Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1978

MAP: A Pascal Macro Preprocessor for Large Program Development

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Report Number:
78-262

Comer, Douglas E., "MAP: A Pascal Macro Preprocessor for Large Program Development" (1978). *Department of Computer Science Technical Reports.* Paper 193.
https://docs.lib.purdue.edu/cstech/193

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

MAP: A Pascal Macro Preprocessor For Large Program Development

Douglas Comer

Computer Science Department
Purdue University
West Lafayette, Indiana 47907

CSD TR #262

March 1978

MAP: A Pascal Macro Preprocessor For Large Program Development

Douglas Comer

Keywords and Phrases: macro, preprocessor, Pascal, software tool

CR Categories: 4.12, 4.22, 4.43

Abstract

The programming language Pascal was originally designed for teaching introductory programming. Currently, however, production systems use it as the primary implementation language. This paper describes extensions of Pascal intended to aid the large program developer. The extensions are implemented in a macro preprocessor MAP, which supports constant expression evaluation, source file inclusion, conditional compilation, and macro substitution. While each of these features can be used independently, they are all implemented with a simple, uniform syntax. Furthermore, in keeping with the spirit of Pascal, an attempt has been made to make the facilities straightforward and simple. The design and implementation details are discussed.

1. Introduction:

While Pascal [14] was originally designed for teaching introductory programming, it has recently received wide attention as a well-structured, general purpose programming language. Most features in Pascal serve quite well in a production programming environment. Of course, there are several flaws in the design that limit its effectiveness; Habermann [4] and conradi [3] provide a comprehensive list. Many suggestions for improving and extending Pascal have appeared (e.g. see [12]). This paper takes a slightly different approach, concentrating on the use of Pascal in an environment where large programs must be written. Our suggested augmentations to Pascal (and to the Zurich implementation [16]) are embodied in MAP (for MAcro Pascal), a macro preprocessor for Pascal that supplies facilities which are not typically provided by a standard Pascal compiler, but which are, nonetheless, crucial to large scale system development and maintenance.

Preprocessors for high level languages are not new. McIlroy[11] notes that macro preprocessors were already in wide use by 1960. Languages such as PL/I [5] and C [10] have preprocessors associated with the language description. Of course, the most well-known macro facilities are those provided by assembler languages (e.g. [6]). And recently, a large number of "structured FORTRAN"-to-FORTRAN preprocessors have appeared (e.g. see [9]).

Basically, a macro preprocessor is a source-to-source translator used to convert a user's "extended" program into a valid source program for a compiler. Brown [2] describes how such processors can aid in software development.

Macro processors need not be linked to any one language. For example, Strachey [13] chose to make the GPM processor both language and machine independent. In GPM, input is viewed as a sequence of characters, including a distinguished character, newline, which marks the end of a line. Thus, GPM can process English text as easily as it can process a Pascal source program. The user simply supplies GPM with a set of macro definitions, where each definition consists of a name and a value. The name of a macro corresponds loosely to a function name in a programming language, while its value is merely a string. Following the definition, the user need only call the macro by giving its name (preceded by a special warning character) to have GPM substitute its value. Formal parameters are also allowed in a macro definition, and actual arguments are substituted by GPM similarly to the way actual arguments are substituted in function calls in a programming language. Indeed, Strachey has shown that GPM is a primitive programming language itself.

Brown [1] suggests an alternative type of macro processor, ML/I. In ML/I the basic unit is a token rather than a single character. Tokens can be English words or the basic syntactic categories of a programming language. In addition, Brown generalized the syntax of macro calls, allowing the user to describe to ML/I the context of the call. While the ML/I notion of tokens as basic units was adopted in MAP, the syntax of macro calls was chosen to resemble calls in GPM.

Another influence on MAP was the macro preprocessor RATFOR [9]. Like macro values in RATFOR, those in MAP must be well-balanced with respect to parentheses.

Finally, Pascal itself has a limited macro facility that had a strong influence on MAP, the constant declaration, CONST. Declaration of a CONST name associates a constant value with the name. It may be thought of as a parameterless macro with the limitations that names are valid Pascal identifiers, and values are language dependent constants. Unlike most macros, however, CONST declarations follow the Pascal scope rules. MAP extends the CONST facility by providing a compile time arithmetic capability.

The next section describes MAP, showing how the CONST declarations are extended, and the macro facility is added to Pascal. The third section discusses the design goals and implementation.

2. An Overview of MAP:

MAP provides four basic additions to Pascal: constant expression evaluation, source file inclusion, parameterized macro substitution, and conditional compilation. This section contains a discussion of each of these facilities.

MAP evaluates constant expressions (expressions where operands are constants or previously defined symbolic constants) on the right hand side of CONST declarations. Expressions may contain the following operators (listed in descending precedence):

function:	name (arguments)
negating:	not -
multiplying:	and * / div mod min max
adding:	or + -
relating:	< <= = <> >= >
concatenating:	(one or more blanks)

All standard operators have the same meaning as in Pascal, and strong typing is observed. The operators min and max require operands of type integer or real and return the smaller and larger of their operands, respectively. Concatenation requires operands of type packed array of char, and returns a packed array of char which is their concatenation (the type char is assumed to be a packed array of one character for concatenation).

MAP recognizes the standard Pascal functions ABS, SQR, CHR, ORD, ROUND, TRUNC, as well as two nonstandard functions, LENGTH and STRINGOF. LENGTH requires an argument of type packed array of char or char, and returns the number of characters in it. STRINGOF requires an integer argument, and returns a packed array of char consisting of its decimal representation.

Operands in CONST expressions may be constants or previously defined CONST names. Of course, Pascal scope rules apply to defined names. MAP also provides several predefined symbolic constants which can be used in CONST expressions. Two especially useful predefined names, TIME and DATE, give the time and date on which the compilation was performed. These predefined constants help when writing production programs that must be time and date stamped. For example, in a production program a heading is usually printed whenever the program runs:

"PROGRAM XYZ COMPILED ON mm/dd/yy AT hh:mm:ss"

Such a heading may provide the only link between an object version of a program and its source. Unfortunately, a programmer may fail to update the heading when making changes to the program. Using the predefined constants

in MAP to create the heading relieves the programmer of the updating task and guarantees the heading will always be accurate:

```
CONST
   HEADING = 'PROGRAM XYZ COMPILED ON ' DATE ' AT ' TIME;
```

In addition to constant expression evaluation, MAP supplies a macro substitution facility. A macro, which may have zero or more formal parameters, may be defined anywhere in the source program using the syntax:

```
$DEFINE(name(formals),value)
```

where "name" is a valid Pascal identifier, "formals" is a list of identifiers separated by commas, and "value" is a sequence of Pascal tokens which is well-balanced with respect to parentheses. Once a macro has been defined, it can be called by coding

```
$name(actuals)
```

where "name" is the name of the macro, and "actuals" is a list of actual parameters separated by commas. Each actual parameter must be a sequence of Pascal tokens which is well-balanced with respect to parentheses.

In addition to the user defined macros, MAP recognized several system macros. Definition of a new macro, as shown above, requires the use of one such system macro, DEFINE. Another system macro, INCLUDE, provides for source file inclusion. When MAP encounters a call:

```
$INCLUDE(file name)
```

it opens the named file, and continues processing, reading input from the new file. Upon encountering an end-of-file condition, MAP closes the included file, and resumes processing the original file. Includes may be nested, but they may not be recursive (even though there is a way to prevent an infinite recursion).

One may think of "include" as a macro whose body is an entire file. This view, however, does not reflect the fact that the user also expects included text to be listed like standard input rather than like the body of a macro. While macro expansions are not usually displayed in the source listing, included files are. Therefore, INCLUDE has a special status among macros.

One other system macro, CODEIF, is provided to support the conditional compilation of code. The syntax of CODEIF is:

$CODEIF(constant Boolean expression, code)

where the constant Boolean expression follows the rules for CONST expressions outlined above, and "code" represents a sequence of PASCAL tokens which is well-balanced with respect to parentheses. If the Boolean expression evaluates to true, the code is compiled; if the expression evaluates to false, the code is skipped.

3. Design Considerations:

MAP was designed after PASCAL had been used to implement several large programs. The primary objectives were to provide:

- A method of generating two or more slightly different versions of a program, say, for different machines,
- A method of saving debug code with the source deck without any runtime overhead,
- A method of compiling a single program divided into several source files.

Obviously, all these objectives can be met by a variety of standard macro processors. Unfortunately, using another macro language on top of Pascal would mean that users would be forced to learn an almost entirely new

language. Furthermore, the typical macro processor has its own rules for computing arithmetic expressions, and they will probably differ from those in Pascal. The approach taken in designing MAP was to keep the syntax and semantics similar to that of the underlying language, wherever possible. An example of this approach is found in the PL/I preprocessor [5].

Unfortunately, a Pascal-like metalanguage to describe Pascal programs presents several problems. The objectives require that the preprocessor conditionally compile tokens as well as entire statements. Pascal's use of the ALGOL 60 type compound statement make it difficult to conditionally compile tokens. One approach to the syntax problem, taken in C [10] uses a list form of compound statement in the preprocessor and an ALGOL 60 form of compound statement in the language. Thus, in the C preprocessor, an #if must match an #endif, while in the language an if does not have an associated endif. The distinction can be confusing.

Since all expression evaluation in MAP follows the rules for expression evaluation in Pascal, the user is not troubled when computing expressions. Using CODEIF, however, presents a serious problem. The syntax is unlike a conditional in Pascal; there is not even an "else" clause. For long code segments, it becomes difficult to spot the trailing right parenthesis. Admittedly, CODEIF does not blend well with the language.

There are advantages to such a simple form, however. A short, simple syntax works nicely if the code segment is short. For example, in a declaration, conditionally coding the packed attribute can be expressed with:

```
$CODEIF(pk,packed)
```

which is probably more readable than an if-then syntax. And while the syntax of CODEIF is unlike Pascal, it is like all macro calls in MAP. At least, it is not a minor variation on the Pascal syntax, a design that can be confusing.

Macros, too, have been restricted to keep their design in the spirit of Pascal:

- A macro must be defined before it is called, even if its value is null (requiring all definitions before the source seems too restrictive).
- Actual arguments to user defined macros may not contain calls of other macros, although macro bodies may contain calls of other macros.
- The number of arguments in a macro call must match the number of formal parameters given in the definition.

These restrictions contrast sharply with most macro preprocessors (cf. [13]). In fact, the second restriction actually changes the computational power of the macro facility; macros cannot be used, by themselves, to simulate a Turing Machine. By restricting actual parameters, however, side-effects from their evaluation are eliminated. Therefore, order of evaluation, multiple evaluations, and time of evaluation can be changed freely without changing the value of the macro; users need not be concerned with the implementation of the macro evaluator.

4. Implementation:

Two implementations of MAP were considered: a modification of the local Pascal compiler, and a separate preprocessor independent of the compiler. The latter was chosen because:

1. A preprocessor would be easy to port to a new machine, while a compiler would not,

2. Macro processing and including source files would increase the size of the (already too large) compiler, probably forcing a two-pass strategy,
3. Changes to the compiler would have to be reapplied to each new release,
4. The time required to develop MAP independent of the existing compiler would be less than the time required to understand and modify the compiler, and
5. MAP was definitely designed as a programming tool rather than a language; modifications to the compiler would blur the distinction between standard Pascal and the extensions provided by MAP.

Perhaps the most interesting reason is #5, fear that users would forget the distinction between Pascal and the extensions provided by MAP. Since MAP was designed in a university environment where students learn Pascal, the author was especially conscious of the tendency among programmers to confuse a language with its implementation. Students already mistake the local FORTRAN extensions for FORTRAN IV; extending the Pascal compiler would only compound the problem. At the same time, maintaining two versions of Pascal simultaneously seemed unattractive.

Several disadvantages of the preprocessor solution became apparent. A preprocessor requires two passes over the source deck instead of one; some errors are not detected until the compiler examines the source; and it is difficult to associate errors with the source program without a compiler generated listing.

Users raised a more subtle point by insisting that MAP does not provide enough facilities. In particular, MAP does not provide an encapsulation mechanism such as those found in MODULA [15] or MODEL[8]. To provide encapsulation, input must be organized into "modules", each with its own CONST, TYPE, VAR, and PROCEDURE declarations. The preprocessor would then parse the program, map identifiers to unique symbols, and reorder declarations to group them correctly for Pascal. While such a facility would be nice, it was eventually rejected because it involved the design of an entirely new language, something that was far beyond the scope of the project.

Several implementation issues plagued the constant expression routine, despite its simple design. Real valued constants suffered loss of precision when they were converted to binary and back to decimal. For example, the declaration

```
CONST
    R = 12.3;
```

was translated into

```
CONST
    R=1.22999999999E+001;
```

It was even more surprising to learn that MAP could not preprocess itself. Trouble arose when a constant greater than MAXINT appeared in the source program. MAP could read the number and convert it to an integer, but could not divide by 10 to convert back to decimal. These problems were eliminated by keeping tokens in string form until they were needed in an expression. Thus, the CONST assignments not involving an expression are never converted to internal form -- MAP passes them on to Pascal unchanged.

Macros, although apparently difficult to manage, were simple to implement. MAP uses four string stacks for macro bodies, actual parameters, CONST strings, and conditional code, growing them from opposite ends of two arrays. The environment is maintained by two independent stacks, one for macro calls and one for included files. To simplify the interaction between them, includes are restricted so that the remainder of the input line on which an INCLUDE appears is ignored. This applies to macro calls in progress too, so the macro stack is always popped when a new file is opened. Without the restriction, input lines would have to be stacked when a new file was opened. Since there is no convenient way to merge lines with any of the existing stacks, yet another string stack would be required.

Once the macro call stack had been devised, the processing of parameters in macros became simple. MAP thinks of a formal parameter in the body as a macro call with no parameters. By pushing an entry on the macro stack that refers to the "body" of the actual parameter, MAP switches the input stream to the actual parameter. The end of the actual parameter is marked like the end of any other macro, so the same mechanism that terminates macro expansion pops the environment stack back to the macro in progress. Since actual parameters cannot call other macros, there can be at most one actual parameter entry on the stack at any time, and it must be the top entry.

One final advantage of the current implementation is that MAP changes most non-standard Pascal into standard Pascal. For example, the CDC extended character set operators are translated into standard Pascal operators (e.g. "≤" is changed to "<="). Thus, the source program that MAP produces is often more portable than the program it received.

5. Conclusions:

A simple tool for developing large Pascal programs has been constructed. By providing constant expression evaluation, macro substitution, source file inclusion, and conditional compilation facilities, MAP eases the development process significantly. In most cases, MAP blends well with Pascal because the design was kept simple, clean, and modular. Each of the extensions provided by MAP can be used independently, and programs which use none of them slip through unscathed.

Most programmers agree that there is no single language or system that satisfies everyone's needs. Pascal is no exception. In our environment, however, Pascal is the only reasonable choice for implementing large systems; we are using a language and a compiler for much more than it was intended. Yet language designers and compiler writers must learn to anticipate the large programs that will be developed using their products. Features such as source file inclusion, constant expression evaluation (with time and date stamps), conditional compilation, macro substitution, and encapsulation should be standard tools. Perhaps the next generation of programming languages will provide them.

References

1. P. J. Brown, 'The ML/I Macro Processor', Comm. ACM 10, 168-173 (1967).

2. P. J. Brown, 'Using a Macro Processor to Aid in Software Implementation', Comput. J. 12, 327-331 (1968).

3. R. Conradi, 'Further Critical Comments on the Programming Language Pascal, Particularly as a Systems Programming Language', Sigplan Notices 11, 8-25 (1976).

4. A. N. Habermann, 'Critical Comments on the Programming Language Pascal', Acta Informatica 3, 47-57 (1973).

5. IBM, OS PL/I (F) Language Reference Manual , Form GC28-8201, IBM Corp.

6. IBM, OS Assembler Language, Form GC28-6514, IBM Corp.

7. K. Jensen, and N. Wirth, Pascal User Manual and Report, Springer Verlag, Berlin, 1974.

8. R. T. Johnson, and J. B. Morris, 'Abstract Data Types in the Model Programming Language', Proc. Sigplan Symposium on Data: Abstraction, Definition, and Structure, Sigplan Notices 11, 36-46 (1976).

9. B. Kernighan, and P. Plauger, Software Tools, Addison Wesley, Reading, Mass., 1976.

10. B. Kernighan, and D. Ritchie, The C Programming Language, Prentice Hall, Englewood Cliffs, New Jersey, 1978.

11. D. McIlroy, 'Macro Instruction Extensions of Compiler Languages', Comm. ACM 3, 214-220 (1960).

12. A. Mickel, ed., Pascal News 1-11, University of Minnesota, Minneapolis, Minnesota, -1978.

13. C. Strachey, 'A General Purpose Macrogenerator', Comput. J. 8, 225-241, (1965).

14. N. Wirth, 'The Programming Language Pascal', Acta Informatica 1, 35-63 (1971).

15. N. Wirth, 'Modula: A Language for Modular Multiprogramming', Software -- Practice and Experience 7, 3-35 (1977).

16. N. Wirth, 'The Design of a Pascal Compiler', Software -- Practice and Experience 1, 309-333 (1971).