SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 13, 295–304 (1983)

The SUPERMAC Macro Processor in Pascal

P. J. BROWN
Computing Laboratory, The University, Canterbury, Kent CT2 7NF

AND

J. A. OGDEN
Department of Computer Science, University of Reading, Whiteknights, Reading RG6 2AX

SUMMARY

SUPERMAC-Pascal is an implementation of the general-purpose macro processor SUPERMAC. It allows the user to write macros using Pascal procedures and to compile these procedures using his normal Pascal compiler.

This paper describes how the macro processor was adapted to fit Pascal, and how the apparently intractable problems of performing string processing in Pascal were tackled.

KEY WORDS Macro Pascal SUPERMAC General-purpose Macro processor

This paper describes the implementation within Pascal of the general-purpose macro processor called SUPERMAC. SUPERMAC is described in detail elsewhere,[1] and here we will confine ourselves to overall philosophy.

The aim of SUPERMAC is to avoid the large learning hump associated with macro processors. This is achieved by presenting macro processing as a run-time library within the user's favourite programming language—here assumed to be Pascal. Thus SUPERMAC can be considered as an ordinary library of string processing routines, but with extra facilities to define macros and then to scan texts to search for calls of these macros. To give a flavour of how it is used, the following is a fragment of a program that uses the SUPERMAC-Pascal library.

In this example the purpose of using macros is to implement a document formatting language. The source file is a text that includes constructions of form

$i[*string*]

to indicate that the *string* is to be in italics. A second construction

startchap[*number, title*]

defines the start of a new Chapter, which has the given *number* and *title*—both of which, like all macro parameters, are represented by strings of characters in the source file. The object of the macros is to convert the source file into an output file in which the above two constructions are replaced by lower-level instructions for a typesetting language. To achieve this the user first defines the macros to match the two

0038–0644/83/040295–10$01.00

Received 29 June 1982

constructions. This is done using the library procedure *Macrop* as follows

Macrop('$*i*[--]', *italics*);
Macrop('*startchap*[--, --]', *chapter*);

Macrop takes two arguments. The first is a string defining the pattern to be recognized. (Certain symbols within this string are interpreted by *Macrop* as metasymbols; as the reader has doubtless already guessed, the metasymbol '--' means an arbitrary string corresponding to an argument of the macro.) The Second argument to *Macrop* is the name of a procedure within the user's program—we call this the *macroroutine* of the pattern. This procedure is called whenever the corresponding pattern is subsequently matched during macro scanning.

(For Pascals that do not support procedures as parameters there is an alternative form of *Macrop* whose second argument is an integer—usually an index to a **case** statement—rather than a procedure name.)

Having defined his macros, the user can then start macro scanning of a source file. To do so he uses the library procedure *Mscanf*, which has the form

Mscanf(*filename*)

This procedure scans the given source file. If it finds any occurrences of a macro pattern, the corresponding macroroutine is called. This is termed a *macro call*. The macroroutine will define some *macro output* text (which may be null) that is to replace the pattern within the source file. Thus the source file is edited, and indeed there is a close parallel between macro processing and editing. At the end of scanning, the revised source file is written to a standard SUPERMAC output file called *Macout*.

As an example consider the occurrence, within the source file, of the macro call

$*i*[*vital*]

This causes the macroroutine *italics* to be called. We must emphasize that *italics* is an ordinary Pascal procedure, no different from any other in the user's program. It can therefore do anything from statistical calculations to having a conversation with the user. Somewhere along the line it will send text to the macro output, thus defining what is to replace the call. SUPERMAC-Pascal provides some *macro output procedures* to help do this. To illustrate the use of three of these procedures, *Mstr, Mln and Marg*, we shall assume our sample macro call is to be replaced by

/it
vital
/rom

The body of the procedure *italics* to do just this would take the form

```
procedure italics;
  begin
    Mstr('/it'); Mln;
    Marg(1); Mln;
    Mstr('/rom'); Mln;
  end;
```

As it stands, the body of *italics* is exclusively made up of calls of library routines. *Mstr* writes to the macro output the string supplied as its argument, *Mln* writes a newline,

and *Marg* writes an argument of the macro call—in our case argument 1. (Actually this description is an oversimplification of a two-stage process, but it will serve for the present.)

An alternative mechanism for macro arguments would have been to pass these as string arguments to the *italics* procedure. However, since macro calls can, in general, have a variable number of arguments and delimiters, the alternative mechanism is impracticable. Instead the *Marg* procedure, together with similar procedures for dealing with delimiters, etc., need to be used.

(Observant readers may have noticed another implementation point; some jiggery-pokery is needed for *Macrop* to remember a procedure name, like *italics*, so that *Mscanf* can call it later.)

THE OVERALL FORM OF A PROGRAM

The following program shows the context in which these example fragments might be embedded

```
program ...;
<<some instruction to include the SUPERMAC library>>
  .
  .
  .
procedure italics;
begin
    Mstr('/it'); Mln;
    Marg(1); Mln;
    Mstr('/rom'); Mln;
end;
  .
  .
  .
procedure chapter;
begin
    .
    .
    .
end;
  .
  .
  .
begin {main program}
    .
    .
    .
    Minit; {initialize SUPERMAC}
    Macrop('$i[--]', italics);
    Macrop('startchap[--, --]', chapter);
```

```
    Mscanf(myfile);
    .
    .
    .
end.
```

AN OPTIMIZATION

Since macro scanning is sequential, text passed over can be written directly to *Macout*, and indeed all SUPERMAC implementations do this. There is no need to keep the entire source file in store until its end is reached.

Macro calls can be embedded within one another. For example our *startchap* macro could include one or more calls of the italics macro. In such cases the embedded calls are expanded first—a kind of call-by-value mechanism. A consequence of this is that if a macro can possibly be embedded within another a macroroutine must not 'jump the gun' by writing directly to *Macout*; instead it should substitute its expanded form in the source file so that this can subsequently be passed as argument to the containing macro. The macro writer should always use the SUPERMAC routines *Mstr*, *Marg*, etc., which automatically send their output to the appropriate place—direct to *Macout* if this is possible, but otherwise to the source file in order to replace the current macro call.

A SHORTHAND

Although the notation for writing a procedure corresponding to each macro is simple and natural to a Pascal user, it does become somewhat tedious, particularly if there are a lot of macros that simply replace one string by another.

To combat this verbosity, SUPERMAC provides a method which allows simple macros to be expressed in a much more concise way. The method is to use the *Macro* procedure, which takes the form

Macro(*pattern, coded replacement string*)

The *coded replacement string* serves as the macroroutine and defines the string that is to replace each occurrence of the pattern. Within this string the escape character $ has a special meaning. In particular $1 means the first argument, $2 the second and so on; $*N* means a newline character.

Using this facility our macro for italics could be defined in the single line

Macro('$*i*[--]', '/*it*$*N*1*N*/*rom*$*N*');

Such a form, which mirrors facilities found in many simple macro processors, is only usable when the macroroutine consists solely of a sequence of calls of macro output procedures. It appeals more for its conciseness than its readability.

POINTS FROM THE EXAMPLE

This example has, we hope, emphasized the two important characteristics of SUPERMAC

(i) The learning hump is small. The user employs his familiar Pascal language and Pascal compiler. All he needs to learn are the specifications of the SUPERMAC-Pascal library procedures.
(ii) SUPERMAC-Pascal is a general-purpose macro processor. The source file is simply a text: it could be a program (in Pascal itself or any other language), some prose (as in our example) or some numerical data. SUPERMAC therefore has completely different uses from a macro processor designed solely for extending Pascal, such as MAP.[2]

THE CHALLENGE OF PASCAL

SUPERMAC has already been implemented in three other 'host languages': BCPL, BASIC and FORTRAN. Each implementation offers similar macro processing facilities, but the way SUPERMAC library routines are called from the host language varies widely depending on what procedural mechanisms, data structures, etc. are available. The way a BASIC programmer likes to view the world differs radically from that of, say, a BCPL programmer, and if SUPERMAC is to fit neatly into each host language it must be adapted to the style of the language.

The following example, which shows the SUPERMAC-BASIC implementation of our *italics* example, demonstrates that the difference between language styles can be dramatic.

```
100 MACRO "$i[--]"
110   PRINT *99: "/it"
120   PRINT *99: ARG$(1)
130   PRINT *99: "/rom"
140   RETURN
```

The adaptation of SUPERMAC to its host language has, of course, a cost in portability. Several macro processors, such as GPM,[3] STAGE2[4] and ML/I[5] are quite easy to move from machine to machine. SUPERMAC is harder, but the extra effort in adapting SUPERMAC to fit the user's preferred environment is felt to be well worth the time and trouble it takes.

With Pascal, the adaptation is a special challenge, since the facilities needed for macro processing appear to match exactly Pascal's worst weaknesses. Specifically there are three requirements

(i) String processing.
(ii) Relatively sophisticated input/output of characters.
(iii) Libraries.

The problems of providing these are compounded by the existence of two separate standards for Pascal: the 'official' standard and the *de facto* standard arising from the wide use of UCSD Pascal on microcomputers. It is desirable to implement SUPERMAC-Pascal in a way that covers both, so that SUPERMAC will run on any computer from an Apple to a VAX.

IMPLEMENTING SUPERMAC

SUPERMAC is essentially based on two machine-independent algorithms. The first decodes and checks the macro patterns supplied by the user. These patterns can be syntactically quite complex, since they can involve alternatives, options and indefinite

repeats. (The level is similar to that of SNOBOL4, but without the rich variety of pattern-matching functions that the latter provides.) The decoding algorithm converts macro patterns into an internal form that is akin to a directed graph; it also gives meaningful error messages to the user if a pattern is wrong.

The second algorithm scans a file for occurrences of the patterns. It deals with nesting (one pattern within another) and recursion (a macroroutine itself initiating a new macro scan). It has sub-procedures for doing such jobs as extracting the arguments and delimiters of a macro call.

The task of implementing SUPERMAC in a new host language consists of

(1) Converting these two algorithms to the host language.
(2) Designing an interface whereby the user communicates with these two underlying algorithms. (Thus in Pascal *Macrop* is the way the user sees the first algorithm.)
(3) Extending the library to remedy any deficiences in the host language—for example adding extra string processing facilities.

The task of implementing SUPERMAC should *not* involve any modifications to compilers (though SUPERMAC-BASIC proved an exception, given the poor foundations it was built on).

The first of the above tasks should be straightforward. If mixed language programming is available (e.g. a Pascal library routine can be written in BCPL) then it is trivial, but, sadly, this rarely happens. Instead the task involves a somewhat tedious transliteration of an existing encoding of the algorithms to the new host language. Normally the BCPL implementation is used as the master version from which other versions are made.

One of us (J. A. O.) together with a colleague, Mrs. S. M. Walmsley, began the implementation of SUPERMAC-Pascal by transliterating the BCPL algorithms. Unfortunately this did not turn out to be the straightforward task it should be. The original implementor (P. J. B.) had planned the BCPL encoding to be as clean and translatable as possible. Many features of BCPL, including recursion, were deliberately avoided in order to allow translation to a wide range of other languages. Nevertheless BCPL certainly allows dirty programming, and in this case the dirt surreptitiously crept in without the author being aware of it. Two horrors were a pair of arrays that overlapped their storage, and flagrant misuse of data types—in one case the constant zero, appearing in a place where an array name should occur, was used to signify a special case. Looking back, the author is both amazed and horrified that such features crept in. The fact that it happened is offered here as a piece of software experience: for some—perhaps most—of us, if dirt is allowed then dirt will indeed come in. In retrospect it was a mistake to use a permissive and somewhat off-beat language like BCPL as a master language. The Pascal implementation, and indeed the FORTRAN one, show the scars of this decision.

DESIGNING THE INTERFACE

The first stage of implementation turned out to be more of a challenge than it should be, but the real problems still lay in the second and third stages: designing a suitable interface to fit the SUPERMAC routines into Pascal, and augmenting Pascal with the extra string and file processing routines needed to make the system usable.

Certainly the worst problems came from the lack of string processing in Pascal. The difficulties in this area are well-known. In the context of macro processing they can be isolated into three areas

(i) *The string data type problem.* A string of varying length needs to be represented as a record (at least in official Pascal), but it then becomes a different data type to a string constant. As a result it is impossible to assign one to another, either directly or via the argument/parameter mechanism.

(ii) *The function problem.* It is not possible to write a function that returns a string as its result. This applies irrespective of whether the string is represented as an array or a record.

(iii) *The length problem.* If a string constant is used as an argument then it must have exactly the same length as the corresponding parameter.

Because of the severity of these problems, string processing packages are often implemented as pre-processors to Pascal.

The first implementation of SUPERMAC, done by J. A. O., used one of these pre-processors: STP[6] for the NORD-10/100 (and portable to other machines as well).

Independently—and intentionally so, since there is value in two people tackling a problem without biasing each other's thinking—P. J. B. attacked the above problems head on by producing a second implementation directly in Pascal. It is this second implementation that is described here for the light it throws on Pascal itself.

THE STRING STACK

Various packages exist to do string processing within Pascal[7,8] but none of these seemed reasonable in the SUPERMAC context. A consequence of our three string processing problems is that, unless precautions are taken, there is a combinatorial explosion in the number of library routines needed (e.g. compare a string constant to a string constant, compare a string constant to a string variable, compare a string constant to a macro argument). Moreover there is a tendency for string processing routines to be low-level, verbose and error-prone. A special concern with SUPERMAC is that perhaps 80 per cent of the references to strings within macroroutines are requests to send strings to the macro output. Thus it is particularly important that this, at least, be expressible in a simple way.

For SUPERMAC-Pascal the solution that was devised was to create a single global object called the *string stack*. The user cannot access this directly but can manipulate it using built-in SUPERMAC-Pascal procedures. These procedures cover the following operations

(a) appending strings to the end of the string stack

(b) marking the top of the string stack. Subsequent strings then form a *current string*

(c) assigning, comparing, scanning and outputting the current string

On return from each macroroutine the string stack defines the macro output.

This device helps combat the combinatorial explosion: all string operands, whatever their data type, are converted to the (secret) internal data type of the string stack and then operated on. Moreover the default mechanism whereby the string stack forms the macro output helps combat verbosity—given that producing macro output is the most common operation.

SOME EXAMPLES

We shall show some examples to give a flavour for how the string stack is used.

As the first example it is instructive to go back to the *italics* procedure already shown and explain more closely what it does. The macro output procedures *Mstr*, *Marg* and *Mln* add strings to the string stack; on return from *italics* this is defined as the macro output of *italics*. (The string stack is empty when a macroroutine is entered.) The overall effect is as if the procedures communicate directly with the macro output, which is how we explained it earlier.

As a second example the lines

```
Mmark; {mark the start of the current string}
Mstr('((('); Marg(2); Mstr('))');
Mtostr(heading);
```

would assign the current macro call's second argument, enclosed within double parentheses, to the string variable *heading* (about which more later). The act of assignment removes the current string from the string stack. If the user also wants to place it in the macro output it must be put on the stack a second time.

As a final example, the statement

```
if Marglength(3)>60 then {length of argument 3 exceeds 60}
begin
  Mmark;
  Mstr('Abbreviate'); Marg(3); Mln;
  Mscans; {start nested macro scan of the current string}
end;
```

would, if the **if** condition was true, initiate a macro scan of the current string, i.e. *Mscans* scans a string as *Mscanf* scans a file. (As an indication of the purpose of such a scan, think of *Abbreviate* as a macro with a pattern that helps divide a string up into units that can be abbreviated.)

STRING DATA TYPES

The user of SUPERMAC-Pascal needs both string constants and string variables. Often he will want the latter to have varying length, and we will assume here that this is the case. Because of the string data type problem, constants and variables must be given separate data types. Therefore SUPERMAC-Pascal provides two string data types. It provides them in a way that is independent of whether the user is in 'official' or UCSD Pascal.

The first data type is *Tstr* and corresponds to the data type of a string constant. In 'official' Pascal it is

packed array [1..*Mmaxstringsize*] **of** *char*

whereas in UCSD it is

string[*Mmaxstringsize*]

Obviously a *Tstr* argument can be used anywhere that a corresponding string constant can, for example as an argument to *write*.

The second data type is a heavy duty string called *Trope*. It is made up of a length

field together with a string. This string has a largish maximum length, the exact value being implementation-defined.

Trope has a varying length—perhaps it should have been called *Telastic*. Some Pascals may support *Tstr*s of varying length, but a user interested in portability must use *Trope*.

Corresponding to the procedures *Mstr* and *Mtostr*, which move *Tstr*s to and from the string stack, there exist procedures *Mrope* and *Mtorope* to do likewise on *Trope*s. (Our previous example that used *Mtostr* to place a string in *heading* might better have represented *heading* as a *Trope* and used *Mtorope* instead of *Mtostr*.)

A TACTICAL RETREAT

The device of the string stack alleviates the first two of our three string processing problems. Its use as the standard internal data structure helps combat the combinatorial explosion caused by the string data type problem; its use as a kind of default result combats the function problem by removing the need for functions.

Having won two partial victories we chickened out of a fight with the length problem. This is because, although this enemy is known to exist, he has not been present in any of the Pascals we have used. If the enemy is present, it is better to use the pre-processor approach, as on the NORD-10/100. All Pascals used for a direct implementation of SUPERMAC have allowed a string constant to be shorter than the variable or parameter to which it is assigned, and, if so, have automatically padded the constant with trailing spaces. SUPERMAC-Pascal procedures therefore delete trailing spaces from the ends of their arguments, though there is a facility for the user to identify trailing spaces that are really required. On a Pascal implementation that did no automatic padding, the user would simply have to do all the padding himself—a thankless task given that maximum string lengths are typically 80 characters or so.

INPUT/OUTPUT AND LIBRARIES

Earlier we outlined three requirements to overcome weaknesses in Pascal; of these, the string processing one is the hardest to satisfy. It is, however, worth spending a little time on the other two: I/O and libraries.

Consider the Pascal built-in procedure *write*. It has four special properties

(i) The first argument can optionally be a file name.
(ii) There can be indefinitely many further arguments.
(iii) These further arguments can have several possible data types (e.g. *char, integer, real, string*).
(iv) Extra formatting information can be appended to arguments.

None of these facilities is available to user-defined procedures.

The ideal way of sending material to the string stack (and thence usually to the macro output) would be to have two SUPERMAC procedures *Mout* and *Moutln* which exactly mirrored *write* and *writeln*. However, the wherewithall to write such procedures is denied to all but writers of the Pascal system itself.

Given that the ideal solution cannot be achieved the following concessions need to be made

(a) Items are sent to the macro output one at a time, rather than as an indefinitely long list of arguments. We have seen this with the sequence of calls of *Mstr*, *Marg*, etc. in our examples.

(b) If items other than strings or characters are to be sent to the string stack these need to be converted to strings first. (SUPERMAC-Pascal provides routines to do this, but fortunately they do not need to be used very often.)
(c) If the user wishes to format his output in some special way he must do it himself by outputting spaces.

The only facility salvaged from those of *write* is the default mechanism: the string stack by default goes to the macro output just as *write*'s output goes by default to the *output* file. Sometimes the macro writer wants to write to other files, for example to the terminal ('Did you really want ... to be your Chapter heading') or to files of, say, statistical information or warning messages. To achieve this SUPERMAC provides a procedure

Mwrite(filename)

which writes the current string to a desired file. (Note that the current string may involve *Tropes* or macro arguments so that *write* cannot be used directly.)

Finally, the library requirement is not hard to satisfy in practice since almost every Pascal supports *include* files or compiled units. (Even if neither is supported, it should be possible to design a shell command that edits the library into the source file before passing it to the Pascal compiler.) The experience of using these facilities is that compiled units are excellent, though a certain amount of sleight-of-hand is needed to allow user-defined macroroutines to be called within library procedures. Included files, on the other hand, make compilation a tedious task. SUPERMAC-Pascal is about 2000 lines and even on a fast personal workstation like the PERQ, it takes an annoyingly long time to compile.

CONCLUSIONS

Although this paper has said a lot about the problems in Pascal, the final message is that Pascal comes out with much credit. It is possible to use the primitive operations present to provide a reasonable user interface, either via a simple preprocessor such as STP or directly in Pascal as described above. It is, moreover, possible to present a mechanism that works with both official and UCSD Pascal, in spite of their different ways of regarding strings.

The end result is, we hope, a macro processor that will appeal to Pascal lovers in two ways: firstly for the ease with which it can be learned and used, and secondly for making the whole power of Pascal available in every macroroutine.

REFERENCES

1. P. J. Brown, 'SUPERMAC—a macro facility that can be added to existing compilers', *Software—Practice and Experience*, **10**, 431–434 (1980).
2. D. Comer, 'MAP: a Pascal macro preprocessor for large program development', *Software—Practice and Experience* **9**, 203–209 (1979).
3. C. Strachey, 'A general purpose macrogenerator', *Computer J.*, **8**, 225–241 (1965).
4. W. M. Waite, 'The mobile programming system: STAGE2', *Comm. ACM*, **13**, 415–421 (1970).
5. P. J. Brown, 'The ML/I macro processor', *Comm. ACM*, **10**, 618–623 (1967).
6. J. A. Ogden, *The String-handling Package STP*, Reading University, 1981.
7. J. M. Bishop, 'Implementing strings in Pascal', *Software—Practice and Experience*, **9**, 711–718 (1979).
8. A. H. J. Sale, 'Implementing strings in Pascal—again', *Software—Practice and Experience*, **9**, 839–842 (1979).