Davis, Olson, and Litecky

ELEMENTARY STRUCTURED COBOL

A Step by Step Approach

ELEMENTARY STRUCTURED COBOL

A Step by Step Approach

Gordon B. Davis
Professor, Management Information Systems
University of Minnesota

Margrethe H. Olson
Assistant Professor, Computer Applications and Information Systems
New York University

Charles R. Litecky
Associate Professor, Accounting
University of Missouri, Columbia

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland Bogotá Düsseldorf
Johannesburg London Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

ELEMENTARY STRUCTURED COBOL
A Step by Step Approach

1011121314 SMSM 8987654

This book was set in Plantin by York Graphic Services, Inc. The editors were Peter D. Nalle, Matthew Cahill, and Theresa J. Kwiatkowski; the cover was designed by Joseph Gillians; the production supervisor was Robert C. Pedersen. The drawings were done by J & R Services, Inc.
Semline, Inc., was printer and binder.

Library of Congress Cataloging in Publication Data

Davis, Gordon Bitter.
Elementary structured COBOL.

Originally published in 1971 under title: Elementary COBOL programming.

1. COBOL (*Computer program language*) I. Olson, Margrethe H., joint author. II. Litecky, Charles R., joint author. III. Title.
QA76.73.C25D38 1977 001.6'424 76-57721
ISBN 0-07-015782-0

PREFACE

This text was written in response to several needs that we and others have experienced in teaching the COBOL language to college students. We felt the need for a COBOL text that would:

1 Adhere to the new 1974 COBOL standard
2 Present COBOL in the context of structured programming practice
3 Apply to both self-instructional and regular lecture-method instruction
4 Use student programs both for motivation and for teaching good programming practice
5 Provide an introduction to COBOL for students not desiring COBOL proficiency or depth of COBOL knowledge while providing a solid foundation for those intending to be engaged in programming in COBOL

We have found no existing COBOL texts which meet these criteria, and therefore this text was written. It has been class-tested with students having a variety of backgrounds—business school undergraduates, liberal arts undergraduates, graduate students, evening students in nondegree courses, and high school students. Some of the students have had previous programming courses in FORTRAN or BASIC; others have had no background. The response from this diverse group of students to the approach and content of the text has been most encouraging.

The text adheres to the 1974 COBOL standard. Most of the newer COBOL compilers will be 1974 standard. Where there is an older compiler, the text can still be used. An explanation of differences is provided in Appendix C. This appendix will allow the student to write the programs in the text in 1968 standard COBOL.

Programming that is done in a structured, disciplined manner is more productive and results in programs that have fewer errors, are simpler to debug, and are more easily maintained than programs written without this approach. This text does not simply teach the rules for COBOL; it teaches the student by explanation and by example how to apply the rules of the COBOL language to write clear, structured programs.

Many students are able to learn a programming language with little assistance; others require lectures to reinforce the language text. In recognition of the need for both instructional approaches, this text was written so that it may be used in a self-instructional mode without lectures. This also makes it suitable for in-company training. When the lecture method is

preferred, the text is still well-suited because the organization of each topic chapter into an A and B section facilitates lecture presentation. Section A presents language features which can be amplified and illustrated in an accompanying lecture. Section B contains a sample program and the programming exercises, one of which will normally be assigned. An instructor may wish to use the sample program as the subject of a lecture, adding to the explanations on programming.

A frequent complaint about COBOL books has been that the texts are designed so that a student does not write programs until late in the study of the language. The result is a lack of the motivation and learning that would come from writing, debugging, and executing programs. This text has the student code and execute a simple but complete COBOL program in connection with each chapter. The programs are used to help students clarify and solidify their understanding of the chapter material. In Chapters 2 to 4, there are three problems at the end of each chapter; the first two are similar in difficulty and follow the pattern of the example program. The third problem is slightly more difficult. In Chapters 5 to 7, there are more problems and the range of difficulty is defined in the problem section.

This COBOL text may be used alone as the major text in the study of COBOL, or it may be used as the language text in a data processing course in which another, general text is used to present basic material on computer hardware, software, and systems. Suggested companion texts for the latter use are Davis, *Computer Data Processing* (New York: McGraw-Hill, 1973) or Davis, *Introduction to Computers in Information Processing*, to be published by McGraw-Hill in 1978.

COBOL courses (and the students taking these courses) have differing learning objectives. These range from "getting a feel for this important language" to learning to be a COBOL programmer. The text is suitable for a wide range of objectives by selective use of part or all of the material. An objective of introducing students to the elements of the COBOL language and the structure of COBOL programs can be achieved by Chapters 1 to 4. Chapter 5 should be included if students need to understand the logic of sequential file updating. Students who need to understand the entire range of capabilities for the language should use all chapters.

A large number of students have provided feedback that has been incorporated in the text. Graduate teaching assistants have provided ideas and pointed out sections in need of improvement. Nancy Kelly Bostrom worked on most of the programs. Alison Davis and Stanley Lau assisted in reviews and in writing sample solutions for the *Instructor's Guide*. Janice DeGross did an outstanding job of typing the manuscript.

We appreciate receiving suggestions for corrections or changes and ideas for additional programming exercises from readers. Comments may be sent to Gordon B. Davis, College of Business Administration, University of Minnesota, Minneapolis, Minnesota 55455.

Gordon B. Davis
Margrethe H. Olson
Charles R. Litecky

CONTENTS

PART 1

FUNDAMENTAL FEATURES OF COBOL

The first four chapters (Part 1) present the basic structure of a COBOL program and explain the instructions for writing simple COBOL programs for processing data and printing reports. Part 1 provides a good introduction to the nature of programming in COBOL and a foundation for the additional features explained in Part 2. The topics covered are:

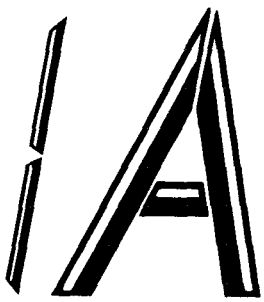
Chapter	Topic
1A	Introduction to programming, programming discipline, and the COBOL language
2A	Instructions for coding a simple program to read, move, and print data
3A	Instructions for doing arithmetic calculations and writing a report
4A	Additional features for input and output, IF . . . ELSE coding, and the COMPUTE verb

The text helps a student to learn COBOL by explaining how to write programs using specified COBOL instructions. The general explanation contained in section A of each chapter is followed by a sample program in section B. The four sample programs for Chapters 1 to 4 illustrate the COBOL features covered in Part 1:

Chapter	Sample program
1B	Card-lister program
2B	Bibliography-list program
3B	Improved bibliography-report program
4B	Book-order-list program

Upon completion of Part 1, the student will be able to write complete COBOL programs in good form for problems involving reading from a file, performing input-data validation, performing processing on the data, and printing well-formatted output for both regular reports and error reports.

CHAPTER



INTRODUCTION TO PROGRAMMING, PROGRAMMING DISCIPLINE, AND THE COBOL LANGUAGE

This chapter introduces the concept of a programming language, programming discipline, the COBOL language, and procedures for submitting a program for compilation and execution. A complete COBOL program is contained in section B of the chapter. Using this example program as a guide, you will be able to code, keypunch, and submit a COBOL program for processing. Section A of Chapter 1 thus provides concepts necessary for understanding the nature of COBOL programming, and section B provides an introductory experience in preparing a COBOL program. The program written as the assignment for section B is also used to provide experience in following the procedures by which a COBOL program is submitted to the computer center and processed.

Instructing a Computer

Before starting the explanation of the COBOL language, it may be helpful to review how a computer is instructed. A computer system requires both hardware and software. The hardware consists of all the equipment; the software includes the programs of instructions which direct the operations of the computer equipment.

Hardware and Software

A hardware computer system (Figure 1-1) has input units (such as card reader), a central processor (CPU), and output units (such as printer). It also has secondary or

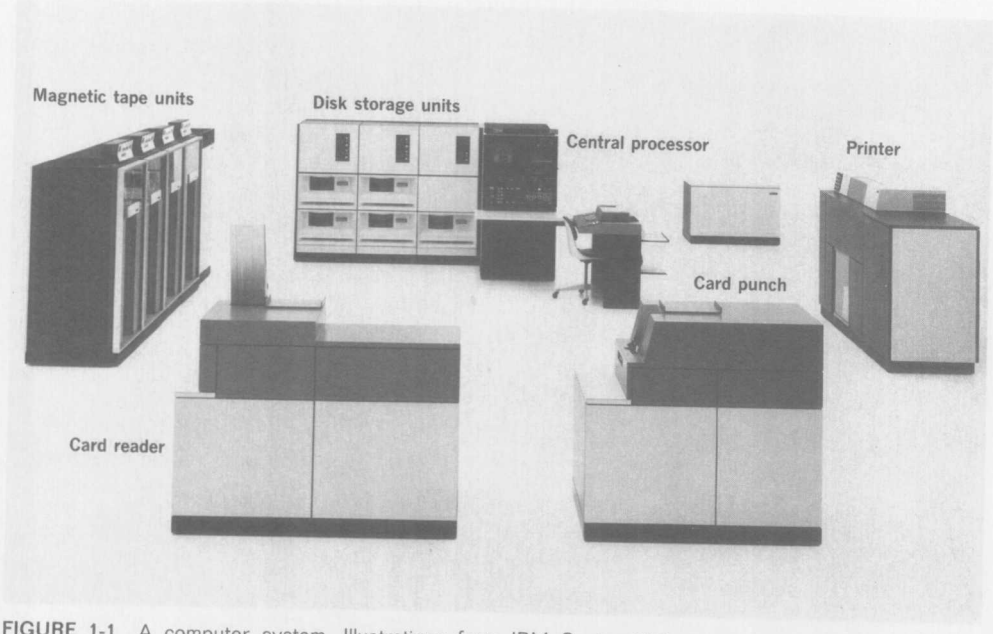


FIGURE 1-1 A computer system. Illustrations from IBM System/370 equipment (*courtesy of International Business Machines Corporation*).

auxiliary storage devices such as magnetic-disk or magnetic-tape units. In a typical processing job, data comes from the input unit (and perhaps from secondary storage) into the central processor, where computation and other processing are performed. After processing, the results are sent to the output device (say a printer) or to a secondary storage device to be held for later output or further processing.

The hardware can perform operations such as read, write, add, etc., but the sequence in which these operations are to be performed and the exact input and output units to be used are specified by instructions stored in the computer memory. The general term applied to these computer processing instructions is *software.* The instructions are organized into sets called *routines* and *programs.* A routine refers to a set of instructions which directs the performance of a specific processing task, such as calculating the square root of a quantity or producing an error message when an error is encountered in input data. A program consists of one or more routines which direct a complete processing job.

There are several major types of software. Three types especially relevant to the study of COBOL are application programs, operating systems, and compilers.

1 *Application programs* Programs which direct processing of an application such as preparation of a report, preparation of payroll checks, etc. The COBOL programs in this text are application programs.

2 *Operating system* A set of routines which directs and manages the operations of the computer. The operating system supports the running of application programs. For example, if an application program has an instruction to read a card from the card reader but the card reader is not operable, the operating system sends a suitable message to the computer operator.

3 *Compilers* Compilers are programs which prepare machine-language programs from programs written in a language such as COBOL.

Application programs may be written by programmers in the organization needing them or purchased from vendors who have prepared programs for specific applications. Operating systems and compilers are generally obtained from the hardware vendor, but independent software vendors are also a source. Programs which are being executed are stored in the main (primary) storage or memory, which is part of the central processing unit (CPU). Programs not currently being executed which need to be available are stored in secondary storage. Some parts of the operating system remain in main storage all the time. According to job-control instructions to be described later, these operating-system routines bring into main memory the routines and programs to be executed and direct their execution. The programs to be run may be compilers, application programs, or other software.

Machine-Level Languages

A program in main memory must be in machine language to be executed. A machine-language instruction is represented as a string of binary digits (bits), which identify the operations to be performed and the data, etc., to be used. For example,

one instruction for the IBM System/370 has the following form (with 1 standing for a 1 bit and 0 for a 0 bit in storage):

01011010001100001011101001000000

Even though the internal machine representation is in this form, it would be very difficult and could lead to error if a programmer were required to deal with such instructions. When machine instructions are printed for operator or programmer use, a condensed notation is employed. For example, the preceding instruction would be printed out for operator or programmer inspection as

5A 30BA40

Although easier than the machine representation, this notation is still difficult to use. Therefore, if a program is to be written in machine-level instructions, the programmer generally uses a symbolic assembly language. Machine-oriented, symbolic assembly languages as a class are often referred to as *low-level languages.* The preceding instruction coded in symbolic assembly language might be A 3,PAY where, for example, A means "add." Since the computer cannot execute the symbolic instructions directly, they must be translated into machine-language instructions. This is done by a program called a *symbolic assembly system,* which converts each symbolic instruction into an equivalent machine-level code instruction. Machine-oriented programming is very useful for some applications because coding can be very machine-sensitive, thus giving very efficient use of the computer. However, an assembly-language program is relatively difficult to code, and logic errors are difficult to find. It is also difficult and time-consuming to change it. A program in a low-level, machine-oriented language also has limited transferability (portability) from one computer to another.

High-Level Languages

A high-level language is oriented to problem solution or processing procedures rather than to the machine-level instructions of a particular computer. There are a number of different high-level languages for different types of problems. Each of these languages consists of a grammar (set of rules) and predefined words for writing instructions. A compiler is used to translate the program written in the high-level language (the source program) into machine-level instructions for the computer on which the program is to be run (the object program). Since the compiler is a computer program, there must be a unique compiler for each computer for which a high-level language program is to be translated. In contrast to symbolic assembly language, in which one symbolic instruction is converted into one machine instruction, a high-level language instruction statement is generally translated into a number of machine-language instructions.

There are two important advantages of high-level languages over symbolic assembly languages: they are machine-independent in the sense that they can be compiled and run on any computer (for which there is a compiler) with little or no change, and they are relatively easy to learn. Today, these languages are generally so

powerful and efficient that they have virtually eliminated the need for symbolic-assembly-language coding except for a few specialized applications. It is also relatively easy to standardize methods of coding with high-level languages. Organizations having a concern for program accuracy and a desire for programming discipline have strongly influenced the trend toward use of high-level languages.

The two most common procedure-oriented high-level languages are FORTRAN and COBOL. FORTRAN is best suited for formula-type mathematical problems, while COBOL is by far the dominant language in use today in business data processing. It is estimated that more than 70 percent of all programs for business use are written in COBOL.

The COBOL Language

COBOL (an acronym for *co*mmon *b*usiness-*o*riented *l*anguage) is a high-level procedure-oriented language designed for coding organizational data processing applications. These types of applications are characterized by the use of large files, a high volume of input and output, and production of reports requiring editing and formatting of output data. COBOL is an English-like language; its vocabulary and grammar are based upon the clause, sentence, paragraph, and word order of the English language. COBOL is nonmathematical, in contrast to other programming languages, like FORTRAN, which are based upon algebraic, symbolic, or algorithmic notations. One can write the procedures for mathematical computation in COBOL, but other more convenient languages are for this purpose.

History of COBOL

Work on a common source language suitable to commercial (as opposed to scientific) data processing began in 1959, when several large business organizations, the federal government, computer manufacturers, and other interested parties formed a committee to develop the language. The committee (named CODASYL for *C*onference *o*n *Da*ta *Sy*stems *L*anguages) developed the specifications for a language called COBOL (see acknowledgment in Appendix A). Their report, issued in April 1960, contained the first version of COBOL, termed COBOL-60. CODASYL formed a maintenance committee to initiate and review recommended changes to keep COBOL up to date. Subsequent revisions were published in 1961 (COBOL-61), 1963 (COBOL-61 Extended), and 1965 (COBOL, Edition 1965). Changes subsequent to the 1965 version have been published in reports titled *The COBOL Journal of Development.*

Although COBOL was developed and is maintained by CODASYL, it has also been established as a standard language by the American National Standards Institute (ANSI).[1] The suppliers of COBOL compilers generally base them on the American National Standard COBOL. An initial standard, issued in 1968, was revised in 1974. The American National Standard COBOL recognizes different levels (of complexity) of COBOL implementation and provides standards for each. The idea is that a small computer may not have memory enough to be able to implement all the COBOL

[1] USA Standard COBOL, X3.23-1974, American National Standards Institute, New York, 1974.

features, but it can implement the more fundamental ones. The basic COBOL language elements are termed the *nucleus* of the language. The nucleus is divided into nucleus, level 1 (low level) and nucleus, level 2 (high level), with level 2 including all elements of the COBOL nucleus and level 1 being a restricted version with some features not required. The same division into levels of implementation is followed for other groupings of COBOL features covering specific processing techniques. These groupings (called *functional modules*) will be explained in Chapter 8.

The COBOL chapters in this text are based on the 1974 ANS COBOL standard, the newest standard version in use at the writing of this text. If the compiler used by a student adheres to the older 1968 ANS standard and has not been updated for the 1974 standard, a few language features described in the text will not be accepted. The text can still be used by making adjustments for the differences. Appendix C describes the differences.

The problems in Part 1 of this text generally use only features contained in the ANS COBOL standard low-level nucleus (level 1). Exceptions are noted in the text and listed in the summary at the end of the chapter.

An Evaluation of COBOL

Why do organizations adopt COBOL as the major data processing language? The reasons suggest why a student should study COBOL in preference to other data processing languages. COBOL has a number of advantages and, like all languages, some disadvantages. Some advantages are:

1 *Wide use* Since COBOL is the major data processing language, many programmers know it, it is well supported by training materials, etc. It has broad support by industry and government users, and all computer manufacturers provide COBOL compilers.

2 *Documentation support* COBOL contains excellent features for self-documentation of programs. English-like sentences make the program fairly understandable.

3 *Portability* COBOL is sufficiently machine-independent to cross computer system lines. Programs compiled and run on the machine of one manufacturer can, with relatively minor modifications, be compiled and run on a computer of another manufacturer.

4 *Standardization* There is an accepted standard COBOL which is supported by the American National Standards Institute.

5 *Growth in response to new requirements* There are regular provisions for adding features to COBOL.

6 *Programming discipline support* COBOL supports the adoption and enforcement of programming standards for effective programming management. COBOL has a block structure suited to the application of principles of modularity, which are important in programming discipline.

7 *Data-handling capabilities* COBOL offers strong capabilities in the areas of data editing and file management.

8 *Program modifiability* The structure of a COBOL program which defines data in a separate part of the program means that the program can easily be modified to accommodate changes in data being processed.

Given these advantages and wide acceptance by organizations, why isn't COBOL taught more widely in the college classroom? Some disadvantages of COBOL provide possible answers to this question.

1 *Verbosity* Due to its self-documenting features, COBOL is wordy. With this language the student must spend more time coding and keypunching programs than with algebraic languages.

2 *Limited scope* COBOL is not a universal language for all kinds of processing. It lacks extensive scientific and mathematical facilities often needed for student course problems. Such features are provided in algebraic languages such as FORTRAN.

3 *For professional programmers* COBOL was designed for the professional programmer. It is therefore difficult to use COBOL without knowing a considerable part of the language.

Of the disadvantages of COBOL, the need to learn or know a great deal of the language in order to write and run simple programs has most significantly limited the use of COBOL in the classroom. The approach used in this text overcomes these disadvantages of size and complexity by presenting first only enough features to write a simple program and by then adding features with each chapter. The important, frequently used features are explained in detail; little used features are surveyed.

Objectives of a Disciplined Approach to Programming

Computer programs frequently do not meet user requirements, are not produced on time, cost considerably more than estimated, contain errors, and are difficult to maintain (to correct or change to meet new requirements). These difficulties have been observed with such frequency that many organizations have attempted to change the practice of programming to improve performance. The revised approach can be termed *programming discipline,* that is, well-defined practices, procedures, and development control processes.

It is apparent from its success in industry that a disciplined approach to program design and development will lead to improved programming performance. The problems in this book have been designed using this approach. The technique of structured programming, one important aspect of programming discipline, will be the basis on which the student is taught to program in COBOL. Because of this approach, the student will be introduced to the elements of the COBOL language in a teaching procedure that is significantly different from that of traditional COBOL texts. The idea is that the student should learn the elements of programming discipline and structured programming while learning the language. The disciplined, structured approach is useful both for those who wish only an introduction to COBOL and for

those who expect to become COBOL programmers. It is not sufficient to merely learn coding rules for COBOL; it is equally important to learn how these features are combined into a high-quality program.

Because programming discipline is an underlying principle for this text, and because programming discipline is important to industry, it will be useful to summarize the major objectives of a disciplined, structured approach to programming. These objectives are:

1 Meeting user needs
2 Development on time within budget
3 Error-free set of instructions
4 Error-resistant operation
5 Maintainable code
6 Portable code

Meeting User Needs

A program has a purpose—to produce a report, prepare a sales invoice, or compute a set of statistics. An assignment to prepare a program is a failure if the program is not used because the potential users of the application find it too complex or too difficult. A disciplined approach to program design includes a careful analysis of user needs and user involvement in key decisions affecting application usability.

Development on Time within Budget

Estimates of time and cost for writing computer programs have frequently been substantially in error. Programmer productivity has generally been lower than expected. Three requirements of a structured, disciplined approach to programming for meeting this on-time objective are:

1 More accurate estimates
2 Greater programmer productivity
3 Closer control of the actual time and cost required

Industry average productivity (in 1975) was about 8 lines of tested code per day for a programmer using a high-level language such as COBOL (about 2000 lines per year). Using a more structured, disciplined approach, installations have achieved dramatic improvements in productivity. For example, one installation boosted productivity to 5000 lines per year, and another expects to achieve an average of 10,000 lines of tested code per programmer per year.

Error-free Set of Instructions

It is generally conceded that all large-scale computer programs contain errors, and it may be impossible to remove every single error from a large set of programs. However, when a disciplined, structured approach is used, programs can be designed