SLAC-PUB-1684
December 1975

EXPERIENCE WITH EXTENSIBLE, PORTABLE FORTRAN EXTENSIONS*

A. James Cook
Computation Research Group, 88
Stanford Linear Accelerator Center
Stanford University, P. O. Box 4349, Stanford, California 94305

ABSTRACT

We assess the impact over a three-year period, of the macro-pre-precessor MORTRAN, and one of the languages it processes. We confine our assessment to SLAC and Stanford since, although MORTRAN has been widely distributed in the United States and to a lesser extent in Europe, we have no personal knowledge of its impact elsewhere. The impact is attributed to three factors: (1) portability, (2) compatibility (with existing FORTRAN libraries), and (3) extensibility, which is sub-divided into (a) extension of control structures,and (b) extension of data structures. We divide the impact into an "initital" impact which we relate to control structure extensions, and a "secondary" impact which we relate to data structure extensions. MORTRAN is currently being used at SLAC to process large production programs, some of which exceed ten thousand lines of MORTRAN source code.

INTRODUCTION

The idea of an intermediate-level target language into which higher-level languages could be translated is not new. In 1958 Strong (1) proposed a "Universal Compiler Oriented Language" (UNCOL) as a solution to the problem which may be briefly stated: Given M machines and N languages, one must write M*N compilers in order that a program in any of the N languages run on any of the M machines. Compilers for the N languages were to produce UNCOL code and the M compilers for the M machines were to translate from UNCOL, thereby reducing to M+N the number of compilers necessary.

A succinct if approximate description of our proposal may be made by analogy to the UNCOL proposal: Our "Intermediate Level Language" (ILL), replaces UNCOL. The N languages are specified by N sets of macro-definitions, all of which are used by one processor to translate the N languages to ILL. Moreover, this one processor can be written in ILL so that portability of the processor is assured. In this way the number of language processors is reduced to M+1, M of which already exist; they are called FORTRAN compilers. Ideally only one new processor need be written. But we are more realistic. We assess the probability that the number of processors will ever approach M+1 at essentially zero.

Specifically, we propose that families of higher-level languages may be specified by sets of macro-definitions, each set defining a language. Each of these languages would then be translated to the base language by a macro-pre-processor, itself written in the base language.

The MORTRAN macro-pre-processor demonstrates the feasibility of this approach using FORTRAN as the base language. (The "MORTRAN language" is only one of the languages currently processed by the MORTRAN processor.)

Even within the context of FORTRAN extensions that are implemented by macro-pre-processors we are not naive enough to suppose that the number of processors will not proliferate. We know of at least two that are derived (with our hearty approval) from MORTRAN, and one that adapts the MORTRAN algorithm for COBOL.

The point that we are trying to make is that macro-pre-processor implementations offer a tremendous advantage over ad hoc implementations; if properly done, they have a "built in" extensibility for the languages they process. (Indeed, we can emulate virtually all other "structured FORTRAN" pre-processors that we have seen (11) by writing macro-definitions.)

We view the problem in terms of extending the base language rather than compiling higher-level languages into the base language for the reason that the compiler view does not imply extensibility.

The issue of extensibility is at the heart of our proposal; portability and compatibility are, in our view, practical constraints that have been too long neglected.

The idea of using a macro processor to extend higher-level languages is almost as old as UNCOL. McIlroy (2) described it in 1960. More recently, Brown (3), Campbell-Kelly (4), and Wegner (5) have discussed such an approach.

We were encouraged to start experimenting by the example of Waite (6) who uses FORTRAN to bootstrap his STAGE2 macro processor, but did not follow his methods which are based on abstract machine modeling. Waite's STAGE2 runs on an abstract machine called the FLUB machine (for First Level Under Bootstrap). In Waite's terminology, we chose not to model our own machine, but instead to use the "FORTRAN machine", whose "design" is approximately fixed by the 1966 ANSI FORTRAN standard. In retrospect, we think MORTRAN is logically more of a descendant of the work of McIlroy and Strachey (7). Waite's ideas regarding portability were and are a strong influence.

* Work supported by the Energy Research and Development Administration.

(Submitted for publication in special issue of SIGPLAN Notices on language pre-processors)

BACKGROUND

MORTRAN was written as the author's own personal tool (weapon?) to deal with the problems of portability, compatibility and extensibility. Macro processors, like compilers, are essentially string manipulators, and since string manipulation in FORTRAN is not a joy, we first bootstrapped up to a string manipulation language in a series of 3 steps, each of which was was a macro-processor that processed the next higher level. (Now MORTRAN processes itself, using a set of special macros.)

(The Stanford Linear Accelerator Center (SLAC) has two IBM 370/168's, one IBM 360/91, a XEROX SIGMA5, and an assortment of PDP's, NOVA's and so on, so we are acutely aware that all FORTRANs are not alike. The answer to this is that one must generate (or program in) a subset of FORTRAN that runs on all the machines. Since most FORTRANs have been extended beyond the ANSI standard, a first cut would be to use the standard as the "subset". Better yet, Ryder (8) has specified a subset of the 1966 ANSI standard called PFORT (for Portable FORTran), and has written a program (The PFORT Verifier) that determines whether a given program meets this specification.)

MORTRAN was first shared with a few of the author's friends around 1971, at which time it was known as BS/360 (BootStrap, of course!). It was poorly documented and frequently changed, and it didn't exactly spread like wildfire. We were convinced of its utility, but reluctant to stabilize and document it. It was stabilized, renamed MORTRAN, and made generally available in late '72. The first "official document" (9) was printed in the summer of '73. It was given to SHARE in August of '73. When MORTRAN2 (10) was written it became necessary to start referring to the earlier processor as "MORTRAN1". We first became aware of the many other pre-processors for FORTRAN at a JPL-SIGNUM-sponsored workshop (11), where we promised that MORTRAN2 would be made generally available in January '75. In January '75 at CompCon75 (12) we promised it for April. In July '75 we made it! We gave it to SHARE and the Argonne Code Center. MORTRAN1 is a FORTRAN program of about 320 statements in one MAIN program. MORTRAN2 has 4 SUBROUTINES, 4 FUNCTIONS, and a MAIN program totaling (exclusive of comments) 777 lines.

INITIAL IMPACT

Since there is no formal instruction in MORTRAN at SLAC, the increase in its use may be attributed to a sort of "grapevine" effect. Those who learn it tell their friends. Before we started this paper we took an informal poll and concluded that the factors affecting programmer acceptance of MORTRAN are (1) programmer investment in learning, (2) ease of transition, (3) compatibility with existing libraries, and (4) extensibility. Extensibility is to some the most important, and to others of little interest at first.

The fact that the initial investment is minimal appears to be important. A programmer may be willing to invest quite a lot of time at later stages, but not initially. A programmer may begin writing MORTRAN programs after learning only four new "coding rules", after which he may proceed at his own pace. A physicist told me that the "break even point" (i.e., the point at which return exceeds investment) was for him about a week. It is only fair to point out that he was already familiar with ALGOL and PL/I, both of which are available at SLAC. This "break-even point" for FORTRAN programmers who have little or no knowledge of other higher-level languages varies, but is probably longer than a week.

Under "ease of transition" we include such things as not requiring programmers to "burn their bridges". GO TO's are allowed as well as alphanumeric labels and FORTRAN statement labels (numbers); we recognize their value during the transition period. There is no need to belabor the merits of (say) the EXIT statement. If a programmer wants to program without the GO TO, he will discover for himself that the EXIT statement is necessary if he is to avoid duplication of code, or setting obscure flags. We do not believe that an abrupt change is either necessary or desirable.

The value of compatibility with existing program libraries is obvious, so we allow programmers to insert sections of MORTRAN code in old FORTRAN programs and vice versa. Some programmers are converting old FORTRAN programs to MORTRAN "piecemeal"; if they must modify an old program, they recode portions in MORTRAN so that future modifications become easier.

Programmers who are involved in large projects are responsible on a day to day basis for the accuracy of their programs. They develop what might be called a "healthy skepticism". It is common for them to examine the generated code for the first week or so to reassure themselves that MORTRAN really works. Without exception, they start debugging from the MORTRAN source after a short period of such reassurance. The situation is not unlike that of the "machine language" programmers of years ago who debugged the code generated by the FORTRAN compiler rather than the FORTRAN source code.

We will resist the temptation to extol the virtues of structured programming (and MORTRAN's control structures in particular), and say in summary that the favorable initial impact was due to the ease of transition to the use of better tools, particularly better control structures.

An excellent example of what we mean by user-defined control structures is Zahn's Situation Case Statement (13, 14, 15) which is implemented by macros. It is not part of the "standard" MORTRAN control structures, but may be added by the user.

SECONDARY IMPACT

While the initial impact is explainable in terms of easy transition to programming with better control structures, the secondary impact, which is concerned with data structures and notation, is a more difficult matter. Control structure issues are much better understood than data structure issues.

To quote Brooks (16), "Beyond craftsmanship lies invention, and it is here that lean, spare, fast programs are born. Almost always these are the result of strategic breakthough rather than tactical cleverness. Sometimes the strategic breakthrough will be a new algorithm.... Much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of the program lies.... Representation is the essence of programming."

Control structures provide perhaps half of the solution to the problem of gaining intellectual mastery of difficult problems. Other tools are needed for representing data. Some data structures exist, others must be invented. Queues, stacks, linked lists and the like can be as useful to the "applications" programmer as they are to the "systems" programmer. Most of us have been thinking in terms of other data structures for years anyway, then laboriously mapping them into arrays and vectors, and obscuring the true form of the data in the process.

A macro facility permits us to perform much of this mapping automatically, so that we may write as well as think in terms of data structures like trees, or others of our own invention. Such mappings may be of general utility or they may be highly specialized.

As an example of the general utility type, Zahn (11) has mapped Algol-W's "records and references" (17) into FORTRAN arrays using MORTRAN macros.

Explaining specialized mapping is more difficult. Before attempting to describe the role of macros, we will quote Hoare (18).

"In presenting a theory of data structuring it is necessary to introduce some convenient notation for expressing the abstractions involved....

"Since these notations are intended to be used (among other things) for the expression of algorithms, it would be natural to conclude that they constitute a form of programming language, and that an automatic translator should be written....

"But this conclusion would be a complete misunderstanding of the reason for introducing the notations.... there are sound reasons why these notations must not be regarded as a programming language. Some of the operations... are grotesquely inefficient when applied to large data objects in a computer; and it is an essential part of the program design process to eliminate such operations in the transition between an abstract and a concrete program. This elimination will sometimes involve quite radical changes to both algorithm and representation, and could not in general be made by an automatic translator. If such expensive operators were part of a language... it is probable that many programmers... would have little feeling for what alternative representations and operations would be economic. In taking such vital decisions, it is actually helpful if a programming language is rather close to the eventual machine, in the sense that the efficiency of the machine code is directly predictable from the form and length of the corresponding source language code."

What we have called a "specialized mapping" is then a mapping by a programmer who understands both the abstract data structure and the base language onto which the he is mapping. The advantage of the macro definition is that he need perform this mapping only once. In the course of this mapping he may concatenate macro-definitions or nest them in order to create more tractable symbols, but he does not lose control of the generated program as he does with totally automatic data structuring facilities.

The power and flexibility of recursive macro-definitions can be a delightful discovery. To those whose experience with macros is limited to some "rigid format" assembler macro facility, we stress that we are talking about something that is qualitatively different. We were, of course, pleased to find that some graduate students in Computer Science at Stanford were using MORTRAN macros, and we were even more pleased when Chuck Zahn designed his own language and implemented it using MORTRAN macros.

But what pleased us most was finding that physicists to whom programming is a "necessary evil" were using macros in elegant and sophisticated ways including operators on data structures. We would like to convey the feeling of excitement that comes with discovery, and we hope to entice the reader to explore the uses of higher-level language macros. We do not claim that mastery of the use of macros comes easy, but that the rewards are worth the investment in time and effort.

Even here the initial investment is minimal. One may begin by writing simple string replacement macros of the form

'pattern string' = 'replacement string'

which causes all occurrences of the pattern string in a program to be replaced at pre-processor time with the replacement string. The need for parameters in macros will soon become obvious and with this need will come the motivation to learn to include parameters. Then to create macros that generate other macros, and so on.

Of course, as we indicated earlier, it is possible to use the processor as a means to write "structured programs" if you understand that term to mean the addition of control structures, like IF-THEN-ELSE, WHILE, FOR loops, EXIT, and so on. To us that is less than half of what is really needed, and within the context of FORTRAN extensions, macros offer so very much more.

SUMMARY

Portability and compatibility are practical issues that have been neglected too long. We have shown that a degree of portability, extensibility, and compability are all achievable by using FORTRAN as a base language and as the language to implement a macro-pre-processor to extend this base language to higher levels. We feel that the macro-pre-processor approach has an advantage over ad hoc pre-processors in that it has a built in facility for user-defined extensions. While MORTRAN demonstrates the feasibility of this approach, it still has many shortcomings. It is, we hope, a step in the right direction. And the right direction as we see it is toward user-defined higher-level languages.

We have indicated that at least one effort is underway to adapt the MORTRAN algorithm to COBOL, and we see no reason that this approach cannot be taken with any other higher-level language.

We do not dismiss lightly the problem of enforcing adherence to local conventions or standards for a given language extension. While we feel that such enforcement is the province of humans and not language processors we concede that in some cases enforcement by processors has merit. MORTRAN is "permissive" because we chose to make it so. We are well aware of techniques, many of which involve only macro-definitions, to enforce adherence to specified standards.

The controversy over control structures has been of educational value. Perhaps the controversy over data structures should also erupt into a public debate.

In any case, we feel that the issue of extensibility includes and transcends both. What Brooks has called the "strategic breakthrough" is and will remain for the forseeable future the province of humans. The macro-pre-processor is is a tool for human use.

ACKNOWLEDGEMENTS

Many, if not most of the sounder ideas incorporated in the processor were contributed by others. At the algorithm design level, two graduate students in Computer Science at Stanford, Len Shustek and John Zolnowsky contributed much their valuable time and efforts as well as their ideas. At the "user interface" level, John Ehrman and Chuck Zahn of were also generous with their time and ideas.

REFERENCES

1 Strong, J., et al., "The Problem of Programming Communication with Changing Machine: A Proposed Solution", Comm. ACM, V. 1, No.8, (Aug., 1958)

2 McIlroy, M., "Macro Instruction Extensions of Compiler Languages", Comm. ACM V. 3, No.4, 1965

3 Brown, P., Macro Processors and Techniques for Portable Software, John Wiley, New York, 1974

4 Campbell-Kelly, M., An Introduction to Macros, Macdonald, New York, 1973

5 Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, 1968

6 Waite, W., Implementing Software for Non-Numeric Applications, Prentice-Hall, 1973

7 Strachey, C., "A general-purpose Macrogenerator" Computer Journal, Vol. 8 pp 225-41. Oct. 1964.

8 Ryder,B., "The PFORT verifier", Software-Practice and Experience, Vol. 4, 1974

9 Cook, A., "A User's Guide to MORTRAN", Stanford Linear Accelerator Center, Computation Research Group Technical Memo No. 150, 1973

10 Cook, A. and Shustek, L., "A User's Guide to MORTRAN2", Stanford Linear Accelerator Center, Computation Research Group Technical Memo 165, 1975

11 Workshop on FORTRAN Preprocessors for Numerical Software", Pasadena, Calif. Nov. 1974

12 Cook, A., and Shustek, L., "MORTRAN2, A Macro-based Structured FORTRAN Extension", Conference Digest of IEEE Spring 75 Compcon.

13 Knuth, D., and Zahn, C. , "Ill-Chosen Use of Event", CACM, Vol. 8, No 6, (June, 1975)

14 Knuth, D., "Structured Programming with the GOTO Statement", Computing Surveys, Vol. 6, No. 4 (1974)

15 Zahn, C., "A Control Statement for Natural Top-down Structured Programming", Proc. of a Programming Symp., Springer-Verlag, Berlin (1974)

16 Brooks, F., The Mythical Man-Month, Addison-Wesley, 1975

17 Sites, R., Algol-W Reference Manual, Stanford Computer Science Department, Stanford University

18 Dahl, O., et al, Structured Programming, Academic Press, 1972