ESSENTIALS OF COBOL PROGRAMMING: A Structured Approach

GERALD N. PITTS
BARRY L. BATEMAN

OBOL COBOL COBOL COBOL COBOL COBO
COBOL COBOL COBOL COBOL COBOL COBO
OBOL COBOL COBOL COBOL COBOL COBO
COBOL COBOL COBOL COBOL COBOL COBO
OBOL COBOL COBOL COBOL COBOL COBO
COBOL COBOL COBOL COBOL COBOL COBO
OBOL COBOL COBOL COBOL COBOL COBO
COBOL COBOL COBOL COBOL COBOL COBO
OBOL COBOL COBOL COBOL COBOL COBO
COBOL COBOL COBOL COBOL COBOL COBO
OBOL COBOL COBOL COBOL COBOL COBO
COBOL COBOL COBOL COBOL COBOL COBO

COBOL COMPUTER SCIENCE PRESS COBO

ESSENTIALS OF COBOL PROGRAMMING: A Structured Approach

GERALD N. PITTS
BARRY L. BATEMAN

COMPUTER SCIENCE PRESS

ESSENTIALS OF COBOL PROGRAMMING:

A Structured Approach

PREFACE

This text begins with structured programming concepts and evolves through the introductory concepts of COBOL clearly and concisely. More advanced topics include tape and disk processing, sorting techniques, and report writing capabilities as applied to the COBOL language. Also contained in the volume are examples of each of the general instruction formats as well as specific examples of their use. Detailed examples of MOVE statements demonstrating editing capabilities are presented. The appendices include error correction (or debugging) concepts, a reserved word list, a summary of general instruction formats for easy reference, and a complete COBOL program which uses the concept of tables and a detailed explanation of the program's design and how execution of the program would proceed.

The text is written for a first course in COBOL programming but contains sufficient depth and emphasis on the relationships among the user's program environment, files accessed, the compiler itself, and the system to allow a more in-depth coverage, if desired.

ACKNOWLEDGEMENT

This book is dedicated to the Computer Science Department at Texas A&M University and Dr. Dan Drew who has shaped that program from its inception.

TABLE OF CONTENTS

Chapter I

PROBLEM ANALYSIS AND STRUCTURED PROGRAMMING

Problem solving is a task that is basic to many everyday situations. Generally, people are not fully aware of what thought processes enter into the solution of their particular problems. A problem presents itself and the solution, if determined, is implemented. Humans are able to solve problems in a relatively informal manner. Computers, due to their limited capabilities, usually cannot solve problems in this fashion. The greatest success in computerized problem solution is attained through strict, rigid expression of the problem and of its solution.

There are various techniques that one can use in formally defining a problem situation. Most theorists agree that one should attempt to put the problem definition in writing. This formalizes the individual aspects of the problem. The written definition can be subdivided many times into lesser problems until the entire problem has been broken down into relatively elementary operations. The problem and all of its subdivisions are then analyzed in order to determine what information or action each segment needs in order to perform its function properly, and what information or action the step produces so that succeeding steps can perform their functions properly or for the achievement of the solution to the problem. Completion of the subdivision and analysis steps is followed by translation of the problem into computer language and by subsequent testing of the solution to insure that it is truly correct.

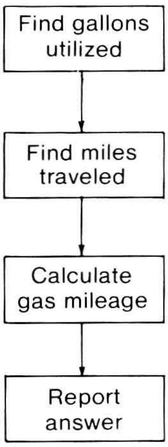
Problem Analysis

The most commonly used method for expressing the analyses of the problem is first to develop a word description of the problem, second to list the major points of the problem, and third, to make a flow-diagram of the problem solution. A flow-diagram of the solution process is nothing more than a step by step explanation of the solution. The main purpose of this method is to define completely the problem and its solution so that every facet of these two items is completely understood. Consequently, you should employ whatever method or combination of methods is easiest to use and promotes the best comprehension of the problem.

Consider the application of the problem analysis discussed above to the following problem. Suppose that one wanted to know the gas mileage of an automobile. The previous sentence becomes the word description or definition of the problem. Subdividing the problem statement into subproblems could be illustrated as follows: 1) How is gas mileage calculated; 2) How many gallons does the gas tank hold; 3) How does one calculate the miles driven?

By analyzing the three subproblems, assuming the gas tank is filled prior to starting, one realizes that required information is the number of gallons used and the miles traveled.

A flow-diagram of the solution to the problem could be:



. . Subtract gallons put into tank after trip from tank capacity

. . Subtract initial odometer reading from final reading (after trip)

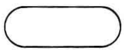
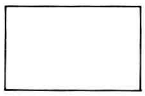
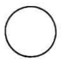
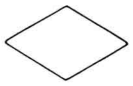
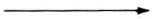
. . Gas mileage $= \dfrac{\text{miles traveled}}{\text{gallons used}}$

Figure 1.1

This flow-diagram is opposite from the problem subdivision shown previously because the first subproblem must be solved before the second and third subproblems become useful.

In order to understand the flow-diagram (flowchart) form, one should become familiar with the following table of basic symbols used in flow-charting:

Table 1.1: Symbols of the Flowchart Language

Symbol	*Meaning*
	Start or Stop box, initial or terminating position of solution, has only an entry or an exit.
	Process box, any action or operation in the solution other than a decision, has one entry and one exit.
	Connector box, all flow lines with symbol "*n*" are connected together, has one or more entries and an exit.
	Decision box, a decision-making operation within the problem solution, has one entry and two exits. It is called a decision box because the computer must decide whether a statement is true or false, as indicated by the two possible exits.
	Flow line, denotes direction of flow for the process, connects all symbols of the language together in the order the various functions are to be performed.

If we were to formalize the gas mileage flow-diagram given previously into a flowchart, only two of the symbols in Table 1 would be used (the terminal and process symbols).

To use the computer to solve a problem, the flowchart must be transformed into instructions to the computer. These instructions must be written in a language that the computer can understand. It is the primary purpose of this book to teach the essentials of one of the languages that a computer understands. However, the process of problem solving must be evolved by readers in their own ways before successful application of this language can be accomplished. It is hoped that the concepts explained previously will provide a basis for this individual evolution.

Structured Programming

Computer solutions to problems can take many forms. It has been found that programmers who use a structured form of program logic tend to write programs with fewer bugs (or errors) than those who do not. Programs written in

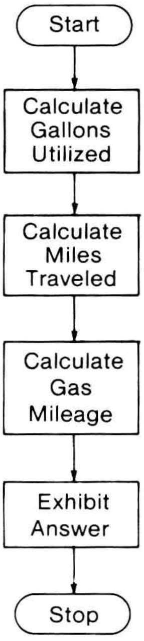


Figure 1.2

a structured form are usually easier to "debug" once the bug is found. The logic forms used are generally one of three types, a sequential form, a simple loop, or a simple decision. These forms, in flowchart language, are shown below. A problem, along with illustrated steps toward its solution, is shown to enable you to follow the logical progression of each form.

The one characteristic of these three forms is that each has only one entry point and one exit point. All problems generally start as a sequential series of processes or one of the other two simple logic forms (simple loop, simple decision). Since the process box generally has only one entry point and one exit point, it is possible to substitute any combination of the three basic forms for a process box and still maintain the exact same logical pattern. This characteristic provides well-structured programs that allow easy modifications and program maintenance. A well-structured program often performs its assigned task with less overall expended effort because there is a higher probability of its being bug-free.

An example of this technique can be seen by constructing a set of instructions for counting by ones from one to any given integer value, *n*. When

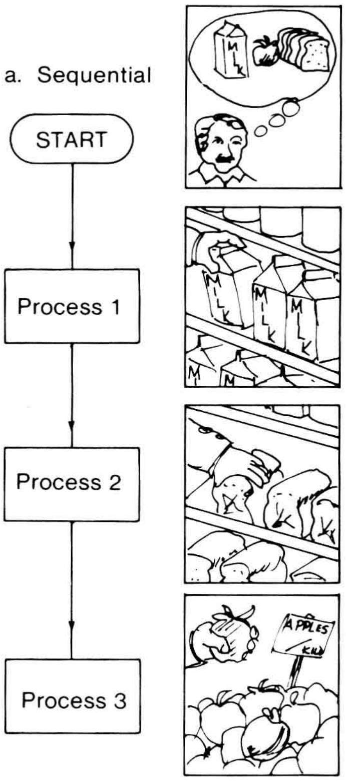


Figure 1.3a

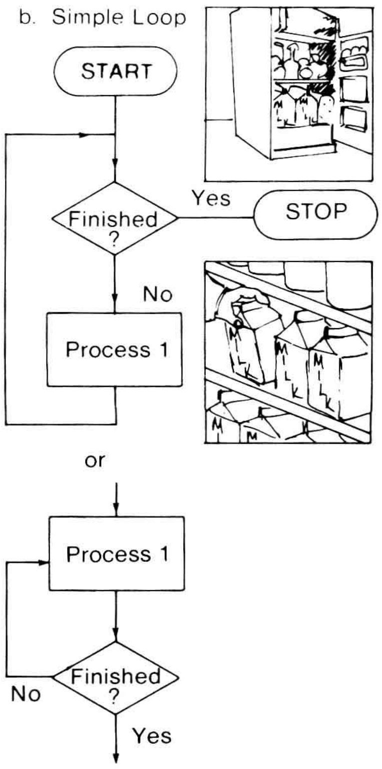


Figure 1.3b

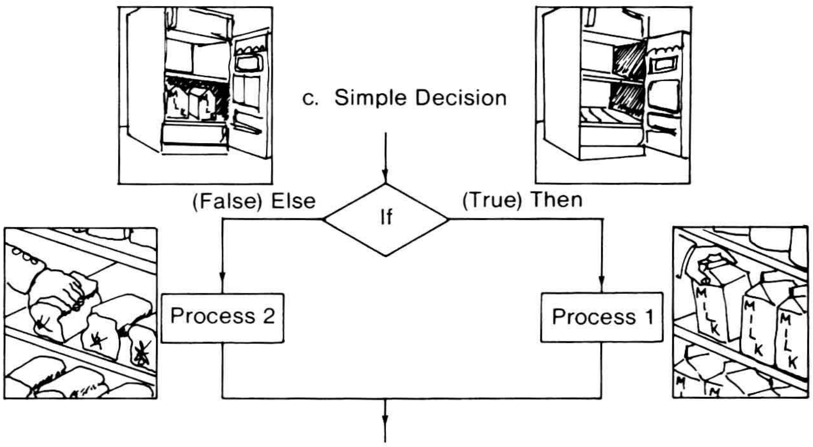


Figure 1.3c

creating an instruction set for something, one must consider the ability of the entity which will follow the instructions. For most people, one could simply say, "count to ten for me," and the person would. However, if the person did not know how to count, then the instructions would be in a considerably different form.

For this individual, it is necessary to define how to perform the action of counting when defining the problem. The problem definition could be stated as follows. Given a number *n*, start at 1 and add 1 successively until the number, *n*, has been reached. This definition still assumes that the user of the definition understands the elementary ideas of number 1, and addition. In some cases, still further redefinition would be necessary. The flowchart language version of the definition/solution to the situation is:

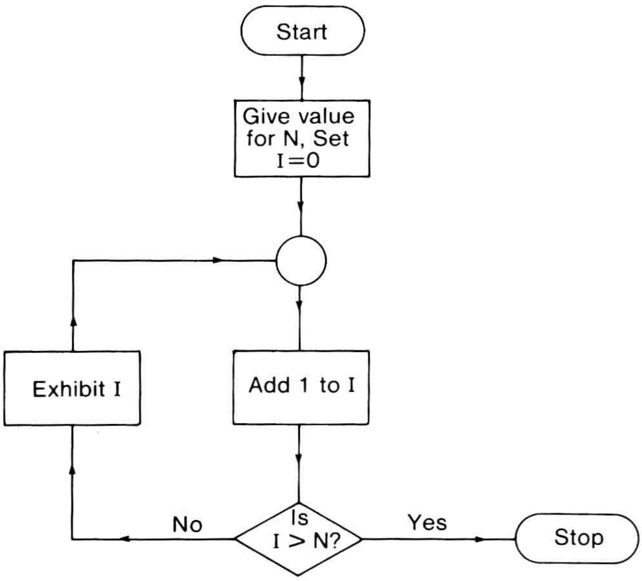


Figure 1.4

Testing the flowchart solution with $n = 10$, $n = 1.2$ and $n = 55$ shows that the solution seems to work, even when given non-integer values for *n*. Further testing with $n = -10$ or $n = -1.2$ illustrates a situation that does not do anything. The problem accepts a number and then stops, without performing the counting procedure. This bug (error) indicates a problem in the original definitions of the solution. (Sometimes bugs are caused by incorrect implementation of the definition).

A better flowchart definition would be:

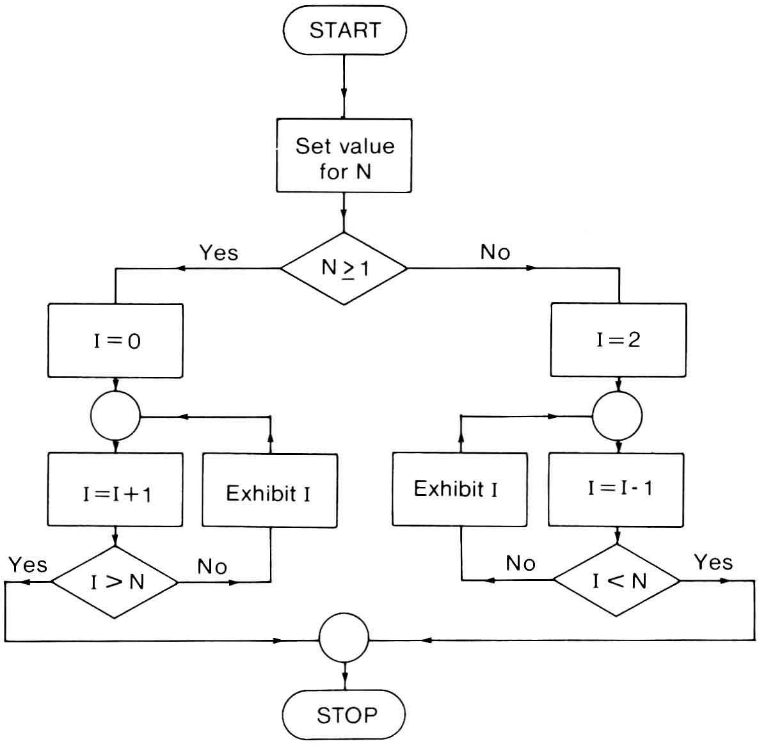


Figure 1.5

Testing this new definition with the previous values illustrates that it works for all of those values. However, it is not generally safe to state that a problem solution is bug-free because it works for the test values. Perhaps the problem has not been tested properly.

Analyzing the better definition for its structure shows the following forms:

Step 1.

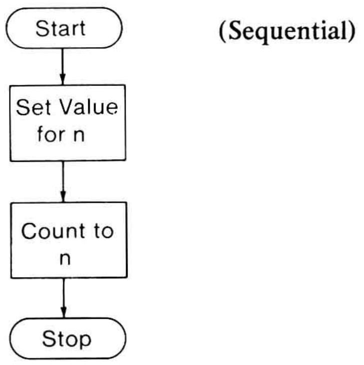


Figure 1.6

Step 2. Count to *n* requires further subdivision:

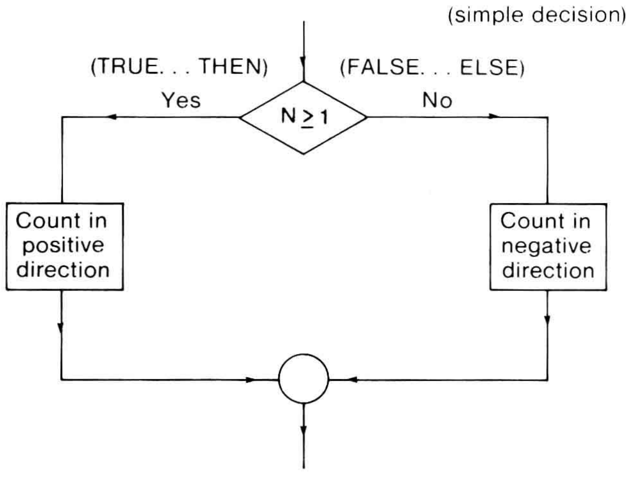


Figure 1.7