Macro Processors for Enhancing High-Level Languages – Some Design Principles*

J. M. TRIANCE AND P. J. LAYZELL†
Department of Computation, University of Manchester Institute of Science and Technology, P.O. Box 88, Manchester M60 1QD

There is a requirement for users to configure programming languages to meet their own particular needs. This requirement can be met by a suitable macro processor. In general existing macro processors are unsuitable because they are not sufficiently supportive towards the popular programming languages.

Alternative design strategies for such a macro processor are examined and a set of design principles is derived.

1. INTRODUCTION

The great emphasis which has been placed on programming language standardisation has obscured the individual programming language requirements of users. However good the standard is it will be unsuitable for all or most of the users of the language. This point is illustrated by the COBOL standard, which has received far more attention than any other programming language standard. Despite this there are in existence hundreds of COBOL dialects supported by different compilers, pre-processors and installation standards. Their existence is a symptom of failure of COBOL to meet the individual needs of the users.

The full extent of this problem and the possible solutions are discussed elsewhere.[19] The main solutions are source text editing, subroutines, special purpose pre-processors and macro processors. Of these only macros offer the power needed to handle all types of language variation, while at the same time permitting each enhancement to be implemented independently of other unrelated enhancements. This independence is vital if users are to be permitted to choose their own combination of language enhancements freely.

The use of macros for language enhancement (or extensibility) is by no means a new concept.[5, 8, 11] Most of the existing schemes have however failed to obtain widespread acceptance. The reasons for this are that many users regard the concept as revolutionary, a significant learning curve is involved in the use of macros and many existing schemes are low level.

The conceptual barrier is beginning to break down with the widespread use of pre-processors, including macro processors such as MetaCOBOL,[1] Cobra[10] and Delta.[15] Solutions to the problems of the learning curve and low-level macro schemes are sought in this paper.

This paper:

(*a*) introduces the concept and terminology of macros;

(*b*) establishes a set of design criteria for language-enhancement macro schemes;

(*c*) identifies the major aspects of language enhancement;

(*d*) derives a set of design principles for language-enhancement macro schemes.

Where examples are used they are drawn from the language COBOL. This is a good source of examples because COBOL is a large and varied language which illustrates most of the problems encountered in other languages.

* Computation Department Report No. 271.

† Correspondence should be addressed to Dr Layzell.

2. CONCEPTS AND TERMINOLOGY

This section introduces the concept of macro processing and the terminology used in this paper. The starting points for a language-enhancement macro scheme are a base language: a language which already exists and is supported (by a compiler or interpreter); and a requirement to enhance the base language by adding new syntax, deleting existing syntax or altering the semantics of existing syntax.

Each enhancement is supported by a *macro* (or macro definition) which translates the enhancement into base-language statements.

For example, Fig. 1 shows a simplified version of the PERFORM WITH TEST AFTER statement which is included in the draft COBOL standard.[3] It could be supported by a macro which translates it into ANS 74 COBOL[2] (see Fig. 2).

PERFORM procedure-name
WITH **TEST AFTER**
UNTIL condition

Figure 1. An enhancement

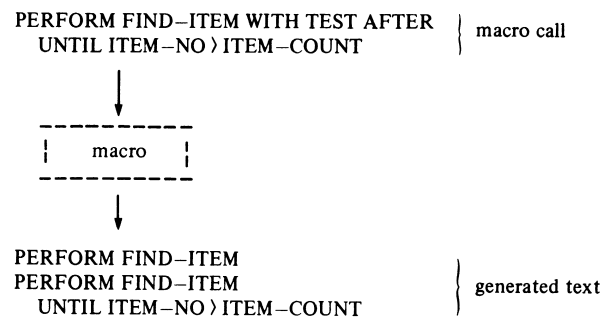


Figure 2. Action of PERFORM WITH TEST AFTER macro

The input to the macro is a *macro call* (an instance of the enhancement). The output is known as *generated text*. The language used for the generated text is known as the *target language*.

The macro call, in general, contains some static parts (such as the words PERFORM, WITH TEST AFTER and UNTIL in the above example) known as *delimiters* and variable parts (the procedure name and the

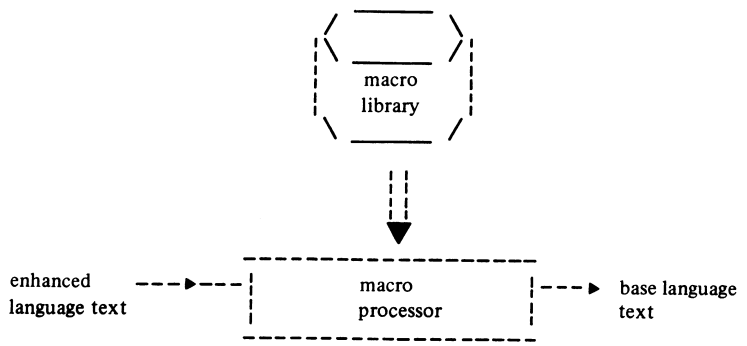


Figure 3. Action of the macro processor

condition) which are known as *arguments* (or *parameters*).

So that the various macro calls can be recognised each macro must have a description of the syntax it supports. This description is known as the *macro template* (or *prototype*). Thus in the example above the macro template would represent the format given in Fig. 1.

The recognition of macro calls is the job of the *macro processor* which scans the source text identifying each macro call and passing it to the appropriate macro. The macros are stored in a *macro library*, and by invoking the macros as required the macro processor translates the *enhanced language* text (containing macro calls) into *base language text* (see Fig. 3).

3. DESIGN CRITERIA

A set of design criteria for a language enhancement scheme can be derived from the requirements of its users. The enhancement of a base language should have no adverse effect on the language in terms of its consistency, level of source text validation or portability. In addition the language used for specifying the enhancements (the *macro writing language*) should attain the standards expected of any modern programming language: it should be high-level, easy to learn and modular. The following design criteria can thus be identified.

(1) *Consistency of the enhanced language*. The enhanced language should display a consistent style: the macro calls should blend into the rest of the language. The applications programmer should ideally be unaware of the existence of the macro processor.

(2) *Full support of enhancements*. The enhancements should be supported to the same level as the rest of the language. They should also be supported in a consistent manner to the base language statements. This implies that the enhancements are fully validated and that any errors are reported in terms of the source text (i.e. the macro call) – not the generated text.

(3) *Portability of macros*. The macro writing language should be as portable between machines (which support the macro scheme) as the base language is. Since one of the major applications of macros is to support portable dialects of a language it is important that the macro scheme itself does not hinder portability.

(4) *High-level macro writing language*. The macro writing language should allow the various aspects of macro definition to be expressed in a natural way and at minimum effort to the macro writer.

(5) *Easy-to-learn macro writing language*. The macro writer should not face a major task in learning the macro writing language.

(6) *Modular macro writing language*. The macro writing language should permit enhancements to be implemented independently of other unrelated enhancements.

These criteria will be used to establish a set of design principles for language-enhancement macro processors.

4. ASPECTS OF LANGUAGE ENHANCEMENT

When designing a macro processor for language enhancement (subsequently referred to as *the macro processor*) the following topics must be considered: the scope of the macro scheme; the evolution of the enhanced language; macro call recognition; specification of the template; generated text; independence; access to the symbol table; the macro writing language.

In each of these areas a number of approaches will be discussed and 'design principles' established.

The macro writing language is discussed last because a choice of language cannot be sensibly made until a list of requirements has been established. The recommendation made at that point in the paper is that the base language (with a few extensions) should be used as the macro writing language. The reader might find it helpful to bear this in mind when reading the sections on the other topics.

5. SCOPE OF THE MACRO SCHEME

The scope of a macro scheme must be defined by specifying the base languages and the types of enhancement it will support.

This paper argues that the macro processor should be designed for one base language (or possibly a family of languages) and that limitations have to be placed on the type of enhancement supported.

5.1 Language dependent versus language independent

In establishing a strategy for macro processing one of the following must be chosen:

(*a*) design a language-independent macro scheme for use with all programming languages; *or*

(*b*) design a language-dependent macro processor for each programming language (or family of programming languages).

Since this paper advocates a general-purpose method

of language enhancement (the macro) it might appear that consistency would demand a general-purpose (language independent) macro processor. However, generality has to be paid for; there are overheads in performance and ease of use. The first level of generality (the macro) is clearly needed because the typical user will want a variety of language enhancements (e.g. database, structured programming and screen handling). A user does not however normally wish to enhance more than one language – so the second level of generality is a minority requirement. In more detail the disadvantages of the language-independent solutions are:

(*a*) generality normally results in some degradation in performance – two levels of generality could result in intolerable degradation;
(*b*) to provide high-level support for the macro writer the language – independent macro processor would have to be separately configured to understand each base language: for languages with completely different structures and lexical forms (e.g. continuation and literal rules) this would be a significant task.

In the existing world of computing the balance is in favour of the language-dependent macro processor.

> Design Principle 1. The macro facility should be designed for a single base language.

5.2 Limitations on enhancements

Macros are very powerful mechanisms, and because of this designers may fall into the trap of placing no explicit limitation on the enhancements supported by the scheme. Such a decision would be incompatible with the design criteria outlined above. Any macro scheme which has the power implied by this decision would of necessity have the capabilities, and therefore the complexity of a compiler.

> Design Principle 2. The requirements placed on the macro processor should be limited.

5.2.1 *Specification of style and philosophy of the enhanced language*

For a programming language to be usable, its structure and the philosophy upon which it is based must be understood by the programmer. Of course a macro scheme can support any number of enhanced languages, but if the scheme is to be easy to use they must all bear some relationship in structure and philosophy to the base language. By defining the structure and philosophy which is common to all enhanced languages the scope of the scheme, and therefore its complexity, will be limited.

For example, in the case of COBOL the following aspects could be considered to be invariant, and thus apply to all enhanced versions of COBOL: data items and files referenced in the program are all defined in the data division; each program is made up of divisions, sections, paragraphs, etc.; the language is free format (within the constraints of areas A and B); the standard COBOL conventions apply to the choice of user defined-names.

5.2.2 *Specification of permitted enhancements*

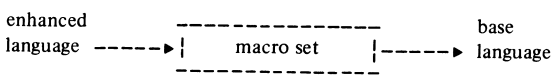
The design of a macro scheme obviously depends on the types of enhancement it supports. It is therefore necessary to make a clear statement of the permitted enhancements.

Most macro schemes would for example support: the addition of new syntax, the deletion of existing syntax and new semantics for existing syntax. On the other hand they might not support new lexical entities and layout-dependent features.

In the case of COBOL these limitations rule out less than 10% of the desired enhancements[13, 17] but would significantly reduce the complexity of a macro scheme.

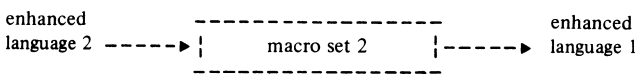
6. EVOLUTION OF THE ENHANCED LANGUAGE

An enhanced language is supported by producing a set of macros which will translate the enhancements into the base language:

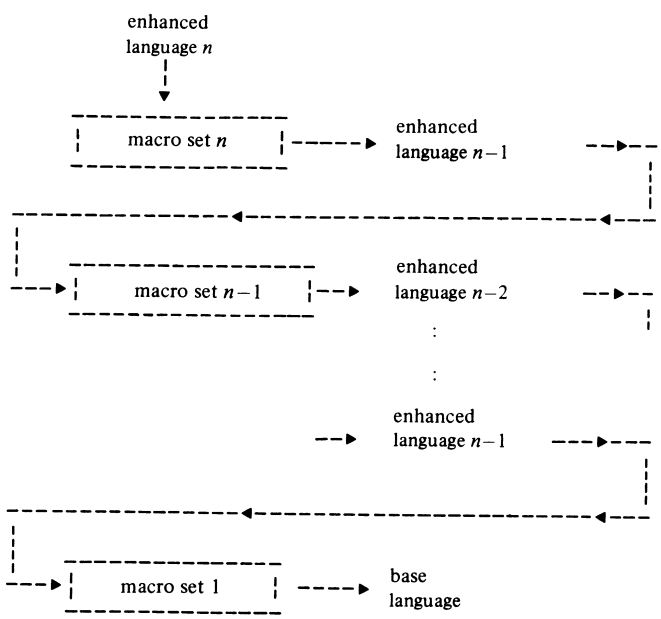


The development of the language will not normally stop there – further enhancements will be devised and further macros will be written.

Having produced an enhanced language it would be foolish to require macro writers to use the 'inferior' base language for the generated text. Why should they be denied the benefits offered to the applications programmers? Thus the enhanced language (now labelled enhanced language 1) should become the target language for the next set of macros.



In general each enhanced language becomes the target language for the next set of macros.



Thus generated text may contain macro calls which are expanded by a lower-level macro set and so on until the code is converted into the base language.

> Design Principle 3. Each enhanced language should become the target language for the next set of macros.

It is the responsibility of the macro processor to support this principle. It must scan the generated text from macros in macro set *n* for macro calls belonging to sets 1 to *n*-1. If it matches two such macros the one from the higher set number is executed. All macro calls take precedence over base language code.

This principle is not essential to the working of a macro processor but it is a major step in controlling complexity. Without it there would be the danger of recursive loops in which, say, macro A generates a call to macro B which in turn generates a call to macro A.

This principle rules out the definition of a language feature in terms of itself. Such features are relatively rare. However, if the language supported requires recursive definitions of enhancements then the principle would be relaxed for those enhancements to permit the generated text of a macro set *n* to be written in enhanced language *n*.

Recursive definitions of enhancements should not be confused with nested code. For example, an enhancement which supports an IF statement which may be nested can be handled within design principle 3.

7. MACRO CALL RECOGNITION

In existing macro schemes combinations of the following are used to determine whether or not a piece of source text constitutes a macro call: the syntax of the piece of source text; the current context; the existence of a herald; the level of nesting; the state of the macro (switched on/off). This section considers each of these criteria for matching.

7.1 Syntax of the macro call

Four levels of matching macro calls by their syntax can be identified. They are: matching on trigger word (e.g. ref. [9]); matching on delimiters (e.g. ref. [4]); matching on delimiters and class of argument (e.g. refs. [12, 14]); complete validation of macro call. These strategies will be illustrated using the COBOL enhancement:

SET condition-name **TO TRUE**

(1) *Matching on trigger word.* The criterion for matching is that the first word of the macro call matches the macro's trigger word. All other words are considered to be arguments whether they are fixed or variable. Thus the template for the SET statement would be equivalent to

SET &1 &2 &3

This defines SET to be the trigger word and for each macro call &1 represents actual argument 1 (the condition-name). &2 and &3 represent the second and third actual arguments (which should always be the words TO and TRUE).

(2) *Matching on delimiters.* The criterion for matching is that all the delimiters (fixed parts) in a macro call match the delimiters in the template. Thus the template for the SET statement would be equivalent to

SET &1 TO TRUE

In the example SET, TO and TRUE are the delimiters and in each macro call &1 represents the argument between SET and TO (i.e. the condition-name).

(3) *Matching on delimiters and class of argument.* The criterion for matching is that all the delimiters are matched and that the class of the arguments is as specified in the template. Thus the template would be equivalent to

SET condition-name TO TRUE

The macro call must thus consist of 'SET', followed by a condition-name, followed by 'TO', followed by 'TRUE'.

(4) *Complete validation of macro call.* The criterion for matching is as in 3 above (all delimiters and classes of arguments) plus a check that all other 'syntax' rules are obeyed. These other rules are referred to as *consistency rules.*

Most rules can be expressed by careful choice of classes for the arguments and by listing all possible combinations in the format. One example of a rule which cannot be readily expressed on this manner occurs with the COBOL statement

USE AFTER STANDARD EXCEPTION
PROCEDURE ON { file-name / INPUT / OUTPUT }

The rule is that within a program INPUT and OUTPUT may each occur only once in a statement of this format.

Design criterion 4 would suggest that the highest-level strategy for macro matching should be adopted: that is strategy 4. However, for many languages the consistency rules are so diverse that any grammar which could express them all would be complex. This would add appreciably to the difficulty of learning the macro writing language (thus offending design criterion 5) for little return – since in most languages there are few rules which cannot be accommodated by strategy 3 above.

> Design Principle 4. Macro call matching should involve as much checking as is practical; as a minimum this should comprise a complete check on the delimiters and the class of each argument.

Strategies 3 and 4 have the following advantages over the other strategies: removing the responsibility of macro call validation from the macro writer; permitting the separation of logical arguments; permitting a greater variety of enhancements. Each of these is discussed below.

(1) *Responsibility of macro call validation.* For the following enhancement (a simplified form of COBOL's case type statement)

EVALUATE identifier
WHEN literal imperative-statement
[**WHEN** literal imperative-statement]...
END-EVALUATE

the macro processor would not only check for the presence of the delimiters EVALUATE, WHEN and END-EVALUATE but would also ensure that the arguments conformed to the appropriate class (identifier, literal and imperative-statement).

The complete validation of the macro call is necessary so that any error message can be expressed in terms of the original source (design criterion 2) rather than leaving

errors to be detected when the generated text is compiled. If the macro processor did not perform complete validation the onus would fall on the macro writer. By design criterion 4 this is unacceptable: validation of the macro call, especially the arguments, can be an extremely tedious job. In fact, in the case when an argument is a COBOL imperative-statement, it is impractical.

(2) *Separation of logical arguments.* If the macro processor is unaware of the syntactic class of the arguments it would be unable to detect their boundaries. This task would then fall on the macro writer. In the enhancement

PERFORM **UNTIL** condition
imperative-statement
END-PERFORM

finding the end of the condition could prove complex and error prone, particularly with conditions such as the following:

(REGION-NO > REGION-COUNT OR SALES (REGION-NO) = SPACES) AND NOT SPECIAL-REGION (REGION-NO)

The best solution is for the macro processor to separate the arguments, thereby enabling the macro writer to refer to 'condition' and 'imperative-statement' as distinct entities.

(3) *Increased variety of macro calls.* If matching is by trigger word or delimiters the macro scheme would be unable to recognize infix macro calls such as:

identifier **EQUALS** identifier.

7.2 The context of the macro call

The context in which a programming language construct may appear is normally specified by stating its class. For example in COBOL the context in which:

SET condition-name TO TRUE

may appear is determined by the fact that it is of the class imperative-statement.

Since macro calls are to become part of the language their context should be described in the same way. The macro processor can then ensure that they appear in the correct context.

This has the advantage of:

(*a*) automatic rejection of macro calls appearing in the wrong context;
(*b*) the ability of the macro processor to distinguish between macros with the same syntax but different contexts (this is relevant for context-sensitive features);
(*c*) enabling the macro processor to validate the classes of all arguments (including those which themselves contain macro calls).

> Design Principle 5. The context in which macro calls may appear should be specified for each macro. Each macro call should be matched only in its specified context.

The implementation of this principle depends on a full syntax check of the language at macro processing time.

Macros which are matched on the basis of context and full validation of delimiters and class of arguments are known as *syntax macros.* In addition to this type of macro there is a need for macros which are invoked purely on the basis of context – these are known as *event-driven macros.* An example is a macro of context 'start of program' which does some initialization for another macro which will be called later in the program.

7.3 Heralds

Many existing macro schemes (e.g. GPM[16]) require all macro calls to be preceded by a symbol known as a *herald.*

Thus if * was a herald

*SET END-FILE TO TRUE

would be a macro call but

SET END-FILE TO TRUE

would not be.

Heralds are used to avoid ambiguity between macro calls and in-built constructs and to make macro recognition more efficient.

With the level of checking performed with syntax macros no ambiguity can remain (without it also being ambiguous to the applications programmer). Furthermore, with the macro processor being aware of the current context of the source program little advantage in terms of efficiency would be gained by the presence of a herald.

To be distinguishable by the macro processor the herald must be inconsistent with the style of the base language. It thus infringes design criterion 1.

> Design Principle 6. Heralds should not be used

7.4 Nested macros

Some macro schemes (e.g. Stage II[20]) forbid the nesting of macro calls.

For most enhanced languages this will be an unacceptable constraint. For example in the COBOL statement:

EVALUATE identifier
WHEN literal imperative-statement
[**WHEN** literal imperative-statement]...
END-EVALUATE

imperative-statement may contain further macro calls including EVALUATE itself.

If nested macro calls were not permitted it would be necessary to support the enhancement by means of three separate macros: EVALUATE identifier, WHEN literal and END-EVALUATE. This would have the disadvantages:

(*a*) of being an unnatural way of viewing the enhancement;
(*b*) of involving the macro writer in the problems of matching the corresponding EVALUATEs, WHENs and END-EVALUATEs (including the handling of nested EVALUATEs),
(*c*) of leaving the imperative-statement unvalidated or partially validated at the source level.

For these reasons nested macros should be supported.

> Design Principle 7. Free nesting, including recursive nesting, of macro calls should be permitted.

7.5 Switching macros on and off

Most languages contain features which can be switched on or off by a compile time indicator. Examples are trace statements or array-bound checks which can be included or excluded from the object code.

These can conveniently be implemented by a macro (A) which implements the optional feature and is initially switched off. Another macro (B) would recognize the compile time indicator and switch macro A on.

> Design Principle 8. It should be possible to switch on and switch off a macro from within the body of another macro.

8. SPECIFICATION OF THE TEMPLATE

The template serves two purposes. One is to provide the macro processor with information about the circumstances in which the macro is to be executed. The other is to provide a means of access to the components of the macro call from within the macro itself. The first purpose was discussed in the previous section. This section discusses the second purpose and establishes principles for the information content of the template and for the method of specifying the template.

8.1 Access to components of the macro call

Within the macro definition reference is made to components of the macro call in order to:

(*a*) transfer part of the call (often an argument) to the generated text;
(*b*) discover whether an optional item is present in the macro call;
(*c*) discover which of a number of alternatives is specified in the macro call;
(*d*) obtain more information about an argument.

These are illustrated by the following examples. In the case of the macro call format:

PERFORM procedure-name-1
[{**THROUGH** / **THRU**} procedure-name-2]
WITH **TEST AFTER**
UNTIL condition

it may be required to transfer the whole of the first line to be generated text (i.e. PERFORM followed by the first procedure-name, followed by the THROUGH phrase if it is specified).

In the case of the macro call format

INITIALIZE identifier-1
[**REPLACING** {**ALPHANUMERIC** / **NUMERIC**} DATA **BY** {identifier-2 / literal-1}]

it may be required:

(*a*) to test for the presence of the REPLACING phrase;
(*b*) to discover whether ALPHANUMERIC or NUMERIC has been specified;
(*c*) to discover the properties of the data item referenced by identifier-1.

In general there is a need to reference any component (elementary or composite) of the macro call. Macro writers should be free to choose the names for each of the components which they wish to reference.

> Design Principle 9. It should be possible for macro writers to assign names of their choice to any component of the macro call.

Most existing methods of specifying templates fall well short of this requirement. Many allow access to single delimiters or arguments by reference to their relative position in the actual macro call. Thus in the above PERFORM statement argument-1 would be the first procedure-name. When the format contains options this could prove to be an inconvenient mode of reference. For example, in some macro calls argument-2 will reference the second procedure-name but in others, where the THROUGH phrase is omitted, it would reference the condition. This problem is overcome, in the case of arguments, but not delimiters, by Campbell[7] by reference to the arguments (in the above example) as procedure-name-1, procedure-name-2 and condition. The language IMP[8] goes a step further and allows the programmer to assign a name to each argument.

A method of fully satisfying the design principle is discussed below.

8.2 The method of specifying the template

To satisfy the preceding design principles the template specification must include the names allocated to the components of the macro call and the full format of the macro call. The method adopted must thus, for most languages, permit the specification of options, alternatives, repeated items and the class of each argument.

The template can be specified in one of the following ways:

(*a*) by using a syntactic metalanguage;
(*b*) by using a syntax diagram;
(*c*) by using the data structure description facilities of the macro writing language.

Examples of these are shown in Figs 4, 5 and 6.

Fig. 6 is an informal definition showing how the statement is repesented as four level 2 data items, some of which are subdivided further. The REDEFINES clause is used to represent alternatives. The OPTIONAL clause has been devised to indicate that an entry may be optional.

The first two approaches (Figs 4 and 5) are more readable but they suffer from two disadvantages. They are unlikely to harmonize with the macro writing language – neither repesentation is used to repesent structures in conventional programming languages. The other problem is that it is difficult with both approaches to allocate names to components within the structure.

For those languages which do have suitable data-

PERFORM procedure-name [{**THROUGH** / **THRU**} procedure-name]
WITH **TEST AFTER**
UNTIL condition

Figure 4. COBOL syntactic metalanguage

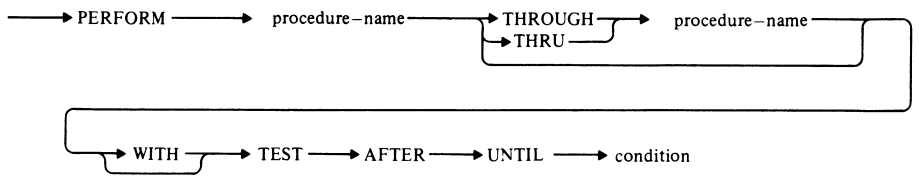


Figure 5. A syntax diagram

```
1     PERFORM-WITH-TEST
 2
 2      PROCEDURES-PHRASE                    'PERFORM'
   3
                                             < procedure-
                                             name >
   3    THROUGH-PHRASE  OPTIONAL
     4    SPELLING-1                         'THROUGH'
     4  SPELLING-2 REDEFINES SPELLING-1      'THRU'
     4
                                             < procedure-
                                             name >
 2      WITH-TEST-PHRASE
   3                       OPTIONAL          'WITH'
   3                                         'TEST AFTER'
 2      UNTIL-PHRASE
   3                                         'UNTIL'
   3                                         < condition >
```

Figure 6. Use of COBOL Style Record Structure

structure description facilities the balance is clearly in favour of using them. In return for some overheads in verbosity we gain consistency between the representation of the template and other data structures, and a convenient method of assigning names to any component of the macro call.

> Design Principle 10. In the specification of the macro template full use should be made of the macro writing language's facilities for describing data structures.

9. GENERATED TEXT

For each piece of generated text the macro writer must:

(*a*) specify the text;
(*b*) specify the destination;
(*c*) output the test to the specified destination.

This section discusses each of these functions.

9.1 The specification of generated text

A problem with macros is that two sets of source text have to co-exist in one program: the macro time code and the generated text. It is thus necessary to clearly distinguish between the two in order to avoid confusion for anyone reading the macro definition. The problem is made more acute if the same language is used for the two sets of source text.

> Design Principle 11. The generated text should be clearly delimited and the macro writing language should permit it to be isolated from the rest of the macro definition.

Many languages permit the manipulation of source text. COBOL for example embeds it in double equals signs:

```
== 1 COUNTER PIC 99. ==
```

It is less common for languages to provide the means for isolating generated text. In COBOL this could easily be achieved by means of a new Data Division Section (say the Generated Text Section) for the definition of generated text.

9.2 The specification of the destination

Many macro processors position all the generated text in place of the macro call. In regimented languages such as COBOL there is a need to distribute the generated text. For example, a procedure division macro call will often generate a definition of a data item which must be placed in the data division.

Having introduced the concept of multiple destinations for generated text it is useful to add a special destination 'diagnostic file' for errors in the macro call detected by the macro definition.

Some of the destinations needed by COBOL would be current position, diagnostic file, file section, working-storage section and end of procedure division.

9.3 Outputting generated text

In most macro schemes the generated text is interspersed with the procedural code and is output as it is encountered during the execution of the macro. This is a hangover from the days of simple macros in which the macro body consisted solely of generated text.

This approach has the disadvantages of: preventing the isolation of generated text (see design principle 11) and of being inconsistent with the normal methods of outputting data from a program.

In COBOL a specially devised statement to output the generated text would be appropriate. In other macro writing languages it might be more appropriate to write it to a file or to pass it to the macro processor as a parameter (i.e. regarding the macro as a subroutine of the macro processor).

10. INDEPENDENCE

A simple enhancement will be implemented as one macro. In more involved cases what is seen by the language designer as a single enhancement can involve more than one macro to implement it. For example, a new data type could be supported by two macros; one macro recognizes the definition of data items of the new data type and the second macro recognizes all references to data items of the new type. A group of macros which supports a single facility will be referred to as a *macro group*.

This section investigates the problems of ensuring that each macro group can be written independently of any other macro group and that communication between macros within a group is possible. These questions of independence and communication affect the macro

definition itself and the generated text. Each is discussed in turn.

10.1 Independence of macro definitions

Within macro definitions independence is required at two levels: the macro group level (data items used to communicate between macros in the same group should not be accessible from any other group); and the level of the individual macro (data items and procedures defined within a macro for use in that macro should not be accessible by any other macro).

Thus there is a need for data items which are local to a macro and others which are local to the macro group. There is also a need for data items which are global to all macro groups. The same arguments apply to procedures.

> Design Principle 12. The macro writing language should support data items and procedures of the following scopes: global to all macros; local to a single macro group; local to a single macro.

The normal method of restricting scope is by means of procedures or subprograms. In COBOL each macro could be a separate subprogram. This supports, by default, local items. Since in COBOL there is currently no direct means of grouping subprograms with shared data the macro writing language would have to be extended to permit 'external data' of scope 'macro group' and 'global'.

10.2 Independence of generated text

Within generated text independence is required at three levels:

(*a*) the macro group (e.g. to generate, in one macro, a data item which is to be referenced by the generated text from another macro in the same group);
(*b*) the individual macro (e.g. to generate, in one call of a macro, a data item which is to be referenced by the generated text of another call of the same macro);
(*c*) each call of an individual macro (e.g. to generate a data item which cannot be referenced by the generated text resulting from any other macro call).

The macro writer can achieve the desired effect by generating names for the local items according to pre-defined conventions. Some macro processors (e.g. ML/I[14] and Cobra[10]) provide registers to assist the programmer in this task.

The high-level solution is for the macro scheme to support the desired scopes in addition to generated data items which are global to the whole program. The same argument applies to generated procedures.

> Design Principle 13. The macro scheme should support generated data items and procedures of the following scopes: global to program; local to a single macro group; local to a single macro; local to a single macro call.

Those languages which support nested procedures (of the Pascal type) provide a convenient method of supporting data items and procedures which are local to a single call of a macro. They are not, however, so well suited for the other two levels of local items because the communication between macro calls cuts across the structure of the program in which they appear.

10.3 Relationship of macro group to macro set

This section has identified the macro group which supports a single enhancement. In section 6 the concept of a macro set was introduced. It is a set of macros which are added to the language as a single step, all using the same target language. The relationship of sets and groups is now considered.

The most powerful arrangement is for sets to be independent of groups, in other words to allow a group to straddle more than one level of the enhanced language. This is only of use when it is desired to code an enhancement at level n of the language and at some later stage enhance the enhancement at level $n+m$. This arrangement has the disadvantage of being somewhat complex. It also is anomalous to have communication by shared data items between two levels of the enhanced language when no such communication is possible with level 0 (the base language).

A far simpler arrangement is for each group to be entirely contained in one set. If this is adopted it is logical in fact for each set to consist of precisely one group. This allows the concepts of sets and groups to be merged without any additional loss of functionality.

11. ACCESS TO THE SYMBOL TABLE

Compilers use the symbol table to validate statements and to determine what object code should be generated. For example to implement the COBOL statement

INITIALIZE SALES-TOTALS

the compiler would:

(*a*) check that SALES-TOTALS is in the symbol table (i.e. it has been defined);
(*b*) locate all subordinate numeric data items and generate code to move zeros to them;
(*c*) locate all other subordinate data items and generate code to move spaces to them.

Just as the compiler needs the information in the symbol table to determine what action to take so also will macros need this information for some enhancements (such as a macro to implement INITIALIZE).

In general macros will need all the information about all the items defined in a program. In COBOL the items will include data items, files, paragraphs and sections. The information includes all attributes (e.g. for files the organization, the access methods, etc.) and all the relationships (e.g. which data items are subordinate to each data item and file).

In theory this information could be accumulated by macros. However, this would be a major task. It would be highly preferable for the macro processor to provide the information for use by the macro definitions.

> Design Principle 14. A comprehensive symbol table should be made available to the macro definitions.

This principle can be fulfilled with minimum duplication of effort when the macro processor is embedded in a compiler.

Since the symbol table contains a set of items with their attributes and relationships it can conveniently be viewed as a data base. The form of access within the macro definition will however depend on the nature of the macro writing language.

12. THE MACRO WRITING LANGUAGE

The macro writing language is required to be as portable as the base language and easy to learn (design criteria 3 and 5). These criteria would both be satisfied if the base language was used as the macro writing language. The first one (portability) is satisfied automatically except in the rare cases where the base language is not available on the machine on which the macro processor is run. The second one (ease of learning) is satisfied because the macro writer will need to know the base language (or at least the current enhanced language) in order to specify the generated text.

The macro writing language must also support the functions of macro writing: the specification of the template and generated text, the procedural code needed to determine the generated text for each macro call, and the interface to the symbol table. Most high-level languages provide facilities which could be used for these functions:

(*a*) record structures with facilities for repetitions, alternatives and options, for use in the template;
(*b*) text-defining facilities; for use with generated text;
(*c*) procedural statements; for examining data (the macro call) and outputting data (the generated text) accordingly;
(*d*) database, file or array accessing facilities: to interrogate the symbol table.

Processing a macro call has very much in common with processing any other data. As a result most existing high-level languages (with a few extensions to support the macro functions in a natural way) are well suited to macro writing. In addition to portability and ease of learning the choice of the base language as macro writing language offers the further benefit of being able to use the macro scheme to enhance the macro writing language.

> Design Principle 15. The base language (with a few extensions) should be used as the macro writing language.

The way in which the base language will be used and modified for macro writing will vary from language to language.

In the case of COBOL, existing language features could be used as follows:

(*a*) a COBOL sub-program for each macro definition;
(*b*) a COBOL record for the template;
(*c*) a data base for the symbol table;
(*d*) the MOVE statement to transfer components of the macro call.

Extensions to COBOL could be made to

(*a*) define the context of a macro;
(*b*) define the class of arguments in the template;
(*c*) support the scope rules;
(*d*) test for the presence of optional components of the macro call.

13. THE PROGRAMMER'S VIEW

The preceding sections have discussed the design of the macro facility. This section summarises its envisaged use from the viewpoint of the applications programmer and the macro writer.

Applications programmers should be unaware of which language features are supported directly by the compiler and which ones by macros. The should see only the original listing (no generated text) and receive one diagnostic listing in which reference is made only to the original source (not the generated text).

This implies that the macro calls are fully validated. The macro writer thus has the responsibility of ensuring that this happens. This is not however an onerous duty since, if the syntax is described in full in the template, the macro processor will do all the checking automatically. The macro writer would thus, for example, specify:

ADD { numeric-identifier / numeric-literal } **TO** numeric-identifier

rather than the normal COBOL

ADD { identifier / literal } **TO** identifier

Any consistency checks which cannot be specified in the template must be supported by procedural code in the macro and all errors reported in the diagnostic file.

The invisibility of macros to the applications programmers demands that the macros are as reliable as the compiler. They have no way of discovering bugs in macros and it is not their job to do so.

Macro writers will be able to work at a high level. Having described an argument or any component of the macro call they will be able to assume that it satisfies all the syntax rules and can simply transfer it to the generated text as required. If an argument is a lexical item, the symbol table can be used to discover its attributes (the organization of a file, the size of a data item, etc.). If it is a syntactic entity there should be no need to dissect it (indeed it would be reasonable for the macro scheme to prevent the programmer from so doing). Everything that needs to be known about an argument should be discoverable by enquiring about its class (e.g. in the case of a COBOL condition, whether it is a relational condition or a sign condition). Thus an argument of any complexity is always handled as a single item.

14. CONCLUSIONS

The following set of principles for designing macro schemes for use in language enhancement of high-level programming languages has been established.

(1) The macro facility should be designed for a single base language.
(2) The requirements placed on the macro processor should be limited.
(3) Each enhanced language should become the target language for the next set of macros.
(4) Macro call matching should involve as much checking as is practical; as a minimum this should comprise a complete check on the delimiters and the class of each argument.
(5) The context in which macro calls may appear should be specified for each macro. Each macro call should be matched only in its specified context.

(6) Heralds should not be used.
(7) Free nesting, including recursive nesting, of macro calls should be permitted.
(8) It should be possible to switch on and switch off a macro from within the body of another macro.
(9) It should be possible for macro writers to assign names of their choice to any component of the macro call.
(10) In the specification of the macro template full use should be made of the macro writing language's facilities for describing data structures.
(11) The generated text should be clearly delimited and the macro writing language should permit it to be isolated from the rest of the macro definition.
(12) The macro writing language should support data items and procedures of the following scopes:
(*a*) global to all macros;
(*b*) local to a single macro group;
(*c*) local to a single macro.
(13) The macro scheme sould support generated data items and procedures of the following scopes:
(*a*) global to program;
(*b*) local to a single macro group;
(*c*) local to a single macro;
(*d*) local to a single macro call.
(14) A comprehensive symbol table should be made available to the macro definitions.
(15) The base language (with a few extensions) should be used as the macro writing language.

These principles can be regarded as guidelines (not absolute rules) for any language enhancement scheme. They aim to provide the applications programmer with fully supported enhancements which blend into the base language without degrading portability. They also assist in providing the macro writer with a high-level, easy-to-learn macro writing language which permits each enhancement to be implemented independently.

The CLEF macro processor[12] has been designed, for the enhancement of COBOL, in accordance with these principles. A discussion of its design appears elsewhere.[18]

Acknowledgements

The CLEF macro scheme and its forerunner MCOBOL were designed by the British Computer Society's COBOL Language Enhancement Facility Working Party. Most of the design principles were established or reinforced at meetings of the working party involving Tony Sale, John Sawbridge, Ken Meyer, Peter Eagling, Ferenc Schustek, Jim Hamilton, Terry Clayton, Brian Reynolds, Colin West, Alan Heywood-Jones, Alan Fryer and the authors.

The authors would like to thank Bob Brooks, Alan Fryer, Jim Hamilton, Alan Heywood-Jones, Ken Meyer, Ferenc Schustek and Francis Warner for their invaluable comments on the first draft of the paper.

This paper was made possible by a grant from the U.K. Science and Engineering Research Council.

REFERENCES

1. ADR, *MetaCOBOL Concepts and Facilities*. Applied Data Res. Princeton, N.J. (1976). [Introduces main features of the MetaCOBOL macropocessor.]
2. ANSI, *American National Standard Programming Language COBOL X3.23-1974*. Amer. Nat. Standards Inst., N.Y. (1974). [Official specification of ANS 74 COBOL.]
3. ANSI, *Draft Proposed Revised X3.23. American National Standard Programming Language COBOL, ANSI* (1981). [The proposed standard to replace ANS 74 COBOL.]
4. P. J. Brown, The ML/I Macroprocessor. *Comm. ACM* **19**, 10 (October 1967), 618–623. [Discusses main features of ML/I macroprocessor.]
5. P. J. Brown, *Macroprocessors and Techniques for Portable Software*. Wiley, New York (1974). [Good introduction to concepts of macros.]
6. P. J. Brown, Macros without tears. *Software Practice and Experience* **9**, 6 (June 1979), 433–438. [Makes the case for using existing programming languages for defining macros.]
7. W. R. Campbell, A compiler definition facility based on the syntactic macro. *Computer Journal* **21**, 1 (February 1978), 35–41. [An example of a syntax macro scheme.]
8. C. Christensen & J. S. Shaw, *Proceedings of the Extensible Languages Symposium SIGPLAN, Notices 4, 8* (August 1969). [Describes seven extensible languages and PL/I macro facility. Also includes introduction to the concept of extensible languages, the motivation behind them, and some alternative approaches.]
9. M. I. Halpen, XPOP: A meta-language without metaphysics. *Proc. FJCC. AFIPS* **26**, 57–68 (1964).
10. Plessey, *Cobra Programming Manual*, Company Software and Systems Programming Department, The Plessey Company plc (1982). [A description of the Cobra macro processor.]
11. P. J. Layzell, *The History of Macro Processors in Programming Language Extensibility*. Computation Department Report No. 277, UMIST (October 1982).
12. P. J. Layzell, *CLEF Journal of Development*. Computation Department Report No. 258 UMIST, (August 1981). [The specification of CLEF macro writing language.]
13. P. J. Layzell & P. Van Der Linden. *Implementing the Proposed 1981 COBOL Standard by a COBOL Language Enhancement Feature*. Computation Department Report No. 249, UMIST (April 1980). [Analyses capability of macro schemes to support new features in next COBOL standard.]
14. B. M. Leavenworth, Syntax Macros and Extended Translations. *Comm. ACM* **9**, **11**, 790–793 (November 1966). [Introduces the concept of syntax macros.]
15. Delta Software Tools Ltd, *Delta MA213 Product Description*. [Includes a description of the Delta macro processor.]
16. C. Strachey, A general purpose macro-generator. *Computer Journal* **8**, 3, 225–241, (1965). [Describes GPM and introduces ideas used in many subsequent macro processors.)
17. J. M. Triance, The Design and Evaluation of a Language Enhancement Facility for COBOL. *M.Sc. Thesis* UMIST (October 1981).
18. J. M. Triance & P. J. Layzell, *CLEF – a COBOL Language Enhancement Facility*, Computation Department Report No. 273, UMIST (December 1982). [Describes and justifies the design of CLEF.]
19. J. M. Triance & P. J. Layzell, *Choose Your Own COBOL*. Proceedings of BCS 81 Information Technology for the 80's Conference (July 1981), pp. 16–33, Heyden. [Makes the case for allowing users to choose their own dialect of COBOL and investigates the ways of achieving it.]
20. W. M. Waite, The Mobile Programming System: Stage 2. *Comm. ACM* **13**, 7, 415–421. (July 1970). [Introduces STAGE 2 macro processor.]