AN EXTENSIBLE PROGRAMMING SYSTEM*

by

E. Kantorowitz

Technical Report No. 26
January 1973

*This work has been sponsored partly by the European Research Office of the U.S. Army under contract JA 37-72-C-3569.
Common programming systems are very wasteful, since facilities for doing practically the same things, e.g. coding a loop, are implemented anew in each of the programming languages of the system. This waste may be avoided if the same compiler is used for all the languages and if the facilities which are needed in more than one language are implemented only once and shared by all the languages which need them. A system satisfying these criteria is being implemented using a syntax macro processor as a common compiler. A macro processor is probably not the fastest compiler, but it provides the programmer with convenient means for extending the programming languages employed to fit the problems encountered, in order that highly efficient problem solving tools may be achieved. The system further enables complete control of the underlying hardware and operating system, such that efficient code may relatively easily be produced. A major disadvantage of a system satisfying the above criteria is that it cannot include some of the common languages, such as FORTRAN and COBOL, because the facilities common for these languages, e.g. loops, are for the most not implemented in the same way, and may thus not be shared. The described system may be useful for system programming, advanced applications and special purpose computing systems.
INTRODUCTION

Most programming systems have to meet a wide range of needs. They must be suitable for very different applications and for programmers whose abilities range from very poor to excellent. There are three principal methods for meeting these needs:

1. Systems which support many programming languages, ideally "a language for each need". This is the most common solution, and it satisfies, by and large, the needs. The cost of producing and maintaining such a large multi-lingual system represents furthermore a considerable waste, because facilities for program control, data specification, etc. are duplicated in each of the languages of the system. There are of course differences between the way these facilities are implemented in the different languages, but they are often small. There is for instance no difference between the power of the IF THEN ELSE constructs of ALGOL 60 and PL/I, and the small syntactical differences represent a difference of taste between the designers of the two languages. These differences are of course a burden for the programmer, who has to learn the many ways in which the practically same thing is said in the different languages.

2. A system which comprises a large "shell language" like PL/I, which ideally "satisfies all needs". Compared with the multi-lingual programming system, we now avoid the duplication of the programming facilities in the different languages as described above. A shell language seems, nevertheless, to result in a large and expensive system. The PL/I compiler for the OS/360 thus comprises of about $10^6$ bytes of code. It is furthermore difficult to
design a shell language which satisfies all the needs in an optimal way, which may be illustrated by Hopkins [1971] analysis of the suitability of PL/I for systems programming.

3. Extensible programming languages, like Algol 68, constitute the third possibility. Here the programmer has a mechanism for extending the "core language" with the data types and operators which are natural for his problem. Extensible languages may, therefore, be expected to be efficient problem solving tools, and Woodward's [1972] report of the experience with the use of Algol 68 at a large computing center is thus very promising.

The programming systems discussed hitherto are based on standard programming languages, such that a program may, relatively easily, be transferred from one computer to another. A disadvantage of the standard languages is that they do not enable a complete control of all the facilities of the underlying hardware and operating system. This is due to the difficulties in devising a machine independent language which has constructs, that corresponds to all the facilities of the many different operating systems. The solution in the conventional multi-lingual system, is to have a special systems programming language, normally an assembly language, through which all the facilities of the hardware and the operating system may be controlled. Programs which require a firm control of these facilities must at least be partly written in the systems programming language.

We shall, in this paper, present the principal ideas of a programming system which avoids some of the insufficiencies of the existing systems.

The design goals of the new system are:
1. The programmer should be able to extend the programming language employed to include constructs which corresponds to the natural elements of the problem encountered. This is an important feature because it increases the efficiency of the problem solving process considerably. Such language extensions may obviously only be made by programmers with appropriate experience.

2. The programmer should within any language in the system, be able to have a complete control over the facilities of the hardware and the operating system. If needed, he will thus be able to produce highly efficient code.

3. The system should be "minimal" for many reasons. The costs of producing and maintaining a small system at a high level of dependability are relatively low. The amount of effort required by the users to learn a small system are reasonable. A small system may be implemented on relatively small computers.

4. The programmer should be provided with powerful aids for debugging and documentation. This is a quite obvious requirement, since any rational program production process incorporates a thorough program verification.

These features include those by which a qualified programmer may produce sophisticated and efficient programs at reasonable costs, i.e. the ability to fit the programming language to the problems to be solved, adequate debugging aids and complete control over the facilities of the hardware and the operating system. On the other hand, design goals 1, 2 and 3 hinder the inclusion of the common standard languages in our programming system, which is a major disadvantage. Programs produced under the system may, nevertheless, often be designed such that they may be transferred from one computer to another at
tolerable costs ("be portable") by employing the techniques reviewed in Newey, Pool, and Waites paper [1972]. In conclusion, three areas of application may be envisaged for our programming system.

a) As the only system on a small or on a special purpose computer;

b) As a base into which a conventional multi-lingual programming system is embedded. Our system here plays the role of the system language in the conventional system. We are, however, in a better position than with a conventional systems programming language because we have an extensible language geared to a wide range of applications.

c) The system may be embedded in an existing multi-lingual system and used as a tool for advanced programming.
THE PROGRAMMING SYSTEM

A programming system which is expected to satisfy the above goals is being implemented. The system utilized Brown's [1967] macro processor ML/I as a common compiler for all the languages of the system. By using a common compiler for all the languages we contribute to the minimization of the system and by choosing a macro processor as a compiler we have automatically, and without extra costs, a mechanism for extending the programming languages. A disadvantage in using a macro processor as a compiler is that compilation is under certain circumstances relatively slow.

A program in our system is composed of ML/I macro calls. The syntax of a macro call in ML/I is

\[
<\text{delimiter}> \{<\text{argument}> <\text{delimiter}>\}. \\
\text{e.g.}
\]

\[
\text{SET} \quad A := B + C;
\]

There are 4 delimiters in this call, namely:

\[
\text{SET} \quad := \quad + \quad ;
\]

They are numbered from left 0, 1, 2, 3. The three arguments in the example are A, B, C and they are numbered from left 1, 2, 3. Delimiter, "SET" in the above example, is the macro name, while the last delimiter, ";" in our case, is denoted the closing delimiter.

A delimiter may be defined such that it may have a number of optional values. Delimiter 2 in our example could be defined to have the optional values "+" and "-". The call
SET A := B - C;
would then also be legal. The value of the actual delimiter, i.e.
"+" or "-" in our example, is tested during the macro expansion
and the corresponding code is generated. A macro may also be defined to
have a variable number of arguments. If this is done in our example, the
macro call
SET A := B + C - D + E;
is legal.

The arguments of a macro call may themselves be macro calls.
We may thus define a macro for the evaluation of a boolean expression and
use it wherever it is needed, for instance as an argument of an IF macro.

It is obvious that quite powerful languages may be translated by ML/I.
The most notable restriction is the requirement that a call is initialized
by the name of the macro. If we didn't have the macro name SET in the
above example, the macro call would have been a legal Algol 60 statement.
By initializing a macro call by its name, the processing is considerably
facilitated and some intricate parsing problems are avoided.

The ML/I processor may be employed for producing code in any desired
language. The choice of target language depends on whether our system is to
be embedded in an existing multi-lingual system or whether it is to be the
basic programming system of the computer. In the latter case, it will be
natural to let the ML/I processor produce relocatable machine code.

Any language of the system is defined by a set of macros, which are
described in the language manual. The programmer may extend this language
by defining new macros. The definitions of all the macros of all the languages are, however, stored in the same library, and a programmer can therefore employ macros belonging to other languages. The different languages of the system may thus be regarded as subsets of one large extensible shell language. We prefer, however, the multi-lingual view of the system because it suggests that the user has only to learn the languages which he needs.

THE COMMON MACROS

There are a number of facilities which are needed in most programming languages, such as mechanisms for producing loops. Following our minimization goal, these facilities should appear only once in our system, e.g. there will be only one set of macros for producing loops in our macro library. We shall now give a brief account of such four sets of common macros.

It is widely accepted that building programs from their constituents, such as blocks and procedures, is useful for both design, coding, verification and documentation. In the many languages which have facilities for program structuring, the programmer has to surround each program constituent by a pair of special symbols, e.g. begin end. Griffiths and Peltier [1969] have demonstrated that structuring of assembly language programs is quite easily achieved by bracketing each program constituent by a pair of special macros, which expand into appropriate branch instructions. We use this technique and define the macro pairs BEGIN-END for blocks, REPEAT-END for loop compounds, PROCEDURE-END for procedures. Program flow is similarly controlled by IF
THEN ELSE and CASE macros, and termination of a REPEAT-END loop is achieved by EXITIF macro.

Macros for specification of data and their types ("Declarations") are needed for many purposes, the most obvious is for storage allocation. The type specification represents useful semantic information, that the programmer should be able to utilize. It should thus be possible to define macros where the code generated depends on the type of the data handled by the macro. The programmer should, for instance, be able to define a macro such that if the data submitted to the macro have wrong types, code for data type conversion or for some error handling is produced. In order to enable such utilizations of the type information, the programmer needs a system macro, which gets the name of a data item, and after consulting the symbol table, it returns the type. Further to basic data types, i.e. data which are operands of machine instructions, such as INTEGER, REAL and CHARACTER, we need user defined structured data types. The structured data are composed of basic data and are defined by the user to fit his problem. Similar structured data types are found in many programming languages, e.g. in PL/I. In these non extensible languages, however, the programmer can only manipulate the structured data with the operators defined in the language, and which do not necessarily fit well to the problem solved. In our system the programmer defines the data handling macros, such that they produce exactly the operation he needs. The implementation of these facilities for specification of data and data types in the framework of ML/I is not obvious, because the processor was not geared to handle symbol tables, and some modifications are therefore necessary.
The structuring of the programs from their constituents, e.g. blocks, facilitates the implementation of some useful debugging aids. It is thus natural to design the macros employed for structuring, e.g. the BEGIN and END macros, such that their expansion will include code for printing messages, indicating entry and exit of the blocks, such that a report of the program flow is generated. The code for printing these messages will of course only be generated in "debugging runs", and will thus not burden "production runs". Tracing the change of values of specified variables is facilitated by the fact that data are handled by a limited number of macros. The body of these user defined data handling macros will have to contain a call of a system macro, which will test whether the variables affected by the macro are in the list of variables to be traced, and when this is the case produce the code for printing the trace message. This system macro will of course only produce code in "debugging runs", and only in the parts of the program that the programmer wishes to verify in the actual run.

The functions of many existing operating systems are activated by calling assembly language macros. This technique is obviously also suitable for our system.

The object of this paper is to present the principal ideas of the system. This description is intentionally sketchy because changes are expected through the experimentation with the system. A detailed account and a critical evaluation is planned when an adequate amount of experience is gained.
REFERENCES


