SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 433–437 (1979)

Macros without Tears

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

SUMMARY

The key to advancing the use of macro processors is not in providing ever more powerful facilities. Instead it lies in minimizing the effort needed to learn to use macros. This can be achieved by embedding the macros as extra run-time facilities—not as compile-time facilities—in existing programming languages.

KEY WORDS Macro General-purpose macro processor SUPERMAC

THE HUMP

In order to use any programming language effectively, you have to cross a hump, and this takes a month or more. There are four parts to the hump:

1. Reading the user manual.
2. Developing a reasonable programming style. The user manual says what you *can* do, but as soon as you use a language you find things it *cannot* do. You must therefore adapt your programming style to exploit the advantages and avoid the defects of your language.
3. Mastering the use of the compiler and its interface with the outside world, notably the operating system.
4. Learning how to debug programs. Even if your language has good error messages, you still need to learn by experience what kinds of bug are the root cause of each error.

The size of the hump is one of the reasons why so few new languages and new compilers gain acceptance.

The same hump applies to macro processors, but its effect is even more dramatic. Macro processors are not mainstream programming languages, and they are generally required for short specialized jobs. These jobs range from a few hours' work to a few months', plus the time needed to master the macro processor. If the latter amounts to a month's work it therefore represents a formidable barrier. This is the reason why the power of macro processors is not properly exploited.

REMOVING THE HUMP

The key to the future of macro processors is not therefore the adding of ever more powerful facilities, as this inevitably makes them even harder to use. Instead it lies in reducing the hump, and this is the problem to which this paper is addressed.

Our view is that the best way to remove the hump is to abandon the idea of the separate stand-alone macro processor, and instead to 'bring the macros to the users' by embedding

0038–0644/79/0609–0433$01.00 *Received 14 December 1978*

them within the user's own favourite programming language. This language is then the *host language* for the macro facilities. For better or for worse, the most widely used programming language is BASIC (though BASIC is hardly the favourite programming language of most computer scientists). We shall therefore use BASIC in this paper as an example of a host language, and shall discuss how BASIC can be extended to offer macro processing capabilities. We assume a version of BASIC that offers facilities for string processing and filing. All but the smallest BASICs do this. The methods presented are not, however, limited to BASIC and we shall discuss other host languages in a later section.

REJECTED APPROACHES

To make the intention of the proposals clear we should state two aims we are not pursuing.

Firstly, we are *not* aiming at a special-purpose macro facility for extending BASIC or some other chosen host language. We want a general-purpose macro processor, which can take a source file containing any kind of text, perform replacements within it and send the result to some output file. BASIC is the language used to specify what the replacements are.

Secondly, we are *not* advocating a kind of compile-time facility like that found in PL/1.[1] Such facilities have a significant learning hump, because they never exactly mirror the corresponding run-time facilities, and are processed by separate mechanisms. We are proposing macro processing facilities which can be brought into action during a normal BASIC run.

THE EXTENSIONS

It is a pleasant surprise to find how easy it is to extend languages to allow run-time macro processing. BASIC only requires two new statements and a few extra built-in functions. The two new statements we propose are the MACRO statement, which defines a macro, and the MSCAN statement, which brings macro processing to bear on a given piece of input—*the macro source.*

The MACRO statement takes the form

MACRO *string expression*

The *string expression*, which in practice is usually a *string constant*, specifies the *macro pattern* that is to be replaced. Every time this pattern is matched in the macro source, the macro is *called* by executing the BASIC statements that follow the MACRO statement. The processing of the call ends when a RETURN statement is executed.

The following example shows a simple macro with no arguments.

```
200 MACRO "PIG"
210 PRINT "BULL"
220 LET P=P+1
230 RETURN
```

The macro replaces each occurrence of PIG, within the macro source, by BULL. It also counts, in the variable P, the number of such replacements made.

Within a macro call, all of the facilities of BASIC can be used. Of particular use may be tables, string processing operations, subroutines and functions, and input/output. The user does not have to learn all these facilities; he knows them already by virtue of being a BASIC user.

Macro processing starts when the MSCAN statement is executed. MSCAN is followed by two arguments that specify the nature of the input (i.e. the macro source) and the output from macro processing (the *macro output*). In the usual BASIC notation a sample MSCAN statement is

```
MSCAN #3, #5
```

This takes channel 3 as the macro source, scans it, replacing all calls of the current MACROs, and sends the macro output to channel 5. A complete BASIC program to perform macro processing is the following.

```
 10  LET P=0
 20  REM ... and any further initialization
 30  FILE #3:  "INPUTFILENAME"
 40  FILE #5:  "OUTPUTFILENAME"
 50  MSCAN #3, #5
 60  REM Control returns here when macro processing has finished
 70  PRINT P, "REPLACEMENTS OF 'PIG' WERE MADE"
 80  STOP
100  REM MACRO definitions can come anywhere in the program
110  REM In this example they are at the end
200  MACRO "PIG"
210    PRINT "BULL"
220    LET P=P+1
230    RETURN
240  MACRO "CAMEL"
250    ...
 :
999    END
RUN
```

We assume the convention that within a macro PRINTing goes, by default, to the macro output. Line 210 is an example of this.

The scanning of the macro source follows the normal method used by macro processors. Text that does not match any macro call is copied directly over to the macro output; macro calls are replaced by whatever the macro PRINTs. In the above sample run, if the macro source consists of the lines

```
THIS LINE CONTAINS ONE PIG AND ANOTHER PIG
AND THIS LINE CONTAINS A THIRD PIG
```

then the macro output is

```
THIS LINE CONTAINS ONE BULL AND ANOTHER BULL
AND THIS LINE CONTAINS A THIRD BULL
```

The MSCAN statement also has a second format, where the macro source is a string expression rather than a file. For example the immediate statement

```
MSCAN "TEST PIG", #0
```

could be applied to give a quick test of the macro PIG. It should produce the macro output TEST BULL. This second format is also useful for the case where one macro wishes to call another, e.g.

```
215  MSCAN "SUBMACRO" & X$, #5
```

where, say, SUBMACRO is a macro name and the string X$ contains a list of arguments (see next section).

ARGUMENTS

Obviously, most practical macros will have arguments, unlike our elementary example of PIG. A simple way of specifying an argument within a macro pattern is to use a pair of dots, as in the example

MACRO "NAME .. , .. ;"

which matches the macro source

NAME *anything1*, *anything2*;

Within a macro the built-in function ARG$(N) has as its value the Nth argument. Thus the NAME macro might contain statements such as

```
PRINT ARG$(1)
IF ARG$(2)="SOMETHING" THEN ...
```

If a macro were allowed to have a variable number of arguments—there would have to be some extra syntax in macro patterns to allow this—then the actual number of arguments on a given call could be ascertained by calling another built-in function.

It is a matter of design choice whether macros are matched a line at a time, as in STAGE2,[2] or whether they are allowed to straddle lines as in ML/1.[3] There are many other design choices of a similar nature, like whether to work one character at a time or one token at a time, and whether an argument is allowed at the start of the macro call. All these design choices concerned with the matching of macro calls, though important, are somewhat outside the main theme of this paper.

OTHER LANGUAGES

The facilities described above are currently being implemented, under the immodest title of SUPERMAC. When SUPERMAC BASIC has been tested and evaluated in practice, the plan is to add SUPERMAC to other host languages. The syntax of SUPERMAC will need to be varied to fit sympathetically with each host language.

BASIC is a good host for SUPERMAC, but any language that supports files and variable length strings should be suitable. If the host has some of the facilities that BASIC lacks, like block structure and records, then all the better. An interactive host language is an advantage, as with any kind of programming, but is not vital.

The only absolute prerequisite for implementing SUPERMAC is the availability of the source code of the compiler for the chosen host language. An essential element of the SUPERMAC philosophy is to extend an existing, well-used, compiler rather than to build a new one.

EVALUATION

Obviously SUPERMAC does not eliminate the learning hump altogether. You, the reader, have spent some time getting this far in the paper, and you would have taken longer if we had spelt out every detail. Moreover, you may require a second reading to get your ideas

straight—paradoxically SUPERMAC may be easier to assimilate for someone *not* familiar with many of the existing macro processors. Nevertheless SUPERMAC makes a significant contribution to eliminating all four aspects of the hump. In detail the situation is as follows.

1. The SUPERMAC user manual is simply an extra chapter or appendix to an existing language manual.
2. The programming style in SUPERMAC is that of the host language. Only when SUPERMAC supports elaborate macro patterns is any further element of style likely to creep in.
3. There is no difference between a run of SUPERMAC and an ordinary run of a host language program.
4. Debugging of macro calls is just like debugging in the host language. Debugging the macro matching process, with the potential errors of wrong or clashing macro patterns, is, however, a new skill for the user to learn.

A reasonable expectation is that, given that the user is familiar with the source language and its compiler, the learning hump is reduced to one-quarter of the size for conventional stand-alone macro facilities of equivalent power. A system four times simpler to use than the norm is at least as worthwhile a goal as a system four times more powerful.

REFERENCES

1. IBM, *Student Text: An Introduction to the Compile-time Facilities of PL/1*, Form C20-1689-0, IBM Corporation, White Plains, New York (1968).
2. W. M. Waite, 'The mobile programming system: STAGE2', *Comm. ACM*, **13**, 7, 415–421 (1970).
3. P. J. Brown, 'The ML/1 macro processor', *Comm. ACM*, **10**, 10, 618–623 (1967).