Some consequences of deferred binding in COBOL

B. G. T. Lowden and I. R. MacCallum

Computing Centre, University of Essex, Wivenhoe Park, Colchester CO4 3SQ

It is a well known and accepted fact in systems design that once the basic objectives of a system have been met, the cost of further expansion frequently outweighs any increased benefits and so effectively deters development.

In many large organisations the implementation of sophisticated data base technology can represent both a profitable and necessary investment which, amongst other things, provides a comprehensive solution to the above problem. In other, perhaps smaller, installations the more important processing requirements are simply those of flexibility in file design and ease of changing formats.

This paper sets out a proposal for a modified COBOL implementation which, by providing a level of data independence adequate to most DP installations, may be seen to offer a viable alternative to a full data-base system.

The approach is based on the transferral of the data division file definitions, from the COBOL program, to the head of the physical files to which they pertain. Only those data elements referred to in the application logic need then be declared in the program itself, and association between the two definition sets may be deferred until run time.

This work is an extension of that being currently undertaken by MacCallum (1973) and MacCallum and Jones (1975) forming part of a research programme aimed at improving information processing techniques.

(Received October 1975)

Data independence (file design problems)

Probably the greatest difficulty in developing integrated systems is that the addition of new applications normally involves extensive changes to existing program suites. File format changes, minor in themselves, can prove extremely costly in terms of program modification and recompilation, one notable example of this being decimalisation.

The size of the problem may be partly reduced by leaving space for expansion in the data records, however this inevitably increases both file sizes and processing overheads and does not allow for changes in record structure.

The concept of data independence or 'file transparency' implies that the application program is independent of the format, organisation and media of the files to which it refers.

To a certain extent all programs are data dependent in that their logic will be bound to the size and appearance of particular data fields. However it is an unfortunate fact of life that changes to the structure, or content, of records within a file usually necessitate alterations to all programs accessing that file *regardless* of whether the changes actually affect fields used by the program. It is this problem that the proposals of the next sections seek to alleviate.

Towards a generalised hierarchical structure

Consider a typical inventory record as depicted in **Fig. 1.** Whilst such a record may be simple to represent in a manual system of multiple stock cards, the converse is true when trying to define its structure in a COBOL data division.

For example there may be a variable number of occurrences of the segments ⟨ORDERS⟩, ⟨REQUIREMENTS⟩ and ⟨CUSTOMER⟩ within the owning segment ⟨PART⟩. In addition the segments ⟨DESCRIPTION⟩ and ⟨CUSTOMER⟩ may themselves be of variable length.

There are not many options open to the systems designer. He may place upper bounds on the size of variable data items and the number of repeats, thus incurring the usual penalties of wasted space and inflexible design, or he may make each segment into a record in its own right. Since, in a hierarchical structure, a segment may only be identified in the context of its superiors, this latter approach implies the overhead of including additional key fields.

Neither of these techniques can be considered particularly attractive to the commercial analyst and both entail program amendment if the level hierarchy (structure) is modified by, say, the insertion of an additional segment ⟨PRODUCT⟩ between ⟨PART⟩ and ⟨ORDERS⟩, ⟨REQUIREMENTS⟩, or even the simple re-arrangement of segments of the same level.

As an example of a more flexible approach, consider the following data structure (**Fig. 2**). In this case, the file consists of an ordered set of record occurrences A each of which consists of a hierarchical arrangement of segment occurrences B, C, . . . K. Segment occurrences which are marked with an asterisk, may own a variable number of filial segment occurrences. All terminal segment occurrences can include variable length fields.

A convenient method for mapping this kind of structure onto linear storage is described more fully in Bowden, MacCallum and Patience (1971). Nodes corresponding to the segment occurrences marked with an asterisk are transformed into two levels, so that the parent node owns an explicit number of filial segment occurrences. For example, if in Fig. 2 D, C and H have *k*, *m* and *n* sons each, the transformed structure is as shown in **Fig. 3.**

Each record is seen to be represented as an ordered tree whose nodes may hold variable length information, and may also possess any number of successors. The nodal representation of a segment occurrence might then appear on backing media as in **Fig. 4.**

Clearly this form of dynamic structure is better suited to the representation of commercial records, than is currently permitted by the file section of COBOL's data division. In fact one may regard the latter as a degenerate case of the above.

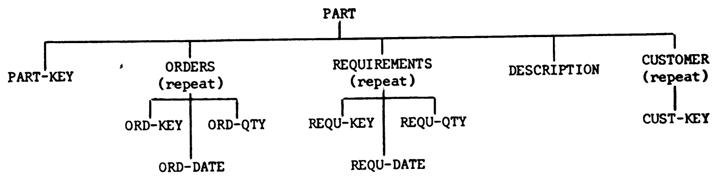

Fig. 1

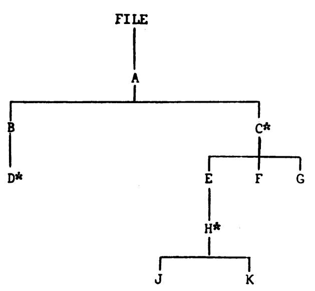

Fig. 2

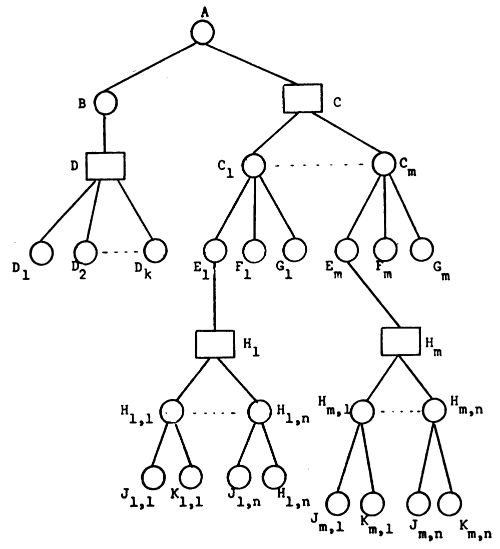

Fig. 3

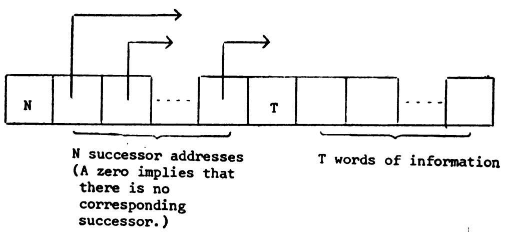
N
T
N successor addresses (A zero implies that there is no corresponding successor.)
T words of information

Fig. 4

A suggested approach

The differences between the structure representations of Figs. 2 and 4 are purely procedural, and can be tabled in the form of a simple, linear mapping function.

Further there is no reason why this function should not be held as a fixed format descriptor record at the head of the file itself, interpretation of data records being carried out at run time. As an example, the mapping function for the structure of Fig. 1 could be as specified in near COBOL format as follows:

Name	*Level*	*Length*	*Class*	*Occurrences*
PART	Ø1	—	—	1
PART-KEY	Ø2	12	X	—
ORDERS	Ø2	—	—	*
ORD-KEY	Ø3	6	9	—
ORD-DATE	Ø3	4	9	—
ORD-QTY	Ø3	4	9	—
REQUIREMENTS	Ø2	—	—	*
REQU-KEY	Ø3	8	9	—
REQU-DATE	Ø3	4	9	—
REQU-QTY	Ø3	4	9	—
DESCRIPTION	Ø2	V	X	—
CUSTOMER	Ø2	—	—	*
CUST-KEY	Ø3	8	9	—

where * denotes a variable number of segment occurrences and V a variable length field.

The existence of an external, as opposed to a data division, representation of a file's structure can now be exploited to provide a degree of data independence within the program itself, since the association between the mapping function and the program's own view of the record structure may also be deferred until run time.

Data definition

To develop the concept of deferred binding we first distinguish four types of data variable associated with the program and files.

1. *True fixed length*
These need to be fixed by virtue of the program logic, e.g. they may be used in output formatting, or redefinition.

2. *True variable length*
Essential in many types of information processing for holding fields of widely ranging sizes such as book titles, part descriptions, etc.

3. *Pseudo variable length*
These are fixed length fields which may be subject to uniform change from time to time within certain defined limits. Thus no length descriptor fields are required within the file since at any one time all occurrences are of equal size, and this size is conveniently held in the mapping record. Examples are numeric fields and descriptor codes.

4. *Non-sensitive fields*
Data which is held on file but not operated on, in any way, by a particular program may be regarded as non-sensitive as far as that program is concerned. Such data is usually described as FILLER in COBOL's data division.

Of the first three types, only true fixed length is currently acceptable in COBOL. The proposed implementation allows both true and pseudo variable length fields to be represented as well as requiring only sensitive data to be declared. Below is an example of data division syntax which embodies these features and depicts a partition of the record format shown in Fig. 1.

```
LFD  PARTFILE.
     VALUE OF ID IS "PTFILE 001".
Ø1   PART.
     Ø2 PART-KEY       PIC X(12).
     Ø2 ORDERS         [IS] SUBRECORD.
        Ø3 ORD-KEY     PIC 9 VARYING 6 TO 10.
        Ø3 ORD-DATE    PIC 9(6).
        Ø3 ORD-QTY     PIC 9 VARYING 4 TO 8.
     Ø2 DESCRIPTION    PIC X(V).
     Ø2 CUSTOMER       [IS] SUBRECORD.
        Ø3 CUST-KEY    PIC 9 VARYING 6 TO 10.
```

As may be seen the syntax is slightly but not significantly changed from standard COBOL; notes of explanation are however needed to describe the semantics.

Firstly the section of code is a declaration of a 'logical' file definition (LFD) rather than the physical definition of standard COBOL (FD). The LFD may best be regarded as the view of the physical file (i.e. that on backing store) as seen by the application program. In this example the program is not concerned with ⟨REQUIREMENTS⟩, i.e. they are non-sensitive and logically, therefore, they are not declared.

Changes to non-sensitive data, e.g. increase in field size, change of type, will not therefore affect the program in any way.

Declaration of multiple occurrence segments is achieved by associating the identifiers with an '[IS] SUBRECORD' clause. Segments so qualified may be regarded as records in their own right as will be further illustrated in the next section.

Examples of true and pseudo variable length fields are DESCRIPTION and CUST-KEY respectively. Use of the VARYING clause enables the bounds of pseudo fields to be declared, and catered for, in the programs whilst still allowing simplified representation on backing media. Clearly any changes effected to data fields on the file which conflict with the logical declaration, will necessarily require changes to the program. However with the proposed syntax this should be reduced to a minimum.

Currency

The concept of multiple occurrences, demonstrated in the last example, implies that only one occurrence is represented at any time in the logical file definition. It is essential therefore to introduce the term 'currency', Codasyl (1971).

In the proposed implementation it will clearly be necessary to hold some indication of which particular segment occurrences are currently being manipulated by the program logic. Thus the LFD, at a particular moment, may refer to the third occurrence of ORDERS and the eighth occurrence of CUSTOMER.

This will have been achieved by viewing occurrences as subrecords within records and using an 'occurrence read' statement

```
READ ⟨occurrence name⟩
```

two and seven times respectively. The first occurrence of each segment will already have been made available by a standard

```
READ ⟨file-name⟩⟨statement⟩
```

The basic idea of currency, then, is to describe the most recently accessed segment occurrence as current.

At this point the reader will have, no doubt, recognised the similarity between the proposed concept of subrecords and COBOL's existing OCCURrence clause, in that each facility enables the repetition of structures to be defined. It is worthwhile, therefore, to emphasise the essential differences.

In standard COBOL the *maximum* number of possible substructures must be declared by means of the OCCURS clause. Furthermore, the length of a record can only be tailored to a variable number of OCCURrences, if an OCCURS . . . DEPENDING clause describes the trailing section of the record. In the dynamic record implementation of COBOL, MacCallum and Jones (1975), already referred to, these restrictions are lifted, though references to identifiers at or below the level of the OCCURS clause must be subscripted.

The SUBRECORD concept similarly permits any number of substructures to exist within a record but does not imply subscripting any more than one would expect to subscript the records of a file to denote their physical location within that file. Specific subrecords are thus selected from a record by statements analogous to those used for selecting records from a file system, namely READ ⟨occurrence name⟩.

The decision as to which structuring method should be used in a particular application, will naturally be dependent on the kind of processing to be performed on the substructure, and its context within the program. For example, if many references are to be made to a particular instance of a substructure without intervening references to others, as in sequential processing, then the SUBRECORD approach is to be preferred.

On the other hand, a subscripting technique would clearly be more suited to randomly accessing a variable sized table by a binary search routine.

The distinction is somewhat analogous to move-mode and index-mode in a multibuffering situation.

If changes are made to a segment while it is current, they may be effected on backing store by the use of an 'occurrence write' statement of the form

```
WRITE ⟨occurrence name⟩
```

and the segment will remain current. The use of a second WRITE statement with no intervening occurrence read will cause a new segment occurrence to be inserted immediately after the one which is current, leaving the new segment as current.

File restructuring

There are, in general, two ways in which COBOL record formats may be restructured:

1. By adding or deleting field names, re-arranging physical field locations or altering field lengths.
2. By modifying the hierarchical nature of the record by inserting or removing levels, or by relinking subordinates of LEVEL A to, say, LEVEL B.

The first transformation may easily be accomplished using conventional update processing and associating the input and output files with the old and modified logical file definitions respectively.

For example the input may be described logically as:

```
Ø1 PART.
   Ø2 PART-KEY        PIC X(12).
   Ø2 ORDERS          [IS] SUBRECORD.
      Ø3 ORD-KEY      PIC 9(6).
      Ø3 ORD-DATE     PIC 9(4).
      Ø3 ORD-QTY      PIC 9(4).
         ———
         ———
         ———
         ———
         ———
```

and the output as:

```
Ø1 PART.
   Ø2 PART-KEY        PIC X(12).
   Ø2 ORDERS          [IS] SUBRECORD.
      Ø3 ORD-KEY      PIC 9(8).
      Ø3 ORD-DATE     PIC 9(4).
      Ø3 ORD-QTY      PIC 9(4).
      Ø3 PRODUCT-CODE  PIC 9(6).
         ———
         ———
         ———
         ———
         ———
```

The transformation, then, has the effect of increasing the field size of ORD-KEY from 6 to 8 numeric characters and appending an additional field PRODUCT-CODE. These changes will be reflected in the mapping function heading the new file and would in no way affect programs associated, say, with the logical file definition of Fig. 1.

To avoid the problem of repeatedly reading, moving and writing individual segment occurrences, a new MOVE option is introduced of the form:

```
MOVE LOGICAL A TO B
```

the semantics of which overcome the present constraints of COBOL's MOVE CORRESPONDING, with regard to subordinate occurs clauses. Using a MOVE LOGICAL statement enables a complete set of segment occurrences to be moved from one area to another.

As stated in 2 above, hierarchical changes are essentially the insertion or deletion of occurrence levels. In practice it is difficult to conceive of a situation which would require the removal of a parent occurrence whilst still retaining its filial links. This is ordinarily because a hierarchy implies some sort of grouping or ordering of suboccurrences within an owning occurrence. Such ordering would be lost if the owning occurrence were deleted.

In our modified implementation, therefore, we have taken the deletion of an individual occurrence to mean the removal of the named occurrence plus all its dependent occurrences, if any. However it may be necessary to caution or even prohibit such action if, in deleting an occurrence declared in a logical file definition, a programmer could unknowingly remove non-sensitive data unseen by the application.

Insertion of a new occurrence *level*, on the other hand, implies a re-ordering of subordinate levels (unless the degree of the new occurrence is unity, when all succeeding level numbers are simply incremented by 1).

It is necessary, therefore, to extend the scope of the COBOL sort verb to include occurrence sequencing. This, in practice, would be fairly straightforward and does not involve any changes to the syntax:
e.g.

```
SORT ⟨occurrence-name⟩ ON { ASCENDING  } KEY
                          { DESCENDING }
                                    data-name etc.
```

Suppose that the structure of Fig. 1 is to be modified to include a ⟨PRODUCT⟩ occurrence, between ⟨PART⟩ and ⟨ORDERS⟩, regrouping the ⟨ORDERS⟩ occurrences on the value of PRODUCT-CODE.

The input and output logical file definitions might then appear as:

```
Ø1 PART.
   Ø2 PART-KEY            PIC X(12).
   Ø2 ORDERS              [IS] SUBRECORD.
      Ø3 PRODUCT-CODE     PIC 9(6).
         ——
         ——
         ——
         ——
         ——
```

and

```
Ø1 PART.
   Ø2 PART-KEY            PIC X(12).
   Ø2 PRODUCT             [IS] SUBRECORD.
      Ø3 PRODUCT-CODE     PIC 9(6).
      Ø3 ORDERS           [IS] SUBRECORD.
         ——
         ——
         ——
         ——
         ——
```

respectively, and the program logic would include a statement of the form:

```
SORT ORDERS OF ⟨input file⟩
          ON ASCENDING KEY PRODUCT-CODE .
```

On writing the restructured output file, all ⟨ORDERS⟩ occurrences associated with a given PRODUCT-CODE must be linked to the appropriate ⟨PRODUCT⟩ occurrence.

In this example ⟨ORDERS⟩ occurrences would be successively read from and written to the input and output files respectively until a change in the sort key PRODUCT-CODE is detected. The current ⟨PRODUCT⟩ occurrence is then written, and the process repeated until the input is exhausted.

Implementation

The proposals outlined in this paper, for achieving a level of independence between physical file layout and file definitions can be implemented by a development of the approach described by MacCallum and Jones (1975). In this system, at compile time, identifiers are associated with logical nodal addresses of the tree defined by the declarations of the DATA DIVISION. Then at run time, whenever an identifier is to be accessed, tree structures are searched so that the logical nodal address may be associated with a particular physical address for the record being processed. File transparency may be achieved by taking the process one stage further. Assuming that, in the LFD of the DATA DIVISION, only those identifiers of interest to the program are declared and that a complete file descriptor appears as the leading record of a file then at *compile time* identifiers are merely checked to ensure that they appear in the DATA DIVISION and retained as identifier strings in a table, to be completed at file opening time. The identifier temporarily becomes associated with the corresponding location in this table. At *file opening time* the file descriptor is read, and traversed, checking each node's identifier against entries in the table created at compile time; on matching, the corresponding nodal address is entered into the table. Finally, at *identifier access time*, the table is looked up at the location determined at compile time (only two or three store accesses) to discover the corresponding nodal address, from which the physical address is computed as before.

The implementation of subrecords in a COBOL-type environment may be achieved in several different ways. If access to subrecords is limited to the sequential mode, then a method by which subrecords are physically read into a logical record area could be devised. The principal objection to this is that heavy overheads are incurred in the reading and writing of subrecords upon which little or no processing may be done.

In the COBOL implementation, already referred to, a mechanism exists for handling variable numbers of sub-structures. To include a subrecord facility, this system requires the following basic modifications:

1. The addition of currency fields at nodes described as SUBRECORD nodes.
2. The creation of mechanisms for resetting all currency fields of a given subtree to 1, and for incrementing a given currency field on the occasion of a subrecord sequential read. The simplest implementation of this is by traversing the subtree and testing whether each node is a currency node.
3. For each reference to a node of a subrecord, the physical address computation is similar to that used for subscripts, except that the value of the appropriate currency field is used instead of the value of a subscript identifier.

Thus the principal overhead consists of a tree traversing operation upon every subrecord read. To avoid the overhead of resetting currency fields on every (full) record read, they can be reset whenever a record is written.

Conclusions

This paper has outlined proposals for a modified implementation of COBOL which endows the language with considerably enhanced capabilities for handling 'real life' record structures.

The intention is to develop the semantic power of its existing statements to meet present day needs whilst at the same time changing the syntax as little as possible. This is considered essential when dealing with a language as widely used as COBOL.

The advantages of data-independent file processing are self-evident in a practical DP environment. It has been estimated (Engles, 1975) that 25% of commercial programming is associated with routine maintenance.

Just as important there is no reason why both logical files and conventional files may not be handled in the same program, or logical definitions used to describe conventional record formats. The process of actual file conversion can therefore be gradual, existing files being 'generalised' as and when the need arises, normally as a result of system integration.

Overall it is felt that the proposals bridge the gap between conventional programming and data base technology.

Acknowledgement

The authors are indebted to P. E. Jones, also of Essex University, for his helpful comments and suggestions during the preparation of this paper.

References

BOWDEN, K. F., MACCALLUM, I. R., and PATIENCE, S. P. (1971). Data Structures for General Practice Records, *Proc. IFIP Congress* 1971.

CODASYL (1971). *Feature Analysis of Generalised Data Base Management Systems*, Section 7.4, May 1971.

ENGLES, R. W. (1972). A Tutorial on Data Base Organisation, *Annual Review Automatic Programming*, Vol. 7, Part 1, July 1972.

MACCALLUM, I. R. (1973). Interpreting Record Structures, proceedings of *Software 73*, Transcripta Books, London.

MACCALLUM, I. R., and JONES, P. E. (1975). Dynamic Record Structures for COBOL, Computer Science Memorandum CSM—13, (Copies available from Computing Centre, University of Essex).

Book reviews

Introduction to Optimization Methods, by P. R. Adby and M. A. H. Dempster, 1974; 204 pages. (*Chapman and Hall Limited*, £2·50)

The problem of optimization is usually regarded as finding **x** which minimises a nonlinear functional $f(\mathbf{x})$, sometimes subject to equality constraints $g(\mathbf{x}) = 0$ or inequality constraints $h(\mathbf{x}) \geq 0$.

The history of the development of the subject is bound up with the growing availability of adequate computational facilities. The main mathematical concepts were worked out a quarter of a century ago before computers really arrived on the scene. The Kuhn-Tucker theory dates from 1951. Conjugate gradients were first used practically about the same time. Levenberg's method for least squares problems goes back further to 1944. The subject never seems to have held much interest for pure mathematics. In the early literature, the names of those developing algorithms are those of chemical engineers, operational research scientists and others with large practical problems demanding solutions. Typical of this phase of development is Rosenbrock whose well known method appeared about 1960—his notorious valley function still survives as a test of the efficiency of new methods.

In the late 1950's numerical analysts became interested. Powell, one of the most prolific of writers on optimization, began publishing about 1962. Marquadt's celebrated algorithm appeared in 1963. In the mid 60's and early 70's there followed a veritable flood of papers introducing what were essentially computing variants of a relatively small number of basic methods. The flood has only recently abated. At its height the research was basically in the field of algorithmics and computing science. Many papers were concerned with the minutiae of computing tactics, few made a distinctive contribution to mathematical knowledge. (Not that this was a bad thing! Many practical people wanted practical answers quickly and economically). By now anyone wishing to write a text book on optimization has a problem of discriminating the really useful from a mass of techniques.

The book by Adby and Dempster sets out to be a typical primer—'suitable for undergraduate and postgraduate courses in mathematics, the physical and social sciences and engineering' as the preface says. After an introductory chapter, the second chapter covers the one dimensional problem, covering search methods and elementary approximate methods. Chapter 3 deals with methods for unconstrained multivariable problems. These three chapters are competently done.

Chapters 4 'Advanced methods' and 5 on 'Constrained optimization' take up some sixty per cent of the text. My personal view is that here the authors try to get too much in, and in places the text reads almost like a catalogue. There is much of value, but readers requiring an introductory text might be better served by a more critical account of a smaller number of variants of methods.

There are well explained algorithms for several of the methods described, and an excellent bibliography, plus a number of good examples. Teachers will find this quite a useful text to keep for delving into.

A. YOUNG (Coleraine)

Simulation with GASP PL/I, by A. A. B. Pritsker and R. E. Young 1976; 335 pages. (*John Wiley*, £9·25)

This book is well presented and gives a good insight into the subject of simulation by practical example, as well as being a manual for GASP PL/I. It compares favourably with other books of this type.

The first two chapters provide a good introduction to simulation and would be suitable for initial reading on the subject. The text of part of the second and the third chapter seems somewhat redundant, as the tables and flowcharts provided, together with the excellent examples that follow later, give a good explanation of the language.

The examples, that comprise two thirds of the book, cover a range of discrete and continuous systems that amply demonstrate the flexibility of GASP PL/I.

There are some good exercises at the end of each chapter and the book also contains algorithms for the random deviate generators and the integration method used by the package.

A. CUNNINGHAM (Manchester)

Integral Equations Via Imbedding Methods, by Harriet H. Kagiwada and Robert Kalaba, 1974; 382 pages. (*Addison-Wesley/W. A. Benjamin*, hardcover US$19·50, paperback US$12·50)

In this book methods are developed for obtaining numerical solutions of Fredholm integral equations by converting them to a set of ordinary first order differential equations with given initial values. In most cases the upper limit of the integral in the integral equation is regarded as a variable which is the independent variable of the differential equations.

The first three chapters deal with degenerate and semi degenerate kernels. In the degenerate case, it is shown that if the kernel is expressed as a sum of M products then the integral equation is equivalent to $M^2 + M$ differential equations. The next two chapters deal with the cases of displacement and composite kernels and in chapter 6 the general linear equation is considered. In the latter case it is shown that if an N point quadrature formula is used, the integral equation is equivalent to N^2 differential equations. Nonlinear equations are similarly considered in chapter 7. In chapter 9 both linear and nonlinear equations whose kernels involve a parameter, which includes the eigenvalue problem, are considered. Some particular integral equations corresponding to problems in radiative transfer are solved in chapter 9, and there is a final short chapter dealing with a pair of dual integral equations occurring in potential theory.

The analysis in the book is set out clearly and in detail so that it is easily followed, although it tends to become tedious because of the repetitions with similar cases. The book is well produced and there are very few misprints. What is not convincingly demonstrated is that the method of replacing the integral equation by a set of ordinary differential equations is superior to the normal numerical methods of solving these integral equations.

V. E. PRICE (London)