A PROPOSAL FOR COMPILE-TIME FACILITIES

by

Robert A. Wagner

TR 71 - 104

September 1971

Department of Computer Science
Cornell University
Ithaca, New York

A Proposal for Compile-time Facilities

ABSTRACT

A novel, simple scheme for providing compile-time facilities to PL/I programmers is proposed. The scheme emphasizes language unity and implementation ease at the expense of syntactic nicety. A similar approach is possible in high-level languages other than PL/I, assuming they include adequate character string processing facilities. This paper describes the scheme, and attempts to analyze its advantage and shortcommings.

Key words and phrases: macros, macro processing, compile-time facilities, compile time macros, compiler macros, high level language macros.

CR categories: 4.21, 4.22

Authors name and address:

Robert A. Wagner

Department of Computer Science

Upson Hall

Cornell University

Ithaca, New York 14850

A Proposal for Compile-Time Facilities

by

Robert A. Wagner

Compile-time, or macro facilities have long been present in assemblers (1,2). In the assembler, they serve several purposes, allowing the programmer to introduce his own syntactic abbreviations, and to cause various, possibly complex, calculations to be used at assembly time in constructing the program which is ultimately assembled. Many workers (3,4,5,6,8) have proposed analogous facilities for use with high-level languages like PL/I or ALGOL, yet none seems entirely adequate.

Part of the difficulty in designing an adequate compile-time facility stems from a confusion of purposes. An ideal macro system apparently should:

1) allow control over the sequence of instructions generated for each "operator" defined in the language
2) allow specification of the syntax of new language elements; and
3) allow specification of the interpretation or expansion of new (or possibly old) language elements into pre-existing ones.

The first of these macro system design objectives must be ruled out, or greatly restricted, if the language-cum-compile-time-facilities is to be machine independent. The second and third objectives are quite feasible. These goals can be achieved by "expanding" a mixture of macros and source program text into basic (macro-free) source program text.

To allow the user adequate flexibility in designing macros, and specifying their expansion, a special procedural language is generally defined. This language, together with certain "substitution rules" allows the user to specify:

1) computations to be done at compile-time,
2) material to be copied unchanged from the macro source to the "expanded" text, and
3) compile-time variables whose values are to be copied into the expanded text.

These three elements seem common to all macro systems which operate by expanding macro-source text into macro-free text in some basic language.

As an example, PL/I-F (3) provides macro facilities based on these principles. Specifically, statements to be executed at macro time (type 1) are preceeded by a '%' character; statements to be copied into the expanded text are simply not preceeded by a '%' character. Within the latter statements, variables declared as "compile-time" are subject to "substitution"; their compile-time values (at the time the statement is expanded) are copied into the expanded text, replacing the variable's name.

The compile-time procedural language of PL/I-F is a rather sparse subset of the basic PL/I language. This leads to some learning difficulty, since someone used to "basic" PL/I must eliminate subscripted variables, non-integer numbers, and complicated DO statements from his vocabulary before using "compile time PL/I."

A very simple and powerful solution to the provision of compile-time facilities is possible: Compile and execute a PL/I program designed to produce a PL/I program. Then, compile the resulting program normally.

The facilities essential to a compile-time macro processor are adequately provided for in PL/I. Statements to be executed at compile-time are simple, ordinary PL/I statements. Material to be copied into the expanded text can be enclosed in string quotation - marks, and placed in the expression - list of a

PUT FILE(SYSUT1) EDIT (list) (SKIP,A)

statement. Similarly, variables whose value is to be substituted into the expanded text can also appear in the expression list of such a PUT statement, but outside string quotation marks. The expanded text appears on auxiliary file SYSUT1 in this example, where it can be immediately compiled and discarded, or punched and compiled later.

Programs to be compiled by the PL/I-F macro processor can be translated quite simply to the new notation.

For example, the PL/I-F procedure:

```
DOT: PROC(X,Y);
% DCL I FIXED;
SUM = 0;
% DO I = 1 TO 10;
   SUM = SUM + X(I)*Y(I);
% END;
RETURN(SUM);
END DOT;
```

becomes:

```
PUT FILE(SYSUT1) EDIT(
    'DOT: PROC(X,Y);',
    'SUM = 0;',
    ('SUM=SUM + X(',I,')*Y(',I,');'
          DO I=1 TO 10),
     'RETURN (SUM);',
     'END DOT;')
(SKIP,A);
```

Using a PL/I program to produce PL/I programs is slightly more awkward than generating PL/I programs via PL/I-F compile-time facilities. The extra typing

is offset by the attractive features of the method:

1) Only one language need be learned, to use compile-time facilities;
2) All the facilities of PL/I can be used in the program generation process.
3) Simplicity of construction: No special preprocessor program is needed to provide this type of compile-time facility; the PL/I compiler is its own pre-processor.

A PL/I program can generate an "expanded" program in some novel and useful ways:

1) Portions of the generation program can be read from an auxiliary file. This text can be modified before being incorporated into the expanded output text. An operation of this sort raises the possibility of translating input text into a PL/I program, using another PL/I program as translator.
2) A "cascade" of program generations can take place; Program 1, when compiled and executed could produce program 2, which when compiled and executed produces program 3, and so on. The depth of such nested program generation must be controlled external to the compilation process (by Job Control Language commands) but is otherwise unlimited.

Obviously, the technique is not limited to the PL/I language alone. Any language with reasonable string processing facilities can act as a "compile-time" pre-processor for any other language. If convenient, a PL/I program could produce assembly-language statements, acting as an assembler macro preprocessor. Or, a SNOBOL program could be used to produce PL/I programs, (or vice versa).

There remains the question of efficiency. A conventional macro system is usually implemented as an interpretive macro preprocessor. Often this interpreter operates as part of the first pass of the base compiler, replacing the base compiler's input mechanism. The use of a PL/I program as a program generator requires that the PL/I program be compiled and executed, before compilation of the "base" program can start.

Under some circumstances, the interpretive macro system clearly has advantages. Where most of the source text is merely copied into expanded text, the interpreter is clearly faster since only one scan of the copied material is needed. On the other hand, if iteration takes place at compile time, as it does in complicated user-defined macros, the advantage lies with the compile-and-execute approach.

The usefulness of the compile-and-execute approach is greatly enhanced by the existence of a fast compiler. For example, the PL/C (7) compiler for PL/I compiles typical program generators in less than 1 second of IBM/360-65 CPU time. Actual program generation-time (program generator compile time plus execution time) might typically be 2.5 seconds. This may well compare favorably with the interpretive macro-system approach in many cases.

The syntax of the macro source program probably constitutes the worst defect of the PL/I program used as a program generator. An interpretive macro processor distinguishes among the three elements of a compile-time program (executed, copied, or substituted elements) by non-local syntactic clues. Usually a statement remote from the statement undergoing substitution announces which identifiers or other language constructs are subject to substitution, say. This allows the "substituted" and "copied" program elements to have familiar, simple forms. It also allows changes in the remote "announcement" statement (ie %DECLARE in PL/I-F) to drastically modify the interpretation of the "copied" and "substituted"

program elements. Changes in the "announcement" statement can totally change the parsing of the source text, causing chunks of macro source text to be switched from "copied", say, to "substituted", or "executed-at-compile-time." This syntactic flexibility is not easily achievable in the program-generator scheme.

One alternative to the missing syntactic flexibility of remote specification of the recognition rule for macro elements is possible, however. A PL/I program can be written to read and translate a source program written in an arbitrary language into PL/I. Such a program could be thought of as a remote specification of a recognition rule for each of the three macro elements. These elements could alternatively appear in the translator's input, rather than as part of the translation. Of course, a translator as described here could equally well be thought of as a compiler. It might have to be quite complicated.

In summary, a very simple, general scheme has been proposed for incorporating compile-time facilities into any high-level language: Write a PL/I program which generates programs in the target high-level language. A few of the implications of such a method have been described.

The advantages of this "program generation" scheme have been shown to include language unity, generality, and implementation ease. In fact, the PL/I implementation of this macro-facility is the compiler itself, requiring no additional preprocessor phase. The scheme's main difficulty is its lack of a sophisticated, built-in "element recognition" mechanism. Such a mechanism would allow its users to specify the syntax of the program elements subject to the operations of "compile-time execution", "copying to expanded text", and "value substituted into expanded text". The scheme seems most usable if a fast PL/I compiler is used to translate the program-generator.

References

1. FAP: Reference Manual. Form C28-6235, IBM Corporation. (September 1962).

2. IBM ØS/360 Assembler Language. Form C28-6514, IBM Corporation. (December 1967).

3. IBM ØS/360 PL/I (F) Language Reference Manual. Form GC28-8201, IBM Corporation. (October 1969).

4. Brown, P. J. The ML/I Macro Processor. Communications of the ACM, 10,10 (October 1967), 618-623.

5. Galler, B. A. and Perlis, A. J. Compiling Matrix Operations. Communications of the ACM, 5,12 (December 1962), 590-594.

6. Galler, B. A. and Perlis, A. J. A Proposal for Definitions in ALGOL. Communications of the ACM, 10,4 (April 1967), 204-219.

7. Morgan, H. L. and Wagner, R. A. PL/C: Design of a High-Performance Processor for PL/I. Proceedings of SJCC (1971), 503-510.

8. Waite, W. M. A Language Independent Macro Processor. Communications of the ACM, 10,7 (July 1967), 433-440.