SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 431–434 (1980)

SUPERMAC—A Macro Facility that can be Added to Existing Compilers

P. J. BROWN
Computing Laboratory, The University, Canterbury, Kent

SUMMARY

SUPERMAC is a macro facility that can be added to an existing high-level language as a run-time library. The design aim is to make macros significantly easier to use; this is done by employing a programming language and an environment already familiar to the user.

KEY WORDS Macro SUPERMAC High-level language BCPL

INTRODUCTION

The SUPERMAC concept, to be described in this paper, represents a new way of thinking about general-purpose macro processors. The aim is not to produce a facility that is more powerful than its predecessors nor one that contains extra clever features; instead the aim is simply to make a product that is many times easier to use than other macro facilities.

To explain SUPERMAC it is best not to relate it to existing macro processors, which may bring out preconceived ideas in readers' minds, but to use a separate analogy. Assume we want to provide a matrix manipulation package for some users. One approach is to design a stand-alone matrix manipulation system, with its own source language, its own compiler, its own debugging system and so on. A second approach is to create a run-time library of matrix routines, available within the users' favourite compiler—we shall assume here it is a FORTRAN compiler. Clearly, the second approach is more likely to attract users, since there is no great learning hump to cross in order to use the matrix facilities. The first approach requires a lot more effort and support from both implementors and users, though the final product might still be quite a successful one—akin to APL, perhaps.

If the library of matrix routines were well used within FORTRAN, similar libraries might be written for use within other languages. Each such language that acted as a *host* for the matrix facilities would doubtless have its own method of calling the routines and interfacing with the library, but the tasks performed by the routines themselves (e.g. inverting a matrix) would be the same for all languages. It is thus possible to design a language-independent set of routines for matrix manipulation and make them available in any suitable host language. (The NAG library[1] is an example of this on a much grander scale.)

If the application is macro processing rather than matrix manipulation, the relative advantages of the second approach are even greater. Macro processing is not a mainstream activity for most users, so they are naturally loath to spend months of their

0038–0644/80/0610–0431$01.00 *Received 24 December 1979*

time mastering a stand-alone macro processor. It is much better if macro processing can be made available as a library of run-time routines within their favourite programming language. Obviously this programming language must support string and file processing, since such activities are the essence of macro processing. (Similarly the host language for the matrix package must support arrays.)

In spite of the advantages of the second approach, macro processors have traditionally been written as separate stand-alone packages. This even applies to macro processors attached to compilers (such as that for PL/I[2]) and to macroassemblers. SUPERMAC takes the opposite view, and tries to exploit the advantages of the second approach. SUPERMAC is therefore a set of macro processing routines, designed to be made available as a run-time library within any suitable programming language. (In this paper we use the word 'routine' to be a generic name for a subroutine or a function; many programming languages use the word 'procedure' instead, but this word is somewhat ambiguous as in some languages it does not include the case of a function.) SUPERMAC has already been made available in BASIC and in BCPL. The latter implementation, which is called *SUPERMAC–BCPL*, is the more interesting as BCPL is much more in the mainstream of programming languages. Implementations for other languages are under way, and these are all based on SUPERMAC–BCPL.

DESCRIPTION OF SUPERMAC

In this paper we describe the main features of SUPERMAC, using SUPERMAC–BCPL as an example. The explanation does not, however, dwell on the subtleties of BCPL, and should be comprehensible to a reader familiar with any block-structured language.

Defining and calling macros

Assume the user wishes to write a macro to convert patterns of the form

INPUT *arg*1, *arg2*, ..., *argN*;

to ouput of the form

*arg*1: = READ; *arg*2: = READ; ...; *argN*: = READ;

This is the kind of task that is done when converting programs from one dialect to another. The example also emphasizes a further point: SUPERMAC is a *general-purpose* macro processing facility in the sense that the input text can be of any form (e.g. a program, some data, some natural language text). When SUPERMAC is embedded in, say, BCPL, this does *not* imply that the input text must be a BCPL program.

To define the above macro in SUPERMAC–BCPL, the library routine MACRO is called in the following way

*MACRO("INPUT — (**, — **);", INPUTMAC) //In BCPL ** means **

Here the first parameter of MACRO is a string representing the pattern to be recognized. Within this pattern the SUPERMAC metasymbol '—' means an argument, and the metasymbols '(* *sub-pattern* *)' mean that the enclosed sub-pattern can be repeated any number of times (including zero). In the above example the sub-pattern that may be repeated is a comma followed by an argument. The second

parameter of MACRO specifies a routine that is to be called whenever the macro pattern is recognized during macro scanning.

The SUPERMAC user can call MACRO as many times as he likes, in order to define all his macros. When this has been done, the routine MSCANF is used to apply macro scanning to a given input file.

Macro scanning works as follows. The given input file is scanned for any pattern corresponding to a macro. Text that does not match any pattern is simply copied to a specified *macro output file*. When a pattern is matched this constitutes a *macro call*. At a macro call, the routine corresponding to the matched pattern (i.e. the routine specified as the second argument to MACRO) is called. This is an ordinary BCPL routine within the user's program. The routine can, if it needs to, call SUPERMAC built-in functions such as *ARG(N)*, which has as its value the string that is the *N*th argument of the current macro call, and *MNA( )*, which has as its value the number of arguments of the current macro call. When the routine is called, all the material that is printed goes to the macro output file, thus replacing the text of the macro call. (There are ways to do normal printing as well, for example to output a message to the user.) Thus the routine INPUTMAC corresponding to our INPUT macro might be specified in BCPL as follows.

```
LET INPUTMAC( ) BE
$( FOR K = 1 TO MNA( ) DO
        $( WRITES(ARG(K)); WRITES(" := READ; ") $)
$)
```

(where *WRITES* is the BCPL routine to print a string). In addition, INPUTMAC could have used any other facility of BCPL. It could, for example, have counted the number of replacements made, or it could have used IF statements to make the text printed dependent on some outside context.

Strings in BCPL

BCPL is rather weak in string facilities, although the underlying primitive operations are present. To remedy this defect the normal SUPERMAC routines are augmented, in SUPERMAC–BCPL, by extra functions and subroutines such as 'compare-strings', 'assign-string', etc. (These are similar in nature to the routines for Pascal described by Bishop,[3] though the latter are more comprehensive.) Such extensions are not, of course, needed in a host language that already supports good string facilities. For example, SUPERMAC–BASIC needed no such extensions.

Summary of SUPERMAC–BCPL

The advantages of SUPERMAC–BCPL can be summarized as follows.

1. It is easy for anyone familiar with BCPL to learn and use the macro facilities.
2. Macro processing takes place within an ordinary BCPL run and can be interspersed with other activities—macro processing is not an isolated stand-alone activity.
3. There is no need, when implementing SUPERMAC–BCPL, to modify the BCPL compiler in any way. All that is provided is a library of routines which can be used when wanted.
4. Since BCPL is implemented as a true compiler rather than as an interpreter, the macro features are compiled and therefore run faster than interpreted macros.

Similar advantages apply to the inclusion of SUPERMAC in other host languages.

APPLICATION TO OTHER LANGUAGES

When SUPERMAC is embedded in other languages the interface will doubtless be different from the BCPL one. In some host languages it may be inconvenient or even impossible to pass a routine name as the second argument to the MACRO routine, and macros will need to be defined in some other way.

Although interfaces may differ, the specification of patterns and the manner of macro scanning should be common to all SUPERMAC implementations. The implementor of SUPERMAC simply translates these routines from some existing form, such as the BCPL form, to his host language, just as an implementor of the NAG library might translate its routines from, say, FORTRAN to ADA.

The only other implementation completed at the time of writing is the one for BASIC. The BASIC compiler that was used had good string and file facilities, but lacked something that is present in almost all other languages: a decent subroutine structure. There was no facility for naming subroutines, structuring them or passing arguments. There was certainly no facility for run–time subroutine libraries. Because of this it was necessary to modify the compiler to allow new statements corresponding to MACRO and MSCANF above. Hopefully this should not be necessary in any other host language. For a discussion of the BASIC implementation see Brown,[4] but ignore the comments that paper makes about the need to modify compilers.

CONCLUSION

The effort needed by a user to master a new stand-alone package involves reading manuals, learning JCL interfaces and, hardest of all, getting a feeling for what error messages really mean, and what debugging techniques are most effective. It is a large and often dispiriting task, and not many users have the time or mental energy to attack such tasks often. Even with a simple tool like an editor, users—and this includes computer professionals—get tied to one editor and are very reluctant to invest the effort to learn a new one.

Thus designers of software tools should, where possible, bring the tool into the user's environment rather than expect the user to make the effort to learn some new environment. SUPERMAC represents an attempt in this direction.

REFERENCES

1. B. Ford and J. Bentley, 'A library design for all parties', in *Numerical Software—Needs and Availability* (Ed. D. A. H. Jacobs), Academic Press, London, 1978.
2. IBM, *Student text: An Introduction to the Compile-time Facilities of PL/I*, Form C20-1689-0, IBM Corporation, White Plains, NY, 1968.
3. J. M. Bishop, 'Implementing strings in Pascal', *Software—Practice and Experience*, 9(9), 779–788 (1979).
4. P. J. Brown, 'Macros without tears', *Software—Practice and Experience*, 9(6), 433–437 (1979).