Systems Management

An Introduction to Structured Programming in PL/I

IBM

GC20-1777-1

An Introduction to Structured Programming in PL/I

This text, intended for programmers, describes and illustrates the use of structured programming. The technique and its supporting practices are generally described in one chapter. A reference chapter illustrates the implementation of the technique in PL/I and is followed by a chapter presenting three sample programs. A knowledge of PL/I is assumed.

Preface

This text describes and illustrates the use of structured programming, a recently formalized programming style in which the structure of a program is made as clear as possible.

Intended for programmers, the publication consists of three chapters:

1. An expository chapter describing the technique, its supporting practices, and its use. General suggestions on getting started are also included.

2. A reference chapter illustrating the implementation of the technique in PL/I. This chapter may be used as a starting point for establishing your own structured programming guidelines.

3. A chapter containing three sample programs written according to the techniques presented earlier.

Familiarity with programming concepts is necessary for the expository chapter, and knowledge of PL/I is needed for the reference and sample program chapters.

Second Edition (June 1977)

This edition is a major revision and obsoletes the previous edition. Changes have been made throughout including those to reflect the availability of Release 3 of the OS/PL/I Optimizing Compiler (5734-PL1), Optimizing Compiler and Libraries (5734-PL3), and Checkout Compiler (5734-PL2), and Release 5 of the DOS PL/I Optimizing Compiler (5736-PL1) and Optimizing Compiler and Libraries (5736-PL3). All of these compilers offer additional support for structured programming techniques.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address comments concerning the contents of this publication to IBM Corporation, Technical Publications/Systems, Dept. 824, 1133 Westchester Avenue, White Plains, New York 10604.

Contents

List of Illustrations

Chapter 1: An Overview of Structured Programming

Introduction

This chapter contains various items of background information about structured programming that should be useful. The topics include:

Definitions
A summary of the potential advantages of structured programming
The relationship of structured programming to other improved programming technologies
A sketch of the theoretical foundation of structured programming
The basic control logic structures
Additional control logic structures
The GO TO question
Segmentation
Indentation
Documentation considerations
Efficiency considerations
Getting started in structured programming

After you have read the material in this overview, you will be ready to study the reference material in Chapter 2 to see how structured programming can be done in PL/I, and to see some of the ideas illustrated in three sample programs in Chapter 3.

Definitions

Structured programming is a style of programming in which the structure of a program (that is, the interrelationship of its parts) is made as clear as possible by using just three control logic structures:

1. Simple *sequence* of functions
2. *Selection* of functions (IFTHENELSE)
3. Loop control, or *iteration*

These three types of control logic structures may be combined to produce programs to handle any information processing task. Statements controlled by the selection and loop structures are indented to make obvious the scope of influence of the structure.

A structured program is composed of *segments*, which may range from a few statements up to about a page of code. Each segment has just one entry and one exit. Such a segment, assuming it has no infinite loops and no unreachable code, is called a *proper program.* When proper programs are combined using the three basic control logic structures (sequence, selection, and iteration), the result is also a proper program.

An important characteristic of a structured program is that it can be read in sequence, from top to bottom, without a great deal of the "skipping around" through the program that is typical of other programming styles. This is important because it is much easier to comprehend fully what a function does if all the statements that influence its action are physically close by. Top-down readability is one consequence of using only three control logic structures, and of avoiding the GO TO statement except in very special circumstances, such as the simulation of a control logic structure in a programming language that lacks it.

A program written according to these principles not only ***has*** a structure, it clearly ***exhibits*** it.

Potential Advantages

A program written in this style tends to be much easier to understand than programs written in other styles. Easier understandability can facilitate code checking and thus may reduce the program testing and debugging time. This is true partly because structured programming concentrates on one of the most error-prone factors in programming, the logic.

A program that is easy to read and which is composed of well-defined segments tends to be simpler, faster, and less expensive to maintain. These benefits can derive in part from the fact that since the program is to a significant extent its own documentation, the documentation tends to always be up to date; this may not be true with conventional methods.

Structured programming offers these benefits, but it should not be thought of as a panacea. Program development is still a demanding task requiring skill, effort, and creativity.

Relationship of Structured Programming to Other Improved Programming Technologies

Structured programming is compatible with, and supportive of, other improved programming technologies, although distinct from them. Other technologies and the relationship of structured programming to them may be sketched briefly.

Top-down program development involves writing and testing the highest-level segments of a program first, in contrast to the more common method in the past, bottom-up development. This approach has the potential benefits of giving the critical top segments the most testing, of giving earlier warning of problems with the interfaces between segments, and of spreading the debugging and testing over a greater part of the development cycle.

Structured programming and top-down program development both emphasize the importance of segments that interact in precisely understood ways. Both involve looking at a program as a hierarchy of segments that are related to each other in a tree-like fashion.

Hierarchy plus Input-Process-Output (HIPO) is an approach to functional specification and documentation of programs. Each function is designed using a HIPO diagram, in which inputs and outputs are listed and the processing that is to be carried out is specified. A visual table of contents diagram points to the HIPO diagrams in the package and therefore shows the functions and subfunctions to be carried out by the various parts of a program, and the relationship between them. At the detailed design level, it also shows the hierarchy of segments.

Structured programming, as the term is used in this publication, refers primarily to the coding phase rather than the design phase of the program development cycle. HIPO is one good way to approach the design task, and one that is complementary to structured programming.

A *structured walkthrough* is a review session in which the originator of program design material or code explains it to colleagues. The intent is to detect errors (which are corrected after the walkthrough) as early in the process as possible, when they should be least expensive to correct.

Structured programming, with its emphasis on easy readability of programs, increases the effectiveness of structured walkthroughs.

A *development support library* consists of a machine-readable library which contains the current versions of all project programming data. It also consists of external library binders which contain current listings of all library members and archives consisting of recently superseded listings. Besides providing easy accessibility of materials, this helps assure that the latest versions of programs are always used.

Structured programming, with its insistence on segmentation of programs, fits in well with development support libraries, although such libraries are useful with any style of programming.

The *chief programmer team concept* involves programming with teams of at least three members: chief programmer, backup programmer, and program librarian. The team may also include other programmers, nonprogramming analysts, and end users. The chief programmer is responsible for the design and coding of all programs produced by the team, either writing or personally checking every piece of code. The program librarian maintains the development support library.

Structured programming is well-suited to chief programmer team methods, since it facilitates one key element, that of code review by the chief programmer.

Structured Programming Theory

The Structure Theorem

The structure theorem states that any *proper program* can be written using only the control logic structures of *sequence, selection* (IFTHENELSE), and *iteration.*

A *proper program* is defined as one that meets the following two requirements:

1. It has exactly one entry point and exactly one exit point for program control.

2. There are paths from the entry to the exit that lead through every part of the program; this means that there are no infinite loops and no unreachable code. This requirement is, of course, no restriction, but simply a statement that the structure theorem applies only to meaningful programs.

The three basic control logic structures are defined as follows:

Sequence is simply a formalization of the idea that unless otherwise stated, program statements are executed in the order in which they appear in the program. This is true of all commonly used programming languages; it is not always realized that sequence is in fact a control logic structure. In flowchart terms, sequence is represented by one function after the other, as shown in Figure 1.

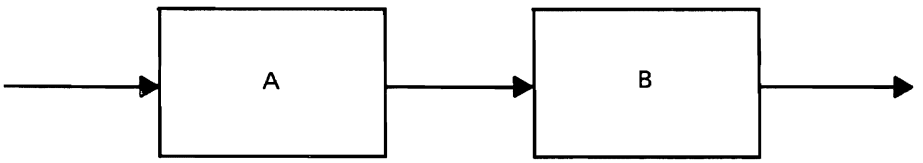


Figure 1. Flowchart for the control logic structure *sequence*

A and *B* are anything from single statements up to complete modules; the concern is only with the abstract idea of a proper program, regardless of its size and internal complexity. *A* and *B* must both be proper programs in the sense just defined (one entry and one exit). The combination of *A* followed by *B* is also a proper program, since it too has one entry and one exit. This can be shown pictorially, as in Figure 2, where the outer box is meant to suggest that the combination of *A* followed by *B* can be treated as a single unit for control purposes.

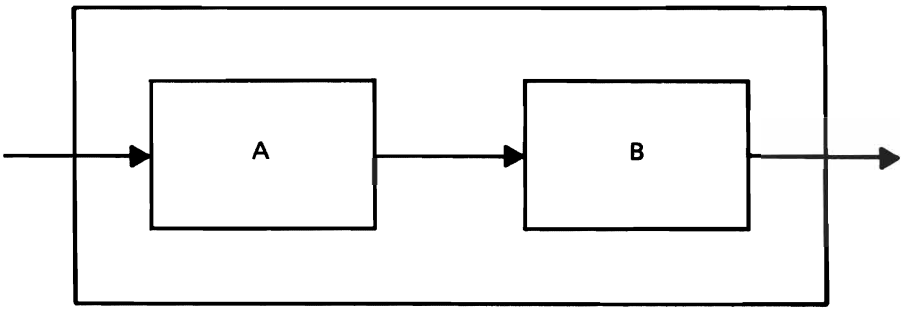


Figure 2. Two proper programs in sequence

Selection is the choice between two actions based on a *predicate;* this is called the IFTHENELSE structure. In PL/I it is implemented with the IF statement, and the predicate is called the *element expression.* The usual flowchart notation for selection is shown in Figure 3, where p is the predicate and A and B are the two functions.

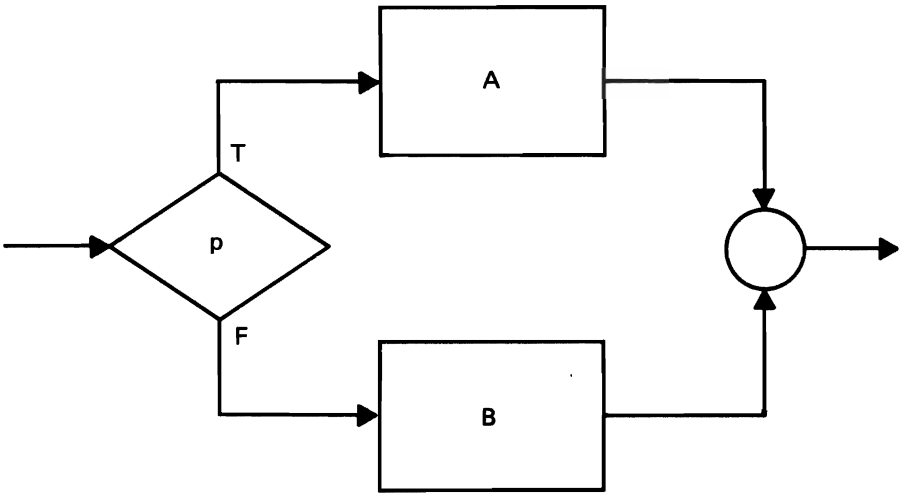


Figure 3. Flowchart for the control logic structure *selection*

The *iteration* structure, used for repeated execution of code while a condition is true (also called loop control), is the DOWHILE. In the flowchart in Figure 4, p is the predicate and A is the controlled code. In PL/I, the DOWHILE is implemented with the DO statement with the WHILE option, as discussed in Chapter 2.

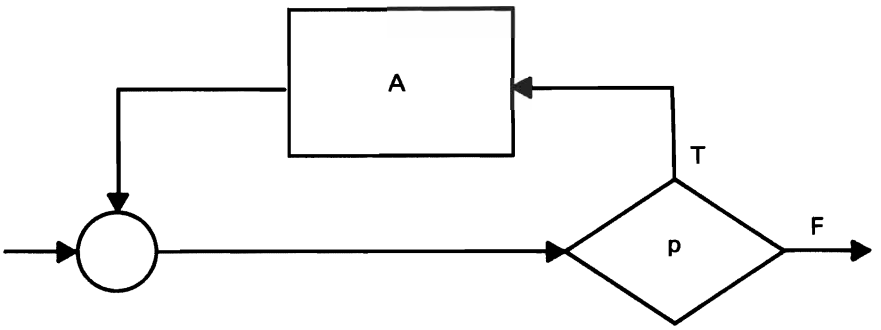


Figure 4. Flowchart for the control logic structure *iteration,* the DOWHILE

A fundamental idea is that anywhere a function box appears, any of the three basic structures may be substituted and still have a proper program. For example, the function box in Figure 4 could be replaced with *selection*, producing the flowchart of Figure 5. The dotted lines show where another structure has been substituted for a function. Or, one function in a *selection* might be replaced with three functions in sequence, and the other replaced with an *iteration*, producing the flowchart of Figure 6. Flowcharts of arbitrary complexity can be built up in this way. Figure 7 shows a flowchart with several control logic structures, drawn this time in top-to-bottom fashion. Three other examples appear in Chapter 3.

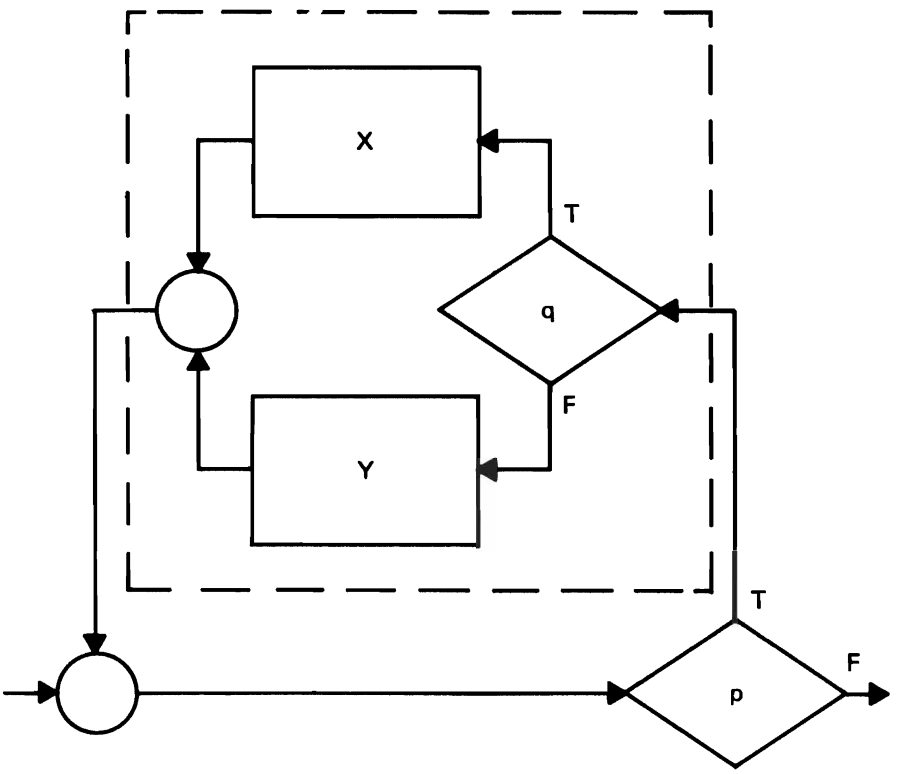


Figure 5. An example of the combination of two control logic structures, in which the function controlled by a DOWHILE is an IFTHENELSE

The ability to substitute control logic structures for functions and still have a proper program is basic to structured programming. This may also be called the *nesting* of structures.

Additional Control Logic Structures

Although all programs can be written using only the three basic structures, it is sometimes helpful to utilize a few others.

The DOUNTIL Structure

The basic iteration structure is the DOWHILE, but there is a closely related structure, DOUNTIL, that is sometimes used, depending on the procedure that is to be expressed and on availability of appropriate language features. The flowchart is shown in Figure 8.

The difference between the DOWHILE and DOUNTIL structures is that with the DOWHILE the predicate is tested *before* executing the function; if the predicate is false, the function is not executed at all. With the DOUNTIL, the predicate is tested *after* executing the function; the function will always be executed at least once, regardless of whether the predicate is true or false.

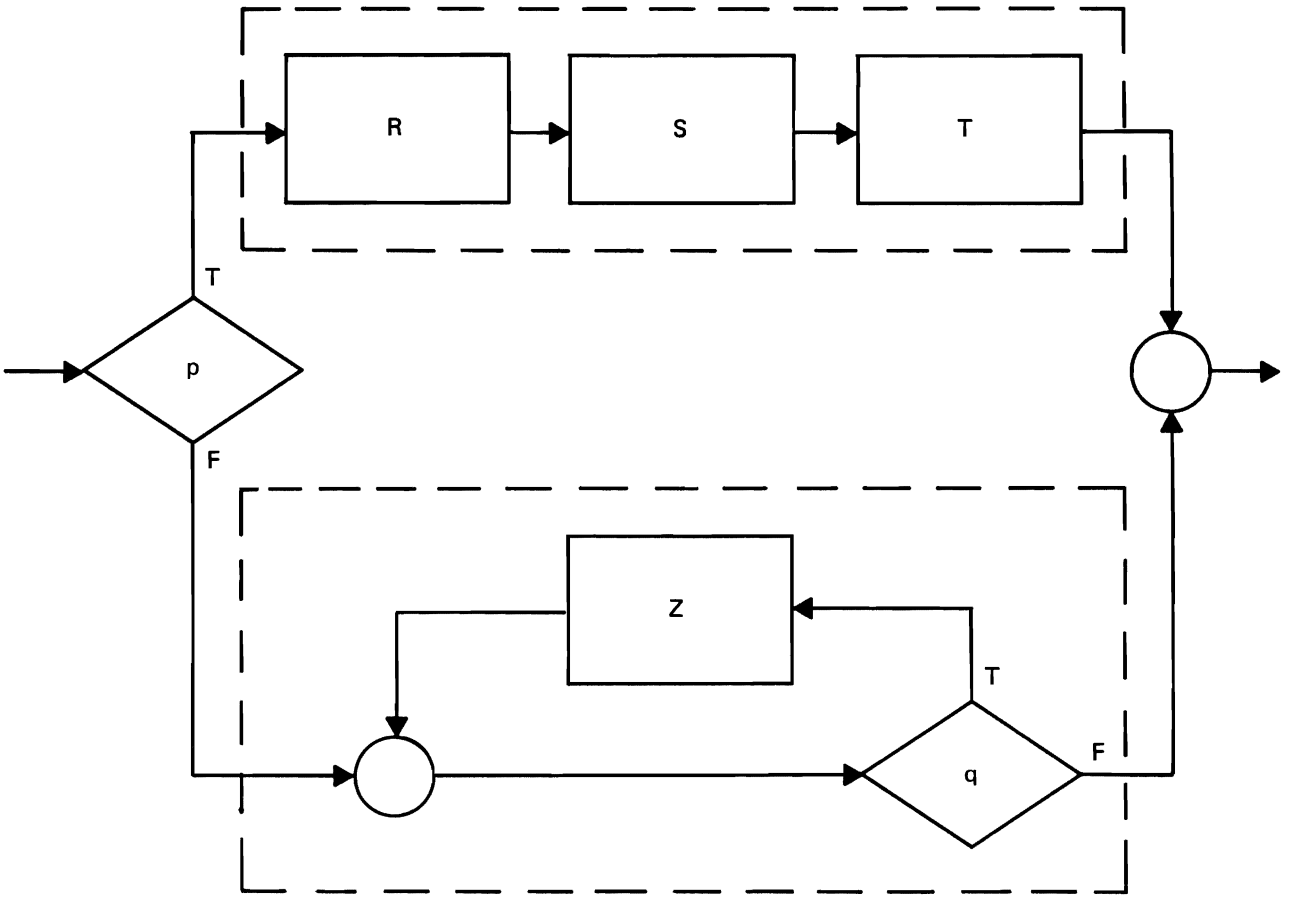


Figure 6. An example of the combination of control logic structures in which a *sequence* and an *iteration* are controlled by a *selection*

The CASE Structure

It is sometimes helpful – from both readability and efficiency standpoints – to have some way to express a multiway branch, commonly referred to as the CASE structure. For example, if it is necessary to execute appropriate routines based on a two-digit decimal code, it certainly is possible to write 100 IF statements, or a compound statement with 99 ELSE IF's, but common sense suggests that there is no reason to adhere so rigidly to the three basic structures.

The CASE structure uses the value of a variable to determine which of several routines is to be executed. The flowchart is shown in Figure 9. Observe that DOUNTIL and CASE are both proper programs.

Efficiency and convenience dictate reasonable use of language elements that may carry out logic functions in ways slightly different from those of the three basic structures. PL/I examples include the use of the DO statement with a control variable. This verb is discussed in Chapter 2 under "DOWHILE".

Labels and GO TO Statements

Structured programming has occasionally been referred to as "GO TO-less programming". Although it is true that well-structured programs have few if any GO TO statements, assuming an appropriate programming language, the absence of GO TO's can be misinterpreted. It may be well to pause for a moment to put this issue in context.

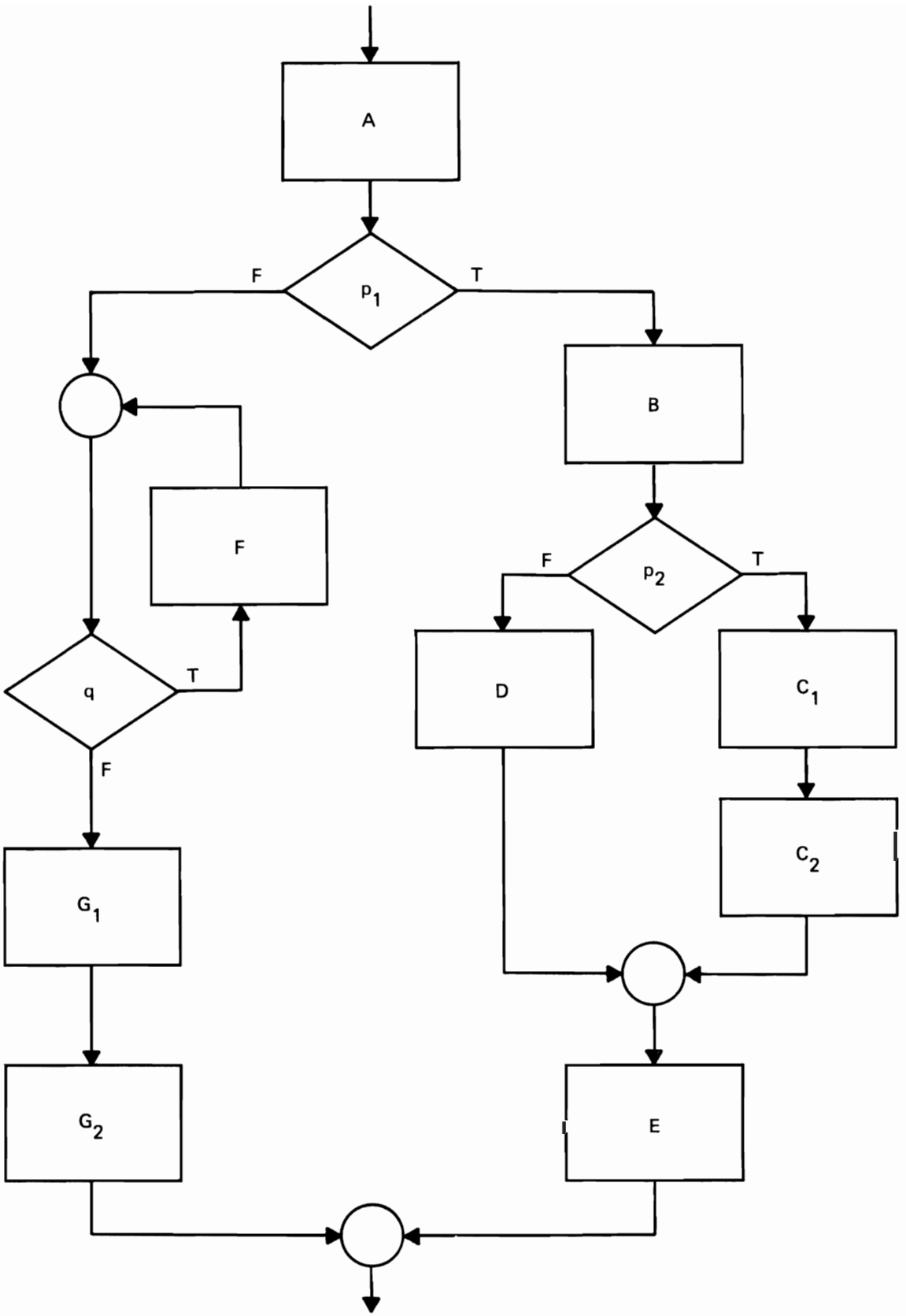


Figure 7. Another example of the combination of control logic structures

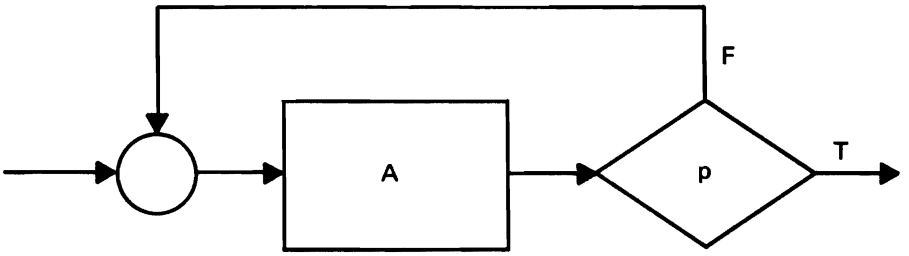


Figure 8. Flowchart for the control logic structure *iteration*, the DOUNTIL

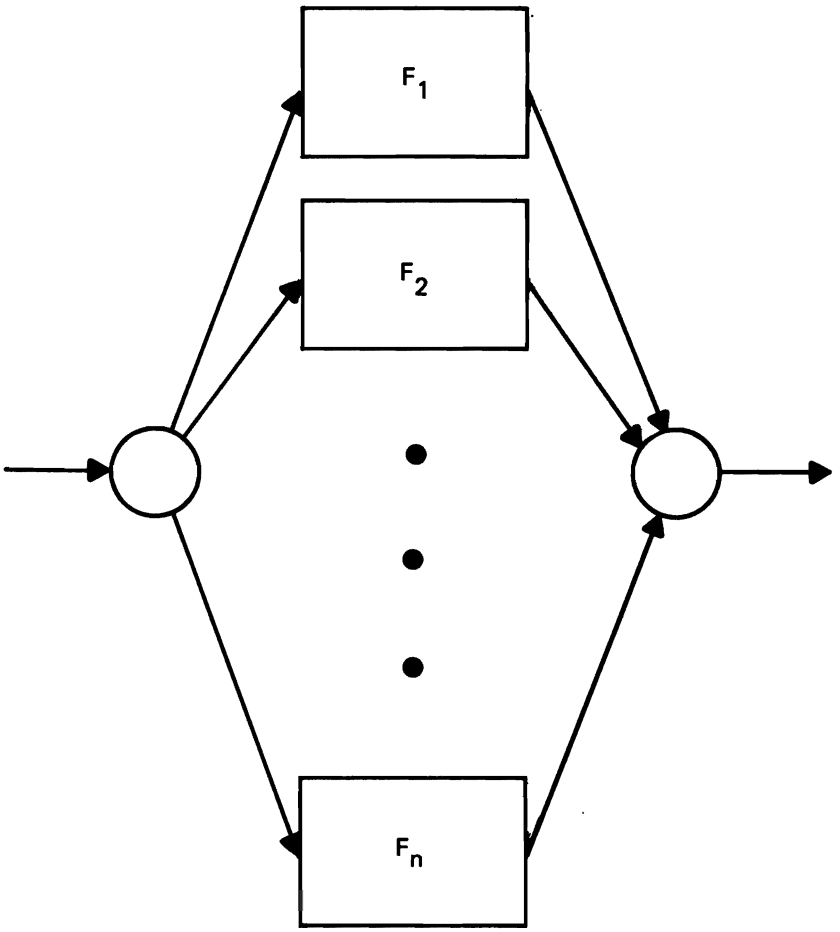


Figure 9. Flowchart for the CASE control logic structure

A well-structured program gains an important part of its easy readability from the fact that it can be read in sequence, without "skipping around" from one part of the program to another. This characteristic is a consequence of the use of only the standard control logic structures (GO TO is not a standard control logic structure). This "sequential readability", or "top-down readability", is beneficial because there is a definite limit to how much detail the human mind can encompass at once. It is far easier to grasp completely what a statement does if its function can be understood in terms of just a few other statements, all of which are physically close by. The trouble with GO TO statements is that they generally defeat this purpose; in extreme cases they can make a program essentially incomprehensible.

No special effort is required to "eliminate GO TO's", which has sometimes been misunderstood as the goal of structured programming. There are indeed good reasons for not wanting to use them, but no extra effort is required to "avoid" them: they just never occur when the standard control logic structures are used. Naturally, if the chosen programming language lacks essential control logic structures, they have to be simulated, and that involves GO TO's. But even this can be done in carefully controlled ways.

There are uncommon situations where the use of GO TO's may improve readability compared with other ways of expressing a procedure. Such examples are exceptional, however, and do not usually occur in everyday programming. The impact of deviations from installation guidelines, such as using GO TO's in other than prescribed ways, should be given careful consideration before such deviations are permitted.

Segmentation

Easy program readability requires that it not be necessary to turn a lot of pages to understand how something works. A practical rule is that a segment (previously defined as consisting of control logic structures and having only one entry and one exit) should not exceed a page of code, about 50 lines. In PL/I terms a segment can consist of one or more external and/or internal procedures, or code incorporated with a %INCLUDE. (The term segment as used here has nothing to do with the different meanings of the term in connection with the functions of operating systems or data base management systems.)

But segmentation is more than just breaking a program into page-size pieces. What characterizes good program segmentation? Three features can be identified:

1. The segmentation should reflect the division of the program into pieces that relate to each other in a hierarchy, that is, a tree structure. This organization, which may be displayed with a HIPO hierarchy chart, makes it simpler to understand how the segments relate to each other. Further, the segments at the top of the hierarchy should contain high-level control functions, whereas the segments at the bottom should contain detailed functions.

2. A well-designed segment carries out a single function or multiple functions that are closely related. This makes it easier to understand and therefore easier to assure that it does what it is meant to do. It also means that when changes have to be made, either during original programming or in maintenance, there is less chance of disturbing portions of the program that do not change.

3. A well-designed segment communicates with other segments only in carefully controlled ways. Some proponents of structured programming urge that segments always consist of procedures and that the only communication between them be through parameter lists; this reduces the chance that segments will interact in unintended and undesirable ways.

Indentation

The use of indentation is important because consistent indentation enhances readability so that the finished program exhibits in a pictorial way the relationships among statements. The basic idea is that all the statements controlled by a control logic structure should be indented by a consistent amount, to show the scope of control of the structure. In PL/I this means that the statements between the IF and the ELSE should be indented a consistent amount, and similarly for the statements between the ELSE and the next sentence. Likewise, in PL/I, the code controlled by a DO group or a BEGIN block should be indented to display the scope of control of the DO or the BEGIN.

Indentation can be a major benefit, as the skeleton programs in Figure 10 show. Both do the same processing, but the second is far easier to understand and, therefore, to verify for correctness.

```
IF P                          IF P
THEN                          THEN
B = A + B                        B = A + B
IF Q                             IF Q
THEN                             THEN
C = 12                              C = 12
ELSE                             ELSE
C = 36                              C = 36
ENDIF                            ENDIF
IF R                                IF R
THEN                                THEN
Y = X + Y                              Y = X + Y
ELSE                                ELSE
Z = X + Z                              Z = X + Z
ENDIF                               ENDIF
ELSE                          ELSE
A = A + B                        A = A + B
ENDIF                         ENDIF
```

Figure 10. Nested IF pseudocode statement, with and without indentation

Establishing Indentation Guidelines

Guidelines for indentation in PL/I programs are suggested in Chapter 2. It should be understood, however, that these are *only* guidelines. Each installation will need to establish local conventions; variation from the suggested guidelines is not important, so long as the installation conventions are followed consistently. For example, it is not of fundamental importance whether the statements controlled by an IF are indented four spaces, or three, or two. Arguments can be made for each, but there is no one way that is absolutely right. Within any one installation, however, *some* set of rules should be followed, or the value of indentation will be lost.

Creating a Structured Program

Structured programming, as the term is used in this publication, refers to the coding portion of the total program development cycle. It may help to sketch the cycle, indicating how structured programming relates to each phase.

The program development process can begin when, in response to a statement of requirements, a *specification* is developed that states the objectives of the application. Then, initial design takes place, during which each major function is identified and then subdivided into lower level functions. HIPO diagrams are a design aid and documentation tool at this stage. It is important in initial design not to become enmeshed in low level details; the strategy is to manage complexity by attacking the problem one level of detail at a time.

It is not to be expected that program design will proceed in a straight-line fashion. The HIPO hierarchy chart may have to be drawn several times, as the expected segment size or the implications of logic flow become clearer. The basic idea is to begin with a top-level attack, with little detail, then fill in the successive levels, refining original plans as necessary until the design is complete.

Once the initial design is complete, programmers refine the design to add the details necessary for the coding process. In *detailed program design,* additional HIPO diagrams are created to specify further detail about each process. If flowcharts are used to express logic flow, they should include only the basic structures. Another technique used in the detailed design phase is pseudocode, an informal means of expressing logic. Although HIPO diagrams can reduce the need for other documentation of logic flow, flowcharts and pseudocode can be used with HIPO diagrams.

In pseudocode, the basic control logic structures and indentation are used in a carefully controlled way, but everything else is at the discretion of the programmer: elements of programming languages may be utilized, or mathematical notation if it is appropriate to the application, etc. Pseudocode is similar to a programming language, but it is not compilable, and it is not bound by formal syntactical rules. Pseudocode is used to depict detailed logic while avoiding the distractions of the details of programming language requirements; it is easier to modify than programming language statements. When detailed program design is finished, the translation from pseudocode to the chosen programming language should be straightforward, since what is normally the most difficult part (the logic) is finished. Examples of pseudocode appear in the illustrations in Chapter 3.

In the *coding* stage of program development, the techniques that have become identified with structured programming, as the term is used here, come into greatest prominence. Program statements implementing control logic structures are used, and they are indented to show the scope of influence of the structures; thus, the details of code are clearly related to the structure of the design. For ease of understanding, no structure is allowed to extend over a page boundary. Meaningful data and procedure names are used, perhaps following conventions that suggest the functions of the data and procedures. Program segments are proper programs (one entry, one exit), and can be read in sequence from top to bottom.

It is becoming increasingly common for completed code to be checked by another programmer, either in a structured walkthrough or in some other kind of code reading process. During *test* program errors are located and it is verified that the program performs according to specifications. With structured programs this stage may tend to take less time than before because errors can be located and corrected more rapidly in the more readable structured code.

Finally, the program has to be *maintained* over the period of its use. Specifications change, equipment configurations are modified, and coding errors are discovered; these may require program modifications. Over the life of a major program, maintenance may require more effort than the original program development.

Structured programming facilitates program maintenance for much the same reason that it facilitates program testing: the program can be easier to understand. Whether the original programmer or a different maintenance programmer is involved, changes can be easier to make and be less likely to cause undesired effects elsewhere in the program.

In summary, program development consists of requirements specification, initial design, detailed design, coding, test, and maintenance. The most difficult task normally is design, which properly should receive the most attention and effort, since errors generally are least costly to correct at this stage.

Documentation

How much documentation of a program's logic is needed in addition to the program itself? In the past it has sometimes been argued that the logic of a program should be documented with a complete set of flowcharts. This contention may need to be reevaluated for structured programs, which can display their own logic better than conventional programs.

To reduce the need for documentation of logic, the code should follow certain guidelines of good programming practice that for many years have been characteristic of the best programmers. Data names and labels should be as indicative as possible of the functions of the data items and program elements, even if this tends to lengthen the names. "Tricky" coding should always be avoided.

When these and other common-sense principles have been followed, and the program has been written according to the principles of structured programming (only a few control logic structures, indentation, top-down readability), there should be little need for documentation of the logic flowchart type. (The need for documentation of function provided by HIPO heirarchy charts and HIPO diagrams, such traditional documentation as data layout charts, as well as data preparation instructions, etc., is not affected by structured programming.)

Efficiency Considerations

Programmers are sometimes concerned that structured programming techniques may lead to object programs which run slowly or which create problems in a virtual storage system. There is nothing inherent in the structured programming approach that leads to inefficiency; the use of a restricted set of control logic structures and of segmentation does not automatically carry any time or space penalty.

Although no systematic study of many users has been made, some users have reported that structured programming techniques usually lead to no performance penalties. Problems, when and if they occur, should be seen in context of the full range of considerations that determine the effectiveness of a data processing operation. For instance, the ability to create programs on time may be much more important than a small object program speed penalty. Or, it may be noted that a "highly efficient" program that is very difficult to maintain is not really efficient in the context of total cost. Finally, efficiency always relates to a specific environment of compilers, hardware, and user code.

If object program speed *does* become a problem, however, the following approaches may be considered.

Identify those portions of the program that are most heavily used; various analysis programs may be helpful in doing so, such as by providing counts of statement executions. It will usually be found that a rather small part of the program has a large influence on speed. Concentrate on those few segments. It may be necessary to recode procedures to inline code, or to "unwind" short, heavily used loops. Attention should be given to the possibility of avoiding certain data conversions or language features that may adversely affect performance. Since usually only a small part of the program needs to be modified, this modification should not take a great deal of effort.

If excessive paging in a virtual storage system is a problem, the basic solution is to place procedures that are used together in the same virtual storage page. Again, analysis programs can be a help. Structured programming can actually be a benefit in this kind of tuning, since procedures are never entered except by a reference to their names. Of course, the scope of the data references must be considered. Further, performance problems, whatever coding techniques are used, can seldom be predicted in advance. Because of the ease of maintaining (changing) structured programs, the likelihood is that performance problems can be more easily corrected.

Getting Started in Structured Programming

One way to evaluate structured programming in an installation is the following:

- Management authorizes the use of structured programming in a project. The first structured programming project should be neither trivial nor extremely difficult, but rather one that would be considered of normal size and level of difficulty. At least two programmers should be assigned to the project so that they can check each other's code.

- Programmers assigned to the project familiarize themselves with the subject. Some installations have implemented structured programming on their own; others have found that attending a class was necessary. Experienced PL/I programmers may be unfamiliar with or reluctant to use the following PL/I language facilities required or permitted in structured PL/I programs:

  - Nested IF's
  - DOWHILE, DOUNTIL
  - Compound conditions for nested IF's, DOWHILE, and DOUNTIL

Therefore, it may be advisable for programmers implementing structured programming in PL/I without attending a class to review these PL/I statements in the appropriate reference manual.

- A set of guidelines for the initial effort should be established. Those in Chapter 2 on PL/I implementation could be used; many installations will prefer to establish their own. The guidelines for the first project should avoid extending the permissible control logic structures; uncontrolled extensions can easily destroy the value of structured programming. Some programmers find it helpful to summarize the guidelines in the form of a checklist or a simple illustrative program.

- After creating the HIPO diagrams and visual table of contents, pseudocode or flowcharts can be used for detailed logic, if appropriate. The code is then written and the program tested.

The evaluation process can be repeated and the guidelines modified until the programmers have sufficient experience with structured programming. At this time structured programming guidelines can be incorporated into the installation's standards.

Chapter 2: Implementing Structured Programming in PL/I

Introduction

Once the principles of structured programming are understood, writing structured programs in PL/I is a matter of habitually following a few simple rules. All control logic structures can be expressed in PL/I.

The subject can be approached in terms of three major considerations:

1. PL/I implementation of the control logic structures.
2. Organization of a structured PL/I program
3. Indentation conventions

Also of interest are some pointers on naming conventions, use of comments, and special conditions.

This chapter assumes the use of one of the following: Release 3 of the OS/PLI Optimizing Compiler (5734-PL1), Optimizing Compiler and Libraries (5734-PL3), or Checkout Compiler (5734-PL2), and Release 5 of the DOS PL/I Optimizing Compiler (5736-PL1) or Optimizing Compiler and Libraries (5736-PL3). All of these compilers offer additional support for structured programming techniques.

Control Logic Structures

SEQUENCE

Sequencing is implemented in the PL/I language simply by writing statements in succession.

IFTHENELSE

The IFTHENELSE structure tests an element expression to determine which of two function blocks will be executed.

The *flowchart* of the IFTHENELSE structure is shown in Figure 11.

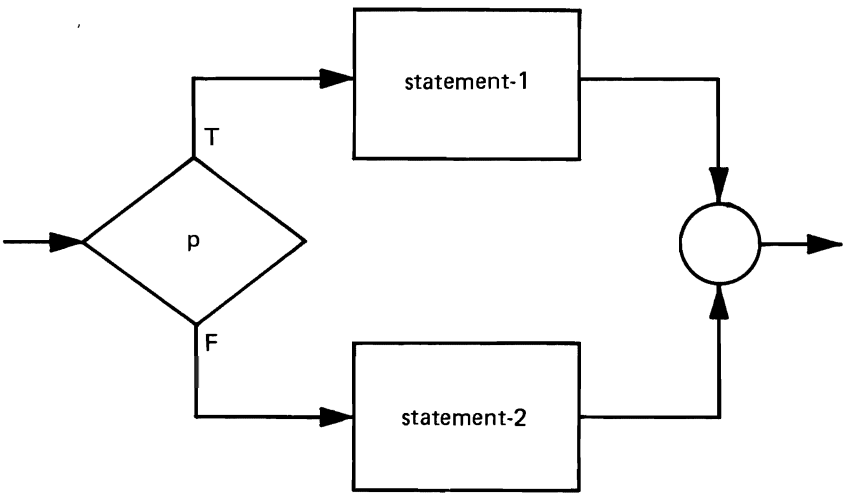


Figure 11. Flowchart for the IFTHENELSE

The *pseudocode* of the IFTHENELSE is:

```
IF condition-p

THEN

    statement-1

ELSE

    statement-2

ENDIF
```

The PL/I IF statement format for the IFTHENELSE may be shown in two common variations:

```
IF p
THEN

   DO;
      statement-1;
      .
      .
      .
   END;

IF p
THEN
   DO;
      statement-1;
      .
      .
      .
   END;
ELSE
   DO;
      statement-n;
      .
      .
      .
   END;
```

The THEN and ELSE are vertically aligned with the IF. The statements controlled by the THEN and ELSE portions are indented to show the span of control of the logic figure.

It is recommended that a DO group be used in the IF statement even when only a few statements are controlled. The compiler's syntax checking will reveal any logic errors caused by missing END's on DO's and BEGIN's.

DOWHILE

The DOWHILE structure tests a predicate and executes a function so long as the predicate is true. The flowchart is shown in Figure 12.

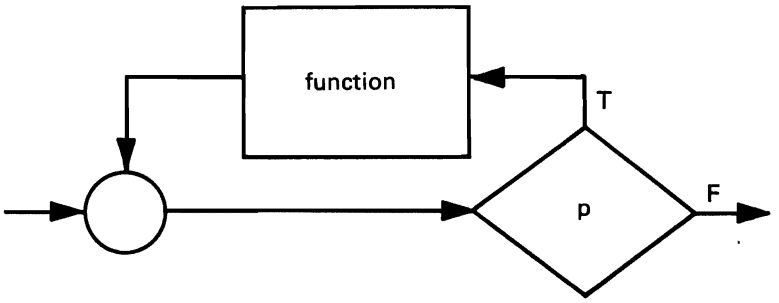


Figure 12. Flowchart for the DOWHILE.

The pseudocode for the DOWHILE is:

```
DOWHILE p
     function
ENDDO
```

The basic PL/I format for the DOWHILE is:

```
DO WHILE (p);
  statement-1;
    .
    .
    .
END;
```

One form of the DOWHILE with indexing, as permitted in PL/I is:

```
DO variable = expression-1 TO expression-2 BY
  expression-3
      WHILE (p);
   statement-1;
    .
    .
    .
END;
```

The REPEAT option provides an alternative method of specifying successive values of the control variable as in:

```
DO variable = expression-1 REPEAT (expression-2)
      WHILE (p);
END;
```

Another variation leaves the predicate implicit in the indexing parameters:

```
DO variable = expression-1 TO expression-2 BY
  expression-3;
   statement-1;
    .
    .
    .
END;
```

Many other forms of indexing are possible, as explained in the PL/I language reference manuals.

DOUNTIL

The DOUNTIL structure executes a function and then tests a predicate to determine whether to repeat it again. The flowchart is shown in Figure 13.

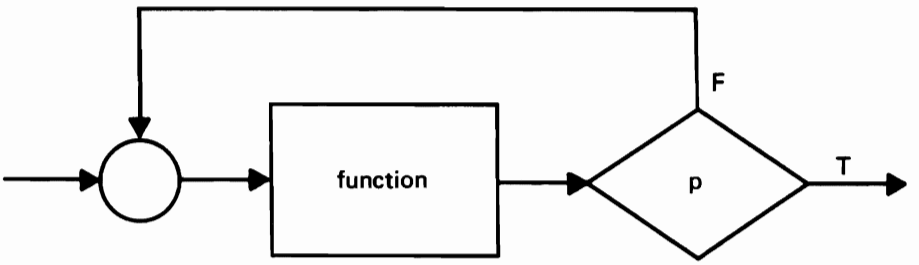


Figure 13. Flowchart for the DOUNTIL

The pseudocode for the DOUNTIL is:

```
DOUNTIL p

     function

ENDDO
```

The basic PL/I format for the DOUNTIL is:

```
DO UNTIL (p);
   statement-1;
   .
   .
   .
END;
```

As with the DOWHILE, variations of the DOUNTIL are permitted in PL/I. And, the DOWHILE and DOUNTIL can be combined in PL/I as in:

```
DO WHILE (A = B) UNTIL (X = 10);
```

CASE

The CASE structure selects one of a set of functions for execution, based on the value of a parameter. The flowchart notation is shown in Figure 14. In PL/I, the CASE structure is implemented with a case-selection unit which has the following form:

```
SELECT (E);
   WHEN (E1) action_1;
   WHEN (E2) action_2;
   :
   :
   OTHERWISE action_n;
END;
next statement;
```

In this example, E, E1, etc., are expressions. When control reaches the SELECT statement, the expression E is evaluated and its value saved. The expression in the first WHEN clause is then evaluated, and its value compared with the value of E. If the two values are equal, the action specified by action_1 is performed; if they are not equal, the expression in the next WHEN clause is similarly evaluated and compared. If none of the expressions in the WHEN clauses is equal to the expression in the SELECT statement, the action specified in the OTHERWISE clause is executed unconditionally.

After the action specified in a WHEN or OTHERWISE clause has been performed, control passes to the first executable statement following the END statement, unless the normal flow is changed by the specified action.

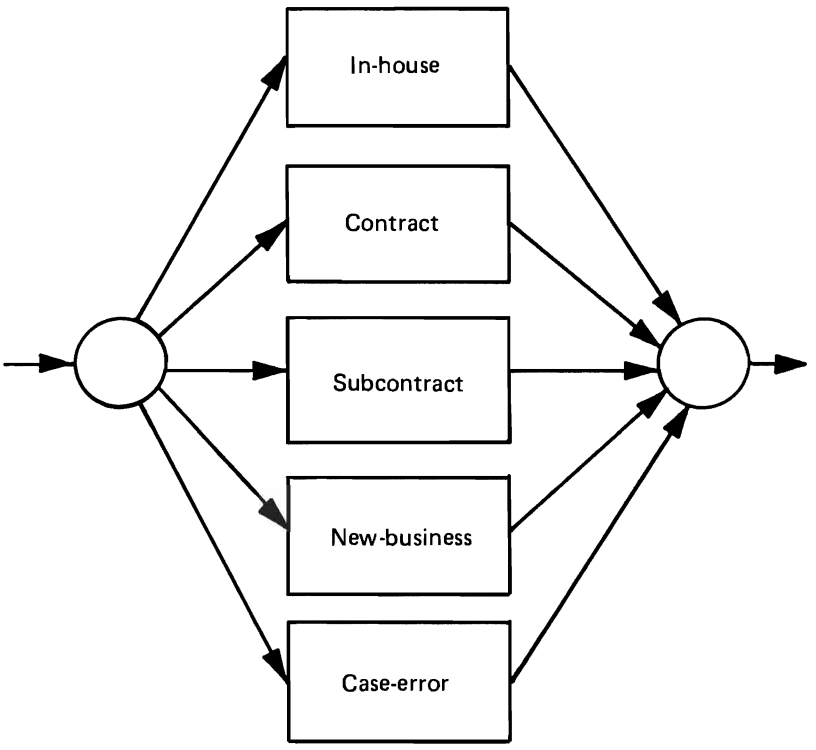


Figure 14. Flowchart for the CASE control logic structure

The example of Figure 14 could be coded as follows:

```
SELECT (CODE);
   WHEN (CODE = 'B') CALL IN_HOUSE_PROC:
   WHEN (CODE = '7') CALL CONTRACT_PROC;
   WHEN (CODE = 'C') CALL SUBCONTRACT_PROC;
   WHEN (CODE = 'D') CALL NEW_BUSINESS_PROC;
   OTHERWISE CALL CASE_ERROR_PROC;
END;
```

The LEAVE Statement

The LEAVE statement is used to transfer control from within a do-group to the first executable statement following the END statement that delimits the group. For example,

```
DO ....;
   .
   .
   LEAVE;
   .
END;
next statement
```

If the LEAVE statement contains a reference to a statement label (for example, LEAVE A), control is transferred to the statement following the END statement that closes the do-group whose DO statement has the specified label. For example:

```
A;
  DO I = 1 to 10;
     DO J = 1 to 5;
        IF X(I,J)=0
        THEN
           LEAVE A;
        ELSE
           ........;
     END;
     statement within group A;
  END;
  statement after group A;
```

A LEAVE statement cannot cause control to leave a block.

When evaluating the use of a LEAVE statement that references a statement label, installations might consider that some users have questioned whether its use is appropriate in a structured programming environment.

Program Organization

A structured PL/I source program is organized into segments. (A segment, in structured programming terminology, has been previously defined as consisting of control logic structures and having only one entry and one exit. The term as used here has nothing to do with the different meanings of the term in connection with the functions of operating systems or data base management systems.)

As previously discussed, a segment in PL/I terms, can consist of one or more external and/or internal procedures, or code incorporated from a library with a %INCLUDE statement.

The first segment of the program contains the MAIN procedure. Other segments may contain data declarations, executable code, or commentary blocks. In each case the segment should be complete -- one or more complete DECLARE statements; a set of comments opened and closed within the segment; or exactly one procedure, BEGIN-block, or DO-group.

The %PAGE listing control statement can be used to place each segment on a separate page of the program listing. If a program contains many short segments, paper conservation becomes a consideration, and it may be preferable to use the %SKIP statement to place a few blank lines between segments rather than putting each one on a separate page. A segment should still not extend over a page boundary.

Indentation Guidelines

The following are *only* suggestions. No standardization of indentation conventions has developed so far, and there seems to be little pressure for it so long as consistent standards are followed within any one organization. The reasons given for the suggestions that follow will be a guide in developing installation standards. The sample programs in Chapter 3 illustrate many of the guidelines.

The key idea in devising helpful indentation guidelines is the production of programs in which the visual layout of the program elements aids the reader in understanding program relationships and functioning.

Following are some ways this may be done:

- Labels stand out better if they always begin in column 2 and appear on a separate line.
- Consistency is obtained by starting all statements in column 4, unless other indentation rules dictate some column to the right of column 4.
- The characteristics of information are better displayed if the attributes in DECLARE statements are vertically aligned.
- The free use of blank lines can exhibit more clearly that relationships exist among items so grouped in the declarations.
- Statements are much easier to locate and to change if no more than one statement is written on a line.
- The scope of control of IFTHENELSE statements, DO groups, and BEGIN blocks is made clearer if the statements controlled by these elements are indented by some consistent amount. Three columns is suggested as a starting guideline, but the number is not critical so long as consistency is maintained within any one organization. An indentation unit of two spaces results in less unused space at the left end of lines; an indentation unit of four columns makes the structure more apparent. Three is a reasonable compromise.

- Statements are easier to locate if the second and following lines of a continued statement are indented by some consistent amount, such as twice the normal indentation unit.

- BEGIN blocks and DO groups can be indented according to the same guidelines. The BEGIN or DO starts in whatever column is determined by previous statements. Statements within the block are indented by three columns from the BEGIN or DO. The END statement is always used and is aligned with the BEGIN or DO.

- When several files are being opened with one OPEN statement (and all files should be explicitly opened and closed), the file names can be vertically aligned:

```
OPEN FILE (TRANFIL),
     FILE (MASTFIL),
     FILE (SYSPRINT);
```

Many other opportunities can be found to use formatting of the source program to enhance ease of understanding, which, to repeat, is the primary goal of all indentation conventions.

Names

Considerable care should be exercised in devising names for data and labels to make them as helpful as possible to the reader in understanding the function and structure of the parts of the program. Therefore, installations should consider adopting conventions that encourage the use of meaningful names. An example of such a convention would be to prefix the names of transaction file items with T, old master file items with M, and new master file items with NM. Another possible convention, not consistent with the first, is that data name qualification be used consistently to convey information about data organization. Another would be that file names must contain the word FILE and record names the word RECORD or perhaps REC. A possible convention for labels would be to require that all labels suggest the procedure's function and begin with numbers denoting sequence of hierarchy in the program. Many other such conventions are possible. Once learned, their use involves little extra effort.

Comments

Experience has shown that well-structured PL/I code can be largely self-documenting, assuming the use of descriptive data and paragraph names. It is recommended that an attempt be made to write programs without comments. If comments are used, however, they should be organized and formatted so that they do not interfere with the readability of the program. Free use of blank lines will make comments stand out from associated code.

Special Conditions

Most programming environments allow for specified unusual conditions to interrupt the normal flow of processing and activate exception-handling routines. Common examples are end-of-file conditions and arithmetic overflow. Whether the structure theorem applies to programs containing such elements depends on whether they violate the one-entry, one-exit principle and thus fail to be proper programs. Certain types of interrupts always break the normal flow; others may or may not, depending on how the program is written.

ON-units can be used to specify the desired processing for events such as data set label processing, input/output error routines, and various other asynchronous operations. The blocks of code in ON-units are "out-of-line" and therefore involve an interruption of sequential control. This is usually considered undesirable in structured programming. However, since this is PL/I's method of handling these essential functions, no attempt has been made to restrict the use of these features. The violation of the spirit of structured programming is lessened if the ON-unit contains no GO TO statements, since control then automatically returns to the statement following the one that caused the interrupt. However, this is not always possible depending on the type of ON-unit.

Occasionally, it is not feasible to handle certain conditions within a series of statements. This situation may arise either within conditional statements or during normal processing, for example, when errors are detected in data editing which prevent further processing. The programmer has at least two methods of handling such situations. One is to set a flag and then return control to the next-higher level routine for further action. This technique usually works well, and there are no violations of structured programming conventions. If, however, the error is detected within the innermost level of many nested levels, many tests may be required (one at each level) to return control up through the nested structures to the point where the error can be handled. Another alternative is to allow the use of GO TO to leave such disabling error routines. Good judgment should be used to determine whether the maintainability of the program is improved by using a few GO TO's in this case.

Chapter 3: Three Illustrative Programs

The best way to get a quick idea of what any programming technique is all about is to see examples of programs that employ it. In that spirit, three illustrative programs are presented that have been written following the principles discussed earlier. The IBM OS PL/I Checkout Compiler (5734-PL2) Version 1 Release 3 Modification 0 was used to compile the examples in this chapter. The programs were executed under VM/370 Version 3 Level 0.

A Two-Level Control Total Program

One of the most common data processing operations is the preparation of a summary report providing totals broken down by several levels of control, as well as a final total. A two-level control total report illustrates the basic ideas, and can easily be extended to any number of levels. In this example, it is assumed that the only report needed is the summary; extension of the program to include other processing and the printing of a detail line for each input record would involve no conceptual difficulties.

For concreteness, it is assumed that the major control is a sales district, and that the minor control is the salesman number. Each record contains a district number, a salesman number, and a dollar amount. The transaction file has already been sorted into sequence on salesman within the district. To keep things simple, the printing of headings and the counting of lines on the pages will be ignored; these matters are considered in the second sample program.

Figure 15 is a HIPO diagram for this processing. A pseudocode representation is shown in Figure 16. Notice how the logic is clearly exhibited by the use of indentation with the basic control structures of sequence, selection, and loop control. The DOWHILE is used for the loop control with the controlled code shown inline. The same logic is shown in flowchart form in Figures 17a and 17b. Working either from the pseudocode or the flowchart, the PL/I program in Figure 18 is not difficult to prepare.

Among general features to be observed are the use of blank lines for readability, the vertical alignment of the attributes in DECLARE statements, and the consistent use of an indentation unit of three spaces. Notice how the procedure can be read in top-down fashion. Its readability makes it unnecessary to explain the program further, assuming that the reader is familiar with the data processing ideas involved.

This program was run with a small sample of test data, and it produced the output shown in Figure 19.

Naturally, the program is quite rudimentary, since it does not include printing of headings, counting of lines, checking for sequence or other errors in the data, or any processing of the records other than the accumulation of totals. All of these operations can be included readily while still following structured programming concepts. Some of these operations are handled in the example that follows.

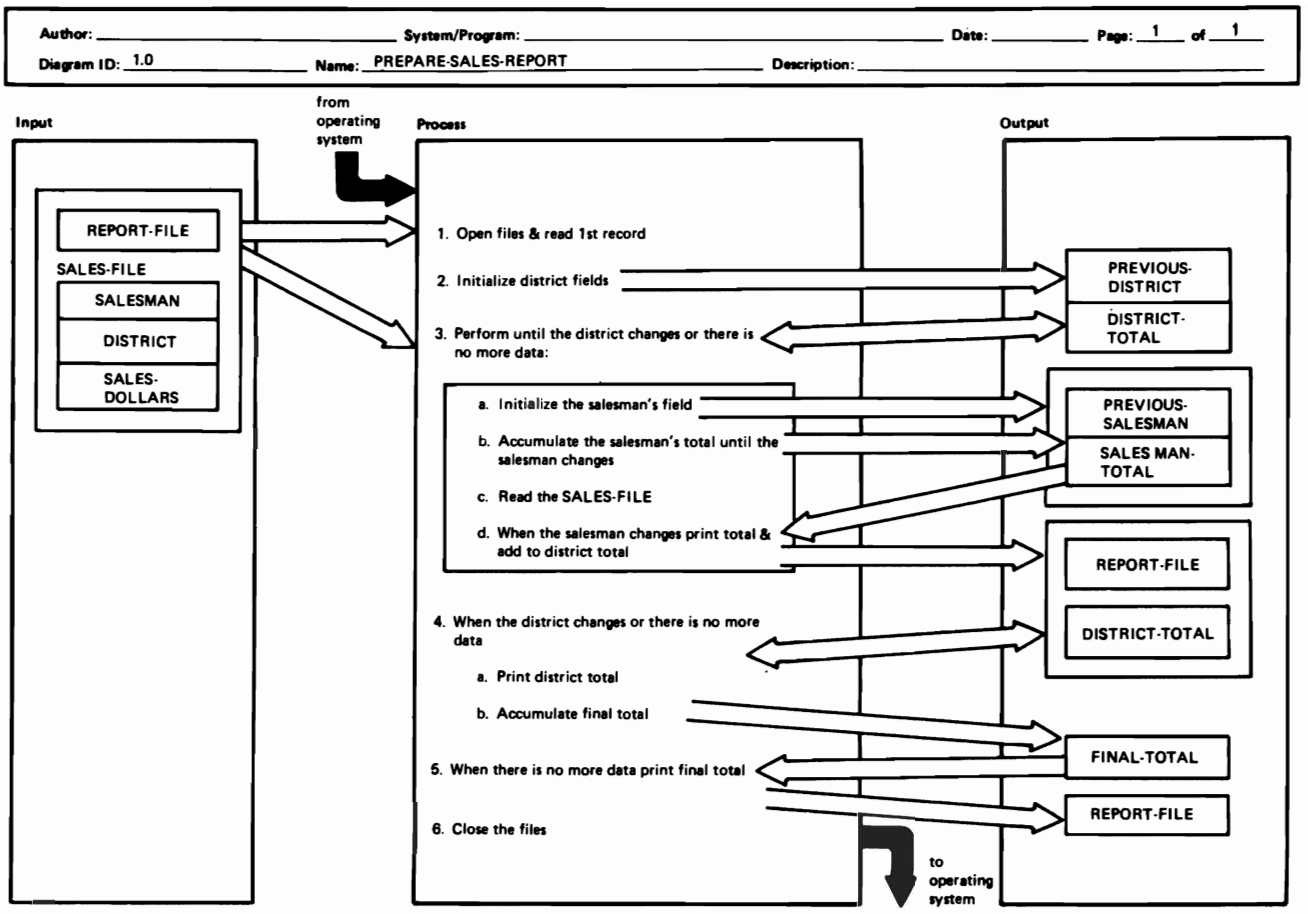


Figure 15. Detailed design level HIPO diagram for a two-level control total processing application

```
Open files
Get a sales record; on endfile indicate no more sales data
Zero final total
DOWHILE there is more sales data
    Zero district total
    PREVIOUS_DISTRICT = DISTRICT
    DOWHILE DISTRICT = PREVIOUS_DISTRICT and there is more sales data
        Zero salesman total
        PREVIOUS_SALESMAN = SALESMAN
        DOWHILE DISTRICT = PREVIOUS_DISTRICT
            and SALESMAN = PREVIOUS_SALESMAN
            and there is more sales data
              Accumulate salesman's total
              Get a sales record; on endfile indicate no more sales data
        ENDDO
        Print salesman's total
        Accumulate district total
    ENDDO
    Print district total
    Accumulate final total
ENDDO
Print final total
Close files
```

Figure 16. Pseudocode for a two-level control total processing application

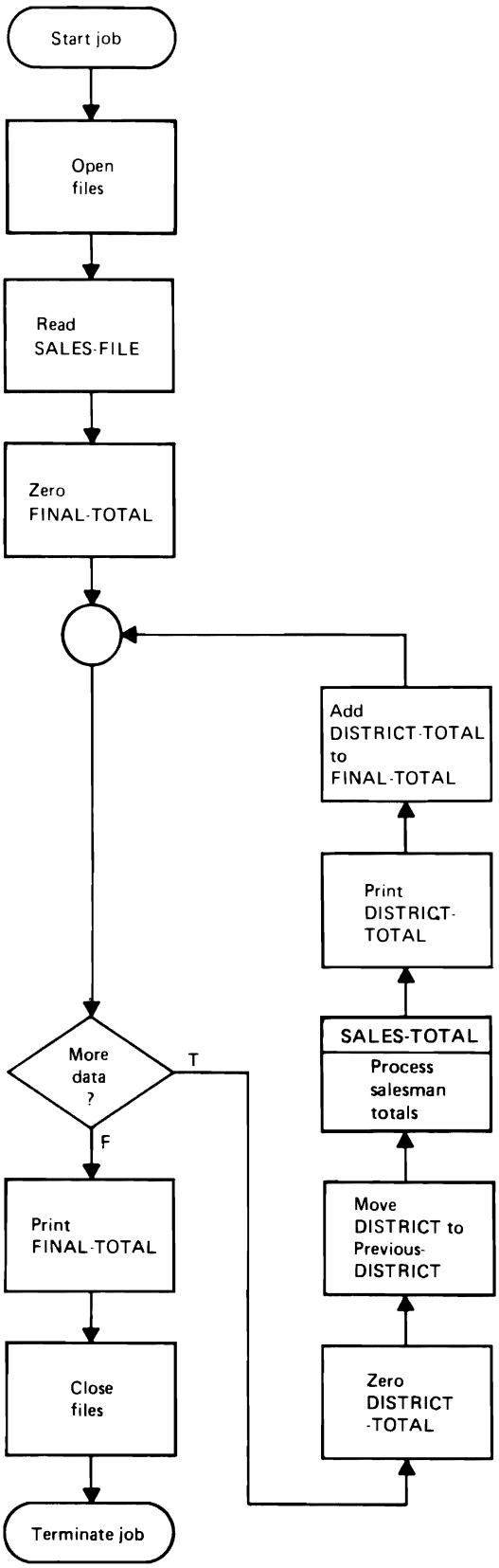


Figure 17a. Flowchart for the mainline processing portion of a two-level control total processing application

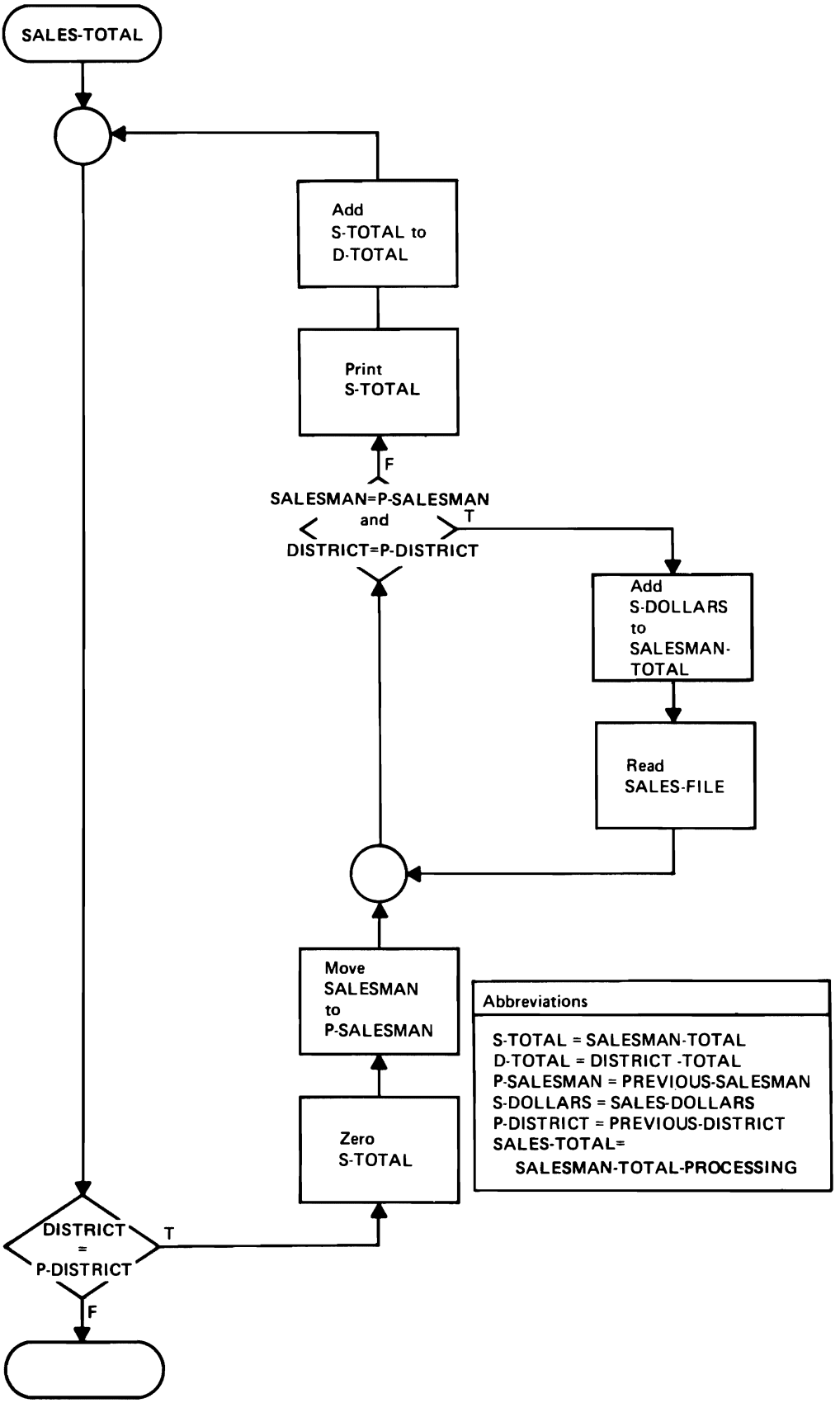


Figure 17b. Flowchart for the record processing portion of a two-level control total processing application

```
TWOLVL:
        PROCEDURE OPTIONS (MAIN);
        DECLARE
           SALESMAN                      FIXED DECIMAL (5),
           PREVIOUS_SALESMAN             FIXED DECIMAL (5),
           DISTRICT                      FIXED DECIMAL (5),
           PREVIOUS_DISTRICT             FIXED DECIMAL (5),
           FINAL_TOTAL                   FIXED DECIMAL (10, 2),
           DISTRICT_TOTAL                FIXED DECIMAL (10, 2),
           SALESMAN_TOTAL                FIXED DECIMAL (10, 2),
           SALES_DOLLARS                 FIXED DECIMAL (7, 2),
           THERE_IS_MORE_SALES_DATA      BIT (1) ALIGNED;
        DECLARE
           SYSIN                         FILE INPUT,
           SYSPRINT                      FILE OUTPUT;

START:
        OPEN FILE (SYSIN),
             FILE (SYSPRINT);

        THERE_IS_MORE_SALES_DATA = '1'B;
        ON ENDFILE (SYSIN)
              THERE_IS_MORE_SALES_DATA = '0'B;

        GET FILE (SYSIN) EDIT
              (SALESMAN, DISTRICT, SALES_DOLLARS)
              (F(5), F(3), F(7, 2));
        FINAL_TOTAL = 0;
        DO WHILE (THERE_IS_MORE_SALES_DATA);
           DISTRICT_TOTAL = 0;
           PREVIOUS_DISTRICT = DISTRICT;
           DO WHILE ( (DISTRICT = PREVIOUS_DISTRICT) &
                      THERE_IS_MORE_SALES_DATA);
              SALESMAN_TOTAL = 0;
              PREVIOUS_SALESMAN = SALESMAN;
              DO WHILE ( (DISTRICT = PREVIOUS_DISTRICT) &
                         (SALESMAN = PREVIOUS_SALESMAN) &
                         THERE_IS_MORE_SALES_DATA);
                 SALESMAN_TOTAL = SALESMAN_TOTAL + SALES_DOLLARS;
                 GET FILE (SYSIN) EDIT
                       (SALESMAN, DISTRICT, SALES_DOLLARS)
                       (SKIP, F(5), F(3), F(7, 2));
              END;
              PUT FILE (SYSPRINT) EDIT
                    (PREVIOUS_SALESMAN, SALESMAN_TOTAL)
                    (SKIP, F(5), P'BBB$$$,$$$,$$9V.99');
              DISTRICT_TOTAL = DISTRICT_TOTAL + SALESMAN_TOTAL;
           END;
           PUT FILE (SYSPRINT) EDIT
                 (PREVIOUS_DISTRICT, DISTRICT_TOTAL)
                 (SKIP, COLUMN(31), F(3), P'BBB$$$,$$$,$$9V.99');
           FINAL_TOTAL = FINAL_TOTAL + DISTRICT_TOTAL;
        END;
        PUT FILE (SYSPRINT) EDIT
              (FINAL_TOTAL)
              (SKIP, COLUMN(59), P'$$$,$$$,$$9V.99');
        CLOSE FILE (SYSIN),
              FILE (SYSPRINT);
END   /* TWOLVL */;
```

Figure 18. Structured program for a two-level control total processing application

```
41          $203.37
52          $110.00
69          $134.65
                        1          $448.02
18          $207.69
32          $185.60
                        2          $393.29
36          $194.15
39          $121.40
50           $51.80
                        3          $367.35
                                              $1,208.66
```

Figure 19. Illustrative output from the two-level control total program of Figure 18

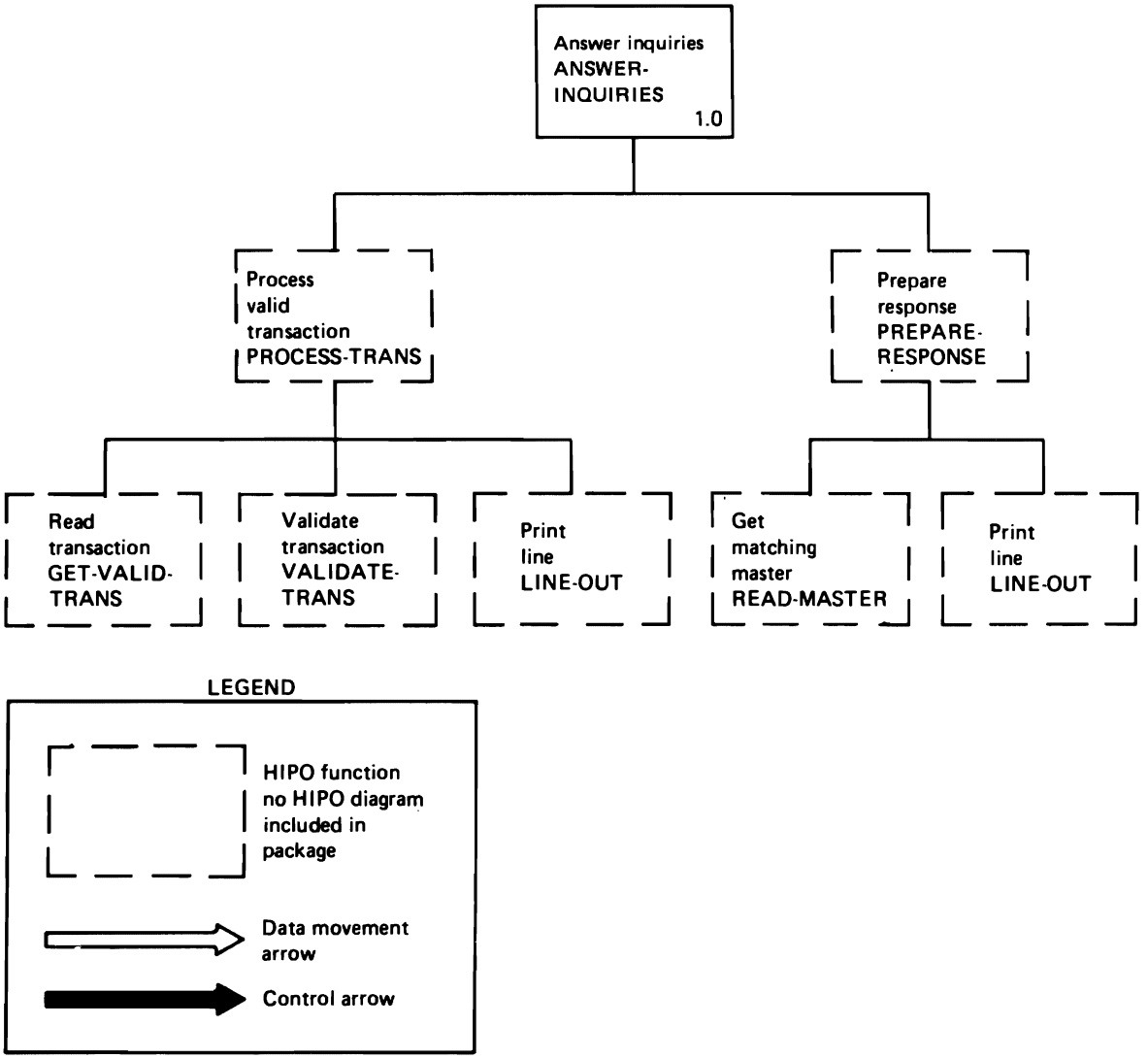


Figure 20. Detailed design level visual table of contents for the inquiry response application

An Inquiry Response Program

In this second example, the structured programming ideas are carried further.

A transaction file is in sequence on stock number; each record also contains a date limit in the form YYDDD. A master file is also in sequence on stock number, with each record containing a description of the product, a unit price, the quantity on hand, and the date of last activity. It is required, for each transaction record, to perform certain error checking, and then, if there has been activity since the transaction date, to produce an inquiry response consisting of the master record contents plus the value of the stock on hand; this is just the product of the quantity and the unit price. (It might be more realistic to assume an interactive environment, in which case the transactions would not be in sequence, and the master file would probably have indexed organization. The sequential organization was chosen to permit this example to display at least a small part of the logic of sequential file processing.)

A HIPO visual table of contents is shown in Figure 20 and a HIPO detail diagram in Figure 21. Observe in Figure 21 how the flow of data from input, through processing, to output, is presented visually. The flowchart, in five sections, is shown in Figure 22, and the program in Figure 23.

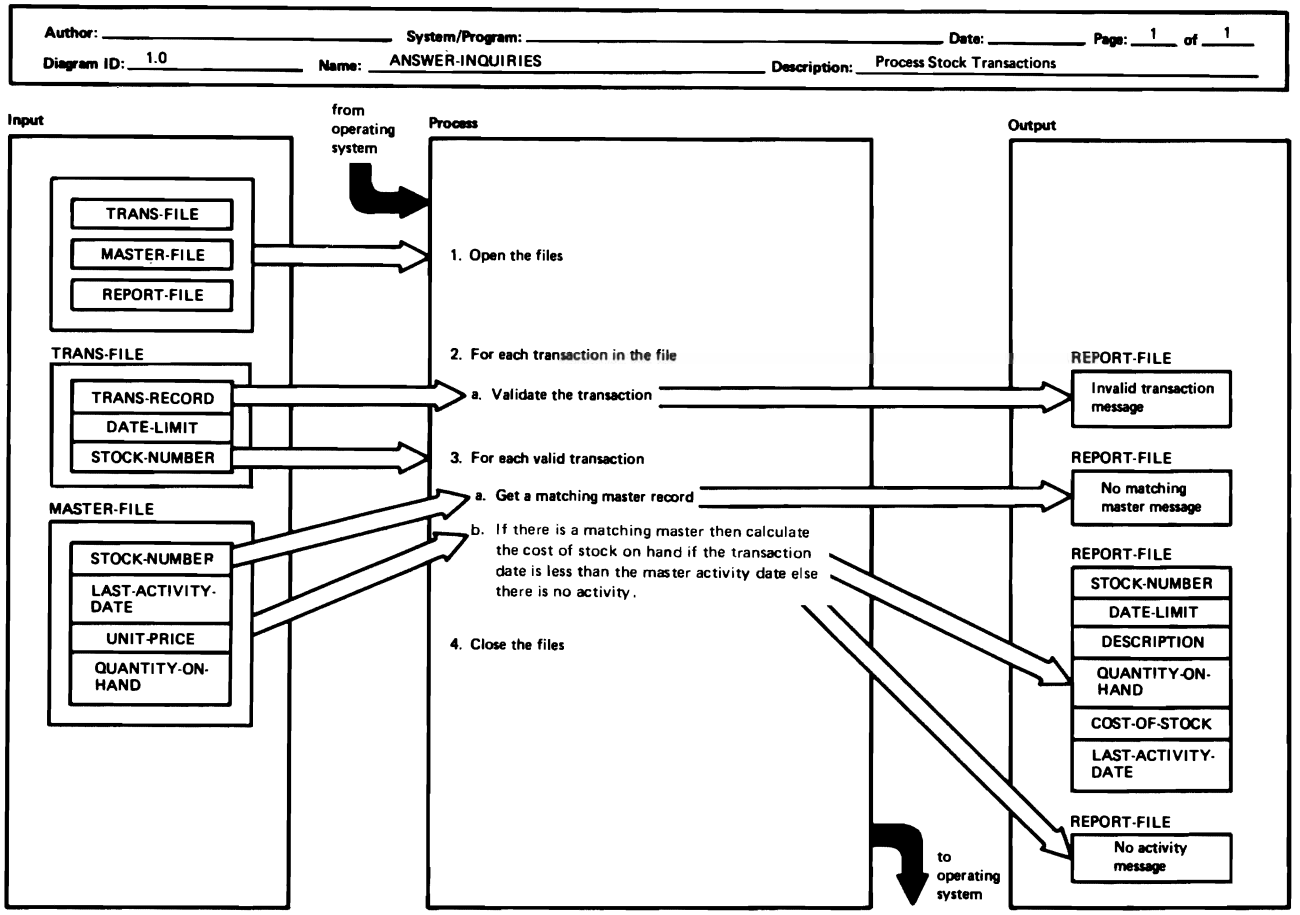
The declarations are more extensive this time, but the concepts should be familiar to most PL/I programmers. RECORD input is used to show that handling it with structured programming involves no problems. The built-in function HIGH is used in the ON-unit for the master file to place in the stock number for that file the largest possible character in the machine's collating sequence, so that when the end of the master file has been reached, any remaining transactions will be correctly flagged as having no matching master. The built-in function VERIFY is used in the internal procedure named GET_VALID_TRANS to determine whether the transaction contains any nonnumeric characters.

The label PROGRAM_LOOP is included to increase readability; there is never a transfer to it. Just before this point, note the call of GET_VALID_TRANS; this gets the first transaction before entering the processing loop. Once a valid transaction has been found, a DOWHILE seeks the matching master. Observe that if there are multiple transactions for the same stock number, this DOWHILE will not read the master file for transactions after the first one in a group.

In GET_VALID_TRANS, observe the use of redundant parentheses in the condition of the DOWHILE to reduce the possibility of (human) misunderstanding. The VERIFY function takes two arguments, and returns a zero if all characters of the first argument are found in the characters of the second argument; as used here, a nonzero result indicates a nonnumeric transaction.

LINE_OUT handles printing report lines, and, if a counter indicates the necessity, prints a heading line with a page number.

Figure 24 shows a sample master file and Figure 25 a sample transaction file for this program. Figure 26 shows the output produced when the program was run with these data files.



Extended Description

Notes	Paragraph	Ref.
2.	PROC-TRANS	
	GET-VALID-TRANS	
2a. All fields numeric, data less than 70,001	VALIDATE-TRANSACTION	
Print	LINE-OUT	
3.	PREPARE-RESPONSE	
3a.	READ-MASTER	
Print	LINE-OUT	

Extended Description

Notes	Paragraph	Ref.

Figure 21. A HIPO diagram for the inquiry response application

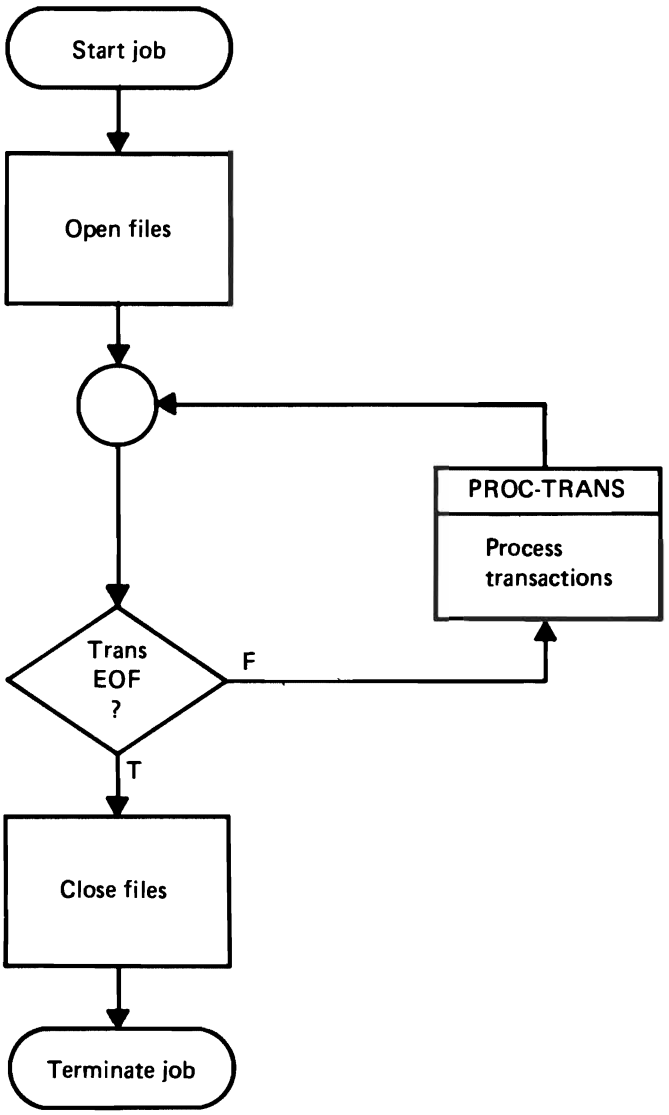


Figure 22a. Flowchart of the mainline processing for an inquiry response application

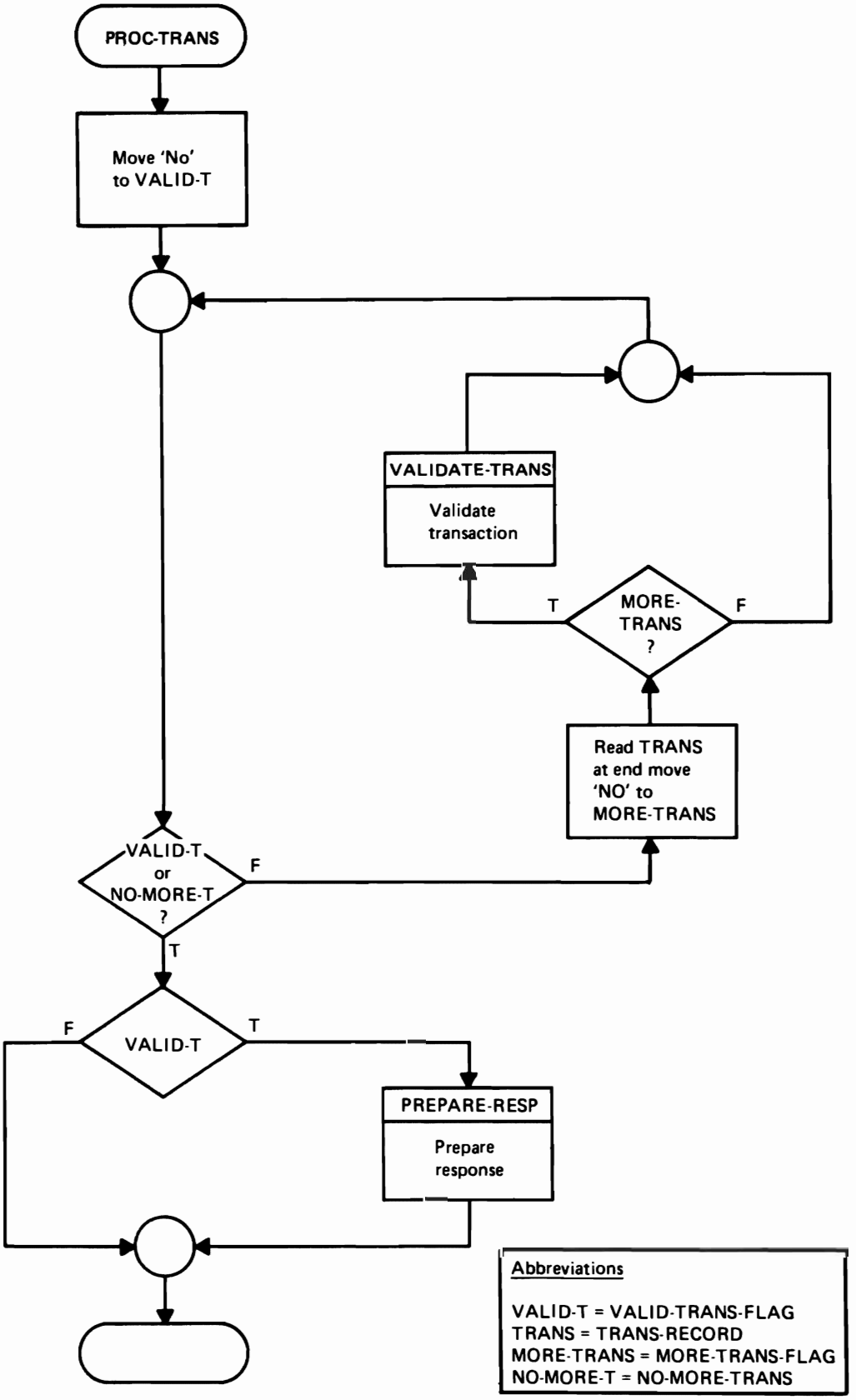


Figure 22b. Flowchart of the transaction processing logic for an inquiry response application

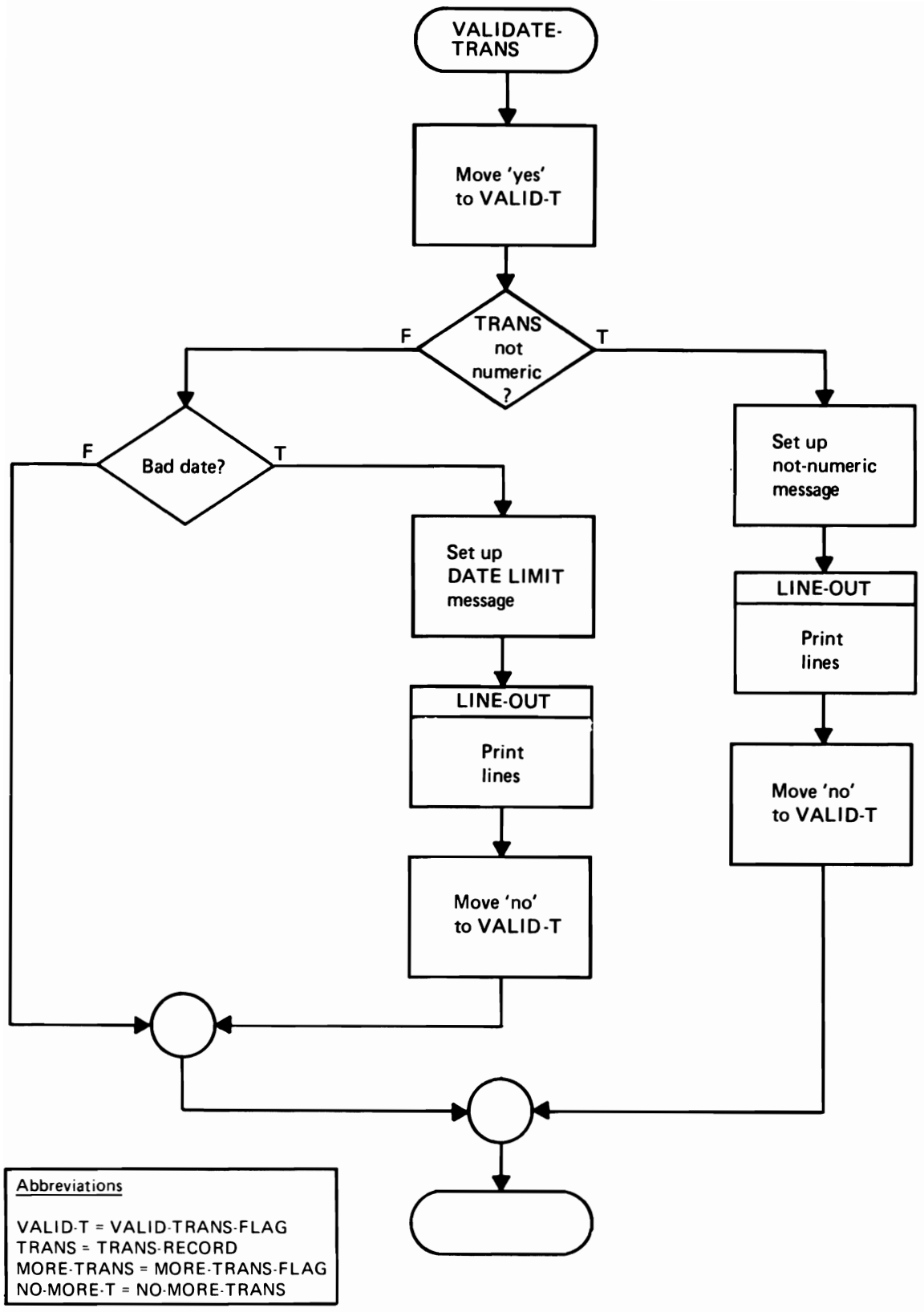


Figure 22c. Flowchart for the validation portion of an inquiry response application

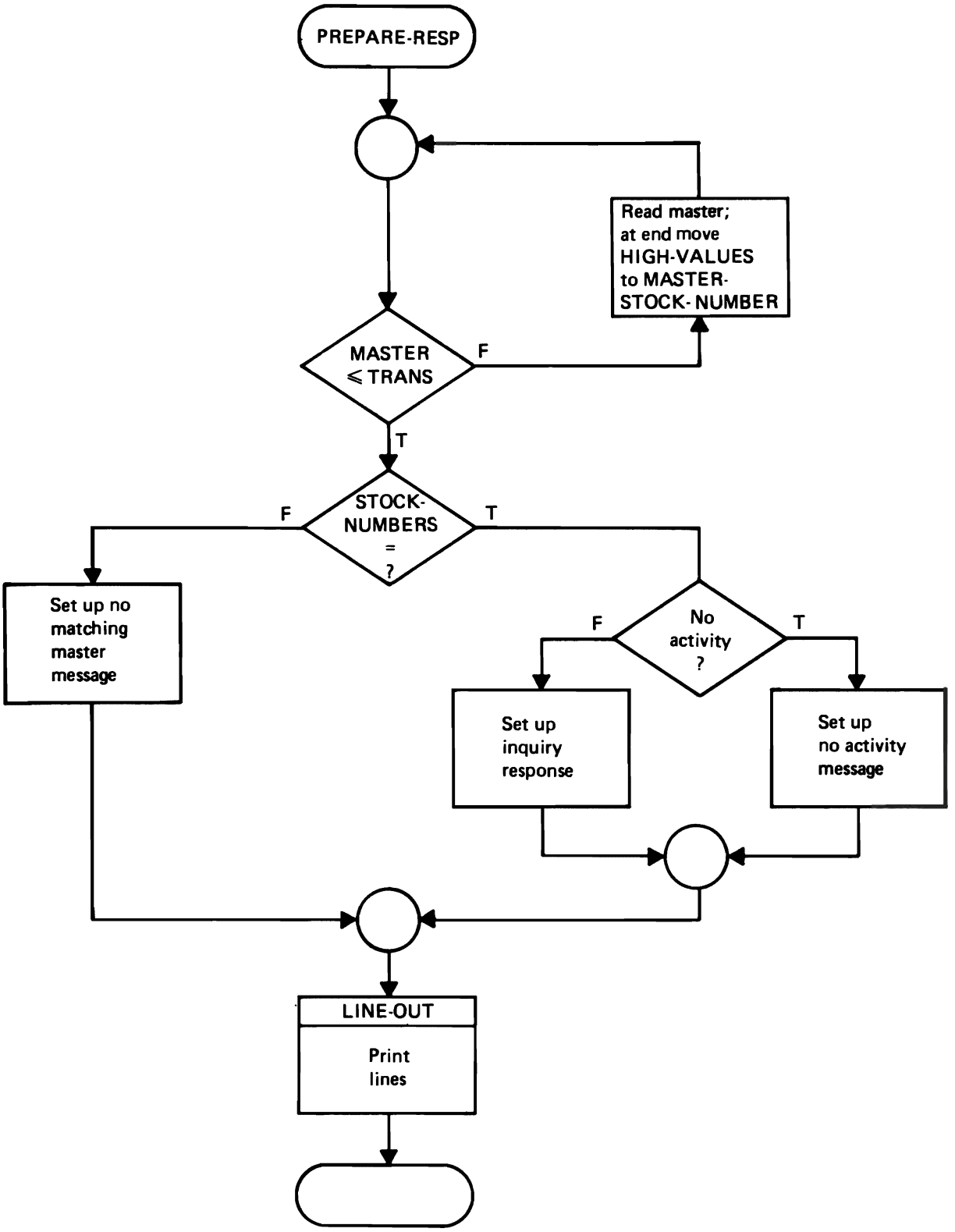


Figure 22d. Flowchart of the logic for preparing a response in an inquiry response application

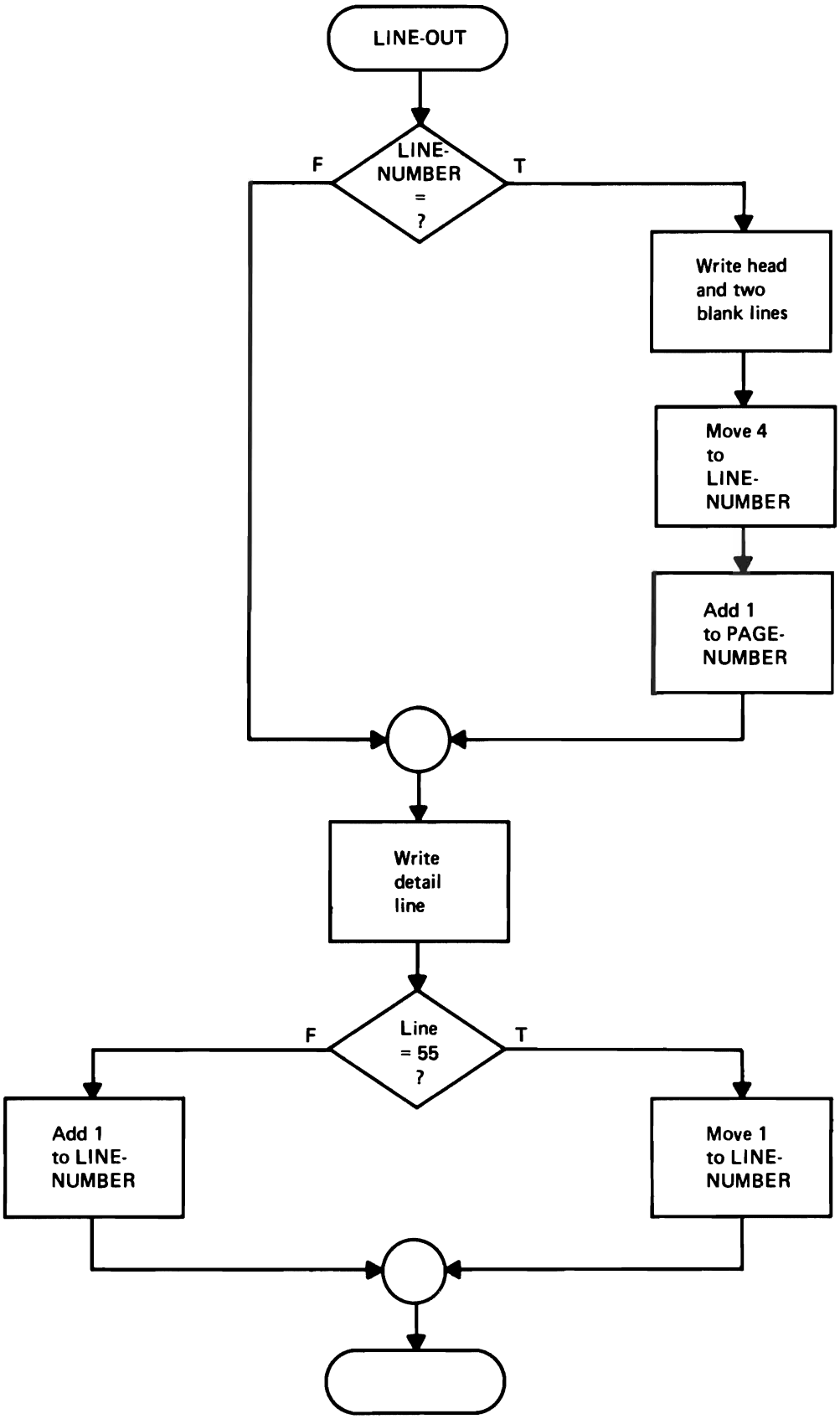


Figure 22e. Flowchart of the logic for printing heading and detail lines in an inquiry response application

```
INQRESP:
        PROCEDURE OPTIONS (MAIN);
        DECLARE
           MASTFIL FILE RECORD SEQUENTIAL
                   ENVIRONMENT (TOTAL F RECSIZE (80) ),
           TRANFIL FILE RECORD SEQUENTIAL
                   ENVIRONMENT (TOTAL F RECSIZE (80) ),
           SYSPRINT FILE OUTPUT;
        DECLARE
           1 TRANS,
              2 STOCK_NUMBER          CHARACTER (6),
              2 DATE_LIMIT            CHARACTER (5),
              2 FILL                  CHARACTER (69);
        DECLARE
           1 MASTER,
              2 STOCK_NUMBER          CHARACTER (6) INITIAL (' '),
              2 DESCRIPTION           CHARACTER (20),
              2 UNIT_PRICE            PICTURE '99999V99',
              2 QOH                   PICTURE '99999V99',
              2 LAST_ACTIVITY_DATE,
                 3 YEAR               CHARACTER (2),
                 3 DAY                CHARACTER (3),
              2 FILL                  CHARACTER (35);
        DECLARE
           1 RESPONSE_FIELDS,
              2 STOCK_NUMBER          CHARACTER (6),
              2 FILL_1                CHARACTER (3) INITIAL (' '),
              2 DATE_LIMIT            CHARACTER (5),
              2 FILL_2                CHARACTER (5) INITIAL (' '),
              2 DESCRIPTION           CHARACTER (20),
              2 FILL_3                CHARACTER (3) INITIAL (' '),
              2 UNIT_PRICE            PICTURE '$$$,$$9V.99',
              2 FILL_4                CHARACTER (3) INITIAL (' '),
              2 QOH                   PICTURE 'ZZZZ9V.99',
              2 FILL_5                CHARACTER (3) INITIAL (' '),
              2 TOTAL_COST            PICTURE '$$,$$$,$$9V.99',
              2 FILL_6                CHARACTER (5) INITIAL ('  19'),
              2 LAST_ACTIVITY_DATE,
                 3 YEAR               CHARACTER (2),
                 3 FILL_7             CHARACTER (3) INITIAL (' '),
                 3 DAY                CHARACTER (3);
        DECLARE
           INQUIRY_RESPONSE DEFINED RESPONSE_FIELDS CHARACTER (92);
        DECLARE
           MORE_TRANS_REMAIN_SW       BIT (1) ALIGNED INITIAL ('1'B);
        DECLARE
           NO_ACTIVITY_MSG            CHARACTER (50) INITIAL
                 ('NO ACTIVITY FOR THIS ITEM SINCE DATE IN INQUIRY');
        DECLARE
           NO_MATCHING_MASTER_MSG     CHARACTER (50) INITIAL
                 ('NO MASTER FOR THIS STOCK NUMBER');
        DECLARE
           HIGH                       BUILTIN,
           VERIFY                     BUILTIN;
```

Figure 23. Structured program for an inquiry response application (1 of 4)

```
        ON ENDFILE (MASTFIL)
              MASTER.STOCK_NUMBER = HIGH (6);

        OPEN FILE (TRANFIL),
             FILE (MASTFIL),
             FILE (SYSPRINT);

        CALL GET_VALID_TRANS (TRANS, MORE_TRANS_REMAIN_SW);
PROGRAM_LOOP:
        DO WHILE (MORE_TRANS_REMAIN_SW);
           DO WHILE (MASTER.STOCK_NUMBER < TRANS.STOCK_NUMBER);
              READ FILE (MASTFIL) INTO (MASTER);
           END;
           IF MASTER.STOCK_NUMBER = TRANS.STOCK_NUMBER
           THEN
              DO;
                 IF TRANS.DATE_LIMIT >= MASTER.YEAR||MASTER.DAY
                 THEN
                    CALL LINE_OUT (TRANS.STOCK_NUMBER||'   '||
                         TRANS.DATE_LIMIT||'     '||NO_ACTIVITY_MSG);
                 ELSE
                    DO;
                       RESPONSE_FIELDS = MASTER, BY NAME;
                       RESPONSE_FIELDS.DATE_LIMIT = TRANS.DATE_LIMIT;
                       RESPONSE_FIELDS.TOTAL_COST =
                             MASTER.UNIT_PRICE * MASTER.QOH;
                       CALL LINE_OUT (INQUIRY_RESPONSE);
                    END;
              END;
           ELSE
              CALL LINE_OUT (TRANS.STOCK_NUMBER||'   '||
                   TRANS.DATE_LIMIT||'     '||NO_MATCHING_MASTER_MSG);
           CALL GET_VALID_TRANS (TRANS, MORE_TRANS_REMAIN_SW);
        END PROGRAM_LOOP;

        CLOSE FILE (TRANFIL),
              FILE (MASTFIL),
              FILE (SYSPRINT);
        RETURN   /* TO OPERATING SYSTEM */;
```

Figure 23. Structured program for an inquiry response application (2 of 4)

```
GET_VALID_TRANS:
        PROCEDURE (TRANS, MORE_TRANS_REMAIN_SW);
        DECLARE
           1 TRANS,
              2 STOCK_NUMBER          CHARACTER (6),
              2 DATE_LIMIT            CHARACTER (5),
              2 FILL                  CHARACTER (69);
        DECLARE
           MORE_TRANS_REMAIN_SW       BIT (1) ALIGNED,
           VALID_TRANS_SW             BIT (1) ALIGNED;
        DECLARE
           EARLIEST_DATE_ALLOWED STATIC CHARACTER (5)
                  INITIAL ('70001');
        DECLARE
           NOT_NUMERIC_MSG       STATIC CHARACTER (50) INITIAL
                 ('ALL ITEMS IN INQUIRY MUST BE NUMERIC');
        DECLARE
           DATE_LIMIT_MSG        STATIC CHARACTER (50) INITIAL
                 ('DATE-LIMIT MUST NOT BE LESS THAN 70001');

        ON ENDFILE (TRANFIL)
              MORE_TRANS_REMAIN_SW = '0'B;

        VALID_TRANS_SW = '0'B;
        DO WHILE ( (VALID_TRANS_SW = '0'B) & MORE_TRANS_REMAIN_SW);
           READ FILE (TRANFIL) INTO (TRANS);
           IF MORE_TRANS_REMAIN_SW
           THEN
              DO;
                 VALID_TRANS_SW = '1'B;
                 IF VERIFY (TRANS.STOCK_NUMBER||TRANS.DATE_LIMIT,
                       '0123456789') ¬= 0
                 THEN
                    DO;
                       CALL LINE_OUT (TRANS.STOCK_NUMBER||'   '||
                             TRANS.DATE_LIMIT||'     '||
                             NOT_NUMERIC_MSG);
                       VALID_TRANS_SW = '0'B;
                    END;
                 ELSE
                    IF TRANS.DATE_LIMIT < EARLIEST_DATE_ALLOWED
                    THEN
                       DO;
                          CALL LINE_OUT (TRANS.STOCK_NUMBER||'   '||
                                TRANS.DATE_LIMIT||'     '||
                                DATE_LIMIT_MSG);
                          VALID_TRANS_SW = '0'B;
                       END;
              END;
        END;
END   /* GET_VALID_TRANS */;
```

Figure 23. Structured program for an inquiry response application (3 of 4)

```
LINE_OUT:
        PROCEDURE (LINE);
        DECLARE
           LINE                          CHARACTER (*);
        DECLARE
           HEAD                          CHARACTER (91) INITIAL
                ('  TRANSACTION          DESCRIPTION          UNIT PRIC
E     QOH     TOTAL COST  LAST ACTIVITY');
        DECLARE
           LINE_NUMBER           STATIC FIXED DECIMAL (3) INITIAL (1),
           PAGE_NUMBER           STATIC FIXED DECIMAL (3) INITIAL (1),
           SYSPRINT              FILE OUTPUT;

        IF LINE_NUMBER = 1
        THEN
           DO;
              PUT FILE (SYSPRINT) EDIT
                    (HEAD, PAGE_NUMBER)
                    (PAGE, A, P'(6)Z9');
              PUT FILE (SYSPRINT) EDIT
                    (LINE)
                    (SKIP(3), A);
              LINE_NUMBER = 4;
              PAGE_NUMBER = PAGE_NUMBER + 1;
           END;
        ELSE
           PUT FILE (SYSPRINT) EDIT
                 (LINE)
                 (SKIP, A);
        IF LINE_NUMBER >= 55
        THEN
           LINE_NUMBER = 1;
        ELSE
           LINE_NUMBER = LINE_NUMBER + 1;
END   /* LINE_OUT */;

END   /* INQRESP */;
```

Figure 23. Structured program for an inquiry response application (4 of 4)

```
000108DESK                    0018500000160075010
000115CHAIR, FOLDING          0001810001270075100
000180LAMP, FLOOR             0003750000120075180
000181LAMP, DESK              0002200001170075093
000200TYPEWRITER STAND        0002490000400074350
000309BOOKCASE, 5 SHELF       0004125000200075105
000310BOOKCASE, 4 SHELF       0003650000310075090
000311BOOKCASE, 3 SHELF       0002800000170075110
000480FILE CABINET, 4 DWR     0006180001000075130
000481FILE CABINET, 2 DWR     0003990000500075150
010684WASTEBASKET, GREEN      0000417000120075190
010686WASTEBASKET, GRAY       0000417001900075120
010687WASTEBASKET, BLUE       0000417000570075182
021732SOFA, LEATHER, BROWN0035620000290075070
021739SOFA, LEATHER, RED      0035620000370075040
```

Figure 24. Illustrative master file for the inquiry response program of Figure 23

```
00010875001
00018075001
00020075001
00025075001
000310 75001
00031075001
00048075140
00048175140
010.68575140
01069075150
02173975030
03194075150
```

Figure 25. Illustrative transaction file for the inquiry response program of Figure 23

```
 TRANSACTION          DESCRIPTION          UNIT PRICE     QOH     TOTAL COST  LAST ACTIVITY

000108   75001    DESK                        $185.00      16.00      $2,960.00   1975   010
000180   75001    LAMP, FLOOR                  $37.50      12.00        $450.00   1975   180
000200   75001    NO ACTIVITY FOR THIS ITEM SINCE DATE IN INQUIRY
000250   75001    NO MASTER FOR THIS STOCK NUMBER
000310    7500    ALL ITEMS IN INQUIRY MUST BE NUMERIC
000310   75001    BOOKCASE, 4 SHELF            $36.50      31.00      $1,131.50   1975   090
000480   75140    NO ACTIVITY FOR THIS ITEM SINCE DATE IN INQUIRY
000481   75140    FILE CABINET, 2 DWR          $39.90      50.00      $1,995.00   1975   150
010.68   57514    ALL ITEMS IN INQUIRY MUST BE NUMERIC
010690   75150    NO MASTER FOR THIS STOCK NUMBER
021739   75030    SOFA, LEATHER, RED          $356.20      37.00     $13,179.40   1975   040
031940   75150    NO MASTER FOR THIS STOCK NUMBER
```

Figure 26. Output of the program of Figure 23 when run with the illustrative files of Figures 24 and 25

Solving a System of Simultaneous Equations by The Gauss-Seidel Method

This example is for the benefit of readers more concerned with technical applications. It assumes some familiarity with simultaneous linear algebraic equations and with their iterative solution by the Gauss-Seidel method.

As many as 80 equations in 80 unknowns are to be permitted; the actual size N, which may be smaller than 80, is read from the first data card. This card also specifies MAX_ITERATIONS, the maximum number of iterations to be permitted, the convergence criterion EPSILON, and the largest absolute value permitted of an element in the system array, BIGGEST. The array is initialized to zero, so that only nonzero elements need be read; row and column numbers are checked for validity as the data cards are read. All data values are checked and errors reported, but the solution is not attempted if any errors are found.

Not all systems of simultaneous equations can be solved by the Gauss-Seidel method. After the coefficients and constant terms have been read, a check is made to determine that the main diagonal element in each row is larger in absolute value than the sum of the absolute values of the other coefficients in the row. If not, the error is reported and the solution is not attempted.

The actual solution proceeds in a succession of sweeps. Starting with all zeros for the unknowns, new values for all unknowns are computed in one sweep. A variable named RESIDUAL holds the largest difference between the old and new values of unknowns. When this residual is found to be less than the convergence criterion, the system has been solved. If convergence cannot be achieved in the specified maximum number of iterations, the nonconvergence is reported.

If all data values are acceptable and the system is suitable for solution by the Gauss-Seidel method, and if the solution converges, then the values of the N unknowns are printed as the solution.

Figure 27 shows pseudocode for the method of solution that is to be used. Observe how the logic of the solution is displayed, without distracting details. For example, the precise form of switch-setting is left to be detailed in the program. Likewise, in the procedure for reading the data, we find the line "IF data card invalid", which conveys the meaning clearly but does not specify exactly what tests are to be made; those details can be found in the program specifications and in the program. Note, too, that a summation sign denotes this commonly used mathematical function, which in the program will become a simple DO loop. If it were necessary to keep the pseudocode in machine-readable form, which is sometimes the practice, the Greek symbols would naturally have to be represented in some transliteration, or the loop could be shown in detail.

The program is shown in Figure 28. The mainline logic, according to which the various tests are made to determine at each stage what further actions are possible, is made clear by the use of meaningful data names, simple IFTHENELSE logic, and consistent indentation.

The internal procedure READ_DATA obtains the data and tests the validity of each element separately. The choice of how much testing to do is a design decision that is taken for granted here; if further tests, such as the reasonableness of the value N, were desired, they could be incorporated easily.

The procedure VALIDATE_SYSTEM is called into play if it is determined that the individual elements are acceptable. This function could, of course, have been made part of READ_DATA, which might then have been renamed READ_DATA_AND_VALIDATE_SYSTEM, or READ_DATA could have called this procedure. The form chosen was picked because it gives the clearest picture of the logic at the top level.

The actual solution of the system, if it is found to be potentially solvable, is done with the procedure named SOLVE_SYSTEM. It involves no unusual concepts. Note, however, the use of the built-in function MAX to establish whether the newly computed difference between the old and new values of an unknown is greater than the previous value of RESIDUAL; this could, naturally, also have been done with an IF statement.

After the program had been tried with various erroneous data to check the error-detection handling, it was tested with the following system:

$$
\begin{aligned}
12.063\,x_1 + 1.018\,x_2 - 4.200\,x_3 + 0.110\,x_4 &= 3.013 \\
1.934\,x_2 + 1.011\,x_3 - 0.500\,x_4 &= 1.165 \\
-0.110\,x_1 + 0.901\,x_2 + 6.914\,x_3 + 0.100\,x_4 &= 18.429 \\
-1.952\,x_1 + 2.139\,x_3 + 5.000\,x_4 &= -15.500
\end{aligned}
$$

Using a convergence criterion (EPSILON) of 0.01, the method found the solution shown in Figure 29.

```
Open files
Initialize bad data switch off
Clear arrays
Read data
IF no errors in data
THEN
  Validate system
  IF system is valid
  THEN
    Attempt to solve system
    IF solution converges
    THEN
       Print results
    ELSE
       Print 'did not converge'
    ENDIF
  ELSE
    Print 'cannot solve this sytem by Gauss-Seidel'
  ENDIF
ELSE
  Print 'bad data'
ENDIF
Close files
```

Figure 27. Pseudocode for a solution of simultaneous equations by the Gauss-Seidel method (1 of 4)

```
Read data:

Get N, maximum iterations, epsilon, biggest
More data switch = yes
DOWHILE more data remains
   Get a card
   IF more data remains
   THEN
      IF data card invalid
      THEN
         Print data values and error message
         Set bad data switch on
      ELSE
         Store element in array
      ENDIF
  ENDIF
ENDDO
```

Figure 27. Pseudocode for a solution of simultaneous equations by the Gauss-Seidel method (2 of 4)

```
Validate system:


DO I = 1 to N
    WHILE no bad rows have been found
  SUM = Σ(i≠j) |a_ij|
  IF |a_ij| ≤ SUM
  THEN
     Set bad row switch on
  ENDIF
ENDDO
```

Figure 27. Pseudocode for a solution of simultaneous equations by the Gauss-Seidel method (3 of 4)

```
Solve system:


Iterations = 1

DOUNTIL iterations > max iterations or residual ≤ epsilon
    Residual = 0
    DO I = 1 to N
       Sum = Σ (i≠j) a_ij x_j
       Temporary = (a_i,n+1 - Sum) /a_ii
       Residual = max(residual, abs(temporary - x_i))
       x_i = temporary
    ENDDO
    Add 1 to iterations
    If iterations > maximum permitted
    THEN
       Set no-converge switch on
    ENDIF
ENDDO
```

Figure 27. Pseudocode for a solution of simultaneous equations by the Gauss-Seidel method (4 of 4)

```
SIMEQ:
        PROCEDURE OPTIONS (MAIN);

        DCL (N, MAX_ITERATIONS)          FIXED BINARY;
        DCL ( A(80, 81), X(80) )              FLOAT;
        DCL (EPSILON, BIGGEST)           FLOAT;
        DCL BAD_DATA_SW                  BIT (1) ALIGNED;
        DCL VALID_SYSTEM_SW              BIT (1) ALIGNED;
        DCL CONVERGE_SW                  BIT (1) ALIGNED;
        DCL SYSIN                        FILE INPUT;
        DCL SYSPRINT                     FILE OUTPUT;

        OPEN FILE (SYSIN),
             FILE (SYSPRINT);
        A = 0;
        X = 0;
        BAD_DATA_SW = '0'B;
        CALL READ_DATA (A, N, MAX_ITERATIONS, EPSILON, BIGGEST,
              BAD_DATA_SW);
        IF BAD_DATA_SW = '0'B
        THEN
           DO;
              VALID_SYSTEM_SW = '1'B;
              CALL VALIDATE_SYSTEM (A, N, VALID_SYSTEM_SW);
              IF VALID_SYSTEM_SW = '1'B
              THEN
                 DO;
                    CONVERGE_SW = '1'B;
                    CALL SOLVE_SYSTEM (A, X, N,
                          EPSILON, MAX_ITERATIONS, CONVERGE_SW);
                    IF CONVERGE_SW = '1'B
                    THEN
                       PUT FILE (SYSPRINT) EDIT
                          ((I, X(I) DO I = 1 TO N))
                          (SKIP, F(2), E(15,6));
                    ELSE
                       PUT FILE (SYSPRINT) EDIT
                          ('SYSTEM DID NOT CONVERGE IN  ',
                                MAX_ITERATIONS, '  ITERATIONS')
                          (SKIP, A, F(2), A);
                 END;
              ELSE
                 PUT FILE (SYSPRINT) EDIT
                       ('CANNOT SOLVE THIS SYSTEM BY GAUSS-SEIDEL')
                       (SKIP, A);
           END;
        ELSE
           PUT FILE (SYSPRINT) EDIT
                 ('BAD DATA -- JOB ABORTED')
                 (SKIP, A);
        CLOSE FILE (SYSIN),
              FILE (SYSPRINT);
        RETURN;
```

Figure 28. A structured program to solve simultaneous equations by the Gauss-Seidel method (1 of 3)

```
%PAGE;
READ_DATA:
        PROCEDURE (A, N, MAX_ITERATIONS, EPSILON, BIGGEST,
              BAD_DATA_SW);

        DCL (I, J, N, MAX_ITERATIONS)       FIXED BINARY;
        DCL A(80, 81)                       FLOAT;
        DCL (EPSILON, BIGGEST, TEMPORARY)   FLOAT;
        DCL MORE_DATA_REMAINS_SW            BIT (1) ALIGNED;
        DCL BAD_DATA_SW                     BIT (1) ALIGNED;

        ON ENDFILE (SYSIN)
              MORE_DATA_REMAINS_SW = '0'B;

        GET FILE (SYSIN) EDIT
              (N, MAX_ITERATIONS, EPSILON, BIGGEST)
              (2 F(2), 2 F(10));
        MORE_DATA_REMAINS_SW = '1'B;
        DO WHILE (MORE_DATA_REMAINS_SW);
           GET FILE (SYSIN) EDIT
                  (I, J, TEMPORARY)
                  (SKIP, 2 F(2), F(10));
           IF MORE_DATA_REMAINS_SW
           THEN
              DO;
                 IF (I < 1) | (I > N) | (J < 1) | (J > N + 1)
                       | ABS (TEMPORARY) > BIGGEST
                 THEN
                    DO;
                       PUT FILE (SYSPRINT) EDIT
                            ('ERROR IN CARD WITH I = ', I, '  J = ', J,
                                  '  VALUE = ', TEMPORARY)
                            (SKIP, A, F(2), A, F(2), A, E(15,6));
                            BAD_DATA_SW = '1'B;
                    END;
                 ELSE
                    A(I, J) = TEMPORARY;
              END;
        END;
END   /* READ DATA */;
```

Figure 28. A structured program to solve simultaneous equations by the Gauss-Seidel method (2 of 3)

```
%PAGE;
VALIDATE_SYSTEM:
        PROCEDURE (A, N, VALID_SYSTEM_SW);

        DCL (I, J, N)                          FIXED BINARY;
        DCL ( A(80, 81), X(80) )               FLOAT;
        DCL SUM                                FLOAT;
        DCL VALID_SYSTEM_SW                    BIT (1) ALIGNED;
        DCL MAX                                BUILTIN;

        DO I = 1 TO N
              WHILE (VALID_SYSTEM_SW = '1'B);
           SUM = 0;
           DO J = 1 TO I - 1, I + 1 TO N;
              SUM = SUM + ABS (A(I, J));
           END;
           IF ABS (A(I, I)) <= SUM
           THEN
              VALID_SYSTEM_SW = '0'B;
        END;
END   /* VALIDATE_SYSTEM */;

%PAGE;
SOLVE_SYSTEM:
        PROCEDURE (A, X, N, EPSILON, MAX_ITERATIONS, CONVERGE_SW);
        DCL (I, J, ITERATIONS, MAX_ITERATIONS, N) FIXED BINARY;
        DCL (RESIDUAL, SUM, TEMPORARY)         FLOAT;
        DCL ( A(80, 81), X(80) )               FLOAT;
        DCL CONVERGE_SW                        BIT (1) ALIGNED;

        DO ITERATIONS = 1 TO MAX_ITERATIONS
              UNTIL (RESIDUAL <= EPSILON);
           RESIDUAL = 0;
           DO I = 1 TO N;
              SUM = 0;
              DO J = 1 TO I - 1, I + 1 TO N;
                 SUM = SUM + A(I, J) * X(J);
              END;
              TEMPORARY = ( A(I, N+1) - SUM ) / A(I, I);
              RESIDUAL = MAX( RESIDUAL, ABS(X(I) - TEMPORARY) );
              X(I) = TEMPORARY;
           END;
        END;
        IF ITERATIONS > MAX_ITERATIONS
        THEN
           CONVERGE_SW = '0'B;
END   /* SOLVE_SYSTEM */;

END   /* SIMEQ */;
```

Figure 28. A structured program to solve simultaneous equations by the Gauss-Seidel method (3 of 3)

```
1   1.493188E+00
2  -1.947272E+00
3   2.997915E+00
4  -3.799566E+00
```

Figure 29. The output of the program of Figure 28 when it was run with sample data corresponding to the system of simultaneous equations shown in the text

Bibliography

Baker, F.T. *System Quality through Structured Programming.* Proceedings of FJCC. December 1972.

Bohm, Corrado and Jacopini, Giuseppe (1966) *Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules.* Comm. ACM Vol. 9 No. 5 May 1966.

Dijkstra, Edsger W. *Go To Statement Considered Harmful.* Letter to Editor, Comm. ACM Vol. 11 No. 3 March 1968.

Dijkstra, Edsger W. *The Humble Programmer.* Comm. ACM Vol. 15 No. 10. October 1972.

Stay, J.F. *HIPO and Integrated Program Design.* IBM Systems Journal, Vol. 15 No. 2 1976. (Reprints are available under order number G321-5031.)

Stevens, W.P. and Myers, G.J. and Constantine, L.L. *Structured Design.* IBM Systems Journal Vol. 13 No. 2 1974. (Reprints are available under order number G320-5323.)

HIPO – A Design Aid and Documentation Technique. IBM Corporation, GC20-1851.

HIPODRAW – Installed User Program 5896-PFF Availability Notice. IBM Corporation, G320-5546.

Improved Programming Technologies – An Overview. IBM Corporation, GC20-1850.

An Introduction to Structured Programming in COBOL. IBM Corporation, GC20-1776.

An Introduction to Structured Programming in PL/I. IBM Corporation, GC20-1777.

Structured Programming (Independent Study Program) Textbook. IBM Corporation, SR20-7149.

Structured Programming (Independent Study Program) Workbook. IBM Corporation, SR20-7150.

An Introduction to Structured Programming in PL/I
Systems Management Manual
GC20-1777-1

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name and mailing address:

_____

_____

_____

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

What is your occupation? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Fold and tape | **Please Do Not Staple** | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 824
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape | **Please Do Not Staple** | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

Cut or Fold Along Line

Systems Management Manual An Introduction to Structured Programming in PL/I Printed in U.S.A. GC20-1777-1

GC20-1777-1

Systems Management Manual An Introduction to Structured Programming in PL/I Printed in U.S.A. GC20-1777-1

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601