The History of Macro Processors in Programming Language Extensibility

P. J. LAYZELL

Department of Computation, University of Manchester Institute of Science and Technology, P.O. Box 88, Manchester M60 1QD

The evolutionary nature of computer programming has necessitated the continual development and enhancement of programming languages. In an effort to make programming languages more responsive to such changes, a number of language extensibility schemes have been designed. This paper examines the history of macro processors as a tool for programming language extensibility.

1. INTRODUCTION

Over the past 20 years, much has been written in computing literature about macro processors. In a recent survey on macro processors and extensible languages over 130 references were given.[1] But despite the wealth of literature in this area, very little appears to have been written about the history of macro processors, especially in the field of programming language extensibility.

It is the purpose of this paper to give a brief résumé of the history of macro processors and their association with programming language extensibility. It is not intended to give a full description of any macro processor or programming language extensibility scheme.[2,3]

2. PROGRAMMING LANGUAGE DEVELOPMENT

Before examining the development and role of macro processors in programing language extensibility, it is first necessary to place such a history in perspective by looking at the need for the development and extensibility of programming languages.

The growth of computer applications has led to the development of numerous specialised programming languages and their translators (compilers, interpreters and assemblers) and has placed a heavy burden upon installations which attempt to support an increasing number of them, since more and more systems programming time has been directed to their maintenance and upkeep.

The major drawback with conventional translators is that the features of a programming language which they support are fossilised at the time they are written. This gives rise to a number of problems.

First, users of translators will often have features built into their particular translator that, in some cases, are never used. For example, many COBOL users do not use the Report Writer and Telecommunications facilities that are provided.

Secondly, programming languages can often undergo considerable revision and development as programming techniques change. In both BASIC and COBOL, for example, standards are now being altered to allow structured programming constructs. However, a conventional translator would have to be extensively modified, if not completely rewritten to accommodate such changes.

Finally, the nature of applications problems may change and so require new language features such as new data types or control structures. This may also require the rewriting of a conventional translator.

As a result of these problems, considerable time and energy has been expended in the development of universal, all-purpose programming languages. The best known example, PL/1, provides a good illustration of the difficulties and drawbacks of attempts at such universal languages.

For a language to be universal, it must provide features for many diverse areas such as numerical analysis, data processing, engineering applications and compiler writing. Hence, the translator for such a language will become unmanageably large, with very few installations having the machine or support staff capacity to maintain it. Furthermore, any language, no matter what diverse areas it attempted to cater for, could never completely satisfy user whims, such as particular new statements or control structures.

As an alternative to numerous special-purpose languages or a single universal language, the notion of extensible languages received widespread support in the late sixties. Such a language would consist of a small but powerful set of 'core' or 'base' features, which could then be extended by the user so as to tailor his translator to his specific needs, thus keeping the translator size as small as possible.

3. THE DEVELOPMENT AND ROLE OF MACRO PROCESSORS IN LANGUAGE EXTENSIBILITY

The history of language extensibility spans a mere twenty years, but in this period it is possible to identify a very clear and definite trend. It can be seen from the published literature on various types of extensibility schemes, shown in Table 1, that there has been a drift from low-level language extension by relatively simple macro processing, through a stage of specially designed extensible languages, up to the present-day development of syntax macro processors, each designed for the extension of a single programming language.

3.1 Initial ideas

Much of the early development work in language extensibility dates back to the early and pre-sixties when macro assemblers were used to make assembly languages

Table 1. Chronology of language extensibility schemes

	Language-orientated text macro processors	Generalised text macro processors	Extensible languages		Generalised syntax macro processors	Language-orientated syntax macro processors
			By altering compiler	By macros		
Pre-1960		McIlroy				
1961						
1962	XPOP					
1963			Compiler–compiler			
1964						
1965		GPM TRAC		IMP	Cheatham	
1966	System/360	LIMP		MAD	Leavenworth	
		ML/1				
1967			ALEC	ALGOL D		
1968	PL/1	STAGE 2		BALM	GPL	COBRA
1969		PLITRAN		EPS PPL		
1970				PROTEUS	MACRO	MP/1
1971			BABEL	ECL	HAMMER WODON	
1972						
1973						
1974						
1975					CAMPBELL	
1976					STEP	
1977					PM	
1978	C					MCOBOL
1979						
1980						CLEF

Note. Dates are approximate, and where dates are not specified by authors of schemes the earliest published reference is used.

'higher-level' by allowing the definition of repetitive coding as macros. The initiative for this use of macros came mainly from computer manufacturers. Consequently little detailed published material exists except on such well-known schemes as the IBM System/360 Macro Assembler,[4] which is typical of most of the macro assembler schemes. The problem with many of these schemes was the limited formats permitted for the macro call. Typically, all macro calls would be restricted to a format such as:

macro-name, parameter-1, parameter–2,...

although XPOP[5] permitted marginally more flexible macro call formats. Thus it would be possible to write:

STORE INTO GAMMA THE SUM OF ALPHA AND BETA

or

STORE GAMMA = ALPHA + BETA

instead of:

STORE GAMMA, ALPHA, BETA

In its infancy, the main aim of language extensibility was to reduce programmer coding time by providing shorthand facilities and, to a lesser extent, improve readability. Therefore the early macro processors were simple text replacement schemes with few other capabilities and can now be found in wide use today in languages such as PL/1 and C.

3.2 Generalised macro processors

With the development in the early and mid-sixties of higher-level programming languages, assembly languages became less popular. The advent of COBOL and FORTRAN meant that programmers no longer had to write in low-level languages, and so the apparent need for readability and high-level statements in programming languages had been fulfilled to a large extent.

Initial stimulus in the area of extending higher-level programming languages came from McIlroy's paper,[6] which was to act as a foundation for many subsequent macro processor and language extensibility schemes.

McIlroy proposed a macro processor which could conditionally replace text, depending upon details within the macro call. Furthermore, he envisaged nested macro calls and the need for a facility to generate unique symbols so as not to conflict with those symbols defined by the applications programmer.

Despite McIlroy's paper, most of the early macro processor schemes did not incorporate all of these features. Many of the schemes, such as GPM,[7] LIMP[8] and TRAC[9] had limited facilities which considerably restricted the format of the input text. LIMP, for example, allows only nine formal parameters to be specified in a macro call format. Furthermore, the macro

defining language of many of the early schemes was clumsy and restrictive and often as unreadable as the assembler languages the earliest schemes had tried to improve.

However, Brown's ML/1 macro processor[10] permitted a much greater degree of flexibility in the format of macro calls, by allowing nested calls and by the specification of optional, alternative and repeated elements.

The power of ML/1 is considerably enhanced by the availability of inter-macro communication through a set of 'macro-time' registers, as well as conditional text replacement and the generation of unique labels and names.

3.3 A move towards extensible languages

Despite the success of the second generation of macro processors to implement and extend programming languages, such as the use of ML/1 to extend BASIC[11] and GPM to implement CPL, there was a move in the late sixties towards new programming languages with built-in extensibility. It seems likely that there were two main reasons for this.

First, there was a question of operating efficiency. Macro processors operating as pre-processors add significatly to compilation costs and the generalised nature of the second generation of macro processors meant that their operation was slow.

Secondly, generalized macro processors do not fully validate macro calls and hence can produce replacement text that contains errors, thus causing problems for the applications programmer with two error reports. For example, Brown in his paper on extending BASIC to permit string handling says that the macros used may generate the erroneous BASIC statement:

151 LET 23 = S (Z1),

which would result from the fact that the macro processor used did not fully validate the macro call. Therefore, it is left to the BASIC compiler to report the error and so provide two separate error reports for the applications programmer to handle.

The new extensible languages were often deliberately developed with minimal facilities which could then be extended to fit a particular user's application area. Such extensions could be new data types such as records for commercial data-processing applications, complex numbers for engineering applications, new operators to manipulate the new data types and new statement forms.

Such was the interest in extensible and universal languages at this time that the Special Interest Group on Programming Languages (SIGPLAN) of the ACM held two extensible languages symposia in 1969 and 1971, attended by many of the 'founding fathers' of language extensibility schemes such as Cheatham, McIlroy, Irons, Perlis and Standish.

Many of the schemes outlined at these symposia, such as GPL,[12] IMP,[13] PROTEUS[14] and PPL[15] resulted in considerable euphoria and excitement about extensible languages, and by May 1975 Standish had noted 27 extensible language schemes including extensible compilers such as ALEC and BABEL.[16]

However, at the 1969 symposium delegates were quick to point out that extensible languages were unlikely to be the complete salvation of the programmer. Cheatham pointed out that programming is complex and that 'maybe the extensible languages idea might help one to organize his [the programer's] complexity a little bit better'.

It was soon discovered that extensible languages were, indeed, not to be the complete salvation of the programmer. As Hoare bluntly put it, extensible languages have had, 'a great lack of success'.[17] The main reasons for the failure of extensible languages are as follows.

First, extensible languages are complex. Brown in a recent paper on macros[18] highlights the problems of having to learn new macro defining languages; similar arguments apply equally well to the learning of new extensible languges especially when the majority of programs are still written in non-extensible languages such as COBOL and FORTRAN.[19]

Secondly, at the second symposium on extensible languages in 1971, Standish exposed the erroneous belief that with a handful of simple mechanisms such as syntax macros, applied in simple ways, one can obtain both the variability and efficiency demanded from extensible languages.[20]

Thirdly, the new extensible languages had to compete with the now well-established languages such as COBOL and FORTRAN, which became increasingly popular. Extensible languages have therefore suffered the fate of many other languages – an inability to present a serious challenge to the already existing languages – despite their obvious advantages.

Finally, when examining the various extensibility schemes, two points become apparent. The more successful schemes are the simpler schemes, the moral being that it is better to produce a simple but usable scheme with only a few features rather than a complex and sophisticated, unusable scheme. In addition, if you start with a simple base language you must often apply considerable effort to extend it to approach anything like a usable high-level language.

3.4 The development of syntax macros

In 1966, before the main development of extensible languages, ideas for improved macro processors to extend already existing languages were being floated by Cheatham and Leavenworth[21, 22] The ideas envisaged a special type of macro called a 'syntax macro'.

A syntax macro takes advantage of the syntactic structures of programming languages. It can do this in a number of ways, the main one being to specify the syntactic class (i.e. statement, identifier, literal, etc.) of formal parameters within a macro call.

For example, in the macro call format:

DO integer TIMES statement

as well as matching the keywords 'DO' and 'TIMES', a syntax macro processor would also check that the first actual parameter was an integer and that the second actual parameter was a statement.

The advantage of a syntax macro-scheme is that much of the responsibility of syntax checking is moved from the macro writer to the macro processor. A number of generalized schemes using this approach have now been developed, such as MACRO[23] and STEP.[24]

Each of these schemes represents a consolidation of the more useful ideas developed in earlier macro schemes. Features included in these schemes are extremely flexible

macro call formats that allow virtually any form of construct to be specified. For example, macro calls do not have to begin with a herald or have a unique delimiter. Alternatives, repetitions and options, as well as the assignment of a syntactic class to formal parameters, are also allowed. Finally, there are flexible macro-time instructions to manipulate generated text and to allow communication between macros.

It is becoming increasingly apparent that whilst these generalized syntax macro processors are unlikely to fail on the basis of a poor macro definition language, they may well fail on an inability to deal conveniently with both the fundamental structures of any language and also its peculiarities.

As a result, a number of single-language orientated schemes have been developed, in particular for COBOL and FORTRAN. For COBOL, these include COBRA[25] and MetaCOBOL,[26] which are both commercially available COBOL macro processors. MCOBOL,[27] later refined and renamed CLEF,[28, 29] is a COBOL extensibility scheme still under development.

All these schemes offer special, COBOL-orientated macro processing facilities such as sophisticated text distribution schemes and, in the case of MetaCOBOL, MCOBOL and CLEF, comprehensive symbol tables.

The CLEF scheme represents a departure from most macro processors since it is envisaged to form an integrated part of a COBOL compiler. It is also a very high-level, powerful, syntax macro processor. The effect is to reduce the inherent inefficiencies in existing COBOL macro processors and to provide a language extensibility scheme for a widely used, commercial language.

The macro processor MP/1[30] is orientated towards FORTRAN and, like the COBOL schemes, automatically handles the formatting of generated text.

Whilst some of these schemes have been in existence for some time, they have had only limited use, and it remains to be seen what degree of success the more recent schemes will have.

4. SUMMARY

The history and development of macro processors in programming language extensibility has been varied and interesting. In their formative years, macro processors were simply used to enhance the readability of assembly language programs. However, this role was outgrown by the development of high-level programming languages which meant that there was less need for assembly language programming. There then arose a need for language extensibility due to the increasingly diverse areas of application which in the words of Halpern[31] meant that 'if the universal language is nowhere in sight, then for a programming system, as for other creatures competing to survive, ability to change is the first law of life'.

Attempts to produce extensibility have been made in a variety of ways. At first there was a move towards new programming languages with built-in extensibility. However, a number of factors including the heavy investment in the already-established languages and the complex nature of the new extensible languages prevented their widespread use.

These were paralleled, though not superseded, by syntax macro processors, some of which are general and others orientated towards specific programming languages. Their aim has been, to a greater or lesser extent, to take advantage of the syntax and idiosyncrasies of the particular languages that they are trying to extend.

Despite all this activity and the many claimed advantages, macros still remain outside mainstream computing – acclaimed by many, but ignored by most.

Acknowledgement

This paper was written with the aid of financial support from the UK Science and Engineering Research Council. The author would like to thank Alan Heywood-Jones of Cobra Systems and Programming and John Triance of UMIST for their valuable help and comments on this paper.

REFERENCES

1. J. R. Metzner, A graded bibliography on macro systems and extensible languages, *SIGPLAN Notices* **14**, 1, 57–68 (1979).
   A comprehensive bibliography of macro and extensibility schemes.
2. P. J. Brown, A survey of macro processors, annual review in *Automatic Programming*, vol. 6, Pergamon Press, pp. 37–88 (1969).
   A survey of the main macro processors.
3. N. Solntseff & A. Yezerski, A survey of extensible programming languages, annual review in *Automatic Programming* 7, part 5, Pergamon Press (1974).
   A comprehensive survey of extensible languages.
4. IBM, *IBM Operating System/360: Assembler Language*, C28–6514 (1966).
   An introduction to the IBM 360 macro facility.
5. M. I. Halpern, XPOP: A meta-language without metaphysics, *Proc. FJCC., AFIPS* **26**, 57–68 (1964).
   An introduction to XPOP.
6. M. D. McIlroy, Macro instruction extensions of compiler languages, *Comm. ACM.* **3**, 4, 214–220 (1960).
   Describes the use of macros in compiler extension.
7. C. Strachey, A general purpose macrogenerator, *Computer Journal* **8**, 3, 225–241 (1965).
   An introduction to GPM.
8. W. M. Waite, A language-independent macro processor, *Comm. ACM.* **13**, 7 (1967).
   An introduction to LIMP and STAGE2.
9. C. N. Mooers & L. P. Deutsch, TRAC – a text handling language, *Proc. ACM. 20th National Conf.*, pp. 229–246 (1965).
   An introduction to TRAC.
10. P. J. Brown, The ML/1 Macro Processor, *Comm. ACM.* **10**, 10 (1967).
    An introduction to ML/1.
11. P. J. Brown, *Extending High-level Languages by Macros – a Practical Evaluation*, Software 72, Transcripta Books, 95–99 (1972).
    Describes extending BASIC using macros.

12. J. V. Garwick, GPL, a truly general purpose language, *Comm. ACM.* **11**, 9, 634–638 (1968).
An introduction to GPL.
13. E. T. Irons, The extension facilities of IMP, *SIGPLAN Notices* **4**, 8, 18–19 (1969).
An introduction to IMP.
14. J. R. Bell, Transformations: the extension facility PROTEUS. *Proceedings of the SIGPLAN Extensible Languages Symposium* (1969).
An introduction to Proteus.
15. T. A. Standish, Some features of PPL, *Proceedings of the SIGPLAN Extensible Languages Symposium* (1969).
An introduction to PPL.
16. T. A. Standish, Extensibility in programming language design, *Proc. SJCC., AFIPS* **44**, 287–290 (1975).
17. C. A. R. Hoare, *Hints on Programming Language Design*, Stanford Artificial Intelligence Lab., Memo Aide 224, CS-403 (1973).
Notes on the good design of programming languages.
18. P. J. Brown, Macros without tears, *Software – Practice and Experience* 6, **9**, 433–438 (1979).
Describes the problems of using macro processors.
19. M. M. Al-Jarrah & I. S. Torsun, An empirical analysis of COBOL programs, *Software – Practice and Experience* **9**, 5, 341–359 (1979).
Presents some statistics on COBOL programs.
20. T. A. Standish, PPL – an extensible language that failed, *SIGPLAN Notices* **6**, 12, 144–145 (1971).
A discussion on the failure of extensible languages.
21. T. E. Cheatham, The introduction of definitional facilities into higher level programming languages, *Proc. FJCC, AFIPS* **29**, 623–637 (1966).
Discusses the use of macros in high level languages.
22. B. M. Leavenworth, Syntax macros and extended translation, *Comm. Acm.* **9**, 11 (1966).
An introduction to the use of syntax macros.
23. S. R. Greenwood, MACRO: A programming language, *SIGPLAN Notices* **14**, 12, 80–91 (1979).
An introduction to MACRO.
24. J. W. Simpson, *The STEP Processor*, Computation Research Group, Stanford Linear Accelerator Center, CA, USA (1977).
An introduction to STEP.
25. Cobra Systems and Programming, 21 Green Hill Road, Camberley, Hampshire, England, *COBRA User Manual* (1982).
An introduction to Cobra.
26. ADR, *Macro facility reference manual*, SM2G-01-00. Princeton, NJ, USA (1977).
An introduction to Meta-COBOL.
27. J. M. Triance & J. F. S. Yow, MCOBOL – a prototype macro facility for COBOL, *Comm. ACM.* **23**, 8, 432–439 (1980).
An introduction to MCOBOL, a COBOL macro facility.
28. BCS. *CLEF Journal of Development*. UMIST, Manchester, England (1980).
An introduction to CLEF.
29. J. M. Triance, The design and evaluation of a language enhancement facility for COBOL, M.Sc. Thesis, UMIST, England (1982).
An introduction to CLEF, a COBOL macro facility.
30. I. A. Macleod, MP/1 – A FORTRAN macroprocessor, *Computer Journal* **14**, 3 (1971).
An introduction to MP/1.
31. M. I. Halpern, Towards a general processor for programming languages, *Comm. ACM.* **11**, 1, 15–25 (1968).