The Proposed COBOL Standard—Its Significance

J. M. Triance

Computation Department, University of Manchester, Institute of Science and Technology, PO Box 88, Manchester M60 1QD, UK

A draft COBOL Standard has recently been published for public comment. The significance of the proposed Standard is examined by looking at the new features it contains and by studying the incompatibilities between it and ANS 74 COBOL.

INTRODUCTION

A draft revision of the current COBOL Standard, referred to as the *proposed standard* in this paper, has been published.[1] On the basis of previous practice the draft may be regarded as an accurate indication of what the next standard will contain when it is published in 1983 or 1984 (by current estimates). In view of the almost universal adherence to the standard by compiler writers it is also a good indication of the features that will be supported by COBOL compilers in the coming years.

This paper provides an introduction to and a commentary on the new features introduced in this revision; investigates the incompatibilities with the current standard;[2] examines the impact of the new standard on portability of programs between compilers; and looks at the opportunities the new standard has missed.

ADDITIONS TO THE LANGUAGE

The draft standard adds many features to COBOL. The most significant of these are (i) structured programming facilities, (ii) nested programs, (iii) the INITIALIZE statement, (iv) access to substrings, (v) the REPLACE statement, (vi) de-editing and (vii) a more flexible form of variable length record. A discussion of each of these and a summary of the other significant additions follows.

Structured programming

A reasonably comprehensive set of structured programming constructs have been added to COBOL. They include (a) conditional statement terminators, such as END–IF, (b) a null statement: CONTINUE, (c) a multi-branch (case-type) statement: EVALUATE, (d) an in-line looping construct: PERFORM with END–PERFORM, and (e) a 'test after' looping construct. Each of these is described and an overall assessment of the structured programming package is made.

Conditional statement terminators. The IF statement now has an explicit terminator—END–IF. Unlike the implicit terminator (full stop) it can be used to terminate an IF statement without at the same time terminating any other statements in which it is nested. Thus the END–IF on line 5 of Fig. 1 terminates the IF which starts on line 2 but does not terminate the IF which starts on line 1. This permits an IF statement to be followed by another statement (S3 in the example) within either branch of an outer IF statement. In this example (and the ones that follow) C1 and C2 represent COBOL conditions and S1, S2, etc. each represent an imperative statement. (Remember that an imperative statement may consist of a string of COBOL statements.)

```
1       IF C1
2       THEN IF C2
3            THEN S1
4            ELSE S2
5            END-IF
6            S3
7       ELSE S4
8       END-IF
```

Figure 1. END–IF in use.

The optional word THEN (which appears in Fig. 1) is also introduced in the new standard.

The problem of terminating nested statements arises with all conditional statements (READ with AT END, ADD with ON SIZE ERROR, etc.). These statements have all been given explicit terminators (END–READ, END–ADD, etc.). A full list appears in Fig. 2.

```
END-ADD         END-IF          END-SEARCH
END-CALL        END-MULTIPLY    END-START
END-COMPUTE     END-READ        END-STRING
END-DELETE      END-RECEIVE     END-SUBTRACT
END-DIVIDE      END-RETURN      END-UNSTRING
END-EVALUATE    END-REWRITE     END-WRITE
```

Figure 2. List of explicit terminators.

Null statement. The proposed standard provides a null operation statement—CONTINUE. It can be used anywhere that an imperative statement may be used but its main value will be for indicating that no action is desired in the true path of an IF statement.

In the example in Fig. 3 the CONTINUE on line 3 indicates that no action is required when C2 is true so control will pass directly to the next statement (on line 6). NEXT SENTENCE, as used in the current standard, would not meet this requirement since it would transfer control to the end of the sentence (somewhere beyond line 8). In fact NEXT SENTENCE may not be used when END–IF is specified.

CCC–0010–4620/83/0026–0060 $04.00

```
1       IF C1
2       THEN IF C2
3            THEN CONTINUE
4            ELSE S2
5            END-IF
6            S3
7       ELSE S4
8       END-IF
```

Figure 3. The CONTINUE statement.

Multi-branch statement. A multi-branch multi-join (case-type) statement has been added to COBOL. It is the EVALUATE statement. Any number of alternative actions may be specified, each accompanied by the condition under which it is executed. Optionally, a final action may be specified which is executed if none of the conditions are satisfied. Thus in the example in Fig. 4, S1 is executed if condition C1 is true, S2 is executed if C2 is true, and S3 is executed if neither C1 nor C2 is true. After executing the appropriate action (S1, S2 or S3 in the example) control is passed to the statement following the EVALUATE statement.

```
1       EVALUATE TRUE
2       WHEN C1 S1
3       WHEN C2 S2
4       WHEN OTHER S3
5       END-EVALUATE
```

Figure 4. An EVALUATE statement.

There is a great deal of flexibility in the form that the conditions may take. It is even possible to associate a set of conditions with each action. This is illustrated in Fig. 5 which shows a decision table and an equivalent EVALUATE statement. This form of EVALUATE statement, is obtained by turning the decision table

```
Decision Table:

    EXAM TAKEN          | 1  -  2
    EXAM PASSED         | Y  N  Y
   -----------------------------
    SEND REVISION       | X
    SEND SECTION 2      |    X
    SEND CERTIFICATE    |       X

Corresponding EVALUATE statement:

    EVALUATE EXAM-TAKEN EXAM-PASSED
    WHEN          1          TRUE     PERFORM SEND-REVISION
    WHEN        ANY          FALSE    PERFORM SEND-SECT-2
    WHEN          2          TRUE     PERFORM SEND-CERT
    END-EVALUATE
```

Figure 5. A more complex EVALUATE statement.

through 90 degrees, replacing the hyphen by ANY, Y by TRUE and N by FALSE. (In Fig. 5, EXAM-TAKEN identifies a numeric data item and EXAM-PASSED is a level 88 condition-name).

Thus in the EVALUATE statement in Fig. 5 SEND-REVISION is executed if EXAM-TAKEN = 1 and EXAM-PASSED is true, SEND-SECT-2 is executed if EXAM-PASSED is not true (regardless of the value of EXAM-TAKEN) and SEND-CERT is executed if EXAM-TAKEN = 2 and EXAM-PASSED is true. Conditions can thus be written in a tabular form but (unlike the decision table) there is no facility for expressing the actions in a tabular form.

In-line looping construct. A variation of the PERFORM verb is used to provide an in-line looping construct (see Fig. 6). The keyword PERFORM is immediately followed by the UNTIL phrase (no procedure-name is specified). This is followed by a string of statements terminated by END–PERFORM. This string of statements (S1 in the example) is executed repeatedly, but before each execution begins the condition (C1 in the example) is tested. When the condition is true control is transferred to the statement following END–PERFORM.

```
PERFORM UNTIL C1
S1
END-PERFORM
```

Figure 6. In-line PERFORM statement.

The other type of loop, in which the condition is not tested until after the first execution, is supported by the WITH TEST AFTER phrase. For example, in Fig. 7, the condition is tested after each execution of S1 (not before as in Fig. 6).

```
PERFORM WITH TEST AFTER UNTIL C1
S1
END-PERFORM
```

Figure 7. WITH TEST AFTER phrase.

The statements which are executed in the in-line PERFORM (S1 in the example) may contain other PERFORM statements or conditional statements with explicit terminators. (In the proposed standard IF with END–IF is classified as an imperative statement.)

Assessment. The new structured programming facilities overcome the main deficiencies in the current standard and thus represent a big step forward.

A major omission however are the alternative branches for the other conditional statements (other than IF, EVALUATE and SEARCH that is). Figure 8 shows the solution chosen by CODASYL.[3] In the proposed standard there is no entirely satisfactory way of representing this logic. However the CODASYL solution involves a

```
READ file-name
AT END S1
NOT AT END S2
END-READ
```

Figure 8. A CODASYL alternative branch.

lot of rather ungainly syntax (including double negatives, e.g. NOT INVALID KEY). The tidy solution to the problem would be a generalized exception handling mechanism which would allow all exceptions to be tested via the IF statement rather than by appendages to intrinsically imperative statements such as READ, ADD and STRING. Such a mechanism appears to be a long way off.

No effort has been made to satisfy the users of the Jackson Method[4] and its derivatives. In particular, there are no additions to support the posit construct, quit or inversion. ANSI have also missed the opportunity of having one verb for selections (like the Jackson select construct). Instead they have two verbs with completely different formats—one which can be used for any number of alternatives (EVALUATE) and another which is reserved for when there happens to be two alternatives (IF).

Finally, although the proposed standard permits the programmer to write structured programs (subject to the limitations mentioned above) it provides no means of enforcing this discipline: the 'unstructured facilities' (IF without END–IF, GO TO, etc.) remain in the proposed standard.

A more comprehensive discussion of structured programming in COBOL appears elsewhere.[5]

Inter-program communication

A number of additions have been made to COBOL's inter-program communication. The main ones are *nested programs* which are defined within other programs; *global data and global files* which can be defined in one program and accessed from within its nested programs; *external data and external files* which can be accessed from any programs in a run unit; *'by content' parameters* which are protected from corruption by the called program; and *initial programs* in which the variables are re-initialized each time the program is called. Each of these will be discussed in turn.

Nested programs. Nested programs are defined at the end of the Procedure Division of the containing program. Each nested program must be terminated with an END PROGRAM header. Thus, in Fig. 9, program P1 contains two programs—program P2 and program P3.

```
IDENTIFICATION DIVISION.                                  }
PROGRAM-ID. P1.                                           }
     :                                                    }
    CALL "P2"                                             }
     :                                                    }
    CALL "P3"                                             }
     :                                                    }
IDENTIFICATION DIVISION.                  }               }
PROGRAM-ID. P2.                           }               }
     :                                    } P2            } P1
     :                                    }               }
END PROGRAM P2.                           }               }
                                                          }
IDENTIFICATION DIVISION.                  }               }
PROGRAM-ID. P3.                           }               }
     :                                    } P3            }
     :                                    }               }
END PROGRAM P3.                           }               }
                                                          }
END PROGRAM P1.                                           }
```

Figure 9. Nested programs.

The nested programs may themselves contain nested programs. Figure 10 is a hierarchy chart for a program P1 which contains P2 and P3, P3 contains P4 and P5, P4 contains P6 and so on. PX is a separately compiled program of the type allowed by the current standard.

Each program may call any program immediately contained in it. Thus in the example P3 may call P4 and P5. Programs may, as in the current standard, call any separately compiled program. Thus in the example P1, P2 . . . P8 may each call PX. By using the COMMON clause in the PROGRAM-ID paragraph a program may be defined as a *common* program. A common program may be called by any programs nested (directly or indirectly) in the program which contains it, provided recursion does not result. Thus if P5 is a common program, in addition to P3, it may be called by P4 and P6 but *not* P5, P7 or P8.

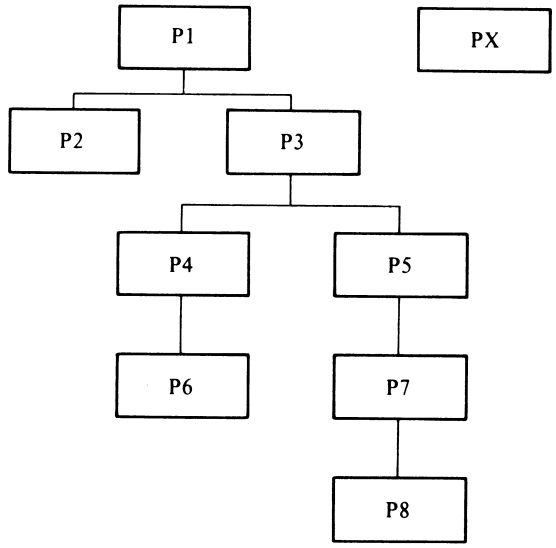

Figure 10. A sample run unit.

Global items. Records, files or reports may be made global by specifying the new GLOBAL clause in their definitions.

In the case of global records the record-name can be used to refer to the record from any program contained directly or indirectly in the program in which its definition appears. Thus if a global record is defined in P3 (see Fig. 10) it may be referenced P4, P5, P6, P7 and P8 (but not P1 or P2). It may also, of course, be referenced from P3. The data items within the record may also be referenced from all these programs.

Global files and reports are accessible to nested programs in accordance with the same rules.

External items. An external record is a record which is associated with the run unit rather than any particular program. Any program in the run unit may reference an external record provided that it is defined in that program with the EXTERNAL clause in its description. Every definition of an external record in a run unit must be identical apart from the possible use of the GLOBAL clause (an external item may be global in one program and local in another).

The EXTERNAL clause may also be used to make files external in the same sense as external records.

Call by content. In ANS 74 COBOL all parameters may be accessed and modified by a called program. In the proposed standard such parameters are known as 'BY REFERENCE'. In addition there is the 'BY CONTENT' parameter which can be accessed but *cannot* be modified in the called program.

Figure 11 gives an example of a CALL statement in the proposed standard. The called program, P1, will be

```
CALL "P1" USING BY REFERENCE I1 I2
                BY CONTENT I3 I4
```

Figure 11. By reference and content parameters.

able to access all the parameters I1, I2, I3 and I4. It will however only be able to modify parameters I1 and I2 (not I3 and I4).

Initial programs. The proposed standard has a new INITIAL clause in the PROGRAM-ID paragraph. When specified this ensures that all data items with VALUE clauses are initialized with the specified values each time the program is called.

The return links from (out-of-line) PERFORM statements are initialized between executions of a called program whether or not the INITIAL clause is specified. (Some ANS 74 compilers already do this.)

Assessment. The concepts of global and external files for the first time ensure that the programmer has a convenient and standard method of processing files in more than one subprogram. This is very welcome.

Global and external data items are completely new to called programs and it remains to be seen how the COBOL programmer will use them. The threat they pose to module independence will worry some purists.

Since the by content parameter and the initial program can be easily simulated in most ANS 74 COBOL programs these two features will not have a major impact.

The INITIALIZE statement

This new statement can be used to initialize a whole record or any category of data item in a record to specified values. The example in Fig. 12 moves zeroes to all numeric data items in SALES-TABLE (that is to each occurrence of SALES).

```
1   SALES-TABLE.
 2  PRODUCT  OCCURS 30.
  3 CODE-NO  PIC X(7).
  3 SALES    PIC S9(5).
    :
    :
    INITIALIZE SALES-TABLE
      REPLACING NUMERIC DATA BY ZERO
```

Figure 12. The INITIALIZE statement.

Any one of the five categories of data may be specified after the key word REPLACING: ALPHABETIC, ALPHANUMERIC, NUMERIC, ALPHANUMERIC-EDITED or NUMERIC-EDITED. The value to be used for initialization can be specified as a literal or an identifier.

If desired the whole REPLACING phrase may be omitted whereupon the numeric and numeric-edited items are set to zero and the other items are set to spaces.

Access to substrings

A new feature known as *reference modification* permits access to any string of consecutive characters in an alphanumeric data item. The reference takes the form of the name of the data item followed (in brackets and separated by a colon) by the starting point of the string and the length of the string.

Figure 13 gives an example of its use. If START-POSN contains 61 and NO-OF-CHARS contains 7 then the MOVE statement would access the 61st to 67th characters of ADDRESS.

This new feature will be very useful for data items whose structure is highly variable. It is, however, open to considerable abuse. A programmer could replace the record definition in Fig. 13 by

```
1   EMPLOYEE-RECORD
  2 EMPLOYEE-CODE    PIC X(6).
  2 DEPARTMENT-CODE  PIC X(4).
  2 ADDRESS          PIC X(72).
1   POST-CODE        PIC X(8).
    :
    :
    MOVE ADDRESS (START-POSN: NO-OF-CHARS)
      TO POST-CODE
```

Figure 13. Use of reference modification.

```
1 EMPLOYEE-RECORD  PIC X(82)
```

and, in the Procedure Division, access the department-code as EMPLOYEE-RECORD (7:4). This would seriously degrade the program's readability.

The REPLACE statement

The REPLACE statement is designed for source text editing. Like the REPLACING phrase of the COPY statement it may be used to replace one string of COBOL words by another. Unlike the COPY statement however the words being replaced may appear anywhere in the COBOL source, not just in text being copied in from a COBOL library. Three sample REPLACE statements appear in Fig. 14.

```
REPLACE ==CSR== BY ==COMP SYNC RIGHT==.

REPLACE ==EXTERNAL== BY ==EXTERNAL-X=
        ==CONTINUE== BY ==PROCEDE==.

REPLACE ==COMP== BY ==COMP-3==
        ==COMPUTATIONAL== BY =COMP-3==.
```

Figure 14. Some REPLACE statements.

The simplest format of REPLACE is

REPLACE ==text-1== BY ==text-2==.

where text-1 and text-2 are both strings of COBOL words. The effect of the statement is to scan the source text for occurrences of text-1 and replace each occurrence with text-2. The scanning begins at the code following the REPLACE statement and ends at the next REPLACE statement or the end of the program (whichever is next encountered).

A list of substitutions may be given in a single REPLACE statement (see second statement in Fig. 14) and if it is merely desired to turn off all previously specified substitutions

```
REPLACE OFF.
```

may be written.

This facility could be used to provide a shorthand facility or to aid conversion (e.g. translating data-names which have become reserved words or translating simple clauses). It, in fact, provides the facilities of a simple text editor without the ability to specify parts of words.

De-editing

In the proposed standard it is possible to move data from an edited data item to an unedited one. For example with

the data definitions in Fig. 15 it has always been possible to move STORED-VALUE to IO-VALUE. Now the move in the reverse direction, shown in Fig. 15, is allowed.

This facility will be useful for handling humanized input but it stops short of providing completely free-

```
1   IO-VALUE      PIC ---9.99.
1   STORED-VALUE  PIC S999V99.
    :
    :
    MOVE IO-VALUE TO STORED-VALUE
```

Figure 15. An example of de-editing.

format input. This added flexibility with edited items has *not* been extended to the arithmetic statements—arithmetic can still only be performed on numeric (unedited) items.

Variable length records

A new method of specifying the length of variable length records is permitted in the proposed standard. The length of each record in turn can be stored in a Working-Storage data item, the name of which is specified in the File Description Entry.

In Fig. 16, DELIVERY-DETAILS can contain anything from 4 to 54 characters of information so in the new RECORD clause in the FD entry the record length is specified as 10 to 60 (characters). The DEPENDING ON phrase is used to specify the data item (in this case DD-RECORD-LENGTH) which contains the length of the current record. This data item may not be defined in the File Section.

```
FD  DELIVERY-DETAILS-FILE
    RECORD IS VARYING IN SIZE FROM 10 TO 60
      DEPENDING ON DD-RECORD-LENGTH.
1   DELIVERY-DETAILS-RECORD.
  2 CUSTOMER-CODE    PIC X(6).
  2 DELIVERY-DETAILS PIC X(54).

WORKING-STORAGE SECTION.
1   DD-RECORD-LENGTH PIC 99.
      :
      :
    store lengtn of current record in
                          DD-RECORD-LENGTH
    WRITE DELIVERY-DETAILS-RECORD
```

Figure 16. New type of variable length record.

In the Procedure Division the length of each record must be calculated and stored in this data item prior to the execution of a WRITE statement for the file. Then only the specified number of characters are transferred to the file.

The same mechanism is used to determine how many characters are transferred by the INTO phrase in a READ statement.

This new mechanism is flexible enough to cope with any structure of variable length record. But as with reference modification the flexibility is achieved by making COBOL more low level.

Accompanying this new type of variable length record are some relaxations on the restrictions currently placed on variable length records. It will be possible to read a record and rewrite it with a different length in Indexed and Relative Files. It will also be possible to SORT and MERGE variable length record files.

Other significant changes

Setting condition-names. It is now possible to set flags explicitly. For example if END-OF-FILE is defined as a (level 88) condition-name it can be set to 'true' by the statement.

```
SET END-OF-FILE TO TRUE
```

The equivalent in ANS 74 COBOL is shown in Fig. 17.

```
1   END-OF-FILE-FLAG PIC 9.
 88 END-OF-FILE      VALUE 1.
    :
    :
    SET END-OF-FILE TO TRUE

 is equivalent to:

    MOVE 1 TO END-OF-FILE-FLAG
```

Figure 17. Setting a condition-name.

This feature can be used to significantly improve program readability. It is however a great shame that there is no feature for clearing condition-names (setting them to 'false').

Optional entries. Those who criticize the verbosity of COBOL will be pleased to discover that the following language elements are now optional: (i) The Environment Division, (ii) The Configuration Section, (iii) SOURCE-COMPUTER Paragraph, (iv) OBJECT-COMPUTER Paragraph, (v) The Data Division, (vi) The LABEL RECORD Clause, (vii) FILLER, and (viii) The Procedure Division. The omission of whole Divisions apart from the Identification Division is permitted in the case of nested programs.

SORT with DUPLICATES. The optional phrase WITH DUPLICATES IN ORDER has been added to the format of the SORT statement. When specified it ensures that records with equal key values are output in the same sequence as they are input.

Punctuation. The punctuation characters comma and semicolon are now interchangeable with the 'separator' space. Thus a comma or a semicolon may appear in place of or in addition to any space which is used to separate COBOL words.

User defined figurative constants. The programmer can now reference any character in the computer's character set. This is done by means of new clause in the SPECIAL-NAMES paragraph which identifies the desired character by means of its position in the collating sequence and assigns it a name. This name may then be used as a figurative constant. In Fig. 18 the name BELL is assigned to the 38th character in the computer's character set.

```
SPECIAL-NAMES.
    SYMBOLIC-CHARACTER BELL IS 38.
    :
    :
    DISPLAY "INVALID CUSTOMER CODE" BELL
```

Figure 18. User defined figurative constant.

DISPLAY WITH NO ADVANCING. It is now possible to suppress the automatic line advance that occurs after

data has been DISPLAYed on a line printer or VDU. The programmer simply writes WITH NO ADVANCING at the end of the DISPLAY statement.

INSPECT CONVERTING. A new format for INSPECT is introduced. It provides a more compact (but less readable) means of specifying replacement within a data-item when each of a set of individual characters is to be replaced by another individual character (see Fig. 19).

```
INSPECT ITEM CONVERTING
  "ABC" TO "XYZ"

is equivalent to:

INSPECT ITEM REPLACING
  "A" BY "X"
  "B" BY "Y"
  "C" BY "Z"
```

Figure 19. INSPECT CONVERTING.

DAY-OF-WEEK. An addition to the FROM phrase of ACCEPT permits a program to access a code number representing the day of the week. (1 for Monday, 2 for Tuesday up to 7 for Sunday). The statement ACCEPT DAY-CODE FROM DAY-OF-WEEK would, for example, store the appropriate code number in DAY-CODE.

Communication with a single terminal. A third format of the Communication Description Entry

CD . . . FOR I-O . . .

is introduced for communication with a single terminal. This simplifies programming for this rather common situation.

SEND REPLACING LINE. When a VDU (or other device to which messages are sent) permits both the superimposition of characters and the replacement of characters the programmer can now select the option he desires.

In such cases he can write

SEND cd-name AFTER ADVANCING 0 LINES REPLACING LINE

to replace the last tine sent with a new message. Alternatively he can omit REPLACING LINE to superimpose a new message on top of the previously sent line.

PURGE. The PURGE statement has been introduced for the purpose of deleting any partial messages released by the most recently executed SEND statement(s).

48 dimensional tables. It is now possible to define tables with up to 48 dimensions—the ANS 74 limit is 3. The main merit of this change is that 48 is somewhat less arbitrary than 3. It is the maximum possible in a COBOL record—an OCCURS clause on each level from level 2 to level 49 inclusive.

Padding partially used blocks. The new PADDING CHARACTER clause may be used in a Select Entry to specify a character for padding blocks in a sequential file. In an output file it is used to fill any unused space at the end of each block. On input any padding characters at the end of a block will be ignored. Its main use is likely to be for transferring sequential files between different machines.

Minor additions

The draft standard makes the following minor additions to the COBOL language. (i) EXIT PROGRAM may appear in the same paragraph as other statements; (ii) the optional word TO may be used in the ADD . . . GIVING statement; (iii) non-numeric literals may be 160 characters long (the current limit is 120); and (iv) parameters in a CALL statement need not be confined to level 1 or level 77 data items. There are a further 47 minor additions. None of them add significantly to the power or readability of COBOL.

INCOMPATIBILITIES

Programs written in ANS 74 COBOL will not necessarily conform to the proposed standard. Some features of the current standard have been deleted, the run-time behaviour of some features has changed and there are some new reserved words. Programmers are also forewarned of longer term incompatibilities.

Deletions from the language

The following features of ANS 74 COBOL have been deleted: the RERUN clause, the ALTER statement, the ENTER statement, the REVERSED option of OPEN, the MEMORY SIZE clause, and double Character substitution.

All of these features could be regarded as obsolete—either because they are inconsistent with current practice (like ALTER) or because they are superseded by other features supported by COBOL or the Operating System (e.g. the implementor is free to support other-language routines by means of CALL instead of ENTER). Another 13 less significant features have been deleted from COBOL. Some of these are features which no-one would envisage using—such as the ADVANCING PAGE option and END-OF-PAGE phrase in the same WRITE statement.

Changes to run time behaviour

There are 26 changes which could affect the run time behaviour of existing ANS 74 programs. The most significant of these are the following.

1. STOP RUN will close all files which are still open.
2. Lower case letters are regarded as alphabetic in the ALPHABETIC condition test.
3. EXIT PROGRAM initializes all PERFORM statement return links.
4. In a string of conditions linked by OR the conditions are evaluated from left to right and the evaluation stops as soon as a true condition is found. Thus the first condition can be used as a trap to prevent the second condition being executed in circumstances which would cause a run time exception.

In the first three examples ANS 74 did not define the run time behaviour. In the last case the rules seemed to imply that all the simple conditions (linked by ORs) should be evaluated before applying the OR(s)—many

compiler writers currently implement the new rule however.

Most COBOL programs will be unaffected by these changes to run time behaviour—either because the appropriate circumstances do not arise in the program or because the implementor already abides by the proposed rules.

New reserved words

The proposed standard introduces 57 new reserved words. Many of them will be in use as user-defined words in existing programs. Figure 20 lists some of the words which are likely to be in use as data-names and procedure-names.

ALPHANUMERIC	END-SEARCH	PADDING
ANY	EVALUATE	PURGE
COMMON	EXTERNAL	REFERENCE
CONTINUE	FALSE	REPLACE
CONVERSION	INITIALIZE	TEST
DEBUG-START	ORDER	TRUE

Figure 20. Some of the new reserved words.

Future incompatibilities

A new departure with this proposed standard is a 'transitional language element' list of those features which are scheduled for deletion in the next revision of the Standard (some time in the 1990s). Those installations that heed the warning will thus have plenty of time to phase out the transitional features. The following features are destined to fade away in this fashion: MULTIPLE FILE TAPE clause, DATA RECORDS clause, LABEL RECORDS clause, RECORD CONTAINS integer-1 TO integer-2 CHARACTERS clause, VALUE OF clause, INSPECT with TALLYING *and* REPLACING in the same statement, STOP literal statement and Debugging Declaratives.

The transitional approach is also being used to achieve a reshuffle of clauses between the Select Entry and the File Description (FD) Entry. The following clauses are transferred from the Select Entry to the FD Entry: ACCESS MODE, RECORD KEY, ALTERNATE RECORD KEY, and FILE STATUS.

The following clauses are transferred from the FD Entry to the Select Entry: BLOCK CONTAINS, and CODE-SET.

The purpose of this rationalization is to use the Select Entry to describe the physical aspects of the file and the FD Entry to describe the logical aspects.

During the life of the proposed standard these six clauses may be written in either entry but after that they will only be accepted in their new positions.

CONFORMANCE RULES

The rules which determine whether an implementation conforms to the standard have been tightened up.

Subsets

The number of approved subsets has been reduced from 104,976 to 54. Three levels of COBOL have been defined.

(i) *Minimum* which consists of level 1 Nucleus (which now includes table handling), sequential I-O and Inter-Program Communication (i.e. called programs).

(ii) *Intermediate* which consists of level 1 of Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, Segmentation, Sort-Merge and Source Text Manipulation (COPY and REPLACE).

(iii) *High* which consists of level 2 of all these modules.

The Debug, Report Writer and Communication (SEND and RECEIVE) modules are classified as optional. This means that an implementor can claim high level implementation (or any other level) without implementing any of these three modules.

Reserved words

All reserved words in the eight required modules (i.e. excluding the optional modules) must be supported as reserved words in all standard implementations even if the features in which the reserved word is used are not supported. This will aid portability between standard conforming implementations but since the optional modules and extensions to the standard are not covered the problem is only partially removed.

Extensions

As with the current standard an implementor may add any extension to the standard and still claim conformance to the standard. Now, however, the implementor is required to document all extensions to the standard.

Implementor defined elements

Some of the features which were not fully specified in ANS 74 COBOL have been specified in the proposed standard. The remaining sources of incompatibilities are listed in the standard under the headings *Implementor defined language elements*, *Hardware dependent language elements* and *Undefined language elements*.

MISSED OPPORTUNITIES

Missing features

The two most significant omissions from the proposed standard are database and screen handling facilities.

The CODASYL database facility was specified many years ago and a great deal of work has been done on its standardization. However, the current obstacle to standardization is the failure of the two ANSI committees to agree on compatible specifications for the COBOL Data Manipulation Language and the Schema. The problem of agreeing on a standard has been aggravated by disagreement on the type of database that should be supported—hierarchy, network, relational or some combination of the three.

Screen handling is much further from standardization. At the present rate of progress screen handling in COBOL is unlikely to be standardized this century.

What should be missing

The transitional language element list will in practice give programmers ten years notice of features to be deleted. Despite the existence of this safety net ANSI have been extremely cautious about removing COBOL's dead wood. The following items have been suggested for deletion but still remain in the language.

debug (D) lines
level 77 entries
CORRESPONDING option
EXIT (but not EXIT PROGRAM) statement
RENAMES clause
segmentation
SYNCHRONIZED clause

The level of COBOL

The proposed standard does not on average make COBOL a higher level language. The higher level features such as INITIALIZE and de-editing are balanced by the lower level features such as reference modification and the new form of variable length records.

The next round of standardization is likely to make more progress with, hopefully, the addition of database and validate facilities. There is also a possibility of support for other data processing functions such as update.

CONCLUSIONS

There is something of value for everyone in the proposed standard but, apart from the structured programming facility, no one feature is likely to make a big impact.

The proposed standard is not upward compatible with ANS 74. However a large and fast growing language like COBOL can only remain healthy if the dead wood is removed. If anything, the pruning could have been more severe. It is of course in the short-term interest of users to maintain compatibility at the expense of perpetuating an inferior language. The proposed incompatibilities are relatively modest however and major manufacturers are in any case likely to support a gradual transition from ANS 74 to the proposed standard.

The proposed standard is undoubtedly a step forward—but not a very big one. It has taken at least two years longer to produce less changes than the ANS 74 COBOL standard did. This slowing down of the standardization process and the failure to incorporate the database facility could mark the beginning of the end—if not for COBOL then for COBOL standardization.

Acknowledgement

The author thanks P. Brown and R. Grealish for their comments on the first draft of the paper.

REFERENCES

1. ANSI, Draft Proposed Revised X3.23 American National Standard Programming Language COBOL, American National Standards Institute (1981).
2. ANSI, American National Standard Programming Language COBOL X3.23-1974, American National Standards Institute (1974).
3. CODASYL, COBOL Journal of Development, Canadian Government, Department of Supply and Services (1981).
4. M. A. Jackson, *Principles of Program Design,* Academic Press, London (1975).
5. J. M. Triance, Structured Programming in COBOL—the Current Options. *The Computer Journal* **23** (No 3), 194 (1980).

Received March 1982