The significance of the 1974 COBOL standard

J. M. Triance

Department of Computation, University of Manchester Institute of Science and Technology, P.O. Box 88, Manchester M60 1QD

This paper studies the American National Standard on COBOL published in 1974. The many new features are described and their significance investigated. Finally the problems of converting existing programs to the new Standard are discussed.

(Received August 1975)

Introduction

This recent American National Standard, which will form the basis of most new COBOL compilers in the next few years, leaves the core of COBOL essentially the same. In fact simple programs written in the two versions will be practically indistinguishable. This is because most of the changes merely allow greater freedom which an individual programmer may or may not choose to use, exclude some usages which no sensible programmer would ever have used or add facilities which are of value to the more sophisticated programmer. Some of these facilities, such as modular programming, indexed sequential file organisation and the REWRITE verb will already be familiar to programmers on some computers as extensions to the old standard. Others (such as the STRING verb and the COMMUNICATIONS module) will be completely new to most COBOL users. Overall the power of the language has been considerably enhanced. The user of existing compilers however will not welcome all the features of the new standard because despite the aim of standardisation to ease the transition from one version of COBOL to another this standard is *not* completely compatible with its predecessor. Thus some conversion of programs will be necessary. This paper first investigates the new facilities in COBOL 74 and then investigates the problems of conversion. It is emphasised that many of the features described already exist in some compilers.

Modular programming

This in-vogue term is normally used to refer to any approach which attempts to produce one object program from more than one relatively self-contained source code-unit (or module). This includes units which are combined prior to compilation (sometimes called Sectional Subroutines) and those which are separately compiled and combined later. The first approach has always been possible in COBOL by use of the PERFORM verb. The new standard also permits modules to be compiled separately.

This facility known as Inter-Program Communication requires that each module be a syntactically correct COBOL program. Thus all four divisions must be present in each module and all data items referred to in the Procedure Division must be defined somewhere in the Data Division of the same module. The main module initiates the execution of a submodule by means of a CALL statement. When a CALL statement is executed the specified module will be executed from the beginning of its Procedure Division until an EXIT PROGRAM statement is encountered whereupon control will return to the statement following the original CALL statement in the calling module. Thus each module has only one point of entry but any number of exits. The called module can in turn call other modules, and so on, provided that a module does not directly or indirectly call itself: recursion is not allowed.

Data items which are transferred between two modules must be listed in the USING clause (see **Fig. 1** which shows parts of a main module and a module which it calls to calculate a check digit). Thus the USING clause lists the data items which are used in both modules. It appears in the CALL statement of the calling module and the Procedure Division heading of the called module. Whenever a data item in the latter is referred to it is processed as if the reference was to the data item which appears in the corresponding position in the USING clause of the CALL statement. Thus in the example (Fig. 1) when CODE-NO is specified in any statement in the Procedure Division of the called module then the data item CUST-NO in the calling module is accessed. A module is no different from any COBOL program in that the name chosen for the data item only has significance within the module so the programmer can choose whether to give corresponding data items in two modules the same name or not. The USING parameters in the CALL statement (CUST-NO and CHARS in the example) must be defined normally in the Data Division. Those in the called

```
Calling module
IDENTIFICATION DIVISION.
PROGRAM-ID. VALCUST.
    :
DATA DIVISION.
    :
WORKING-STORAGE SECTION.
77  CUST-NO  PIC X(10).
77  CHARS    PIC 99.
PROCEDURE DIVISION.
    :
    CALL "CHECKDIG"
      USING CUST-NO CHARS.
Called module
IDENTIFICATION DIVISION.
PROGRAM-ID. CHECKDIG.
    :
DATA DIVISION.
WORKING-STORAGE SECTION.
77  CURR-DIG  PIC 99.
77  CALC-DIG  PIC 99.
    :
LINKAGE SECTION.
77  CHARS     PIC 99.
1   CODE-NO.
  5 DIGIT      PIC 9 OCCURS 10.
PROCEDURE DIVISION
    USING CODE-NO CHARS.
INIT.
    MOVE 1 TO CURR-DIG.
    :
    MOVE CALC-DIG TO DIGIT (CURR-DIG).
END-CD.
    EXIT PROGRAM.
```

Fig. 1 Example of Inter-Program Communication

module however (CODE-NO and CHARS) are really only synonyms and therefore require no additional allocation of core. However the compiler does need to have a description of these items so that it can determine such things as their length and data format. To achieve this all the items listed in the USING clause of the Procedure Division heading must be defined in the Linkage Section of the Data Division which allows the structure of the data items to be specified without any core being allocated for them.

Other facilities available are the option of deciding which module any CALL statement executes at run time and the overlaying of modules. When a data-name rather than a literal is specified after the word CALL then the module whose name is currently stored in the data-name is executed. This can of course, be altered as desired during the running of the program.

When a CALL statement is executed it is possible to detect and take corrective action if there is insufficient room in core for the called module. This corrective action could consist of logically removing from core another module by means of the CANCEL verb. When a module which has been overlayed in this manner is next accessed it will be in the same state as it was initially. Any module which is only partially executed cannot be cancelled.

This facility undoubtedly satisfies the basic requirements of modular programming. However problems might result from the fact that parameters in a using clause are restricted to level 77 or level 1 data items. Another problem is that it is not clear from the standard to what extent any file can be processed in more than one module. Finally the programmer faces the chore of having to repeat all the Identification and Environment Division coding in each module.

File organisation and access

The existing techniques for accessing sequential files based on the OPEN, CLOSE, READ and WRITE verbs remain unaltered. There are, however, many new facilities for use with magnetic files.

'*Piggy-backing*' is now possible: in other words, records can be added to the end of an existing sequential file by opening the file with the EXTEND option and then simply issuing WRITE instructions as usual.

Random access by means of the ACTUAL KEY has been dropped and in its place there are the Indexed and Relative file organisations. With the former the Standard makes no mention of indexes or overflow areas but it is, as its name suggests, compatible with the typical indexed sequential file organisations supported by many operating systems. The programmer must indicate the position of the key in the record by specifying its data-name. The READ verb can then be used to access records sequentially or obtain a record with a given key value. The WRITE verb can, similarly, be used to output records sequentially or randomly. Besides specifying the key on which the file is sequenced (the prime key) it is also possible to specify one or more alternate keys. When an alternate key is specified (that is another field within each record of the file) records can be accessed by the value of this key field. The programmer can also read the file in sequence by the alternate key. This, presumably, would prove to be a rather inefficient way of accessing the whole file, but will be useful for obtaining all the records with the same key once the first one has been located: the value of an alternate key (unlike the prime key) need not be unique in each record on the file. Thus the Indexed I-O module offers partially *inverted files* as well as indexed sequential.

The Relative file organisation corresponds to one possible implementation of its predecessor (random access by ACTUAL KEY) with some additional features. A relative file can simply be regarded as a big one-dimensional table on backing store rather than in core. Records can be written and read sequentially but it is also possible to access individual records by specifying their relative position in the file. If, for example, 6 is moved to the field specified in the RELATIVE KEY clause of the SELECT entry the next READ statement will access the sixth record in the file.

On some computers random access is currently achieved by specifying the hardware address, such as cylinder and track numbers, in the ACTUAL KEY and in some cases synonyms are handled by software. Compared with such implementations Relative I-O is simpler and it is device independent. On the other hand if the programmer wishes to use a randomising algorithm to locate his records it is no longer possible for the software to handle synonyms automatically. Furthermore because the programmer has no knowledge of where the cylinder boundaries lie any method he devises for handling overflows could prove inefficient. These problems are unlikely to inconvenience most users, however, since when records must be located by key values the Indexed organisation will normally be an adequate (or desirable) substitute for randomising algorithms. It is however likely that some compilers will support Relative but not Indexed I-O.

It has already been indicated that Indexed and Relative files can be accessed either sequentially or randomly. In addition they can be accessed *dynamically* which means that records can be accessed by key value (i.e. position in the case of relative files) and sequentially in the same run. The NEXT option of the READ verb is specified to indicate that the next record in sequence is to be obtained regardless of the type of access that preceded or will follow it.

There are also two new verbs which can be used with Indexed and Relative files. They are START and DELETE. Sequential access can begin with the record having any specified key value simply by preceding the READ statements by an appropriate START statement. In the case of an alternate key the access begins at the first record with the given key value. With other variations of this verb it is possible to begin access at the record whose key is 'greater than' or 'greater than or equal to' a given value. The DELETE verb, as its name suggests, can be used to prevent any record from being accessed in the future—the data is logically destroyed.

There are two new facilities, REWRITE and FILE STATUS which can be used with any of the file organisations. The REWRITE verb is used to update records *in place*. With sequential access the record most recently read is replaced in its amended form. With random access the record in core replaces the record already on the file which has the specified key value. In either case the new record and the replaced one must be of the same length.

The FILE STATUS facility in the new Standard is designed to supplement, or if the programmer desires, replace the traditional exception handling for files. To use this facility with a file a two character data item must be specified in the FILE STATUS clause in the SELECT entry. After each operation on the file this data item is automatically set to indicate the outcome of the operation (such as 'success', 'end of file' or 'sequence error' in an Indexed file). The programmer can specify a declarative section which will be executed when the operation is unsuccessful. This section can check the File Status data item to discover which type of exception has occurred and then take the appropriate action. This facility does not prevent the programmer from continuing to use the AT END & INVALID KEY clauses except that the latter can no longer be used with sequential files.

Finally the Standard presents many operating systems with a sizeable problem. None of the file organisations are restricted to fixed length records only. So in theory the programmer will be able to have variable length record Relative and Indexed files in any full standard compiler. The user should check carefully to see if the manufacturer solves this problem by

padding out all records to the maximum length with the resulting waste of space in the files.

Sequencing data

As file sorting has been supported by COBOL for some time the addition of the simpler task of merging comes as no surprise. The MERGE statement is similar in format to the SORT statement (see **Fig. 2**).

It specifies the name of the work file (SALES-MERGE in the example), the keys (REGION and SALES-VALUE) in decreasing significance with an indication of whether the data in each key field is descending or ascending. Finally it lists the names of the input files (CUMULATIVE-SALES and TODAYS-SALES) and the output file (NEW-CUM-SALES). For consistency with SORT the work file is defined in an SD the associated record of which includes the definition of the keys. Furthermore the work file name must appear in a SELECT statement. However since the same file cannot be used for input and output from the merge there appears to be no need for any actual working space on backing store. So unlike SORT the work file appears to be a purely imaginary file which nonetheless must satisfy the rules of COBOL.

As with the SORT verb an output procedure can be specified instead of an output file but an input procedure cannot be specified presumably because the type of input procedure used in SORT would not be suitable. Consistency in the USING option has been maintained by permitting more than one file to be input to the SORT verb. Both verbs can only be used for sequentially organised files.

One of the problems previously posed when files and COBOL programs are transferred from one computer to another has been the different character sets and collating sequences. To alleviate this problem each COBOL compiler is now required to support the American National Standard Code for Information Interchange and may, if desired, support any other character sets. The programmer can then use the CODE-SET clause in an FD entry to indicate which of these character sets should be used for that file. This will often be of value when a magnetic tape is used to transfer data between the computer and some other equipment.

Besides this external use the programmer can specify which collating sequence should be used for all comparisons in the program. He can specify one of the sets supported by the compiler or define his own in the Special-Names paragraph. The collating sequence chosen would be used in all relation conditions and any SORT or MERGE statements in which an alternate collating sequence had not been specified.

Producing reports

The Report Writer facility has not been altered in principle. There are however many changes of detail most of which are aimed at removing ambiguities or imposing rules which the sensible programmer will already be adhering to. The one completely new feature is the SUPPRESS verb which can be used to prevent a specific report group from being output whenever required. Thus if for example a control heading was only required when no higher level control break had occurred this situation could be detected and a SUPPRESS statement executed to prevent the heading from appearing.

```
MERGE SALES-MERGE
    ASCENDING  KEY REGION
    DESCENDING KEY SALES-VALUE
    USING CUMULATIVE-SALES TODAYS-SALES
    GIVING NEW-CUM-SALES.
```

Fig. 2 A Merge Statement

Some of the most useful facilities in the Report Writer have also been made available for report production by means of the WRITE verb. These facilities are concerned with the end of page situation. In the FD entry of a print file the programmer can indicate by means of the LINAGE clause the number of lines on the page required (*a*) for printing, (*b*) as a top margin and (*c*) as a bottom margin. Furthermore he can, if he wishes, specify where within the section used for printing a page 'footing' (the opposite to heading for those who are not familiar with Report Writer terminology) is to appear. **Fig. 3** shows an example of the LINAGE clause showing the page format which results.

```
FD PRINT-FILE
    LABEL RECORDS OMITTED
    LINAGE 60 LINES
      FOOTING AT 58
      LINES AT TOP 3
      LINES AT BOTTOM 3.
```

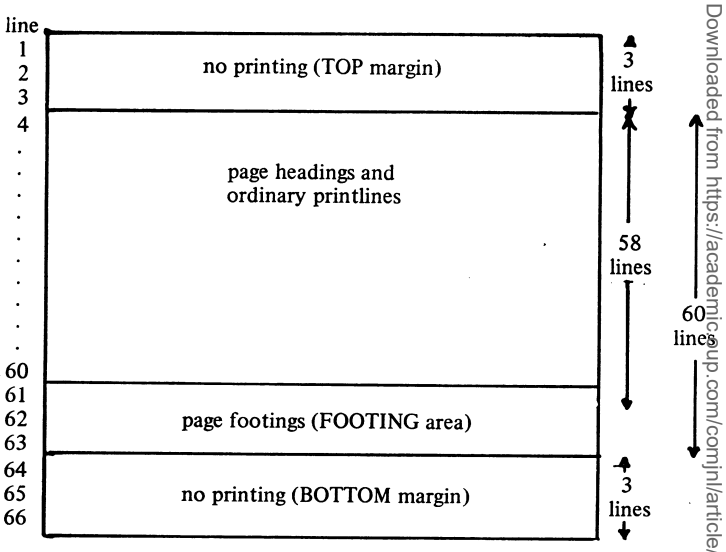

Fig. 3 Use of the LINAGE clause

Users of the Report Writer facility should note that unlike the PAGE LIMIT clause this clause does not use absolute line numbers (the reason for this inconsistency is not obvious). However an advantage that this clause offers over the corresponding Report Writer facility is that it is dynamic: data names can be specified instead of integers and by altering their values each new page can, if required, have a different format.

The advantage of describing the page format is that the page end will be detected automatically and, if required, handled automatically. Thus in this latter case an attempt to print in the bottom margin will cause an automatic advance to the new page and past its top margin. Alternatively the programmer can take his own action whenever an attempt is made to print in the footing area or beyond by using the END-OF-PAGE clause in the WRITE statement. When such an attempt is made the statement in this clause is executed allowing the programmer to print any page footings and then headings he requires. Another clause in the WRITE verb allows the stationery to be advanced to a new page, whether or not the previous page is full. This is achieved by WRITE . . . AFTER ADVANCING PAGE.

These new facilities mean that the programmer need no longer concern himself with machine dependent conventions for advancing to the top of a page and furthermore need no longer keep a count of the lines printed so that he knows when the page is full. This latter task is done automatically when the

LINAGE clause is used and the current line number can, if desired, be accessed in the special register LINAGE-COUNTER.

Debugging

The previous Standard has no facilities designed specifically for debugging. This standard does not provide new debugging verbs (such as EXHIBIT and TRACE) but does provide the opportunity for the programmer to take whatever action is required at predetermined points in the program.

When a USE FOR DEBUGGING declarative is included in the program the progress of the program is automatically monitored. After executing a procedure whose name appears in such a statement the relevant declarative is executed. This gives the programmer the opportunity to print out the procedure-name and the values of any data items or take any other appropriate action. The execution of a dubugging declarative can also be initiated by reference to specified file-names or identifiers. Thus the progress of any data-item, file or the transfer of control within the Procedure Division can be monitored. Each time one of these declaratives is executed a special register (DEBUG-ITEM) is automatically provided with details of the cause of execution such as the relevant source-line number, the name of the relevant procedure, data item or file and an explanatory message. This information is provided in such a form that it can be printed (or DISPLAYED) without further editing if required.

In addition to debugging declaratives extra lines of coding can be included anywhere in the program after the OBJECT-COMPUTER paragraph, specifically for debugging purposes. These lines are identified by the 'D' in column 7. Such lines and any debugging declaratives will only be compiled when the DEBUGGING MODE clause is specified in the SOURCE-COMPUTER paragraph, otherwise they are treated as comments. Furthermore when such debugging coding is compiled its use is still optional and can be included or excluded, as required, in each run. A run time switch is set or cleared by the operator or operating system.

Character handling

There are three new verbs which are designed for identifying and manipulating strings of characters (including strings of one, i.e. individual characters) within a data item. They are INSPECT, STRING and UNSTRING.

INSPECT which supersedes EXAMINE can count the number of occurrences of a particular string of characters and replace each occurrence by another string. The main advantages compared to EXAMINE are

(*a*) strings of characters rather than just single characters can be counted and replaced;

(*b*) the relevant characters can be specified by means of an identifier instead of a literal if required;

(*c*) the counting and replacing can be confined to part of the data item by specifying a delimiting character string;

(*d*) more than one character string can be counted or replaced in a data item by a single execution of an INSPECT statement.

Some of these capabilities are shown in **Fig. 4** where the statement replaces leading spaces in INPUT-VAL by zeroes and counts how many characters follow the decimal point storing the result in DEC-PLACES.

The STRING verb is used to combine two or more strings of characters. The example in **Fig. 5** shows how a date can be set up by combining three fields and two literals.

The entry immediately following DELIMITED BY indicates how much of the preceding data items are transferred to the destination field (in this case FULL-DATE). When SIZE is specified the full field is transferred (in this case both characters of DAY-IN-MONTH and a single space are placed at the

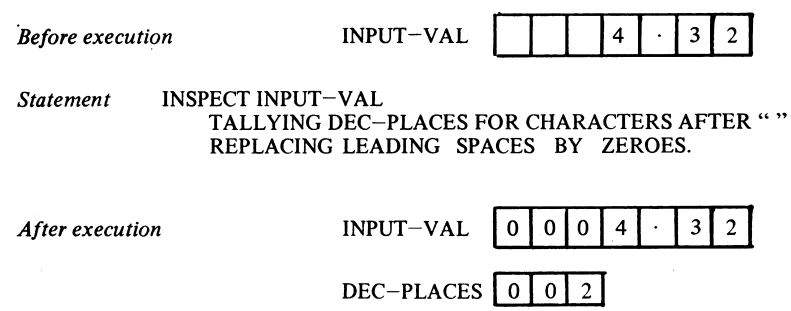

Fig. 4 Action of the INSPECT verb

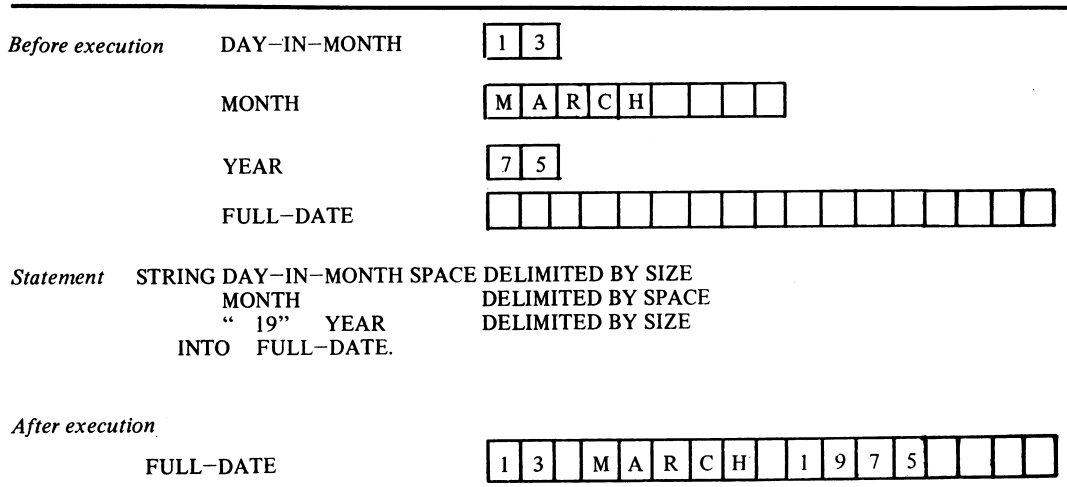

Fig. 5 An example of the STRING verb

beginning of FULL-DATE). When anything other than SIZE is specified it acts as a delimiter which when encountered, prevents any more characters being transferred (thus characters are transferred from MONTH to the next available positions in FULL-DATE until a space is encountered in MONTH whereupon characters are transferred from the next entry in this case "19"). Besides figurative constants the delimiter can be any non-numeric literal or the contents of an identifier. An additional feature not shown in the example is to define as a pointer a numeric data item. This will indicate the relative position in the destination field where the next character is to be inserted. Thus character strings can be copied into the middle of data items without disturbing the surrounding characters. In Fig. 5 the same result would have occurred if a pointer had been defined and preset to 1. In some situations it is possible for the destination field to become full before a STRING statement has successfully concluded. This condition can be detected by an OVERFLOW phrase.

The UNSTRING statement, as would be expected, performs the opposite activity of dividing a data item up into several separate strings of characters. An added bonus is the option of being provided with a count of the number of characters transferred to each substring. This verb will be of great value when reading in records which contain several variable length data items separated by identifiable characters.

Communications

The new communications module allows a COBOL object program to send messages to or receive messages from any remote devices (teletypes, visual display units, paper tape punches, card readers, printers or even other computers). It depends on the existence of a 'Message Control System' which performs similar managerial and device dependent tasks to the ones that an operating system performs for the local peripherals such as discs and card readers. After any editing or code conversion which is necessary the messages arriving at the computer are placed in a queue to await processing by a COBOL program. The user may specify several queues which can be used according to a variety of criteria (such as one queue for each device, or group of devices, or one queue for each application). A COBOL program can then obtain the next available message by specifying the relevant queue and executing a RECEIVE statement (just as the next disc record can be obtained by a READ statement). It is also possible to give the Message Control System a variable amount of freedom in selecting which queue the message is to be taken from. Conversely when records are to be transmitted to remote devices the COBOL program must specify the names of the relevant destinations and execute a SEND statement. This will cause the message generated by the program to be placed in a queue to await transmission by the Message Control System.

Although such action will often be taken outside the program it is nonetheless possible to logically connect and disconnect the remote devices by means of the ENABLE and DISABLE verbs. A facility that is, however, essential to the nature of telecommunications is the ability to initiate the execution of a COBOL program in two ways. Besides causing the program to be executed by normal means (such as by Job Control language) the program can also, if required, be initiated by the Message Control System when it discovers that there are messages which await processing by the program.

This module fills a gap which has existed in COBOL for too long. In so doing it has further eroded the concept that all input/output in COBOL is in the form of files. In place of SELECT and FD entries this facility has a CD (communication description) entry which works on completely different principles. Thus to transfer data out of a COBOL program in various situations the programmer can use WRITE, DISPLAY, RELEASE, GENERATE and SEND (and if the proposed database extensions are approved INSERT, MODIFY and STORE). The feeling that this approach to telecommunications could be improved upon is supported by the fact that at least one major manufacturer is omitting the Communications Module from an otherwise complete implementation of this standard.

Other changes

In addition to the substantial changes already described a number of smaller amendments have been made largely to remove anomalies and ambiguities. Some of the more significant ones are described in this section.

The programmer will now have more freedom in the way he writes his program. Spaces can precede punctuation (such as full stops and commas), the use of commas and semicolons is completely interchangeable and optional and comments can be placed anywhere in the program. To indicate that a line contains comments an asterisk is placed in column 7. The inflexible REMARKS and NOTE entries have been dropped.

Several changes have been made in the Data Division. Level 77 entries need no longer precede all other entries in the Working Storage Section. The character "/" can be used in a picture string as a simple insertion character like B and O. The OCCURS clause with the DEPENDING ON option is now restricted to use at the end of a record. This appreciably reduces the potential of this clause (see Triance, 1974). However, on the positive side the actual size of a variable lengthed record will be used whenever it is processed (within core as well as during input or output). Another enhancement to the data description entry is the new SIGN clause which allows the programmer to specify where and, to a certain extent, how an operational sign is to be stored in a numeric data item. This has previously only been possible with numeric edited items. Thus if the programmer specifies SIGN IS LEADING SEPARATE CHARACTER numbers such as -14 or $+23$ can be read in from such media as punched cards with the sign punched as a separate character. Reading negative numbers with the old Standard was machine dependent and in some cases extremely cumbersome.

Two changes in the Procedure Division which have not been mentioned yet are enhancements to ACCEPT and SET. ACCEPT can now be used to access the system's date and time and an index can be SET UP (or DOWN) by negative as well as positive amounts.

Finally the COPY statement has been made a lot more flexible. It may now be used anywhere that a COBOL word may appear. Furthermore groups of words, as well as individual words may be replaced while text is being copied from a library.

The problem of conversion

Anyone changing to a pure American National Standard (ANS) 74 compiler will face a conversion problem for existing programs because there are an appreciable number of features which an ANS 68 compiler will accept but an ANS 74 compiler will not. Worse still there are a few statements which will be equally acceptable to both compilers but will generate different object code. The reasons for the incompatibility are:

(*a*) the clarification of rules,

(*b*) the removal of facilities which would be of little use in the new Standard, and

(*c*) additions to the list of reserved words.

In general the clarification of rules has the effect of preventing programmers from doing things which were never intended in COBOL. For example the new standard states that SEARCH ALL (which is normally implemented as a binary search), will only work correctly on a table which is in the correct sequence as indicated by the ASCENDING/DESCENDING KEY

option of the OCCURS clause. Thus these clarifications are unlikely to cause serious problems when converting existing programs. This is because the average programmer, abiding by the spirit of COBOL as well as the rules, would not have broken the new rules and furthermore with most compilers it would not have been possible to do so anyway.

The facilities being dropped because they are of little use are those such as EXAMINE and NOTE, which are superseded and those which were not used much under the last standard. SEEK and user label processing fall into this category.

Each new facility in COBOL requires new reserved words. This standard introduces more than sixty of them, many of which (such as DATE, DESTINATION, LENGTH and POINTER) are used widely as data and procedure names. These will all have to be located and changed before a program can be re-compiled using an ANS 74 compiler.

Admittedly none of these problems need be faced until a program is re-compiled for some other reason (e.g. to remove a bug or implement a systems change). Even then re-compilation will cause no problems as long as the old compiler is available. But when support on the old compiler has been withdrawn the user could face an appreciable delay in re-compiling a faulty program if it had not already been converted.

Once the user has accepted the necessity of converting programs most of the changes will cause few problems because for them the conversion method is obvious and failure to convert will be highlighted upon compilation with the new compiler. Thus Remarks and Notes can be converted simply by placing an asterisk in column 7 of the lines in which they appear with some copying of the code necessary if any NOTE sentences share a line with other coding. EXAMINE statements can be replaced with an equivalent INSPECT statement. Any SEEK statement can simply be removed from the program and there are several other changes where the modification will require little effect.

However there are a few changes which could cause appreciable problems. Firstly there are existing facilities which have no exact equivalent in the new standard. Most notably is the superseded random access (based on the ACTUAL KEY). Most of these files can be replaced by either Relative or Indexed files. But problems will arise for those users who, for example, can place records on specific cylinders and need very fast access to every record (including overflows). Another major problem could be where an OCCURS DEPENDING clause has been used within the body of the record. Achieving the same effect by other means would be extremely cumbersome.

Another potential cause of significant problems is those changes which effect only run time behaviour. Thus the old coding compiles correctly but different results are produced. These problems result from the removal of ambiguities in the language and will not therefore affect all compilers. One example already mentioned is the use of the actual length of a record containing an OCCURS DEPENDING clause when the record is moved within the computer's core: many compilers move the maximum length. Another example is contained in the statement

```
PERFORM SALES-ANALYSIS
  VARYING DISTRICT FROM 1 BY 1 UNTIL DSITRICT >
    D-COUNT
  AFTER DAY-NO FROM FIRST-DAY BY 1 UNTIL DAY-NO
    > LAST DAY.
```

If FIRST-DAY is altered during the execution of this statement it can now affect the number of times the routine is performed since the latest value will be used for re-initialising DAY-NO.

The conversion problem described in this section is from a pure 1968 Standard compiler to a pure 1974 Standard compiler. In practice the move will be between two compilers which have extensions and, quite possibly, deviations. This actual conversion would be made worse if the writer of the old compiler had incorrectly anticipated the new standard with the result that such things as Indexed files and the Modular Programming statements had to be converted as well. However in general it is hoped that the compiler writer will ease the conversion problem. His main contribution would be to allow the defunct features to die out gracefully rather than abruptly removing them from the language. The accompanying manual should however indicate that these features are on the way out and should not be used in any new programs. Another service the compiler writer should offer is to issue warning disgnostics whenever coding is used for which the compiler will produce different object coding from its predecessor. An alternative approach would be for the old compiler to be supported in parallel for a reasonable period of time. This approach however results in extra work for the manufacturer and the user. In any case the user should ensure that the manufacturer takes appropriate action to ease the changeover to ANS COBOL 74.

Conclusions

The new standard undoubtedly makes COBOL a much more powerful language. It now encompasses all the major techniques currently popular in commercial programming with the notable exceptions of data bases and structured programming. Furthermore some steps have been taken to reduce the implementor's freedom which should, in the future, ease the transfer of programs between computers. On the negative side upward compatibility between the standards has not been maintained and it is to be hoped that the compiler writers will take steps to ease the transition where the standard has failed to do so. Finally COBOL has become a much bigger language without any major steps being taken to simplify it or make it more consistent. On balance, however, the new standard can be regarded as a considerable step forward for COBOL.

References

ANSI. *American National Standard Programming Language COBOL X*3.23-1974, American National Standards Institute, 1430 Broadway, New York, New York 10018 (Price $17).

ANSI. *American National Standard COBOL X*3.23-1968. American National Standards Institute, 1430 Broadway, New York, New York 10018.

TRIANCE, J. M. (1974). Handling records with a variable structure in COBOL, *The Computer Journal*, Vol. 17, No. 1, pp. 93-94.