procedure

The hierarchy chart, flowchart, and pseudocode for this example are shown in Figure 4–H.

RESPONSE is the input area. It holds the responses entered by the user. DISPLAY-LINE and INVENTORY-FILE are the output areas. INVENTORY-FILE is used for the inventory file. DISPLAY-LINE is used for the display. The actual lines to be displayed are described as work areas and used as local constants in the program.

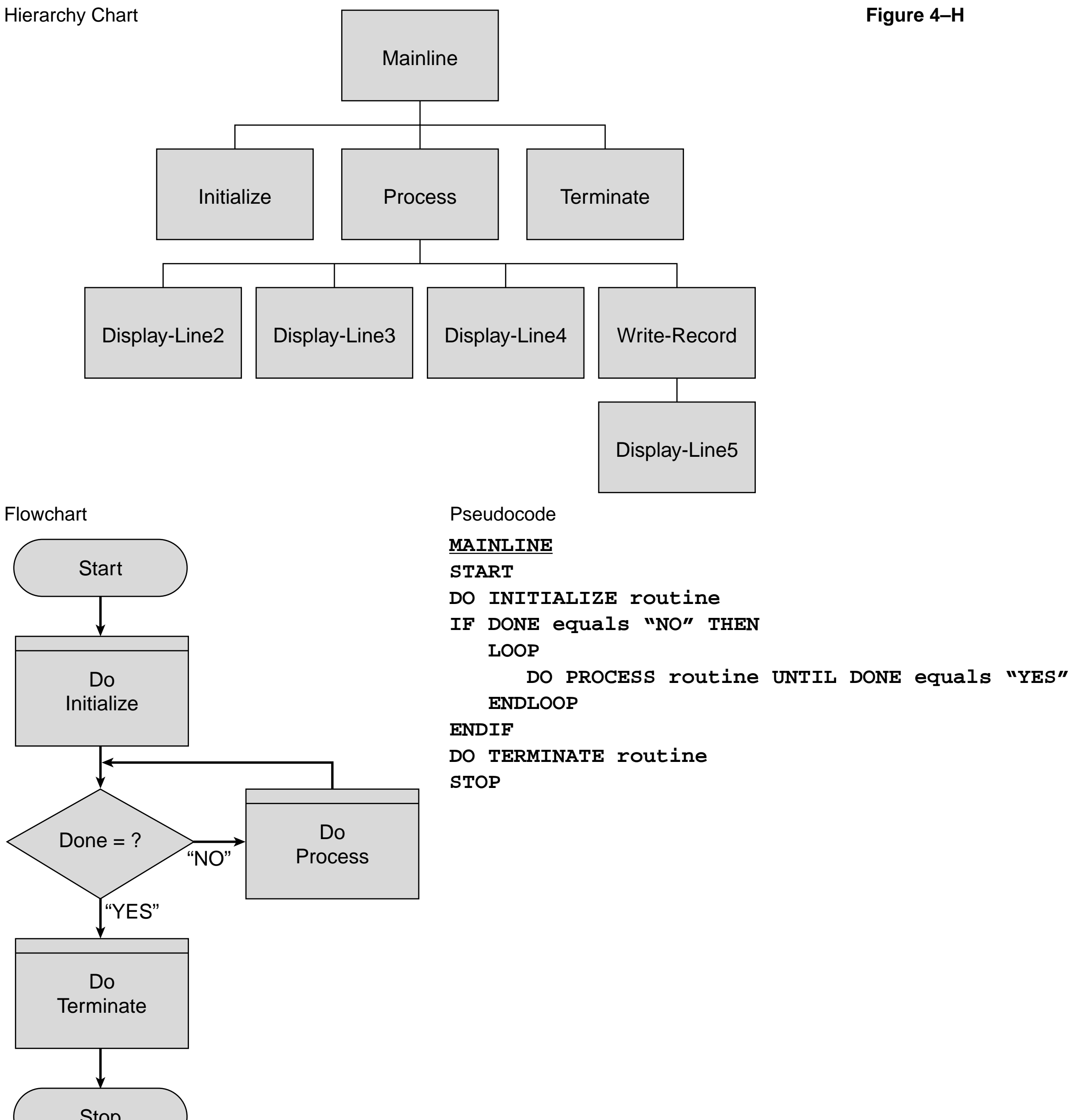


Figure 4–H

A **VALID-RESPONSE** indicator is defined as a work area. All routines requesting user input contain a loop to request and validate the user's response. VALID-RESPONSE tracks whether the response entered by the user is valid. If the response is valid, "YES" is moved to VALID-RESPONSE, and the program exits the loop. If the response is not valid, then VALID-RESPONSE becomes "NO," and the program repeats the loop.

The PROCESS routine illustrates how VALID-RESPONSE is used. PROCESS controls the lines displayed. It is called once for each record written to the output file. PROCESS first does DISPLAY-LINE2. It does this routine until a valid response is obtained. DISPLAY-LINE2 displays the line and reads the response. If the response is in the range 0 to 100, "YES" is moved to

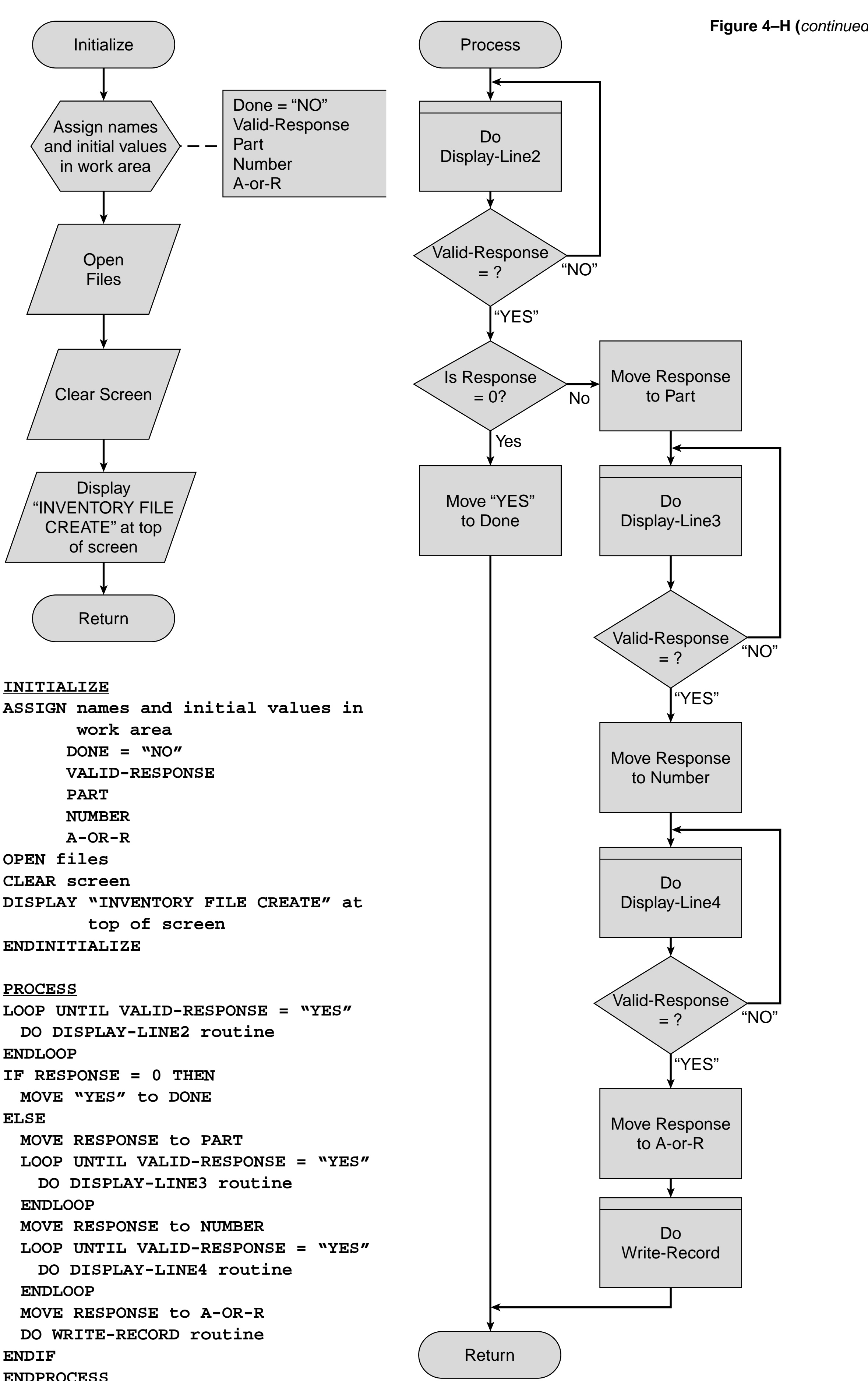
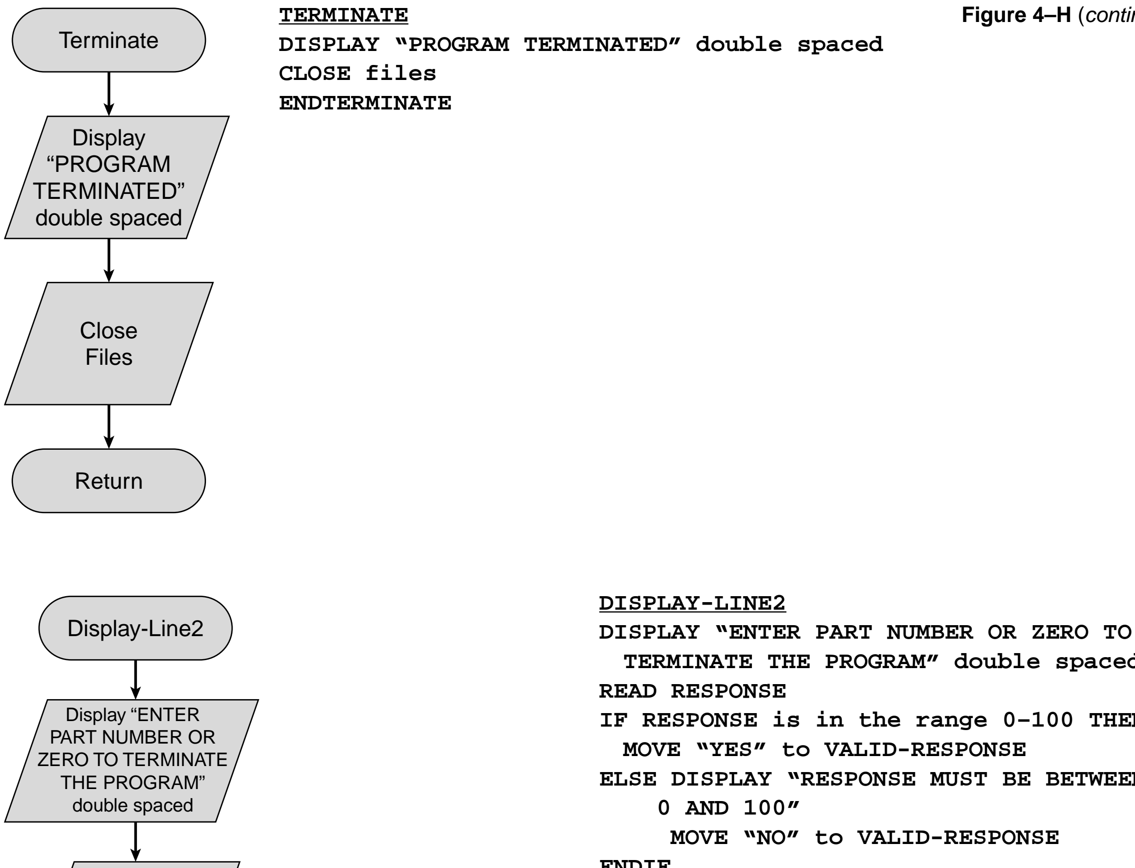


Figure 4–H (*continued*)

```
INITIALIZE
ASSIGN names and initial values in
       work area
      DONE = "NO"
      VALID-RESPONSE
      PART
      NUMBER
      A-OR-R
OPEN files
CLEAR screen
DISPLAY "INVENTORY FILE CREATE" at
        top of screen
ENDINITIALIZE

PROCESS
LOOP UNTIL VALID-RESPONSE = "YES"
  DO DISPLAY-LINE2 routine
ENDLOOP
IF RESPONSE = 0 THEN
  MOVE "YES" to DONE
ELSE
  MOVE RESPONSE to PART
  LOOP UNTIL VALID-RESPONSE = "YES"
    DO DISPLAY-LINE3 routine
  ENDLOOP
  MOVE RESPONSE to NUMBER
  LOOP UNTIL VALID-RESPONSE = "YES"
    DO DISPLAY-LINE4 routine
  ENDLOOP
  MOVE RESPONSE to A-OR-R
  DO WRITE-RECORD routine
ENDIF
ENDPROCESS
```

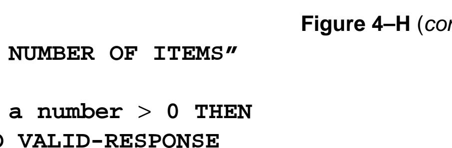
Figure 4–H (*continued*)



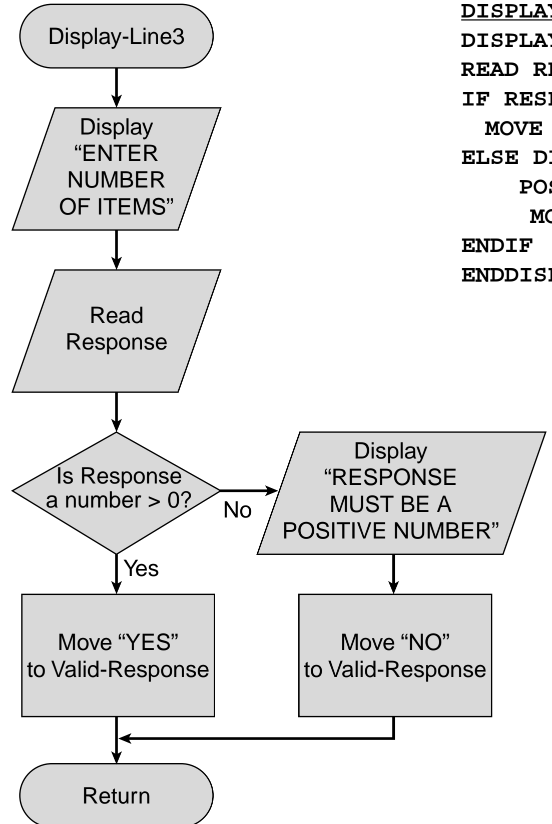

```
TERMINATE
DISPLAY "PROGRAM TERMINATED" double spaced
CLOSE files
ENDTERMINATE
```

```
DISPLAY-LINE2
DISPLAY "ENTER PART NUMBER OR ZERO TO
  TERMINATE THE PROGRAM" double spaced
READ RESPONSE
IF RESPONSE is in the range 0-100 THEN
  MOVE "YES" to VALID-RESPONSE
ELSE DISPLAY "RESPONSE MUST BE BETWEEN
     0 AND 100"
      MOVE "NO" to VALID-RESPONSE
ENDIF
ENDDISPLAY-LINE2
```

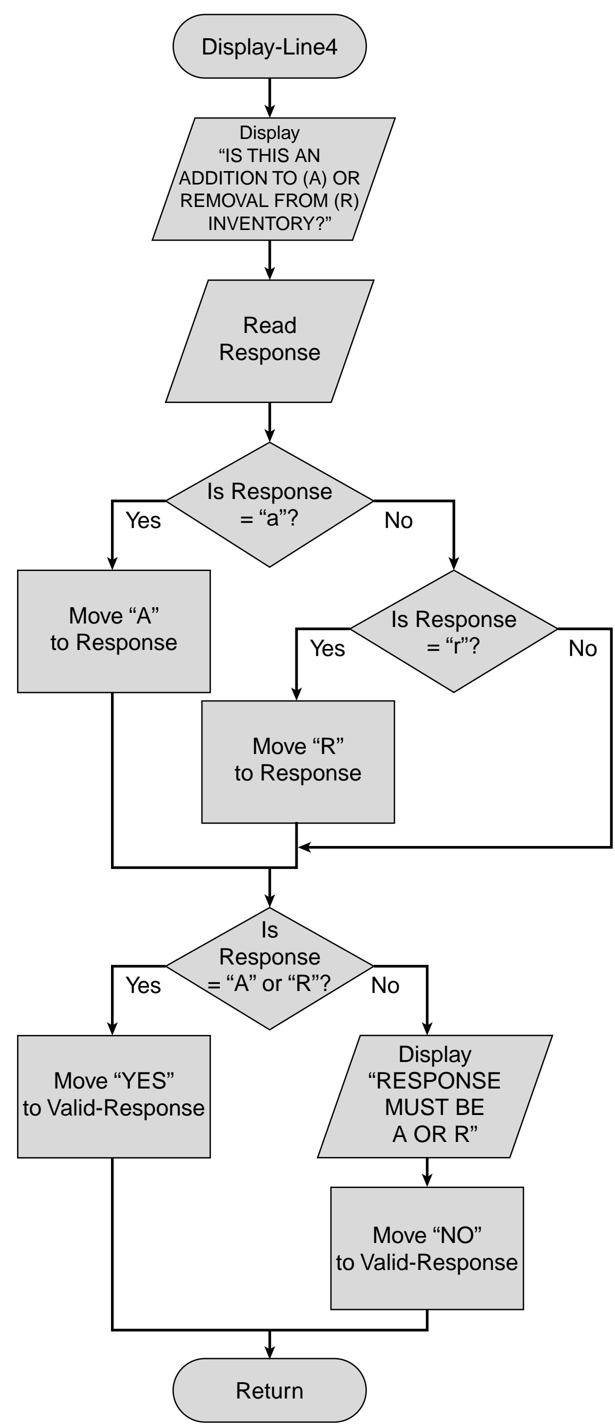
Figure 4–H (*continued*)



```
DISPLAY-LINE3
DISPLAY "ENTER NUMBER OF ITEMS"
READ RESPONSE
IF RESPONSE is a number > 0 THEN
  MOVE "YES" TO VALID-RESPONSE
ELSE DISPLAY "RESPONSE MUST BE A
    POSITIVE NUMBER"
     MOVE "NO" to VALID-RESPONSE
ENDIF
ENDDISPLAY-LINE3
```




```
DISPLAY-LINE4
DISPLAY "IS THIS AN ADDITION TO (A) OR
        REMOVAL FROM (R) INVENTORY?"
READ RESPONSE
IF RESPONSE = "a" THEN MOVE "A" to
        RESPONSE
ELSE IF RESPONSE = "r" THEN MOVE "R"
        to RESPONSE
     ENDIF
ENDIF
IF RESPONSE = "A" or "R" THEN
     MOVE "YES" to VALID-RESPONSE
ELSE DISPLAY "RESPONSE MUST BE A OR R"
     MOVE "NO" to VALID-RESPONSE
ENDIF
ENDDISPLAY-LINE4
```



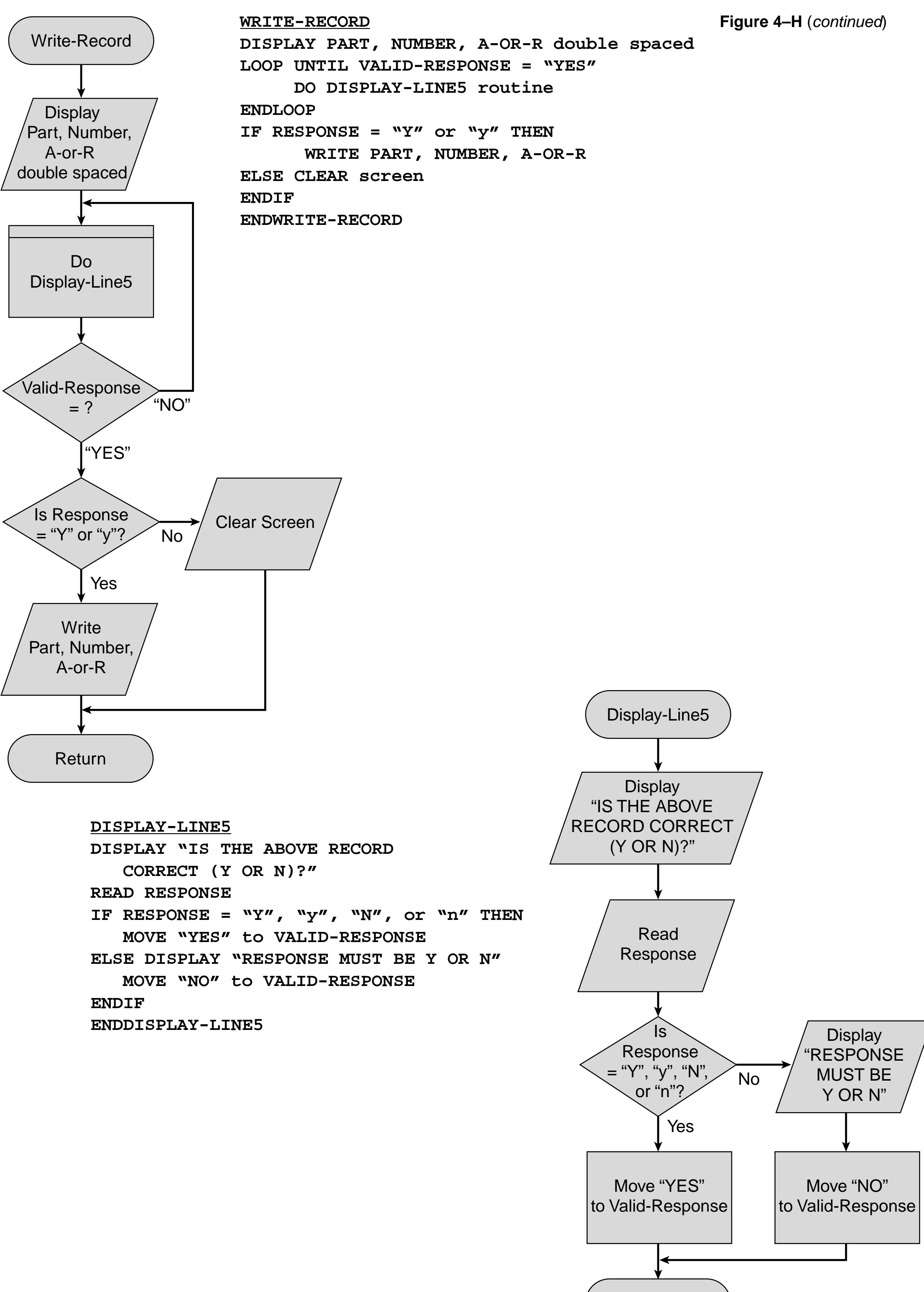


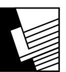
Figure 4–H (*continued*)

VALID-RESPONSE, and the program returns to PROCESS. If the response is not valid, an appropriate error message is displayed, and "NO" is moved to VALID-RESPONSE before returning to PROCESS and calling DISPLAY-LINE2 again.

Upon returning to PROCESS with a valid response, the value of RESPONSE is checked. If it is 0, "YES" is moved to **DONE** (an indicator used to track whether processing is complete), and the program goes back to the MAINLINE routine, calls TERMINATE, and ends processing. If RESPONSE is not 0, then the response is moved to PART.

PROCESS then enters a loop to display LINE3 and get a response for NUMBER. After receiving a valid response for NUMBER, PROCESS enters the loop that displays LINE4 and gets a value for A-OR-R. After obtaining a value for each field in the output record, WRITE-RECORD is called. WRITE-RECORD uses visual verification to determine whether the inventory record should be written to the output file. If the user does not want to write the inventory record, input for the next record is requested. If the user wants to write the inventory record, processing continues after writing the record.

- Before continuing to the next section, answer the following question at the end of the chapter: Multiple Choice 23.



4.13 What Is a Menu-Driven Program?

Some interactive programs require that the user make choices that control the processing. In such programs, creating **menus** to display the choices is one way to make a program **user-friendly,** that is, easy to use. Presenting a menu, or list, of functions that a program can do guides the user through the program. Programs that present a list of options this way, rather than by requiring that users memorize a list of commands, are said to be **menu-driven.**

In a typical menu-driven program, the first screen the user sees contains the **main menu,** that is, a list of the program's major functions. Selecting an option from the main menu either leads to another, more detailed submenu or directly to the specified task. A program that leads directly to the specified task has a one-level menu. Programs that lead to more detailed submenus have multiple-level menus. Some menu-driven programs have very complicated logic with many levels of menus.

The following is an example of a main menu that allows the user to choose a report based on a student file:

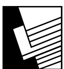
```
                STUDENT REPORT OPTIONS

   Enter the number corresponding to your choice:

         1 Report for all students
         2 Report for all students by department
         3 Report for all students by teacher
         4 Report for an individual student
         5 Exit program
```

The content of the report generated depends on the option selected by the user. If this program had multiple menu levels, a choice from the preceding main menu would generate an additional menu. For example, if the user selects choice 3, a list of teachers might be displayed on the next screen. The user would then choose from this menu which teacher or teachers to include on the report. The program must validate the menu choice to make sure it is a valid number.

- Before continuing to the next section, answer the following questions at the end of the chapter: True/False 20, 21.



4.14 How Do Event-Driven Programming Languages Enter and Validate Data?

Event-driven languages such as Visual Basic have a menu control as one of the objects that can be included on a screen. This control facilitates writing instructions or methods that are executed when a particular menu selection is made. Visual Basic has other objects including text boxes, command buttons, list boxes, input boxes, message boxes, option buttons, and check boxes that make the program more attractive visually as well as assist with data entry and validation. Figure 4–I shows two of these objects. In addition, Visual Basic also allows input to be entered interactively. Instead of the program making an input request as the example in Section 4.12 illustrates, the program displays an entire screen (called a **form**), and the values can be entered by the user in any order desired.

Figure 4–Ja is a form used by a pizza restaurant to process phone orders. The person taking the order asks the caller whether the order is for a pick-up or delivery. Data entry is done by using a mouse and clicking on one of the option buttons (Pick-Up or Delivery). Only one of the option buttons can be pressed at a time. Pressing or selecting one of the buttons by clicking a mouse on the button causes the other button to be deselected. Next, the order taker requests the customer information shown at the bottom of the screen. As data are entered into the text boxes, they can be validated if the programmer so chooses. Then the order taker clicks on the Place Pizza Order command button. When the button is clicked, a click event is generated. This event will contain methods for validating the input as follows:

- Either pick-up or delivery must be selected.
- If pick-up is selected, there must be data for customer name and telephone number.
- If delivery is selected, there must be data for customer name, address, and telephone number.

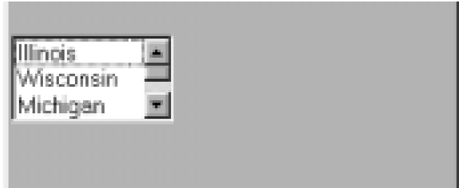


Figure 4–I

Sample List Box. When the user selects an item, it is highlighted. Scroll bars are provided to allow the user to easily select other items in the list.

Sample message box. The user selects one of the buttons shown.

If any errors occur, messages are displayed on the screen using a message box instructing the order taker to correct the input before continuing.

After validating the input, the screen shown in Figure 4–Jb is displayed.

The order taker requests the size of pizza desired. Again, this is done using option buttons, so only one size pizza may be selected. Then data are collected regarding ingredients. This is done using check boxes. When check boxes are used, more than one item in the group can be selected, so, in this case, more than one ingredient can be selected. After taking the order, the person doing the data entry will read the order back to the customer, thus verifying the size of pizza and ingredients requested. The order taker will then click on the Calc button to calculate the amount owed, click on the Print button to print the order for the chef, and click on Next Order to take the next phone order.

Figure 4–Ja

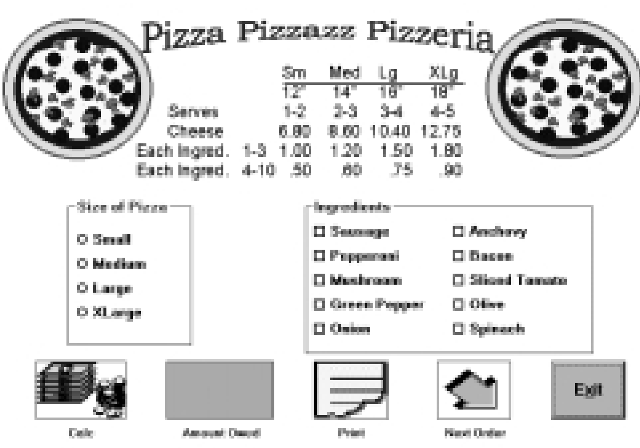
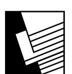


Figure 4–Jb

- Before continuing to the next section, answer the following questions at the end of the chapter: Matching Part 3; True/False 22, 23.



Summary

- Reports can be printed on paper or displayed on a screen. The four basic types of reports are detail reports, summary reports, exception reports, and query reports.
- The types of lines that can be written on a report are heading (header) lines, footing (footer) lines, detail (data) lines, and total (summary) lines.
- Heading and footing lines identify the report and describe the information it contains. These lines can consist of report headings/footings, page headings/footings, column headings, and control headings.
- Detail lines, which contain information from the input records, comprise the body of the report.
- Total lines summarize information contained on a report.
- The program must tell the computer how to space lines vertically on the page.
- The number of lines that can fit on a page or screen depends on the size of the paper and the size and type of the font selected.
- Programs must count the number of lines and tell the computer when headings should be printed or displayed. If a screen report contains multiple "screens," the programmer must avoid scrolling and pause the screen to allow the user time to read it. Screen reports require that the screen be cleared or refreshed prior to displaying each screen.
- Printer and display layout charts are used to design printed and screen output. Special characters are used on layout charts to indicate whether fields are nonnumeric or numeric, whether zero suppression is to occur, and if special characters, such as dollar signs and commas are included.
- It usually makes no sense to write a program when there is no input file. Sometimes, however, such programs are useful. An example of such a program is one that does mathematical computations on a range of numbers.
- External subroutines exist outside the program being coded. The way in which an external subroutine is accessed depends on the particular programming language. In general, however, you must provide the name of the routine being accessed, the field name(s) used in the subroutine, and the field name(s) where the results are to be placed. External subroutines are not shown on the hierarchy chart. Examples of external subroutines are routines that calculate the square root and obtain the current date from the computer system.
- When detail lines are single-spaced, the blank line between the headings and first detail line is written in the HEADINGS routine. When detail lines are double-spaced, the blank line is written when the detail line is written.
- Including the current date on a report is often desirable. Usually the current date can be accessed from the computer's operating system. If a date other than the current date is desired, the date is generally entered as input.
- Many types of errors can occur during program translation and execution. Translation (syntax) errors result from violating the language rules of a programming language. Execution (logic) errors occur after the translation process while the program is being executed.

- Some execution errors, such as data-exception errors, cause a program interrupt. Others allow the program to go to normal completion but incorrect output is produced.

- Data-validation techniques check for input errors prior to processing. Such techniques help prevent program errors due to incorrect data. These techniques include validation for data type, range of values, exact value or values, sign, data presence, data consistency, data reasonableness, data sequence, check-digit calculations, table or file look-up, and visual verification.

- Validation techniques can confirm that the input data will not "cause problems for" a program, but cannot confirm that the value entered is the one intended by the user.

- When codes or values are entered as alphabetic characters rather than numbers, checking for both uppercase and lowercase responses is important. This type of checking is necessary because the computer stores uppercase and lowercase characters differently. Therefore, comparing "Y" to "y" results in an unequal comparison.

- Sequence checking can show that records or values are out of sequence; however, the out-of-sequence values are not always the ones shown on the report.

- The two basic modes of processing data are batch and on-line. Batch systems operate by collecting groups (batches) of data that are processed at specific intervals such as a day, week, or month. On-line systems process data as they become available. In batch mode a separate validation program is usually used. In on-line mode the same program is used to enter, validate, and process the data.

- Because error corrections are generally made by a clerk, who corrects only indicated errors, it is important that the report lists all errors contained in the input file.

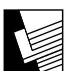
- FILE-ERROR indicators track whether any error messages are written on the report. If no errors are found, the program can issue a message indicating that all the input is valid and no corrections are needed.

- RECORD-ERROR indicators are used to facilitate writing records without errors to a file containing correct records. These indicators are also used to show all errors on the report, while showing the incorrect record one time only.

- Interactive programs are used for on-line data entry. A VALID-RESPONSE indicator is frequently used to control loops requesting input.

- Menus help make programs more user-friendly. A menu (list) of functions that a program can do helps guide the user through the program. Programs that use menus are said to be menu-driven. The first menu the user sees is called the main menu.

- Event-driven languages such as Visual Basic have a menu control and other objects including text boxes, command buttons, list boxes, message boxes, option buttons, and check boxes that can be placed on forms and displayed. These objects make programs more attractive visually as well as assist with data entry and validations. With event-driven programs, much more flexibility is provided for the user to enter data in any order desired.



Key Terms and Concepts

Detail reports (4.1)
Summary reports (4.1)
Exception reports (4.1)
Query reports (4.1)
Heading (header) lines (4.2)
Footing (footer) lines (4.2)
Report heading/footing lines (4.2)
Page (screen) heading/footing lines (4.2)
Column heading lines (4.2)
Control heading lines (4.2)
Detail (data) lines (4.2)
Total (summary) lines (4.2)
Carriage control characters (4.3)
Single-spaced lines (4.3)
Double-spaced lines (4.3)
Lines counter (4.3)
Font (4.3)
Font size (4.3)
Scrolling (4.3)
Printer and display layout charts (4.3)
Zero suppression (4.3)
External subroutine (4.4)
MAX-LINES (4.4)
Refreshing the screen (4.5)
Syntax (translation) error (4.8)
Execution (logic) error (4.8)
Program interrupt (4.8)
Data-exception error (4.8)
GIGO (4.8)
Data-validation techniques (4.8)
Data type validation (4.9)
Range of values validation (4.9)
Exact value or values validation (4.9)
Sign validation (4.9)
Data presence validation (4.9)
Data consistency validation (4.9)
Data reasonableness validation (4.9)
Data sequence validation (4.9)
Check-digit calculations (4.9)
Table (file) lookup (4.9)
Visual verification (4.9)
Batch mode (4.10)
On-line mode (4.10)
Validation program (4.10)
Interactive program (4.10)
FILE-ERROR indicator (4.11)
RECORD-ERROR indicator (4.11)
VALID-RESPONSE indicator (4.12)
DONE indicator (4.12)
Menu (4.13)
User-friendly (4.13)
Menu-driven (4.13)
Main menu (4.13)
Form (4.14)

Questions and Exercises

Matching Part 1 (Sections 4.1 to 4.5)

Indicate which term best fits the definition.

_______ **1.** No blank lines appear between detail lines on a report.

_______ **2.** Occurs on a screen display when lines are erased at the top of the screen to make room for more lines on the bottom of the screen.

_______ **3.** Page numbers shown on the bottom of each page of the report.

_______ **4.** A customer list showing one line for each record of the input file.

_______ **5.** Lines that give summary information.

_______ **6.** Describes how large each character is.

_______ **7.** A routine that exists outside of the program.

_______ **8.** The process of erasing only that part of the screen that needs to be updated.

a. Carriage control characters (4.3)
b. Column heading lines (4.2)
c. Control heading lines (4.2)
d. Detail (data) lines (4.2)
e. Detail report (4.1)
f. Double-spaced lines (4.3)
g. Exception report (4.1)

_______ **9.** Allows the programmer to keep track of the number of lines written on a page or displayed on a screen.

_______ **10.** Used to contain the maximum number of lines that can fit on a page or screen.

_______ **11.** Used to design printed output and screen displays.

_______ **12.** Describes how the letters, numbers, and other characters look.

_______ **13.** Lines generated from input records; comprise the body of the report.

_______ **14.** Replacing lead zeros with spaces or blanks.

_______ **15.** A report listing the average amount sold last month by all salespeople rather than the amounts for the individual salespeople contained on the input file.

_______ **16.** Describe the fields that appear on the report.

_______ **17.** A report listing only those students who are earning an A instead of listing all students contained on the input file.

_______ **18.** Generated from asking questions about a record or records contained on a database.

_______ **19.** Appear only at the beginning of the report, usually on a separate page or screen.

_______ **20.** Tells the printer the vertical line spacing for a report.

_______ **21.** Appear at the end of the report indicating that the report is complete.

_______ **22.** Appear at the top of each page of the report and contain such things as the name of the report, the page number, and the current date.

_______ **23.** Separate one group of data from another on a report.

_______ **24.** One blank line appears between lines on a report.

h. External subroutine (4.4)
i. Font (4.3)
j. Font size (4.3)
k. Lines counter (4.3)
l. MAX-LINES (4.4)
m. Page (screen) footing lines (4.2)
n. Page (screen) heading lines (4.2)
o. Printer and display layout charts (4.3)
p. Query report (4.1)
q. Refreshing the screen (4.5)
r. Report footing lines (4.2)
s. Report heading lines (4.2)
t. Scrolling (4.3)
u. Single-spaced lines (4.3)
v. Summary report (4.1)
w. Total lines (4.2)
x. Zero suppression (4.3)

Matching Part 2 (Sections 4.4 to 4.7)

Indicate which routine should be used to accomplish each function. (*Note:* You can select a choice zero, one, or more than one time.)

_______ **1.** Add 1 to PAGE-NUMBER.

_______ **2.** Get the current date for a heading line.

_______ **3.** Check LINES-COUNTER to see if there is enough room on the current page to print (display) the detail line.

_______ **4.** Print or display a blank line between the last heading line and first detail line when the detail lines are single-spaced.

_______ **5.** Print or display a blank line between the last heading line and first detail line when the detail lines are double-spaced.

_______ **6.** Reset the LINES-COUNTER to 0.

_______ **7.** Request that the last set of footings be printed or displayed.

a. Headings (or Footings)
b. Initialize
c. Mainline
d. Process
e. Terminate
f. Write-Detail (or Display-Detail)

Matching Part 3 (Sections 4.8 to 4.14)

Indicate which term best fits the definition.

_______ **1.** Tracks whether any error messages are written on the report.

_______ **2.** An error that results from violating the rules of a programming language.

_______ **3.** An error that occurs after the translation process while a program is being executed.

_______ **4.** An error that occurs as a result of an incorrect data type being in a storage location (for example, attempting to do arithmetic on a field that does not contain a number).

_______ **5.** Garbage in, Garbage out.

_______ **6.** Techniques that check for errors on input data.

_______ **7.** Processing is controlled by allowing the user to make choices from a list of options.

_______ **8.** An execution error that causes the program to terminate prior to normal completion.

_______ **9.** A program that is easy to use.

_______ **10.** A list of choices presented to the user.

_______ **11.** The menu that shows the major functions of a program; usually the first screen or menu displayed for the user.

_______ **12.** Indicator that tracks whether any errors occur when checking the record currently being processed.

_______ **13.** Collects groups of data that are processed at a specific time such as daily, weekly, or monthly.

_______ **14.** Indicator that tracks whether processing is complete.

_______ **15.** A program that checks input data for errors.

_______ **16.** A program that asks questions of and receives answers from an on-line user.

_______ **17.** Processes data immediately as they become available.

_______ **18.** Indicator that tracks whether the response entered by the user is valid.

_______ **19.** Screen displayed for the user.

a. Batch mode (4.10)
b. Data-exception error (4.8)
c. Data-validation techniques (4.8)
d. DONE indicator (4.12)
e. Execution (logic) error (4.8)
f. Form (4.14)
g. GIGO (4.8)
h. Interactive program (4.10)
i. Main menu (4.13)
j. Menu (4.13)
k. Menu-driven (4.13)
l. On-line mode (4.10)
m. Program interrupt (4.8)
n. RECORD-ERROR indicator (4.11)
o. FILE-ERROR indicator (4.11)
p. Syntax (translation) error (4.8)
q. User-friendly (4.13)
r. Validation program (4.10)
s. VALID-RESPONSE indicator (4.12)

Matching Part 4 (Section 4.9)

Indicate which type of validation is best for detecting each of the following errors. Choices can be used more than once.

_______ **1.** Ensures that an input field is not blank.

_______ **2.** Ensures that an amount entered is greater than 0.

_______ **3.** Ensures that the correct item was ordered.

_______ **4.** Ensures that a test grade is between 0 and 100.

a. Data type
b. Range of values
c. Exact value or values
d. Sign

_______ **5.** Ensures that the date entered is for the current year.

_______ **6.** Ensures that all five test grades are entered for a particular student.

_______ **7.** Ensures that TYPE-IN is a 1, 2, or 3.

_______ **8.** Ensures that a test grade is numeric.

_______ **9.** Ensures that the part number ordered is an actual part stocked by the company.

_______ **10.** Ensures that the number of grades entered for a student is the same as the number of courses taken.

_______ **11.** Ensures that as each record is read, ORDER-NUMBER is greater than ORDER-NUMBER of the previous record read.

_______ **12.** Ensures that digits are not reversed or transcribed.

e. Data presence
f. Data consistency
g. Data reasonableness
h. Data sequence
i. Check-digit calculations
j. Table (file) lookup
k. Visual verification

True/False

Indicate whether each statement is true or false. If the statement is false, change it to make it true.

1. Any report can contain either heading lines or footing lines, but not both. (4.2)
2. Font size affects the number of lines that can be displayed on a screen or printed on a piece of paper. (4.3)
3. The computer does not automatically know how many lines to print on a page. As a result, the computer will even print over the perforations in the paper unless the program tells it to do otherwise. (4.3)
4. Every program requires at least one input file. (4.4)
5. The date obtained from the computer system should be used when you do *not* want to use the current date. (4.6)
6. Consider the following spacing chart.

	0 123456789	1 0123456789	2 0123456789	3 0123456789
1	MM/DD/YY	STU	DENT AGES	
2				
3		NAME		AGE
4				
5	XXXXX	XXXXXXXXXX	XXXX	XX
6	XXXXX	XXXXXXXXXX	XXXX	XX

   The best place to write the blank line (line 4) between the headings and detail lines is in the HEADINGS routine. (4.7)

7. Suppose the detail lines in question 6 above were double-spaced. The best place to write the blank line (line 4) between the headings and detail lines is in the HEADINGS routine. (4.7)
8. If the current date is obtained in the HEADINGS routine, it is possible that the date will change in the middle of the report. This is *not* a problem if the date is obtained in the INITIALIZE routine. (4.7)

9. A program that translates with no errors will sometimes produce incorrect output when executed. (4.8)
10. When a logic error occurs, a message is printed or displayed during translation, and the program is not executed. (4.8)
11. Translation errors are also called logic errors. (4.8)
12. A syntax error occurs when rules of a programming language are violated. (4.8)
13. Some logic errors cause a program interrupt. (4.8)
14. Data-exception errors generate program interrupts. (4.8)
15. Data-validation techniques check for syntax errors. (4.8)
16. When responses are alphabetic, it is not necessary to check for both uppercase and lowercase responses because the computer stores both responses identically. (4.9)
17. Sequence checking a file using a validation program sometimes results in an error message being shown for a record that is "in sequence" rather than the one that is "out of sequence." (4.9)
18. In a validation program, after finding the first error for a particular record, you do not need to continue checking the record for other errors that might exist. (4.10)
19. On-line systems operate by collecting groups of data that are processed at a specific time such as daily, weekly, or monthly. (4.10)
20. A user-friendly program is one that is easy to use. (4.13)
21. Interactive programs always contain menus. (4.13)
22. When using a form, the user can choose the order in which to enter data. (4.14)
23. Visual languages provide more flexibility for data input than nonvisual languages. (4.14)

Multiple Choice

Select the best answer.

1. Which of the following types of lines is used to provide user instructions explaining how to continue displaying a report on the next screen? (4.2)
   - **a.** Summary lines.
   - **b.** Control footing lines.
   - **c.** Page (screen) footing lines.
   - **d.** Report footing lines.
2. Scrolling occurs ________. (4.3)
   - **a.** On a printed page when the computer runs out of room on the page and prints over the perforation between pages.
   - **b.** On a display screen when the information on the top line of the screen is erased to make room for a new line on the bottom.
   - **c.** On a display screen when the information on the bottom line of the screen is erased to make room for a new line.
   - **d.** When footings are displayed.

Questions 3 to 6 refer to the following spacing chart.

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789	6 0123456789
1			MONTHLY INVENTORY REPORT				
2							
3	ITEM	AMOUNT--START	AMOUNT	AMOUNT--END		REORDER	REORDER
4	NUMBER	OF MONTH	USED	OF MONTH		QUANTITY	
5							
6	XXXX	ZZZ,ZZ9	ZZZ,ZZ9	ZZZ,ZZ9		ZZZ,ZZ9	XXX
7							
8	XXXX	ZZZ,ZZ9	ZZZ,ZZ9	ZZZ,ZZ9		ZZZ,ZZ9	XXX

3. The detail lines are ________. (4.3)

a. Single-spaced.
b. Double-spaced.
c. Triple-spaced.
d. None of the above.

4. The Xs in the column under ITEM NUMBER indicate that ________. (4.3)

a. Numeric data are to be printed in that column.
b. Nonnumeric data are to be printed in that column.
c. Xs are to be printed in that column.
d. None of the above.

5. The report has _____ column heading lines. (4.3)

a. 1
b. 2
c. 3
d. 4

6. The number 124 would be printed in the AMOUNT USED column as ________. (4.3)

a. 000,124
b. ,124
c. 124
d. None of the above.

7. The following pseudocode is for a program that has an error. The error causes the program to print headings prior to every detail line.

```
MAINLINE
START
DO INITIALIZE routine
READ a record (SALESPERSON, SALES)
IF EOF THEN
  DO ERROR routine
ELSE LOOP
     DO PROCESS routine UNTIL EOF
     ENDLOOP
ENDIF
CLOSE files
STOP

INITIALIZE
ASSIGN names and initial values in work area
  HEADING1
  "PAGE"
  PAGE-NUMBER = 0
  HEADING2
  "SALESPERSON"
  "AMOUNT OF SALE"
  OTHER
  LINES-COUNTER = 0
  MAX-LINES = 54
```

```
OPEN files
LINES-COUNTER = MAX-LINES
ENDINITIALIZE

ERROR
WRITE "NO INPUT RECORDS"
ENDERROR

PROCESS
DO WRITE-DETAIL routine
READ a record (SALESPERSON, SALES)
ENDPROCESS

WRITE-DETAIL
IF LINES-COUNTER >= MAX-LINES THEN
  DO HEADINGS routine
ENDIF
WRITE a line (SALESPERSON, SALES)
ADD 1 to LINES-COUNTER
ENDWRITE-DETAIL

HEADINGS
ADD 1 TO PAGE-NUMBER
WRITE HEADING1 at top of page
WRITE HEADING2
WRITE a blank line
ENDHEADINGS
```

To fix the problem, the program must _______. (4.4)

a. Move 3 to LINES-COUNTER in HEADINGS.
b. Move 0 to LINES-COUNTER in HEADINGS.
c. Move 0 to MAX-LINES in INITIALIZE.
d. Move 3 to MAX-LINES in HEADINGS.

8. Refreshing the screen _______ (4.5)

a. Is the same as clearing the screen.
b. Is done when detail lines are double-spaced on the screen.
c. Is done when detail lines are single-spaced on the screen.
d. Involves erasing only that part of the screen that is to be changed.

Questions 9 to 14 refer to Figure 4–K.

9. What is the maximum number of lines that this program will display on a screen (include blank lines, heading lines, detail lines, and the like)? Be careful! (4.5)

a. 16
b. 19
c. 20
d. 17
e. 18
f. 21

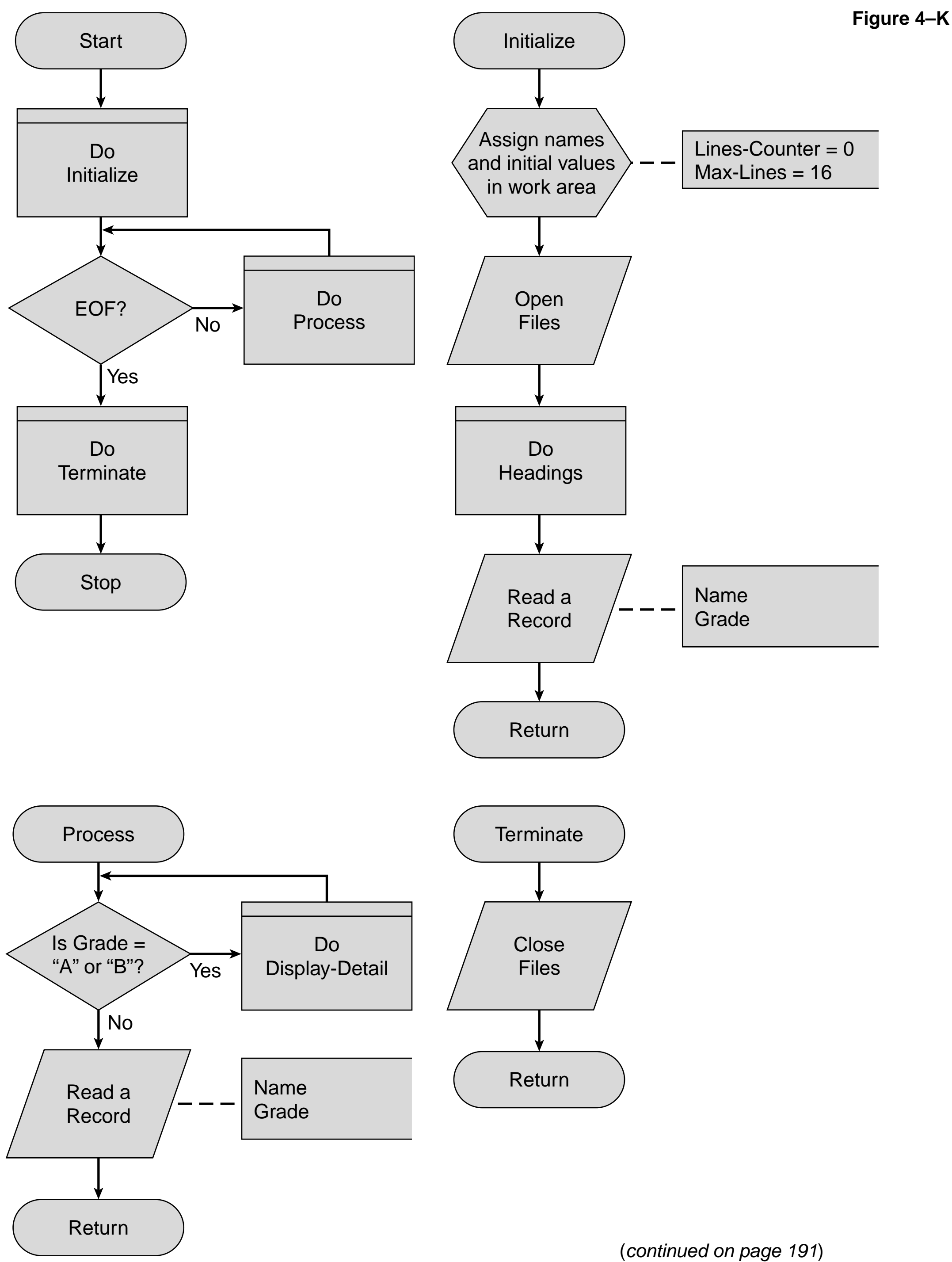
10. What routine should be modified to display the line REPORT COMPLETE? (4.5)

a. Initialize.
b. Process.
c. Terminate.
d. Display-Detail.
e. None of the above.

11. The detail lines are _______. (4.5)

a. Single-spaced.
b. Double-spaced.
c. Shown only at the bottom of the screen.
d. Shown only at the top of the screen.

Figure 4–K



(*continued on page 191*)

12. If the input file contains 75 records, how many "screens" (pages) of data will be displayed? (4.5)

a. 5
b. 4
c. 75
d. Unable to determine the number of screens from the information given.

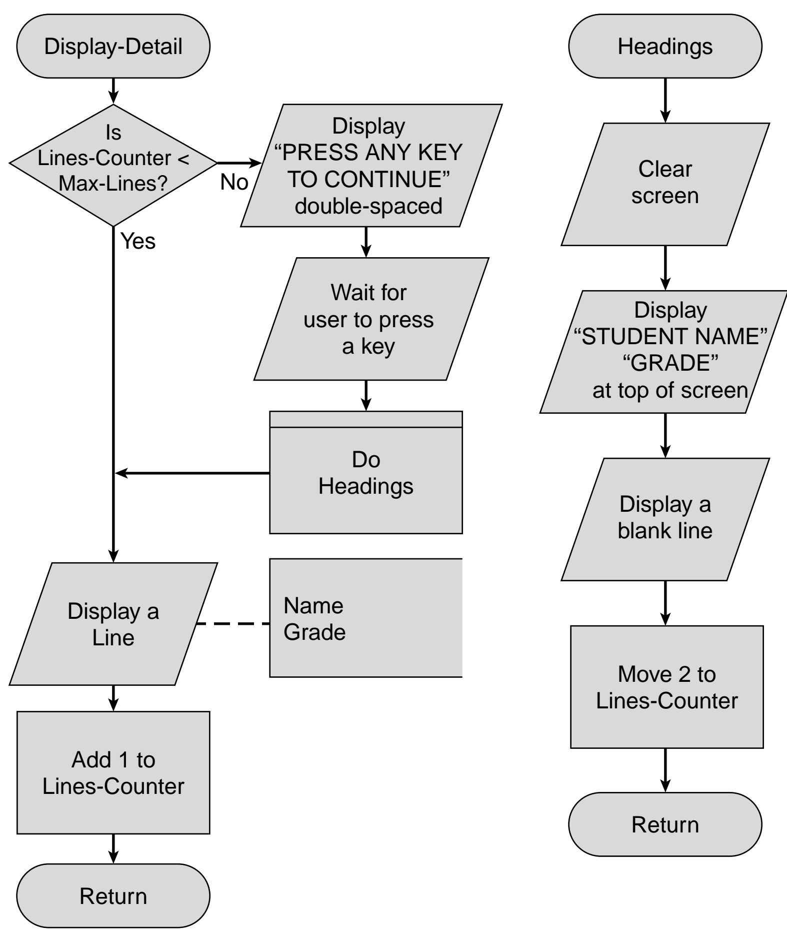


Figure 4–K (*continued*)

13. The report generated is a(n) ________ report. (4.5)

a. Detail.
b. Exception.
c. Summary.
d. Query.

14. Suppose you are asked to modify the flowchart shown in Figure 4–K to obtain the current date in the most efficient manner. What routine should be modified to obtain the date? (4.7)

a. Initialize.
b. Process.
c. Display-Detail.
d. Headings.

15. In your flowchart or pseudocode, obtain the current date stored in the computer system using _______. (4.7)

a. An external subroutine.
b. An internal subroutine.
c. A separate file.
d. None of the above.

16. Suppose a program displays headings prior to every detail line. The maximum number of lines per page is 50. There are three heading lines on the report. The logic error most likely is that _______. (4.7)

a. The program moves 0 to LINES-COUNTER in HEADINGS.
b. The program moves 3 to LINES-COUNTER in HEADINGS.
c. The program moves 50 to LINES-COUNTER in HEADINGS.
d. The program adds 1 to LINES-COUNTER in WRITE-DETAIL.
e. The program adds 1 to LINES-COUNTER in PROCESS.

17. Which of the following will result in a data-exception error? (4.8)

a. Incorrect spelling of a variable or field name.
b. Using a subtract instruction instead of an add instruction.
c. Adding AMOUNT to TOTAL when AMOUNT is not numeric.
d. More than one of the above.

18. Which of the following errors *cannot* be detected using a validation program? (4.9)

a. Using the instruction COMMISSION = 0.10 * SALES-AMOUNT instead of COMMISSION = 0.20 * SALES-AMOUNT when writing a program.
b. Entering a month of 13.
c. Entering ABC for a field that should be numeric.
d. Entering – 5 for a field that should be a number greater than 0.
e. Entering y instead of Y for a yes/no response.

19. Attempting to do arithmetic on a field that is nonnumeric results in a(n) ________. (4.9)

a. Out-of-sequence error.
b. Data-exception error.
c. Visual-verification error.
d. Check-digit error.
e. None of the above.

20. Which indicator tracks whether an input file contains any errors? (4.11)

a. RECORD-ERROR.
b. FILE-ERROR.

Questions 21 and 22 refer to the pseudocode shown below.

```
MAINLINE
START
DO INITIALIZE routine
IF EOF THEN
   WRITE "NO INPUT RECORDS" at top of page
ELSE LOOP
         DO PROCESS routine UNTIL EOF
     ENDLOOP
ENDIF
DO TERMINATE routine
STOP

INITIALIZE
ASSIGN names and initial values in work area
  HEADING1
  "INVALID RECORDS"
  HEADING2
  "RECORD"
  "ERROR"
  ERROR-LINE
  BAD-RECORD
  ERROR-MESSAGE
  OTHER
  LINES-COUNTER = 0
  MAX-LINES = 50
  RECORD-ERROR
  FILE-ERROR = "NO"
OPEN files
READ a record (STU-NAME, NUM-OF-COURSES, GPA)
LINES-COUNTER = MAX-LINES
ENDINITIALIZE
```

```
PROCESS
IF STU-NAME is blank THEN
   MOVE "STUDENT NAME BLANK" to ERROR-MESSAGE
   DO WRITE-ERROR routine
ENDIF
IF NUM-OF-COURSES is not numeric THEN
   MOVE "NUMBER OF COURSES NOT NUMERIC" to ERROR-MESSAGE
   DO WRITE-ERROR routine
ELSE IF NUM-OF-COURSES is less than 0 THEN
        MOVE "NUMBER OF COURSES < 0" to ERROR-MESSAGE
        DO WRITE-ERROR routine
      ENDIF
ENDIF
IF RECORD-ERROR = "NO" THEN
   WRITE a record (STU-NAME, NUM-OF-COURSES, GPA) on
      STUDENT-OUT file
ENDIF
READ a record (STU-NAME, NUM-OF-COURSES, GPA)
ENDPROCESS

TERMINATE
IF FILE-ERROR = "NO" THEN
  DO HEADINGS routine
  WRITE "NO ERRORS FOUND—ALL RECORDS CORRECT"
ENDIF
CLOSE files
ENDTERMINATE

WRITE-ERROR
IF LINES-COUNTER is greater than or equal to MAX-LINES THEN
  DO HEADINGS routine
ENDIF
IF RECORD-ERROR equals "YES" THEN
  MOVE spaces to BAD-RECORD
ELSE MOVE STU-NAME, NUM-OF-COURSES, GPA to BAD-RECORD
     MOVE "YES" to RECORD-ERROR
ENDIF
WRITE BAD-RECORD, ERROR-MESSAGE
ADD 1 to LINES-COUNTER
ENDWRITE-ERROR

HEADINGS
WRITE HEADING1 at top of page
WRITE HEADING2 double spaced
WRITE a blank line
MOVE 4 to LINES-COUNTER
ENDHEADINGS
```

21. A programmer's goal was to write a message indicating when no errors resulted from processing the entire file. The logic doesn't work. Why? (4.11)

a. FILE-ERROR must be changed to "YES" in HEADINGS.
b. FILE-ERROR must be initialized to "YES" in INITIALIZE.
c. FILE-ERROR must be changed to "YES" whenever a record is read.
d. The message indicating that no errors occurred should be written in PROCESS rather than in TERMINATE.

22. A programmer's goal was to write all records without errors to a disk file. Why doesn't the program work? (4.11)
    a. RECORD-ERROR must be set to "NO" whenever a record is read.
    b. RECORD-ERROR must be changed to "YES" in HEADINGS.
    c. The record should be written to the disk file at the beginning, instead of at the end of PROCESS.
    d. None of the above.

23. VALID-RESPONSE is given a value of _________ if the input is not valid. If the input is valid, the value in VALID-RESPONSE is _________. (4.12)
    a. YES, NO
    b. NO, YES

Short Answer

1. Indicate whether each of the following describes a detail report (D), summary report (S), exception report (E), or query report (Q). (4.1)
    a. A list of names for all registered voters in a particular precinct. Input consists of one record for each registered voter.
    b. A list showing names of those voters who did not vote in the previous election. Input consists of one record for each registered voter. Note: In order to vote, a person must be registered. Some registered voters do not vote.
    c. A list of the number of registered voters for each precinct in the state. Input consists of one record for each registered voter.
    d. A report showing the names of students who are Computer Information Systems majors and have a GPA of 3.5 or above. Input consists of a student database.
    e. A class list showing each student's name and the number of times he or she was absent. Input consists of one record for each student in the class.
    f. A report showing names of students who have been absent more than ten times. Input consists of one record for each student in the class.
    g. A report showing whether a driver pulled over for a traffic violation has any outstanding warrants for arrest. Input consists of a state database showing outstanding warrants by driver license number.

2. Using a date other than the current date on a report is sometimes desirable. Give two examples (not the one in the book) of such cases. (4.6)

3. Give an example (other than one in the book) of each type of error described below. (4.8)
    a. A logic error that causes incorrect output even though the program goes to normal completion.
    b. A translation (syntax) error.
    c. A data-exception error.
    d. An input error that cannot be detected using a validation program.

4. Give an example that illustrates why records listed as sequence errors in an input file might not actually be the ones that are out of sequence. (4.9)

Logic Exercises

1. Construct a data dictionary and draw a hierarchy chart and flowchart or pseudocode for a program to solve the following problem. Suppose a business borrows $10,000 at an interest rate of 1.5 percent per month on

the unpaid balance. If the business pays $1,000 at the end of each month, what is the remaining balance at the end of each month?

Output. Output is a screen report showing the remaining balance each month.

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789	6 0123456789
1	MM/DD/YY						
2							
3	MONTH		REMAINING				
4			BALANCE				
5							
6	Z9		$ZZZ9.99				
7	Z9		$ZZZ9.99				

Input. There is no input for this program.

Processing. The mathematics involved is as follows. The interest for the first month is the balance $10,000 * .015 = $150. Since $1,000 is paid each month, the remaining balance after the first payment is $10,000 + $150 – $1,000 = $9,150. The interest for the second month is based on the remaining balance from the first month. You will need to do this until the remaining balance is less than $1,000. *Note:* You will not write each calculation. The calculation occurs in a loop. Use work areas to hold the interest amount for the month and the remaining balance. Manually check your results to ensure your loop is correct.

2. Construct a data dictionary and draw a hierarchy chart and flowchart or pseudocode for a program to produce an inventory report.

   Output. Output is the report. Each line of the report contains the item number, item name, quantity at start of inventory period, and quantity at end of inventory period. Each page of the report contains at most 40 detail lines. Include headings with page numbers.

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789	6 0123456789
1							PAGE ZZ9
2							
3	ITEM N	UMBER I	TEM NAME	QUANTI	TY AT START	QUANTI	TY AT END
4				OF	PERIOD	OF	PERIOD
5							
6	XX	X XXXX	XXXXXXXXXXX	X	ZZ9	Z	Z9
7	XX	X XXXX	XXXXXXXXXXX	X	ZZ9	Z	Z9

   Input. Input consists of records containing the item number, item name, quantity at start of period, and quantity used.

   Processing. Calculate the quantity at the end of the period by subtracting the quantity used from the quantity at the start of the period.

3. Construct a data dictionary and draw a hierarchy chart and flowchart or pseudocode to print a list of patients in a hospital.

Output. Output is the report. Use the current date in the heading line. Include the page number in a footer line (line 51 on the page). Include at most 45 detail lines on each page.

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789	6 0123456789
1		LIST OF PA	TIENTS AS O	F MM/DD/YY			
2							
3	PATIEN	T NAME	RO	OM NUMBER	DATE ADMIT	TED	
4							
5	XXXXXXXXX	XXXXXXXXXX	XXXXX	XXX	XX/XX/XX		
6	XXXXXXXXX	XXXXXXXXXX	XXXXX	XXX	XX/XX/XX		
.							
.							
51	PAGE ZZ9						

Input. Input consists of records, each of which contains a patient name, room number, and date admitted.

4. Construct a data dictionary and draw a hierarchy chart and flowchart or pseudocode to generate an exception report showing products with a percent increase in price of at least 10 percent.

   Output. Output is a screen display report. Use the current date in the heading line. Include at most eight double-spaced detail lines per screen. After displaying an entire screen, hold the screen to enable the user to read it. Request the user to enter any key to go to a new screen to display additional lines. At the end of the report, display the line REPORT COMPLETE double-spaced.

	0 123456789	1 0123456789	2 0123456789	3 0123456789	4 0123456789	5 0123456789	6 0123456789
1		PRODUCTS W	ITH PRICE	INCREASES O	F AT LEAST	10%	MM/DD/YY
2							
3		PRODUCT	PREVIOUS	PRICE	CURRENT PR	ICE	% INCREASE
4							
5		XXXXXXXXX	$ZZ9.	99	$ZZ9.99		Z,ZZ9.99
6							
7		XXXXXXXXX	$ZZ9.	99	$ZZ9.99		Z,ZZ9.99

Input. Input consists of records containing the product number, previous price, and current price.

Processing. Use the following formula to calculate the percent increase:

$$\text{PERCENT-INCREASE} = \frac{\text{CURENT-PRICE} - \text{PREVIOUS-PRICE}}{\text{CURRENT-PRICE}} * 100$$

On the report, include only records having a calculated increase of at least 10 percent.

5. Construct a data dictionary and draw a hierarchy chart and flowchart or pseudocode to print the economic order quantity (EOQ) for products stocked by a mail-order company. The EOQ is the quantity at which products should be reordered by the company in order to maximize profits.

Output. Output is the report. At most, 50 detail lines are on each page. Show the current date and page numbers on each page of the report.

	0	1	2	3	4	5	6
	123456789	0123456789	0123456789	0123456789	0123456789	0123456789	0123456789
1	MM/DD/YY			PAGE ZZ9			
2							
3	ITEM	NUMBER	REORDER	QUANTITY			
4							
5	X	XXX	ZZZ	Z9			
6	X	XXX	ZZZ	Z9			

Input. Input consists of records, each of which contains an item number, purchase cost, storage cost, and total demand for the item during the period.

Processing. Use the following formula to calculate the reorder quantity (EOQ):

$$EOQ = \sqrt{\frac{2 * \text{PURCHASE-COST} * \text{TOTAL-DEMAND}}{\text{STORAGE-COST}}}$$

Note: Stock-out costs are not included in this calculation. Determine the square root (√) using an external subroutine.

6. Construct a data dictionary and draw a hierarchy chart and flowchart or pseudocode to validate a file.

Output. Output consists of a report showing the invalid records and a file containing the valid records. The valid record file is to use the same format as the input file. If no errors occur during processing, the message NO ERRORS is to be written on the report. Print a maximum of 25 detail lines on each page of the report. If multiple errors occur for a particular record, list the record only once and show each of the error messages. Include page numbers.

	0	1	2	3	4	5	6	7
	123456789	0123456789	0123456789	0123456789	0123456789	0123456789	0123456789	0123456789
1	PAGE ZZ9							
2				VALIDATION	REPORT			
3								
4		REC	ORD			ERR	OR	
5	XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	X	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX
6	XXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	X	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX	XXXXXXXXXX

Input. Input consists of records, each of which contains a salesperson number (must be numeric), salesperson name (must be present), units sold this month (must be numeric and greater than or equal to 0), month (must be numeric and in the range 1 to 12).

Processing.

- All fields must be checked according to the validation information in the input description.
- In addition, check the salesperson number to ensure that it is increasing sequence (that is, the number on the second record is greater than the first, and so on).

- Use the following error messages:
  1. SALESPERSON NUMBER NOT NUMERIC
  2. SALESPERSON NUMBER OUT OF SEQUENCE
  3. SALESPERSON NAME MISSING
  4. UNITS SOLD NOT NUMERIC
  5. UNITS SOLD NOT GREATER THAN OR EQUAL TO 0
  6. MONTH NOT NUMERIC
  7. MONTH NOT IN THE RANGE 1 TO 12

  Note: Check for the errors in the above order. Check for all errors with the following exceptions: If error 1 occurs, do not check for error 2; if error 4 occurs, do not check for error 5; if error 6 occurs, do not check for error 7.

7. Construct a data dictionary and draw a hierarchy chart and flowchart or pseudocode to interactively enter student test scores, calculate the test average and course grade, and write records to an output file.

   Output. Output consists of lines displayed on a screen and a grade file. Each record on the grade file should contain the student number, three test scores, the test average, and the course grade. The interactive procedure showing the lines to be displayed, the possible responses, and programming paths to be followed based on the responses is shown below. *Note:* For ease of reading, all displayed lines are underlined in the figure. The underlines are not displayed on the screen.

GRADE FILE CREATE

Clear the screen prior to displaying this line.

ENTER STUDENT NUMBER OR ZERO (0) TO TERMINATE THE PROGRAM

Display a blank line prior to displaying this line.

Make sure that the response is numeric and is in the range 1 to 25.

If the response is invalid, display an appropriate message and repeat the request for student number.

If the response is 0, display a blank line followed by PROGRAM TERMINATED and terminate processing.

If the response is in the range of 1 to 25, continue with the next line to be displayed.

ENTER TEST SCORE

This prompt will be given three times so that all three scores can be entered.

Validate each score to confirm that it is a number in the range of 0 to 100.

If the score is invalid, display an appropriate error message and request the score again.

After receiving three valid scores, display a line asking the user if the information is correct and should be written to the grade file. If the user responds affirmatively, calculate the average, determine the grade, and write a record to the output file. Then, clear the screen and continue with the next student. If the user responds negatively, clear the screen and continue with the next student without writing the record to the output file.

Input and Processing. Input consists of the values described in the interactive procedure above. Calculate the test average by adding the three test scores and dividing by 3. Determine the letter grade as follows:

A 92 < = Average < = 100

B 84 < = Average < 92

C 75 < = Average < 84

D 65 < = Average < 75

F 0 < = Average <65

8. Construct a data dictionary and draw a hierarchy chart and flowchart or pseudocode for a menu-driven program that computes interest using the type of interest and terms selected by the user. Design the screens to give the user choices to compute simple interest, compound interest, continuously compounded interest or to terminate the program.

Use the following formulas in your program:

Simple Interest: AMOUNT = PRINCIPAL * (1 + YEARS * RATE)
Compound Interest: AMOUNT = PRINCIPAL * (1 + RATE / NCMPD) ^ (NCMPD * YEARS)
Continuously Compounded Interest: AMOUNT = PRINCIPAL * E ^ (RATE * YEARS)

In the above formulas:

AMOUNT is the total amount accumulated.
PRINCIPAL is the principal invested.
YEARS is the number of years.
RATE is the interest rate expressed as a decimal.
NCMPD is the number of times compounding occurs each year.
E is the base of the natural log (E = 2.71828).

Notes on input of values:

- The above calculations require many of the same variables. Use a single subroutine to request the common variables (principal, interest rate, number of years).

- Validate input as follows:

  1. PRINCIPAL and YEARS > 0
  2. 0 < RATE < 1
  3. 0 < NCMPD < = 366
  4. Validate all other input as appropriate.