Methodical Programming in COBOL

Ray Welland

Methodical Programming in COBOL

Ray Welland

Department of Computer Science
University of Strathclyde

Pitman

PITMAN BOOKS LIMITED
128 Long Acre, London WC2E 9AN

PITMAN PUBLISHING INC.
1020 Plain Street, Marshfield, Massachusetts

Associated Companies
Pitman Publishing Pty Ltd, Melbourne
Pitman Publishing New Zealand Ltd, Wellington
Copp Clark Pitman, Toronto

First published in Great Britain 1983

British Library Cataloguing in Publication Data

Welland, Ray
Methodical programming in COBOL.
1. COBOL (Computer program language)
I. Title
001.64'24 QA76.73.C25

ISBN 0-273-01820-5

Printed and bound in Great Britain by
Biddles Ltd, Guildford, Surrey

Preface

At the time of writing, the most widely used commercial programming language is COBOL which has been in existence for more than 20 years. Similarly the most widely used scientific programming language is FORTRAN, first formulated in 1957. It has often been claimed that better languages exist for both fields but the difficulty with introducing a new language is the enormous investment in existing software written in both COBOL and FORTRAN. Therefore it seems likely that both these languages will remain with us for a long time yet.

The objective of this book is to try to combine the modern ideas of structured programming with the existing COBOL language to show how good programs can be constructed.

The main targets of this book are students of computing or data processing who may or may not have had previous programming experience. The ideas have been developed while teaching COBOL as a first programming language to accountancy students and teaching COBOL to computing science students already conversant with Pascal.

The approach used in this book is to separate the tasks of program design and coding. Programs are designed using an abstract program design language which can then be translated into COBOL using a defined set of transformations. One attraction of this approach is that students can learn to program without getting overwhelmed by the mass of detail involved in even a simple COBOL program. It would also be possible to design a different set of transformations from the design language into another programming language.

Program designs can be checked by the lecturer, or demonstrator, and discussed before coding commences (and also marked if required). This means that many logical errors can be trapped before the program is actually coded in COBOL.

Use of the book

This book is structured so that the student should be able to design quite substantial programs by the end of Chapter 2. This will enable useful practical work to be undertaken while the fundamental elements of COBOL are explained in Chapters 3 and 4. By the end of Chapter 4 the student should be able to write and run complete, simple COBOL programs.

The next two chapters introduce more features of the language concerned with the handling of input and output, and the expression of more complex algorithms. At the end of Chapter 6 the student has been introduced to all the language elements required to write quite complex COBOL programs. Chapters 7 and 8 are designed to give a 'breathing space' while the student consolidates his understanding by practical experience of COBOL programming.

From Chapter 9 onwards more features of the language are introduced and the student's repertoire of programming techniques is steadily increased.

The Appendices are designed to give summaries of various features of

COBOL for quick reference when writing programs.

Exercises

Programming is essentially a practical subject and to take full advantage of this book the student must actually design and run some programs. Programming is like swimming - one can read all the books available but until one "takes the plunge" one doesn't start really learning.

The exercises are divided into two categories: pencil and paper exercises, to follow up points made in the text, and programming exercises which involve the design, writing and running of complete computer programs. It is recommended that the student should do all of the pencil and paper exercises, and at least one of the programming exercises at the end of each chapter.

Hints on the solutions to the pencil and paper exercises are given at the end of the book.

The COBOL Language

The language used in this book is that defined in the 1974 American National Standard for COBOL, usually called ANS COBOL '74. The definition of ANS COBOL '74 is modular and within each module there are levels of implementation. Therefore what is legitimately described as an ANS COBOL '74 compiler may not have all the features introduced in this book. To assist readers using low-level ANS COBOL compilers, notes are inserted in the text where problems may arise, and Appendix 5 gives some hints on alternative constructs for methodical programming.

Acknowledgements

I would like to thank Charles Clarke and Roy McAllister of the Management Services Division, Strathclyde University, for their help in using the local system to test the COBOL programs in this book. I would also like to acknowledge the strong influence of two ex-colleagues: Bill Findlay and David Watt, of Glasgow University, who taught me much about structured programming.

This book also owes a great deal to another ex-colleague John Jeacocke, of Glasgow University, who encouraged me to use the style of teaching advocated in this book, helped to formulate the design language and made many helpful criticisms of various drafts of the book. Finally, I would like to thank the consulting editor, David Hatter, for his constructive criticisms, and Alfred Waller of Pitmans for his support throughout the writing of this book.

Ray Welland
University of Strathclyde

January 1983

Contents

1 Introduction

This chapter includes a very brief history of COBOL and a survey of some of the 'computer basics' which are necessary for this book. However, it is not intended that this book should be read in isolation. Pressure on space dictates that it cannot cover fundamental concepts of computing in any depth while doing justice to its main theme of methodical programming in COBOL. Therefore it will be assumed that the readers are familiar with the basic concepts of computing through attendance at formal classes, from background reading or from using their own home computer.

1.1 History and Objectives of COBOL

In the late 1950s it was recognized that the existing programming languages were inadequate for building large commercial data processing systems. Most programming at that time was being carried out in assembly languages which meant that programs, and skilled staff, were tied to one particular type of computer. Therefore the American Department of Defense, one of the biggest users of computers for inventory control and accounting, pulled together a group of experts with the objective of creating a new high-level programming language suitable for commercial applications of computing. The objectives which this group, called the CODASYL Committee, were given for the design of the language were as follows.

(a) To specify a language independent of any make or model of computer, open-ended and stated in both an English notation and a narrative form.
(b) The language should be extensible so that it could be run on future ranges of machines.
(c) It should be easy to learn so that relatively inexperienced programmers could make a significant contribution.
(d) Programs written in the language should be "self-documenting" and also "readable" by managers and non-technical people.

The committee proposed the language now known as COBOL (COmmon Business Oriented Language) which has been revised regularly since its inception in 1960. The language is reviewed regularly and new proposals published in the COBOL Journal of Development. Various manufacturers have also added their own enhancements to the languages giving rise to a wide range of 'dialects' of the language. In order to maintain the portability of the language, one of its most important advantages, an American National Standard for COBOL was published in 1968 and a new standard in 1974. This book is based on the 1974 standard, commonly called ANS COBOL 74, as this is currently the most widely used version of COBOL. It is expected that a new standard will be agreed in 1983 but this will take some time to become widely implemented and accepted.

Cobol is a very 'rich' and powerful programming language, it has a wide variety of features designed to handle the most complex problems in commercial data processing. A danger for the student is being overwhelmed by this plethora of facilities. This book uses only a minimal useful subset of COBOL initially so that the trainee programmer can grasp the basic principles before learning to use the more advanced features of the language.

1.2 The COBOL Computer

When designing a programming language it is necessary to have a conceptual model of the computer to be used, which is called the abstract machine. For the purposes of this book we shall assume that the model shown in Figure 1.1 is the first approximation to the COBOL abstract machine.

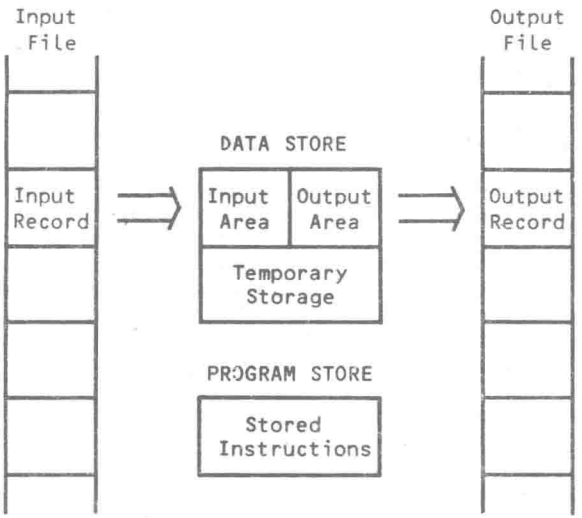

Figure 1.1

The input and output files are divided into records. Each record in a file is in a similar format and usually contains data about one of a group of related 'things'. For example, a file might contain details about a company's employees; each employee would have a record in the file and each record will contain similar data: works number, name, address, national insurance number, salary etc.

A file is accessed in such a way that only one record from a file is available to the program at any given time, the current record. Initially it will be assumed that there is one input file and one output file. The current record of the input file is held in the input area and records are written to the output file via an output area. In a program which does anything useful there will need to be internal storage for intermediate values in calculations, for example. Values held in such temporary storage exist only for the lifetime of a program and if a permanent copy of the values is required they must be sent to

the output file. Therefore in the simple model of the COBOL computer the data store is divided into three parts: the input area, output area and temporary storage.

Each of the areas of the data store is divided into a hierarchy of units of storage called records, group-items and elementary-items. At the lowest level, the elementary-item is a named location which can hold a value which can either be a number or a string of characters. Data processing is built around using these stored values in arithmetic operations, transferring them from location to location and making simple decisions based on the values stored in specific locations.

The simple machine will need a number of basic operations which can then be combined to create programs. A program is a sequence of instructions which, in the absence of instructions which change the 'flow of control', will be executed in the order they appear in the program store. Obviously there will need to be an operation to get a record from the input file, making it available in the input area, and a complementary operation to put a record, which has been built up in the data store, into the output file. There will also need to be a transfer operation to allow data to be copied from one storage location to another and basic arithmetic operations: add, subtract, multiply and divide, operating on the stored data to create new values.

It would be possible to write a simple computer program consisting only of a sequence of basic operations to be carried out one after the other from the first operation to the last. Such a simple computer program would consist of three parts: initialization, processing and termination. In the majority of programs the processing part will be more complex than a simple sequence of instructions but these three basic parts will still exist.

The initialization will define an initial state for the computer, processing will change the state of the machine by transforming data in some way and termination will tidy up before the program finishes. For example, initialization will define the initial status of the input and output files and the initial values in the data store. At any given time during the processing it may be necessary to find out the current state of the machine, e.g., are there any more input records? Termination in a simple machine might consist of releasing the input and output files and printing some message about the final state of the machine.

Most useful programs, particularly in commercial data processing, will involve repetitions of sequences of instructions. For example, going back to the employee file it should be possible to establish a sequence of operations, called an algorithm, for calculating an employee's pay given certain facts such as salary, tax code, superannuation rules etc. This algorithm can then be repeated for each employee's record in the file. Therefore in addition to the basic operations outlined above a special operation to allow controlled repetition of sequences of operations will be required. This implies that a method of recognizing conditions which arise will have to be included in the language. Conditions are formalized questions which can be asked and the result returned will be either true or false, a 'Boolean' value.

Many algorithms will involve the choice of alternative courses of action depending on the value of data. Going back to the employee example there might be two different superannuation schemes available to employees, each requiring a special deduction calculation routine. The type of superannuation scheme might be distinguished by a special

code 'A' or 'B' in the employee record which needs to be recognized by the program and the appropriate routine used. Therefore in addition to repetition a selection operation is required.

Therefore, to summarize, the basic design language used in this book will have the following operations:

(a) record input/output: get and put,
(b) data transfer, within the data store,
(c) calculation, to perform simple arithmetic,
(d) controlled repetition,
(e) selection.

In addition to these operations there will also have to be facilities for the description of the form of data, and condition tests, to support (d) and (e), above.

These features together with one additional concept called refinement, which is purely a program construction technique, will define the subset of COBOL described in the first part of this book. This subset is powerful enough to write quite complex programs.

1.3 The Real Computer: hardware and software

The machine shown in Figure 1.1 and described in the previous section is called an abstract machine because no such machine actually exists. A program written in COBOL, and designed to run on this abstract machine, will have to be translated into a lower level language which can be run on a real machine. This real machine, or hardware, need not concern the programmer since it can be made to obey the rules for the abstract machine.

In running a COBOL program, there are two distinct phases: compilation and execution. Compilation is the process of translating a COBOL program into a suitable low-level language and is carried out by a special program called a compiler, which is part of the software for a particular computer system. Because this translation process is mechanical, the COBOL program needs to be written in a precise form, according to certain syntactic rules. Therefore part of compilation is checking that the syntax of the program is correct and identifying errors. However, the fact that a program is syntactically correct does not mean that it will do anything useful. In the same way sentences in English can be syntactically correct but meaningless: 'The mat sat on the cat', for example. Once a program has been successfully compiled it needs to be executed and tested with a variety of types of input data to prove that it works. The process of debugging a program is concerned with the location and correction of errors shown to exist by testing. The steps in program development are summarized in Figure 1.2.

Students beginning in programming are often discouraged because their programs do not work at the first attempt. Creating a working program is an iterative process; one frequently takes one step forward and two steps back. The first stage in this process is to design the program and 'desk check' the design by stepping through the design manually. The design can then be translated into COBOL, turned into machine readable form using a computer terminal perhaps, and an attempt made to compile this program. This will almost certainly result in syntax errors, detected by the compiler, which will have to be corrected by the programmer.

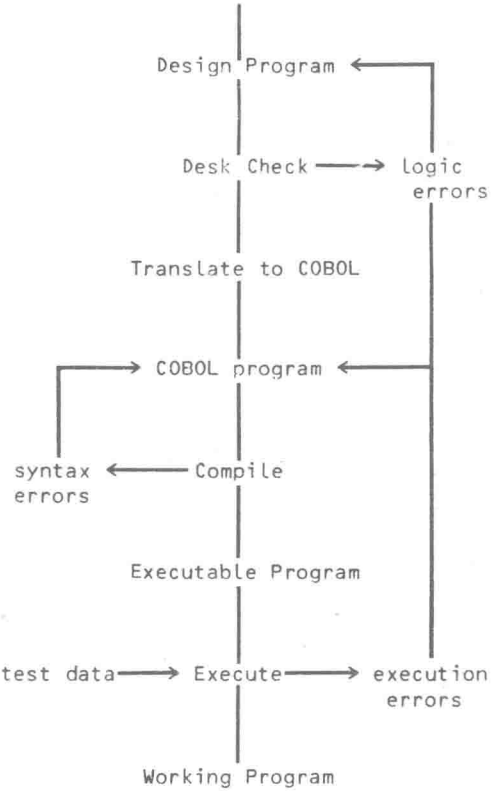

Figure 1.2

Eventually the program will compile, producing an executable program, which can then be executed with an input file of test data. The result of this test will most probably be an execution error. Execution errors can be divided into two categories: run-time errors, where the program stops because some unexpected condition occurs, and logic errors, where the program produces the wrong results. To correct an execution error may involve going back and modifying the COBOL program, possibly modifying the design as well, recompiling the program and then executing it again. Alternatively the execution errors may be caused by incorrect test data which causes unexpected results to be produced by the program. Once the program successfully processes one input file it will then need to be tested with other input files, as a single file of test data is usually insufficient to test all possibilities.

2 The Program Design Language

This chapter introduces the design language which will be used throughout the book for the design of algorithms. Initially the operations of the design language are defined in terms of the 'COBOL computer' described in Section 1.2. These definitions are based on the particular discipline imposed by the use of a specific COBOL program structure and are not definitions of COBOL language constructs. The operations are then used to build up some simple programs which are followed through step by step to illustrate how the operations work.

In the descriptions of the operations anything appearing between the angle brackets '<' and '>' is a thing which the programmer chooses. For example get-next-<filename> indicates that an operation can be defined by substituting the name of a file in place of '<filename>'. Therefore 'get-next-employee' and 'get-next-order-form' could be generated for files called 'employee' and 'order-form', respectively. However, 'get-another-employee' and 'get-employee' could not be generated from the pattern get-next-<filename>.

2.1 Basic Operations

The operations defined in this section are those which carry out a simple task without changing the order of execution of operations.

get-next-<filename>

Attempts to read the next record from the input file, called <filename>, into the input area. If there is no such record then the condition end-of-<filename> is set to true. Initially end-of-<filename> is false and the contents of the input area are undefined.

e.g. get-next-sales
get-next-customer-file

put-<recordname>

Transfers the contents of the area of the data store called <recordname> to the output file, via the output area. After the transfer the area defined by <recordname> is then blanked out. Initially the <recordname> area will be set to all blanks.

e.g. put-details
put-summary-line

transfer <item-1> to <item-2>

Copies the value stored in <item-1> to the location called <item-2>. The contents of <item-1> are unchanged but any existing value in <item-2> is lost. A special case of this operation is:

transfer <constant> to <item-2>

where <constant> is a fixed value which is to be stored in the location <item-2>.

e.g. transfer total-price to summary-tot-price
transfer 25 to discount-rate
transfer "special customer" to details-comment

calculate <result-item> = <arithmetic-expression>

The <arithmetic-expression> is evaluated and the result put into the location called <result-name>. The arithmetic expression will be made up of the names of storage locations, constants, arithmetic operators: plus, minus, multiply and divide, and parentheses.

To define the meaning of an arithmetic expression the idea of an 'order of precedence' is required. For example, without a defined order of precedence it is not possible to say whether the result of 2 + 3 *4 is equal to 14 or 20. The order of precedence is defined as:

() parenthesized expressions
* / multiply and divide
+ - add and subtract

If two or more operators at the same level of precedence appear in an expression then they are evaluated from left to right.

This means that expressions in parentheses are evaluated first, followed by multiplications and divisions, and finally additions and subtractions. These rules may need to be applied repeatedly if expressions contain 'nested' parentheses, that is parentheses within parentheses ...

e.g. calculate total-price = sale-price * sale-quantity
calculate discounted-price =
(1 - discount-rate) * total-price

It should be noted that the equals sign, in this context, means 'becomes equal to' rather than 'is equal to'. Therefore

calculate total-records = total-records + 1

is perfectly sensible and means take the value in the data-item total-records, add 1 to it and store the result in total-records.

A sequence of basic operations will be executed in the order written, one after the other from top to bottom. Therefore the sequence:

transfer first-item to second-item
transfer second-item to first-item

will not exchange the contents of first-item and second-item. The value in first-item will be copied into second-item then this new value will be copied back from second-item to first-item, leaving both

locations containing the same value: the original value of first-item.

2.2 Selection and Repetition

Before considering the details of the selection and repetition operations it is necessary to define the conditions which can be used to control these operations.

Conditions

The conditions which will be allowed initially in selection and repetition operations fall into two groups.

The comparative conditions are of the form:

```
<item-name> <comparison-operator> <compared-value>
```

where the <comparison-operator> is one of:

<	less than	not <	not less than
>	greater than	not >	not greater than
=	equals	not =	not equal to

and the <compared-value> is either another item-name or a constant.

e.g.
```
sales-quantity < 20
taxable-pay > tax-threshold
employee-super-code not = "A"
```

Note that the operators '>' and '<' are not included but are equivalent to 'not <' and 'not >' respectively.

The file conditions test the current state of the input file and are of the form:

```
end-of-<filename>                    not end-of-<filename>
```

The condition end-of-<filename> is true if there are no more records available in <filename> because the last get-next-<filename> failed. This condition is false after a successful get-next-<filename> or at the beginning of the program, when no information is available. The condition not end-of-<filename> is simply the converse of end-of-<filename>.

e.g.
```
end-of-sales
not end-of-customer-file
```

Selection

The selection operation has two forms:

```
if <condition> then
    <procedure-1>
else
    <procedure-2>
endif
```

(a)

```
if <condition> then
    <procedure-3>
endif
```

(b)

The selection operation (a) works as follows:

(1) the <condition> is evaluated to give true or false,
(2) if the <condition> is true then carry out <procedure-1>,
(3) if the <condition> is false carry out <procedure-2>,
(4) after selecting the appropriate procedure continue by carrying out the next operation after endif.

The shortened version of the selection operation, shown in (b) above, works in a similar way except that when the <condition> is false the operation does nothing. Therefore <procedure-3> is executed if and only if the <condition> is true.

The 'procedures' specified can be a sequence of one or more basic operations, or the name of a sequence of operations to be executed (see 2.4) or a mixture of the two.

Examples of selection operations:

```
if taxable-pay not < tax-threshold then
   calculate total-higher-payers = total-higher-payers + 1
   higher-tax-routine
else
   calculate tax-to-pay = taxable-pay * standard-rate
endif

if house-price > stamp-limit then
   add-stamp-duty
endif
```

Repetition

Initially only one repetition operation will be introduced and it has the general form:

```
until <condition> do
    <procedure>
enduntil
```

This operation is defined as follows:

(1) the <condition> is evaluated to true or false,
(2) if the <condition> is true then the operation following enduntil is carried out,
(3) if the <condition> is false then carry out the <procedure> and return to step (1) above.

Informally, the <procedure> is executed repeatedly until the

<condition> becomes true when the operation following enduntil is executed. If the <condition> is true when the until is first executed then this whole operation has no effect and the operation following the enduntil is executed immediately.

The <procedure> specified is similar to that for selection: a sequence of one or more basic operations, or the name of a sequence of operations, or a mixture of the two. For example:

```
until end-of-employee do
   calculate-tax
   get-next-employee
enduntil
```

In this example 'calculate-tax' must be the name of a sequence of operations to be executed.

2.3 Simple Programs

This section discusses some examples of programs to illustrate the material presented in the preceding two sections.

Example 2.1

A very simple but complete program which does something useful is:

```
display-sales-records
   get-next-sales
   until end-of-sales do
       transfer sales-record to printer-line
       put-printer-line
       get-next-sales
   enduntil
```

Here there is an input file called 'sales' which contains records called 'sales-record', and there is an output record called 'printer-line' which is assumed to belong to a lineprinter file. The program starts by attempting to get a record from the sales file. If there are no records in this file then end-of-sales will be true, the until will have no effect and the program will terminate.

If there is at least one record in the sales file then end-of-sales will be false and the operations between until and enduntil will be repeated until end-of-sales becomes true. The operations inside the loop simply transfer the current input record to the output record, transfer the output record to the output file (printing a line in this case) and attempt to get another record from the sales file.

Therefore this simple program will print the entire contents of the sales file on the lineprinter, exactly as they appeared in the input file. In itself this may be occasionally useful but it is important because it forms the basic framework for a large number of programs which read records from an input file, process each record and output the results.

The indentation shown in the example above is not mandatory but it is useful for emphasizing the structure of the program.