A MACRO FACILITY FOR COBOL

J. M. Triance,
Computation Department,
University of Manchester Institute
of Science and Technology,
Sackville Street,
Manchester, England.

Abstract

The British Computer Society's COBOL Specialist Group has designed a Macro Facility for COBOL for the purpose of permitting users to obtain a more up-to-date portable version of COBOL. This paper describes the facility and explores its implications.

INTRODUCTION

In terms of its wide usage COBOL must be regarded as the most successful of all the programming languages. This continued success can be largely attributed to two factors. Firstly, it is kept up-to-date: the current version of CODASYL COBOL [1] contains extensive facilities for Database Handling, Telecommunications and Structured Programming. Secondly, COBOL is, "all things to all men". For example, the addition of Structured Programming facilities will not prevent (or even discourage) those users, who so desire, from writing programs in the same unstructured way as has always been possible in COBOL.

But any advantages which CODASYL COBOL offers exist only in a diluted form in most compilers. It is intended as the ideal form of COBOL which will, in due course, be implemented on all relevant computers. But most users must wait until new features are standardised and then until their compiler is modified or rewritten. Typically this involves a delay of eight or nine years from when a facility is incorporated into CODASYL COBOL. Perhaps even more serious is the unbounded variation in this delay. At one extreme CODASYL can adopt a facility which is already provided as an extension in a particular implementation of COBOL and at the other extreme the same facility might never be supported in another implementation.

This has led to vast variations amongst the COBOL compilers currently in use. Some are based on the 1968 Standard [2] and some on the 1974 Standard [3]. Some are small subsets of the Standard, others full implementations with many extensions. But this vast selection of compilers is not available to the individual user since on many ranges of computer there is only one COBOL compiler available.

Problems arise when the compiler available is not suitable for the programming techniques used or is incompatible with other compilers which are also to be used. In fact, a measure of users' dissatisfaction with their compilers is provided by the large number of COBOL pre-processors currently available. The dissatisfaction is with features in the compiler which are regarded as undesirable as well as with the omission of features which are considered desirable. The undesirable features can, of course, be avoided by installation standards and in some cases by compiler options. On the other hand, extensions to the compiler can be handled by the pre-processors. But pre-processors can only be regarded as a stop-gap measure, not as the ideal solution.

A better solution would be for the user to configure his compiler so that it provides the version of COBOL he needs. Currently there is no mechanism for doing this, but the British Computer Society's COBOL Specialist Group has designed a suitable mechanism based on Macros. This differs from many other macro facilities [4] in that the macro calls are COBOL-like and the Macro Processor is an integral part of the compiler.

The Macro Facility was designed to meet the following objectives:

(1) It should not be necessary for the programmer to know which parts of the programming language are macro calls and which parts are pure COBOL.

(2) Macro definitions should be portable between different compilers and different computer systems.

(3) Macro definitions will be no more difficult to write, test and maintain than any COBOL program of comparable complexity.

THE MAIN FEATURES OF THE MACRO FACILITY

Macro Calls are COBOL-like

One of the requirements of the Macro Facility is that it can implement features of CODASYL COBOL which are not yet generally available. In order to satisfy this requirement the facility must accept any macro call which can be expressed in the COBOL Metalanguage, an example of which is shown in Figure 1.

```
SET index-name-1 [index-name-2] ...
    { UP   BY }   { identifier }
    { DOWN BY }   { literal    }
```

Figure 1. Example of COBOL Metalanguage

The implication of this for the applications programmer is that the macro calls will look like normal COBOL. For example, CODASYL COBOL's in-line PERFORM (Figure 2) could be a macro call with the condition (in this case IND > LIMIT) and the imperative-statement (SET IND UP BY INCR) as arguments.

```
PERFORM  UNTIL  IND  >  LIMIT
SET IND UP  BY INCR
END-PERFORM
```

Figure 2. Example of a Macro Call

As with any macro processor the macro call is replaced by equivalent coding referred to in this paper as Substitute Coding. The possible substitute coding for this example is shown in Figure 3.

```
    PERFORM  P1  UNTIL IND > LIMIT
    GO TO P2.
P1. SET IND UP BY INCR.
P2.
```

Figure 3. Substitute Coding for in-line PERFORM

Substitute Coding is Lower Level COBOL

The Substitute Coding is processed by the compiler as if it, rather than the macro call, had appeared in the source program. This takes place within the compiler unseen by the applications programmers who need not know any more about the Substitute Coding than they do about the object code.

In order to achieve portability of macros the Substitute Coding is always a lower level of COBOL. If this rule is interpreted strictly it is estimated that it would still be possible to represent more than half of CODASYL COBOL by means of macros including the Structured Programming facilities, the Report Writer, the String Processing verbs, SORT, MERGE and the more involved formats of many other verbs.

Macros may be Nested

When the Substitute Coding is processed by the compiler a search is made for further macro calls. For example, if SET IND UP BY INCR is a macro call with Substitute Coding ADD INCR TO IND, then after expanding the nested macro call the Substitute Coding in Figure 3 becomes :

```
    PERFORM P1 UNTIL IND > LIMIT
    GO TO P2.
P1. ADD INCR TO IND.
P2.
```

Figure 4. Expansion of a nested macro call

This process is repeated until all levels of nested macro calls have been expanded, at which stage the Substitute Coding is compiled in the normal manner. Recursive macro calls are also permitted.

The Macro Call Format is Expressed in COBOL

Each macro contains a description of the format of the macro call (known in some macro processors as a prototype or template). The obvious representation in this case would appear to be the COBOL Metalanguage (Figure 1) but this cannot readily be produced on a line-printer. The solution adopted is to represent the macro call format by means of a COBOL record as shown in Figure 5.

The record definition tells us that, to satisfy this format, a macro call must consist of four parts (corresponding to the four level 3's). The first part must be the words PERFORM UNTIL, the second part must be a condition, the third part must be an imperative statement and the final part must be the word END-PERFORM. The USAGE ARGUMENT and CLASS clauses are extensions to COBOL which will be explained later.

```
PERFORM UNTIL condition
imperative-statement
END-PERFORM
1     IN-LINE-PERFORM.
  3   PERF-UNTIL PIC X(14).
   88 P-U                  VALUE "PERFORM UNTIL ".
  3   COND       USAGE ARGUMENT.
   88 CD                   CLASS "CONDITION".
  3   IMP-ST     USAGE ARGUMENT.
   88 I-S                  CLASS "IMPERATIVE-STATEMENT".
  3   END-PERF   PIC X(12).
   88 E-P                  VALUE " END-PERFORM".
```

Figure 5. Macro Call Format in Metalanguage and Equivalent COBOL

Figure 6 shows how all the metalanguage constructs used in ANS 74 can be represented. A new clause OPTIONAL is introduced for optional parts of macro calls. There is also a need for an UNORDERED clause for the case when the sequence of components in a macro call is immaterial.

COBOL Meta-language	Meaning	Representation
CAPS	Key words	VALUE clause in level 88
CAPS	optional words	VALUE clause in level 88 following OPTIONAL clause
lower-case	semantic-class	CLASS clause in level 88
[ ]	optional clause	OPTIONAL clause
{ }	alternatives	REDEFINES clause for each alternative after the first or (when all alternatives are semantic classes) consecutive level 88's with CLASS clauses
...	repetitions	OCCURS clause with DEPENDING option

Figure 6. Method of Representation Formats of Macro Calls

The macro call formats are stored in a file and used by the macro processing phase of the compiler to identify macro calls.

Each Macro is a Cobol Subprogram

Most macro processors employ a special language for writing the macro definitions. However, given the language extensions highlighted in the preceding section COBOL seems to be just as well suited for writing macros as it is for writing applications programs. It has the distinct advantages over existing special purpose languages of

(1) avoiding the need for extra training for the macro writers;

(2) being a complete and self documenting language; and

(3) being portable.

The fact that it is verbose and will lead to unnecessarily long macro definitions would seem unimportant when we consider that most COBOL users are willing to sacrifice brevity in return for readability.

The macro definition is written as a sub-program to achieve the greatest possible independence from other macro definitions. An example of the macro definition for the in-line PERFORM is shown in Figure 7.

```
 1 IDENTIFICATION DIVISION.
 2 PROGRAM-ID. PERFORM-FORMAT-5
 3             CLASS IS "IMPERATIVE-STATEMENT".
 4 ENVIRONMENT DIVISION.
 5 INPUT-OUTPUT SECTION.
 6 FILE-CONTROL.
 7     SELECT SUBSTITUTE-CODE ASSIGN SUBCODE.
 8 DATA DIVISION.
 9 FILE SECTION.
10 FD    SUBSTITUTE-CODE.
11 1     SUBST-CODE-REC.
12   3   SUBST-ST   PIC X(17) VALUE     "PERFORM P1 UNTIL ".
13   3   SUBST-COND USAGE ARGUMENT.
14   3   SUBST-MID  PIC X(15) VALUE    " GO TO P2.
15-                                   "P1. ".
16   3   SUBST-STMT USAGE ARGUMENT.
17   3   SUBST-END  PIC X(6)  VALUE      ".
18-                                   "P2.  ".
19 MACRO-LINKAGE SECTION.
20 1     IN-LINE-PERFORM.
21   3   PERF-UNTIL PIC X(14).
22    88 P-U                  VALUE "PERFORM UNTIL ".
23   3   COND       USAGE ARGUMENT.
24    88 CD                   CLASS "CONDITION".
25   3   IMP-ST     USAGE ARGUMENT.
26    88 I-S                  CLASS "IMPERATIVE-STATEMENT".
27   3   END-PERF   PIC X(12).
28    88 E-P                  VALUE "END-PERFORM ".
29 PROCEDURE DIVISION USING IN-LINE-PERFORM.
30 SET-UP-SUBST-CODE.
31     MOVE COND TO SUBST-COND.
32     MOVE IMP-ST TO SUBST-STMT.
33     WRITE SUBST-CODE-REC.
34 END-MACRO.
35     EXIT PROGRAM.
```

Figure 7. A Macro Definition

This macro definition is compiled and placed in a library. When, subsequently, a call for the macro is encountered in an applications program the appropriate macro definition is executed after the details of the call have been placed in the Macro Linkage Section. (It will be noted that this Section contains the macro call format previously shown in Figure 5.)

Thus for the Macro Call shown in Figure 2 the contents of the Macro Linkage Section will be as follows :

field	contents
PERF-UNTIL	PERFORM UNTIL
COND	IND > LIMIT
IMP-ST	SET IND UP BY INCR
END-PERF	END-PERFORM

The Procedure Division coding uses these data items to set up the Substitute Coding and write it to the Substitute Coding File before returning control to the Macro Processor. The contents of this file are then processed as if they had been part of the original source program.
The non-standard use of VALUE in the File Section is explained later.

The Class of the Arguments is Specified

Most macro processors process arguments as strings of text and leave any error checking to the macro writer. This Macro Facility, however, requires the macro writer to specify explicitly the class of each argument. In Figure 5 the classes were CONDITION and IMPERATIVE-STATEMENT but in general could be any syntactic type in COBOL. This scheme which could be regarded as a development of Levenworth's Syntax Macros [5] offers many advantages:

(1) Greater precision in the identification of macro calls is possible. For example,

```
PERFORM UNTIL STOP RUN
SET IND UP BY INCR
END-PERFORM
```

would not be accepted as a valid call of our sample macro since STOP RUN is not a condition.
This precision permits the early detection of errors and the generation of error diagnostics relating to the original macro call rather than to the Substitute Coding. This is vital if the error messages are to be meaningful to the applications programmer.

(2) Greater precision is possible in the generation of Substitute Coding. Imagine that, for a compiler which does not support the option, a macro is being defined for PERFORM with the VARYING option:

```
PERFORM procedure-name-1 [THRU procedure-name-2]
  VARYING {identifier-1 } FROM {identifier-2 }
          {index-name-1 }      {literal-1    }
                               {index-name-2 }
  BY {identifier-3} UNTIL condition.
     {literal-2   }
```

In the Macro Linkage Section the argument following VARYING will be defined as

```
3   COUNTER-ITEM   USAGE ARGUMENT.
 88 ID             CLASS "IDENTIFIER".
 88 IND            CLASS "INDEX-NAME".
```

to indicate that the argument is either an identifier or an index-name. The condition-names ID and IND can then be used in the normal way to discover whether

the Substitute Coding should contain MOVE's and ADD's or SET verbs for initialising and incrementing the counter.

(3) Macro calls may be identified more efficiently. Since an argument of a macro call may itself contain macro calls it is necessary for each macro to have a class associated with it to permit full validation of the argument. The class of the macro is specified by the CLASS clause in the PROGRAM-ID of each macro definition. Having been specified this information can then be used by the Macro Processor to ensure that the source text is scanned only for Macro Calls of a permissible class in each context.

All Macros are Stored in a Library in a Separate Run

Before macros can be used they must be submitted to a library run. This extracts the macro call format from the Macro Linkage Section and stores it in a Macro Call Formats File. At the same time a Reserved Word File is updated to include any additional reserved words. Finally the macro definition is compiled and stored in a macro library file. These three files, when used in conjunction with the compiler's inbuilt features, determine the language which is available for applications programs.

IMPLEMENTATION CONSIDERATIONS

Handling Variable Lengthed Arguments

Most arguments are variable in length without any moderate maximum size. Even an identifier in CODASYL COBOL can exceed 72,000 characters without counting the embedded spaces. Since COBOL isn't well equipped to handle such fields the solution adopted is to store a pointer to the argument in the fields described as USAGE ARGUMENT rather than the argument itself. When this pointer is transferred to the Substitute Coding the Macro Processor will replace it by the original argument. The Macro Writer will not be able to access the actual argument.

The length of fields described as USAGE ARGUMENT would be implementor defined with the result that the macro writer should make no assumptions about them. But the macro writer must define the correctly lengthed Substitute Coding File record. The preferred solution to this problem is to set up the Substitute Coding in the File Section rather than in the Working-Storage Section and to enhance COBOL to treat VALUE's in the File Section in the same way as in the Report Section. In other words the field is automatically initialised immediately before writing each record. If this approach is unacceptable to CODASYL various other solutions are possible including using MOVE statements to achieve the same effect.

Distribution of Substitute Coding

Not all Substitute Coding will belong in the same physical location as the macro call. The actual location (e.g. Working Storage, End of Procedure Division) is indicated at the start of each Substitute Coding record. (This indicator was omitted from the example in Figure 7 to avoid confusion.) Error diagnostics are also output to the Sub-

stitute Coding File with an indicator to signify that they should appear with the compilation error list.

Accessing Data Definitions

In some macro definitions it will be necessary to access the data definition of identifiers which are arguments. For example a macro for the CODASYL COBOL verb

INITIALISE identifier

would need to establish whether identifier was numeric or not to determine whether the Substitute Coding should move zeros or spaces to the identifier.

The data descriptions are all stored in a compiler generated indexed file and the data definition of any item can be accessed by using its identifier as the key.

Generating Unique Names

The Macro Definition in Figure 7 would not work if it was called twice in the same program because two sets of identical paragraph-names P1 & P2 would be produced. This is obviously unacceptable.

The problem is overcome by appending the desired combination of the following special registers to such names :

MACRO-ID - the unique identifier of the macro definition currently being executed

CALLS-OF-CURRENT - the number of times the current macro has been called in the current source program

CALLS-OF-ALL - the total number of calls of all macros in the current source program.

Each of the special registers also contains a symbol which will make any of these generated names distinct from names defined in the original source text. The Macro Writer can also obtain a unique name by terminating any name in the Substitute Coding with a hyphen, whereupon CALLS-OF-ALL will automatically be appended.

IMPLICATIONS OF THE FACILITY

The Envisaged Use of Macros

The primary purpose of the facility is to allow each installation to configure its compiler to support any subset of CODASYL COBOL that is desired. The examples given so far have shown how a compiler which falls short of CODASYL COBOL can be extended by defining macros. Since a macro for a given construct has priority over the same construct supported by the compiler itself it is also possible to override the implementation provided by the compiler. This is of use in avoiding compiler bugs and non-standard implementations or in prohibiting or highlighting features which are considered undesirable. For example, if the installation standards banned the use of the ALTER verb a macro could be defined for ALTER which would produce a fatal diagnostic message but no Substitute Coding.

When transferring COBOL programs from one compiler to another, the Macros can be used to achieve compatibility. Any desired features of COBOL which are lacking on both compilers will normally be provided on the new compiler by the same macro as on the old compiler. But when the Substitute Coding includes incompatible features there will be a need for a new version of the macro which generates different Substitute Code. If the new compiler supports some features that were previously supplied by macros then the relevant macros can be discarded. If, on the other hand, the new compiler lacks some of the desirable features which were built into the old compiler then additional macros will be needed. In all cases the applications programs can remain unaltered.

In effect these steps will make the compilers upward compatible. But in some cases (such as a permanent move from an ANS 68 Compiler to an ANS 74 Compiler) it might be preferable to actually translate some of the constructs in the applications programs once and for all. Macros could also be used for this provided the compiler offers the option of outputting the applications programs with the macro calls replaced by substitute coding (in other words, acting like a pre-processor). Then, for example, all EXAMINE statements (ANS 68) could be replaced by the equivalent INSPECT statements (ANS 74).

The Macros required for the various purposes described above would be fairly standard and would probably be supplied by manufacturers, software houses or other bodies. There is, of course, nothing to stop the individual user from writing his own macros. If the user restricts himself to implementing a subset of CODASYL COBOL the process will no doubt be onerous in some cases but will be relatively straightforward. The user could go a step further and design his own extensions to COBOL. If done on any scale this would probably require a similar commitment and degree of central control as is necessary for setting up a data base.

Whoever writes the macros it is essential that they are thoroughly tested. Since the applications programmers regard the macros as part of the compiler the programmers must be able to place as much trust in the macro definitions as they would in a good compiler.

Impact on COBOL Development

The addition of the macro facility requires surprisingly few enhancements to COBOL. The proposed changes, which only effect the macro definitions, involve eight minor additions to the language and the relaxation of rules on four existing features. There is, however, a far bigger impact on the way that compilers are written.

With the help of Macros, users would be able to update their versions of COBOL at their own speed. If, for example, level 77's are dropped from the latest release of a compiler, the users could continue to use them via a macro for as long as was desired. This should remove from CODASYL the spectre of universal disapproval when-

ever they do anything to stop COBOL being upward compatible. Thus decisons to delete or amend features of the language could be based more on merit and less on the extent to which existing programs depend on the feature. In the other direction macros could also serve CODASYL members by allowing them to test some of the proposed additions prior to incorporation in the language.

The Macro facility could also simplify the present system of subsetting COBOL in the American National Standards. In a standard which contained the macro facility all the features could be divided into two classes - those which could be written as macros and the remaining set of "elementary" features. Hopefully this class of elementary features would be small enough to regard as the minimum subset. Unlike the present minimum subset this one would contain no redundant features and would be sufficient for any application for which full COBOL is sufficient. The rest of the language would merely provide a more convenient method of achieving the same results.

There would be two subsets in this Standard. One would be the minimum subset described above and the other would be the minimum subset plus the macro facility. With this macro facility the rest of the language could be provided by macros if necessary. Any of these remaining features which were supported directly by the compiler would not increase the power of language but could increase its efficiency.

Possible Drawbacks

There appear to be two potential drawbacks with macros. The most serious for COBOL is probably the danger that the language will diverge into a large number of incompatible dialects. At worst, there is the worry that every programmer might define a different set of ill-conceived macros for each program. The fact that all macros must be placed in a central library makes this worst situation infeasible. No doubt some of the larger installations will devise their own dialect of COBOL but these are the type of installation which is already doing just that by means of pre-processors.

The other drawback traditionally associated with Macros is in the area of efficiency. In particular Macros will tend to slow down compilations. This facility includes some features which should alleviate this problem: the macro definitions are executed not interpreted and unduly sophisticated features are avoided. Despite this it is likely that some users will find the overhead of macros too great just as some find certain current features of COBOL too extravagant. Such cases are likely to be in the minority and for other users the feasibility of macro processors with COBOL has been well demonstrated with the Cobra [6] and Meta-Cobol [7] pre-processors.

Planned Developments

A pilot implementation of the facility is in progress at UMIST. The objective is to demonstrate its functional feasibility. It is hoped that this will be followed by a full systems study and a complete implementation which would indicate its economic viability. The long term objective is its incorporation into CODASYL and then ANS

COBOL and its general availability in COBOL compilers.

Conclusions

There are problems associated with the use of Macros but they are ones which have been overcome to the satisfaction of current users of macro pre-processors. The incorporation of macros into COBOL overcomes shortcomings inherent in pre-processors and could offer all users the opportunity to select their own version of COBOL. This would allow COBOL, at last, to satisfy its design objectives of being an up-to-date portable language.

Acknowledgements

The macro facility described in this paper was designed by a working group consisting of K. H. Meyer of British Gas, A. Morrison of the Central Computer Agency, A. E. Sale of Alpha Systems Limited, J. E. Sawbridge of Plessey and Chaired by the author. In addition, J. Yow of UMIST is implementing the pilot system in consultation with R. M. Gallimore of UMIST and the author.

The opinions expressed in this paper are not necessarily shared by those named above.

References

1. CODASYL (1978). COBOL Journal of Development 1978, Canadian Government, Department of Supply and Services.
2. ANSI (1968), American National Standard Cobol X3.23-1968, American National Standards Institute.
3. ANSI (1974). American National Standard Programming Language Cobol X3.23-1974, American National Standards Institute.
4. BROWN, P. J. (1974). Macro Processors and Techniques for Portable Software, Wiley.
5. LEVENWORTH, B.M. (1966). Syntax Macros and Extended Translation. C.A.C.M., Vol. 9, pp.790-93, Nov. '66.
6. HAMILTON, J.G.A., FINLAYSON, E.D., HEYWOOD-JONES, A.H. (1973). Computer-aided Program Production, Datafair 1973.
7. ADR (1976). MetaCOBOL Concepts and Facilities, Applied Data Research.