The influence of the structure of high-level languages on the efficiency of object code

G. MUSSTOPF
Scientific Control Systems GmbH, Hamburg, W.Germany

1. Introduction

Programming languages are a means of communication between man and machine. For each of these languages an alphabet (set of symbols), a syntax and a set of semantic rules are defined. Applying the syntax and semantics statements for a computer can be constructed from the symbols of the alphabet. If a language consists of only a few simple elements which are strongly adapted to the hardware of a given computer, it is called a low-level language. Algorithms which are to be described using such languages must first be manually prepared, i.e. translated. Other languages have a substantially larger set of elements and rules. The representation of formulae in such languages, for example, is that used by the mathematician. These languages are called high-level languages. The translation into a form acceptable to the computer is carried out by a special computer program. It is noteworthy that no measure exists for the level of a programming language. It is also important to note that an arbitrary number of levels exists between the two extremes.

One of the aims of developing a program with the help of a language is to perform the computation defined by the program. The object program as end product of the development, whether from manual pretranslation or from automatic translation, can be of varying quality. This can be seen, for example, from the speed and/or size of the object program. The quality of an object program is influenced by the quality of the translator, the language and the source program.

People who program computers have the wish to express themselves at as high a level as possible, i.e. they want to use the jargon with which they are familiar. The advantages of being able to do this are:

1. Short training period.
2. Short program development time.
3. High legibility of the source text.
4. Low test and maintenance effort.
5. Later modification easily carried out.

The comparison of an object program, which has been generated automatically by a translator from a high-level language, with a functionally equivalent object program which has been optimally written in a low-level language shows a large difference in the quality, i.e. efficiency, of the object programs. The size of this factor with respect to speed or size depends also on the structure of the computer (in addition to the reasons given above).

For many applications this loss is rightly accepted. There exist, however, apart from system software, very many problems in the field of real-time applications which require that the object programs be of high quality. The assertions in the following sections should be considered in connection with these problems.

2. Programming languages and programs

The classical machine and assembler languages are considered here solely as objects of comparison for the quality of object programs. This section classifies languages such as PL360 [1], PS440 [2], CORAL66 [3], BLISS [4], PASCAL [5] and POLYP [6].

A program consists of two different kinds of elements. The first kind is used to describe the application (application elements), e.g. testing measurements for critical boundary values. The second kind is required owing to the 'general or specific structure of the computer' (EDP elements), e.g. the specification of the lengths of items of information or the use of address variables. The use of this kind of element helps the translator to produce a better object program.

These two classes of elements cannot be distinguished by the operators of operands they contain.

The minus sign in

```
TN - TO
```

is used to represent a temperature difference, whereas that in

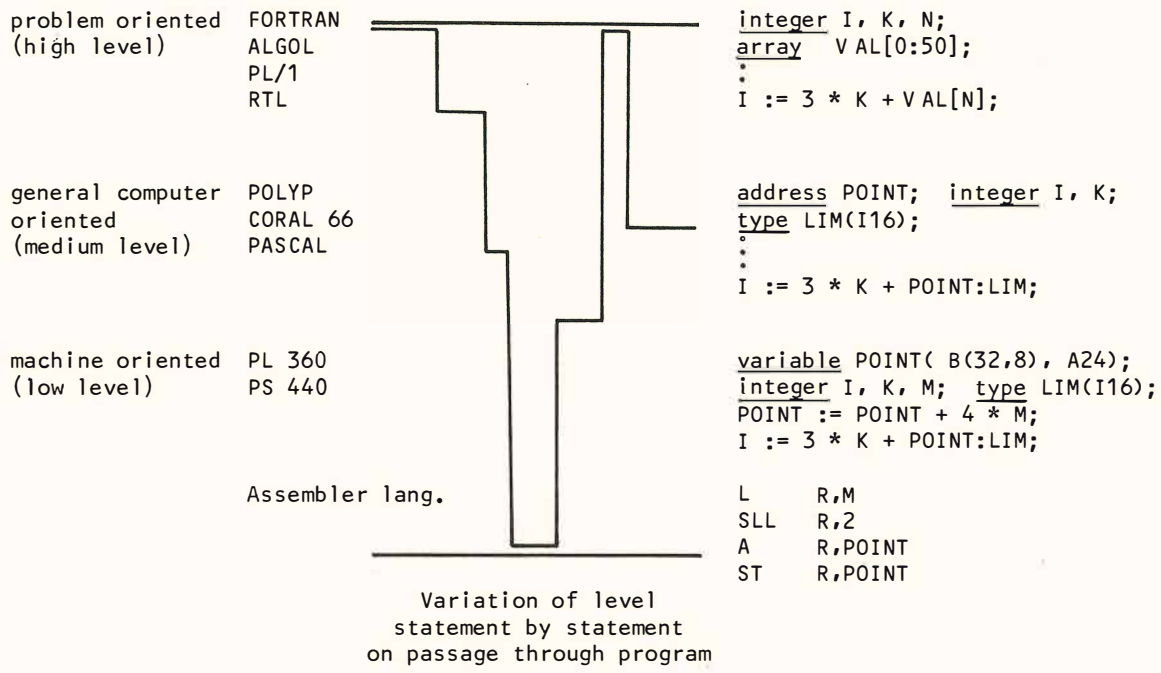

Fig. 1 Language levels

is used to modify an address constant. From this it can be seen that one must speak not only of the level of language but also of the level of a program. If address arithmetic is available in a language it does not mean in general that the programmer is forced to use it in his programs.

The EDP elements of a program can be divided into three subclasses:

1. Elements which refer to the general structure of computers (e.g. data overlay).
2. Elements which refer to the properties of a specific computer but which only influence the efficiency of the program (e.g. use of pointers as operand).
3. Elements which refer to the properties of a specific computer and which lead to errors (among other things) if the same source text is used when the program is transferred to another computer.

Programs which only contain applications elements and EDP elements of the first class are called machine-independent, those which in addition contain EDP elements of the second class are called conditionally machine-dependent and those containing EDP elements of the third class are called machine-dependent.

This classification proves useful when the languages themselves are analysed. The positioning of information in POLYP is taken as an example. The declaration:

```
variable AV(B(32,8),A24);
```

defines a variable with the name AV of type address and of length 24 bits (A24). The specification B(32,8) states that the address variable should have a bit address which is modulus 32 plus 8. For a word machine (smallest addressable storage unit is a word) with a 32-bit word the declaration above means that the variable is to be positioned into the rightmost 24 bits of a word. The declaration is also interpretable for a 16- or 8-bit machine.

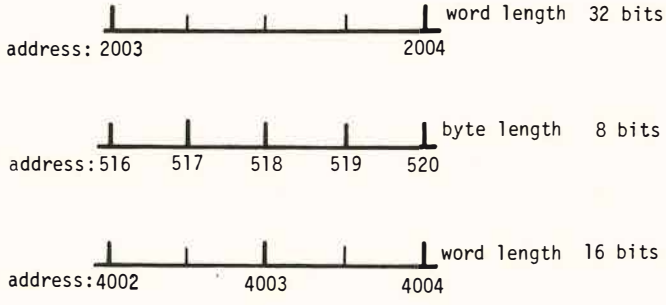

Fig. 2 Positioning

The language elements used in the declaration above are machine-independent since the syntactic and semantic definition makes no reference to a particular computer. Their occurrence in program elements leads to conditionally machine-dependent or even to machine-independent programs. The notion machine-independence is connected with the notion transferability of programs. Languages like FORTRAN, ALGOL 60 and COBOL were developed in order to allow for the exchange of programs regardless of the

computers being used. A program was transferable if it produced identical results when translated and run on two different computers. This condition is in general not sufficient for real-time applications. In addition, timing conditions (reaction time) must be fulfilled. This is, however, difficult to attain.

Another difficulty leads to similar wishes. Small and medium-sized real-time computers often have peripherals whose performance is too low and software which is not extensive enough for the development of medium and large program systems. For this reason many users are in favour of transferring the development work onto a large computer. This should be possible without having to use a simulator. It is mostly sufficient if the logic of the program modules can be tested on the large computer. Furthermore, it is not only possible to produce translators for such computers more cheaply, but they are also capable of accepting the full language.

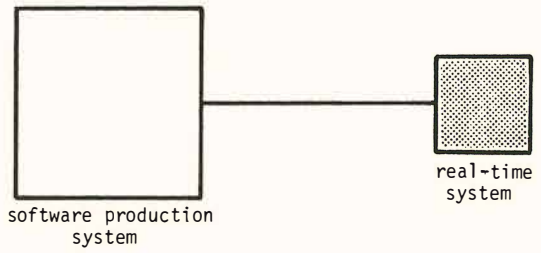

Fig. 3

If an object code of high quality is required it is necessary to make use of additional aids such as general macro processors.

3. Compilers

For the following considerations it is necessary to say something about the structure and the development of compilers. It is the job of a compiler to analyse a source text and translate it into a form which is directly or indirectly executable by the hardware. It is useful to differentiate between those problems which are associated with 'compile time' and those which are associated with 'run time'.

A compiler must be integrated into an operating system which, for example, takes over the management of data files. It is important that the compiler must be able to differentiate between various kinds of language elements (as opposed to the programmer). For example, the compiler must normally be able to distinguish between calls to user procedures, calls to standard procedures and calls to I/O procedures. One reason for this is that often different instruction sequences must be generated in each of the three cases. In order to make the distinction possible, the compiler must have a list of the names of all standard and I/O procedures (e.g. in its identifier list).

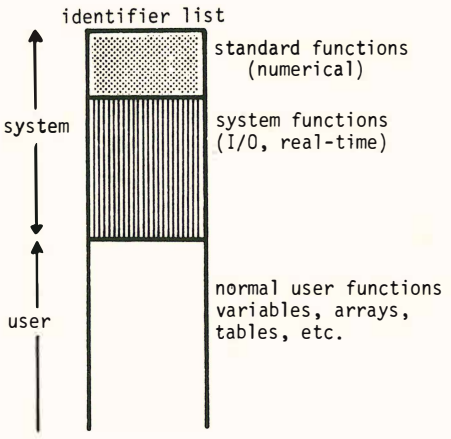

Fig. 4 Functions

The program which is generated by the compiler must normally run under the control of an operating system. This run-time operating system need not necessarily be identical to the compile time operating system. An existing operating system is usually used, whereby interface routines and extensions are necessary. This often requires an effort which is half the total effort of producing the compiler. Also, adaptation causes a reduction in the efficiency of the system. This problem occurs especially with real-time systems where the advantages of a universal system are discarded in favour of the higher efficiency of numerous small systems, each of which is tailored to the demands of a given application. Thus the necessity arises of adapting a language and its compiler to a given operating system. This should be possible without having to modify the compiler. It is possible to meet this demand if a distinction can be made at the definition level between the classes of procedure mentioned above [6]. In the compiler only the call interface need be defined. For each procedure class a different instruction sequence for the call is generated. The number and nature of the individual standard and I/O (or system) procedures need not be known to the compiler.

From the programmer's point of view, the main nucleus together with the procedures must represent a unit. By this method a system independence can be achieved which, at least for the next few years, exists in no process control programming language, compiler or operating system.

4. The reasons for bad object programs

It is often thought and claimed that the reason for

the low quality of object programs which have been generated from source programs written in high-level languages lies in the fact that the optimisation methods used in the compilers are not good enough. Much could certainly be done to improve these methods but optimising alone cannot solve all of the problems being discussed. For example, the problem of choosing between an optimisation which produces a fast object program and one which produces a short object program cannot be solved satisfactorily since the compiler sees the source program as a static unit and has little or no infromation about its dynamic nature. In the following considerations it will be assumed that the compilers considered all perform a sufficient degree of optimisation. Three typical properties of program elements have been chosen which exist in languages such as FORTRAN.

In the first example the definition of information (data) is analysed. Information types such as integer, real, boolean and string are usually available. The information length is normally predefined although alternatives such as double length are often allowed. In considering the information type boolean, which is most interesting in real-time applications, the question of the representation of a boolean value immediately arises. Should it be represented by a bit, a byte or maybe a short word? The information can be stored so as to optimise access time or so as to optimise storage requirement. The source text of the program must be analysed in order to determine which optimisation method should be used. Although it is relatively easy to calculate the storage requirement of a program, it is usually impossible to discover much about the effect of an access time optimisation, since the dynamic properties of the program cannot in general be determined. As already mentioned, it is also necessary, along with the type and length, to specify the positioning of an item of information. This is especially of interest in the case of tables, where the elements possess various types and lengths.

The second example is the sequential processing of information. Program cycles or loops (e.g. for statement), in which rows or columns of arrays or tables are processed, are mostly used for this purpose. The well known methods of optimising the organisation of the loop do not give the solution to the problem. More critical is the access to the information which is processed within the loop. The use of index registers relies on the fact that the loop controlled variable is incremented/decremented by a fixed constant value for each circuit of the loop, one condition for which is that its value is neither directly nor indirectly (side effects of procedures!) altered within the loop. If a static analysis of the source program shows even a remote possibility of this occurring, index registers cannot be used. The sequential processing of information outside loops is normally left unexamined. The problem can be solved in critical cases only by using address variables for which an address arithmetic is provided. An improvement can be achieved in POLYP [6], however, by using the vectorial statements which are an extension of the PL/1 array cross-section principle.

The last example is the call of a procedure with the transfer of parameters. Procedures are one of the most important aids in producing program systems and at the same time one of the most common causes of low quality in object programs. The reason can be found in the universality of the procedure concept. The parameter list in the object program consists, with hardly an exception, of a vector of pointers. The possibility of transferring values in registers or in special lists does not exist. The volume of instructions required for a procedure call and in the prolog and epilog of the procedure is out of proportion for short procedures.

The reason is that the construction and management of the parameter lists must be partly generated by the compiler and partly undertaken by run-time routines. It is completely sufficient for critical applications if an address variable as parameter is allowed. Thus the organisation and management of parameter lists becomes flexible and can be programmed to suit the situation. For example, the same parameter list can be used for different procedures, or the parameter list can consist of a mixture of pointers and values.

5 Conclusion

Considering the demand for language elements for the positioning of information, address variables, address arithmetic and address variables as parameters in procedure calls, it can be seen that the price to be paid for an improvement in the quality of object programs is the necessity for a better knowledge of the hardware properties of the computer(s) being programmed. It is not, however, required that the syntax and semantics of the assembler languages in question be known.

The problems mentioned are often solved by inserting assembler language text into the high-level source. This, however, does nothing to improve the legibility of the source program. The solution adapted by languages such as BLISS, PASCAL and POLYP is to provide language elements like those discussed above which can be used to improve the declarations and in some cases which can be used at critical positions in the dynamic parts of the program. The use of a second language is thus not necessary (Fig. 1). A more exact examination shows that a relatively small number of fixed or special positionings of information are required for a given computer or project. The generality of, and thus the amount to be written for, a given declaration can be

disturbing. This difficulty can be removed by introducing a general control (or macro) language. With the aid of a suitable library the required 'modifications' can be made to the source text.

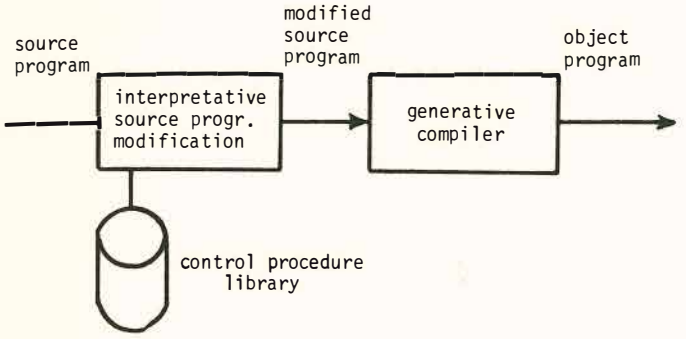

Fig. 5 General macro processor

This method can be used to great advantage, together with the system procedures mentioned in Section 3, to extend the language, without changing the compiler or operating system, with real-time oriented function calls. It is also possible, after the design phase of a project, to construct a control procedure library (Fig. 7) so that during the programming phase programs can be written without special knowledge of the computer and which can be translated into efficient object programs.

The methods described here also allow the development of real-time program systems using larger computers (software production system). It is only necessary that the required control procedure libraries be available.

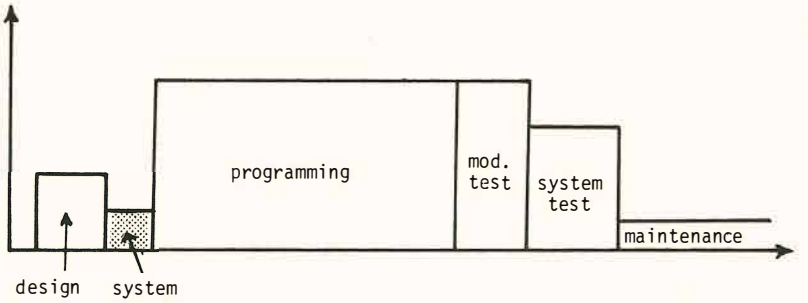

Fig. 6 Project phases

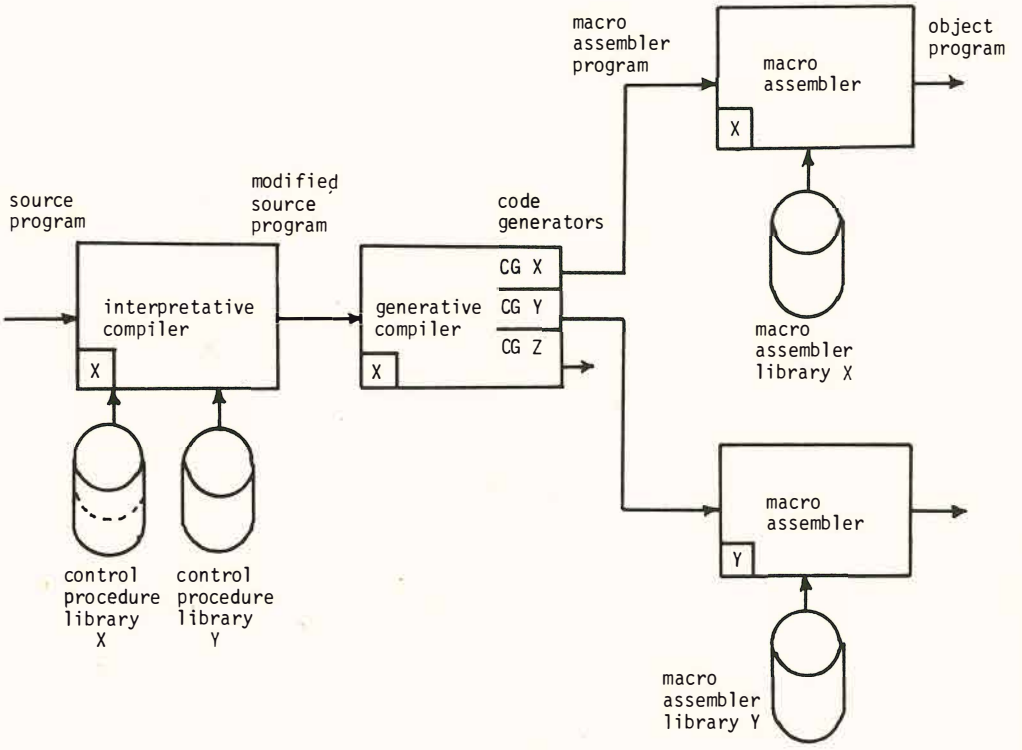

Fig. 7 Full compiler

1. WIRTH, N., 'PL 360, a programming language for the 360 computers', Journ. ACM, V15, pp. 37-74 (1968).
2. GOOS, G., LAGALLY, K., and SAPPER, G., 'PS440 – Eine niedere Programmiersprache', Bericht 7002 RZ der TH München.
3. Inter-establishment Committee for Computer Applications, 'Official definition of CORAL 66' (1970).
4. WULF, W.A., RUSSELL, D.B., and HABERMANN, A.N., 'BLISS: a language for systems programming', Comm. of the ACM, V14(12), pp. 780-790 (December 1971).
5. WIRTH, N., 'The programming language Pascal', Acta Informatica, V1, pp. 35-63 (1971).
6. MUSSTOPF, G., 'Definition der Programmiersprache POLYP' (noch unveröffentlicht).
7. MUSSTOPF, G., 'Sprachgesteuerte Modifikationen von Quell-programmen', GI-Tagung (März 1972).

Discussion

Q. In Fig. 1, what does the horizontal axis denote?

A. The stage during execution of a program – different statements may be at different levels.

Q. In Fig. 6 (project phases), how significant are the relative sizes?

A. It depends on the problem.

Q. Is a description of POLYP available?

A. Only in German (see reference [6]).