Interaction between user's needs and language–compiler–computer systems

By A. S. Cormack*

The intention of this paper is to explain some of the difficulties which surround the implementation and design of commercial languages for computers. It is aimed at helping the user or potential user to decide what system is best suited to his purpose. An abbreviated version of this paper was presented at the Joint Computer Conference held in Edinburgh in April 1964.

This paper admits the existence of commercial languages and advances no particular arguments to support or oppose them. In the opinion of the author, computer languages have been the subject of sufficient publicity, comment and criticism, over the past few years to enable any interested party to form his own opinion on their general merits.

The small minority of confirmed agnostics, who derive aesthetic pleasure from bit manipulation and intricate order modification, are unlikely to be seduced by any system less elegant than their own; but the hardened core of realists who have suffered the tedious process of preparing and testing large commercial programs are now convinced of the value of machine-independent languages as programming tools, and are merely concerned about the effectiveness of the ones currently available.

To these two groups may be added a third, consisting of all those on the fringe of the computer world who have not yet used a computer but have decided, for a variety of reasons, to purchase one and are shopping around for the one most likely to suit their purpose.

The object of this paper is to examine some of the problems that arise, both in the design of languages which are independent of the machines with which they are intended to communicate, and in writing compilers to translate these languages for any particular machine.

Three classes of languages are considered and the interrelation of each language with a range of computers is examined. It is hoped that by taking these imaginary language-compiler-computer systems and investigating the advantages and disadvantages of each, a clearer picture may emerge which will help the user or potential user to assess more accurately his own needs.

Language concepts

It cannot be too strongly emphasized that any procedure-oriented language, whatever its class or complexity, must have as its foundation a set of clearly defined concepts, all of which are related to each other in such a way that the final structure is stable, although not necessarily inflexible. Any inconsistencies will, of necessity, inhibit the growth of such a language since they may produce disastrous side effects when the language is extended.

It is only too easy to seize upon a feature that appears to be both elegant and powerful, as indeed it may be in its proper context, and try to work it into a system which is not designed to accept it. One of the misfortunes of the language designer is the necessary shelving of good ideas which do not fit the system, and one of the disadvantages of some of the languages which have been designed lies in the fact that the originator was not sufficiently ruthless in this direction.

It is perhaps advisable at this stage to clarify one of the misconceptions that has developed with programming languages concerning their ease of learning and use. No criticism is directed intentionally, implicitly or explicitly, towards any language on the grounds of complexity alone. Clearly, a language is merely a sophisticated tool and, as such, depends to a large extent upon the intelligence of the user for maximum effect. The more a language does for you the harder it is to learn, but the greater the effect it has, both in speed of writing and conciseness. Only where the complexity has an adverse effect on the system as a whole, both from the implementation angle and the efficiency of the final object program, can it be justifiably criticized.

Language-compiler-computer systems

The emphasis throughout the paper is placed upon the whole computer system, and it must be clearly understood that the evaluation of commercial programming languages on their own is an extremely difficult task. Misleading conclusions may be drawn if the relationship between one such language and its implementation on a particular computer is not also taken into account.

As software support improves so does the compiler become more and more just a part of the total operating system. A rather loose analogy may be drawn between a computer system and a modern car, where the engine represents the compiler and the transmission represents the input-output control system handling the peripherals. The major effort in the development of the modern car, from the functional standpoint, is directed towards increasing the power/weight ratio of the engines, and distributing the load evenly across the various working parts to minimize mechanical failure. In a similar way the computer system, instead of exhibiting the uneasy relationship which used to exist between the

* *National Cash Register Co. Ltd., St. Alphage House, Fore St., London, E.C.2.*

compiler and the rather primitive peripheral routines then available, is, or should be, a balanced whole with the compiler providing the driving power under the control of the operating system, and the input-output system providing the transmission to the peripherals.

It is precisely this balance which is so difficult to achieve, and it is here that the analogy starts to break down. The car manufacturer is not in the least concerned with producing a universal engine which will fit the whole range of his models, but the computer manufacturer ideally would like the input to his compiler to be restricted to one language. Unfortunately, this is not currently possible, and the reason it is not possible is because of the continual, and as yet unresolved, struggle to define a language which is sufficiently powerful for effective use on large computer systems and, at the same time, can be efficiently implemented on smaller systems. The addition of options and features which the language does not need but efficiency demands, has made subsequent efforts at extracting effective subsets for smaller systems extremely difficult.

As a result, a need has arisen for a lower-level language which, while maintaining the desirable feature of machine independence, can be readily implemented on smaller computers. Provided that sufficient care is taken in the design of such a language, it has the additional merit that it can be extended to take advantage of the more powerful features of the larger systems.

Language X

Most of the major commercial languages that have been developed over the past few years have the same general characteristics and structure, and the first language to be considered belongs in this category.

Data are considered to be unrelated to the procedures that act upon them, and are therefore defined separately. Within the data division, the concept of levels is introduced so that a field within a group within a record is recognized in a single statement.

Within the procedure division, a full range of commands is defined which include complex compound conditions, Boolean and logical operators, asynchronous processing and recursive procedures, powerful arithmetic facilities, and a large number of options to allow different machines to take full advantage of individual hardware features. Segmentation is specified implicitly by the source programmer, and storage allocation is handled dynamically by the system supervisor.

Potentially, then, this is the most powerful language of the three which are discussed in this paper, by virtue of its size and scope. The language is assumed, hopefully, to be both well defined and free from unnecessary restrictions, and the only major criticism to which it is open concerns the arbitrary separation of the procedures from the data upon which they act.

This imposes an unnecessary rigidity on the language since it ignores the inter-relation between procedures and data. If the division between the two were removed, a number of the unavoidable static parameters could be replaced by dynamic ones with a consequent gain in flexibility.

Now let us consider the implementation of Language X on a large, fast computer.

Because of the computer's size the compiler may be assumed to fit fairly comfortably without undue restraint, and because of its speed the time taken to translate from source program to final object program may be assumed to lie within acceptable limits. In addition to the problem of producing an efficient translation pass it is the aim of every compiler writer to generate optimum coding in the run-time program, and it is in this area that the generality of the language causes the greatest difficulty.

An unrestricted system of dynamic segmentation may be extremely helpful to the user but it creates an intolerable situation when the program is being obeyed. It presupposes relocatability of all segments without any indication to the compiler of segment priority.

The supervisory system has to create a table carrying all the information about the various segments into which the program has been divided. This table must be made available to the executive loader which activates the segments as and when they are required, and the loader must, in turn, arrange to shuffle the segments as they come in and out of use in order to make the best possible use of the storage space available. In the process of doing this it has to alter the references to all segments which have changed position before it can hand back control to the main program.

Clearly, if the segments are fairly small and of high activity, the time taken changing segments will be a large proportion of the actual running time. Again, if recursive procedures are allowed and do not have to be defined as such, one of the following two situations will arise.

Firstly, the compiler must allow for the possibility that every procedure may be called recursively, and generate coding accordingly. This is the more naïve approach since it is expensive both in time and space for the object program. Secondly, there must be a mechanism built into the compiler which examines each procedure and generates the coding for a recursive procedure only when it is necessary.

The latter course, even though it will add to the compiling time, is the more acceptable, but even this method has its limitations. It is possible, for instance, to imagine a case where the programmer may know that a particular procedure will never be called recursively, because he has access to the logic of the program. The compiler, on the other hand, has no such special knowledge and therefore must cater for the most general case.

The concept of levels of data was introduced because it was considered to reflect more closely the current procedures of manual filing which computer systems are replacing, and thus it was considered easier for the apprentice programmer to understand. In no way does

it improve run-time efficiency, and indeed it creates particular problems of its own.

Great care must be taken by the compiler writer to ensure that any reference to an element at a low level, which is defined explicitly in the source program, gives rise to the generation of a single direct access in machine code. It is only too easy, under the continual pressure of effort to meet unrealistic deadlines, to access each level indirectly using the simple cascading link technique. Since inner loops and arithmetic expressions usually refer to elements at the lowest level it is of great importance that these references generate minimum coding.

It is also important for the programmer to remember that not all procedure commands may refer to all levels of data, and it is necessary to keep clearly in mind the distinction between those procedures that are permissible only with primitive elements of data and those that may refer to higher levels.

The preceding remarks describe very briefly the major problem areas in the implementation of Language X, and clearly they all apply to an even greater extent when related to medium-sized computer systems.

If the speed and capacity of the memory are sufficiently great, the loss of efficiency at run-time may, to a large extent, be disguised; but the less peripheral-bound the system becomes, the more apparent becomes the discrepancy between generated coding and optimum hand-coding.

In addition, with medium-sized systems, considerable restraints may be put upon the compiler, forcing the designer to increase the number of passes to such an extent that the compiling time becomes an important criterion in the assessment of the complete system. The argument that short cuts which streamline the compiler are justified in this case has little appeal if the efficiency of the run-time program is materially affected.

A large number of commercial data-processing jobs are relatively simple and may be processed using standard techniques for updating, analysis, sorting, etc. A rough estimate of 20% to 30% of the available facilities of Language X are needed for such jobs, but the mechanisms for handling all the available facilities are of course present, and one of the penalties that must be paid for the considerable generality of Language X is that unused mechanisms take up space.

In addition to the space taken by these mechanisms a number of them are activated automatically during every compilation. Even though the response in most cases may be of the form "no action" and the time taken for a specific case is trivial, the cumulative effect of activities which occur for every primitive of the language at every occurrence during the source program may be considerable.

On the medium-sized computer, therefore, priorities such as speed of execution and length of compilation assume a greater significance, and the potential user would be well advised to consider carefully whether his requirements justify the use of Language X or whether a simpler language would fulfil his purpose better.

If we now move on to the implementation of Language X on a small computer, it becomes immediately apparent that the compilation time has increased to such an extent that it lies well outside the acceptable limits, and that any attempt to use Language X on such a system would be totally impracticable.

Language Y

The underlying philosophy of Language Y is one of simplicity, clarity and flexibility. Attempts to introduce features which may, on the surface, be attractive to programmers, but can only have an adverse effect on either the compiler or the generated object program, have been resolutely resisted. What is offered is intended to be the minimum effective language for commercial data processing.

Concessions have been made in cases where it has been considered justifiable in the interests of easier programming so that, although basically data are treated as elementary items throughout, the concept of a record exists to permit the easy interchange of data between backing store and main memory. In general, however, the wealth of possible expressions, such as compound conditional phrases, that could have been permitted, has been deliberately reduced in order to make learning and accurate use easier.

In contrast to Language X, Language Y accepts the relationship between data and procedures, and no separate description of data is required of the user. Items may be defined during the procedures which use them, and provision is also made for the definition of contiguous strings of information so that list handling and record filing may be handled quickly and efficiently.

Since computer systems exist where the relationship of the logical record to the physical record must be considered in order to generate efficient object coding for filing on a backing store, only one level of data is permitted. Clearly, a considerable amount of special knowledge of how the compiler handles files, which contain records of complex structure, is necessary to program such computer systems efficiently. A very limited amount of such special knowledge is necessary where only one level is allowed.

The procedure division contains a logically complete set of imperative statements, simple conditions, Boolean expressions and the usual arithmetic operations such as add, subtract, multiply and divide. More complex arithmetic is handled by means of a set of calculate functions which may be extended to include those operations that are specific to particular fields of business. Input/output functions are provided to handle paper tape, magnetic tape, punches, line printers, card readers, etc. Segmentation is permitted but must be specified by the user.

Language Y, then, lies much closer to the machine than Language X, and enables the user to exercise tighter control over the operation of the program and the positioning of the data.

The greater flexibility of Language Y is achieved at the expense of slightly greater effort on the part of the programmer. The emphasis has shifted to a certain degree from the compiler to the user so that he has slightly more work to do, in the sense that he has to build his system from "bricks" rather than "prefabricated" slabs.

The implementation of Language Y on any of the range of computers mentioned presents no particular problems, although the minimum system must of necessity have some form of backing store.

Since the installation of a computer system represents a sizeable capital investment, potential users usually have a well defined major role for the computer to play before considering a purchase. Once the purchase has been completed and the system is operating satisfactorily they are reluctant to experiment further because of the considerable programming effort that is required. This is perhaps a pity, because the potential of the computer extends far beyond the normal day-to-day processing for which it may have originally been bought.

The structure of Language Y is such that it is comparatively simple to extract the basic features of the language without regard to the more sophisticated facilities. It is to be hoped that the considerable programming effort saved by the use of this fundamental set will encourage users of the future to be more adventurous in the field of experimental investigation than has hitherto been feasible.

Language Z

The third language to be considered possesses totally different characteristics in that it is a problem-oriented and not a procedure-oriented language. In other words, it has been designed specifically to cater for one particular problem or, more generally, one particular class of problem.

A survey covering a wide spectrum of jobs passing through a number of service bureaux yielded a large number of functions which were common to most commercial data-processing problems. These functions were then analyzed carefully, extended to full generality, and incorporated in a function library.

Activation of these functions, including the automatic linking of one function with another, is achieved by the input of a string of parameters to the generator. All the parameters necessary for each function are carefully defined and specified on pre-printed parameter sheets which represent the only form of input necessary to the generator.

The criticism normally levelled at packaged programs is that they are either not general enough to cover all cases, or that they are too general, and therefore too wasteful, to cater for a specific case. This does not apply with Language Z since the technique of selective generation, under control of the parameters, is used, and the functions may therefore be as general as is necessary.

One of the attractive features of this language is the rapidity with which it is possible to transform the overall system specification into a running program. All that is necessary is the preparation of a flow chart indicating the functions that are to be used and the order in which they are to be linked, the completion of the parameter sheets relating to these functions, and the transfer of the information from the parameter sheets to punched cards or paper tape for input to the generator.

The definition of the data which are to be used by each function is included on the parameter sheet for that function and, although it is advisable to keep a check of the data on date record layout forms, this is purely for the sake of documentation and does not form part of the input.

Another feature which appeals is the logical soundness of the language structure. It is apparent from the specification that the basic concepts have been rigidly adhered to throughout and that no attempt has been made to go over the prescribed limits set by the language definition. This consistency is by no means evident in a number of other languages which have nevertheless gained considerable support.

The language was designed to handle at least 50% of all commercial data-processing problems, and although practice indicates that the figure may be nearer 80%, it must be realized that a language of this sort cannot be expected to satisfy the needs of a software support system on its own.

Within the defined scope it is extremely effective, but the rigidity of the structure causes a very rapid falling off of efficiency if attempts are made to apply it in areas outside its range. The implementation of such a language on any of the range of computers considered in this paper presents no particular problems since the larger part of the generator consists of the function library which may be held on the backing store of the system.

In any system which is equipped with a reasonably sophisticated assembler, it is more likely to be the assembler which defines the minimum configuration rather than the Language Z generator. The only proviso that is added concerns the main memory size which must be sufficiently large to hold the main control program. This program inputs the parameters, accesses the functions and hands over control to the individual function generators which must in turn output the source lines to the assembler.

E.C.M.A.

Throughout this paper attention has been concentrated on explaining in general terms some of the difficulties that face compiler writers. To most experienced programmers and all software specialists these remarks may seem naïve and obvious, but to those users who are not specialists in this field and who merely want guidance in choosing their systems, it is hoped that this paper may be of some assistance. As an additional aid the user may like to refer to a paper originating from one of the technical committees of the European Computer Manufacturers Association. (ECMA/TC2/64/11.)

Briefly, the paper defines a matrix where the rows represent system features and the columns, users' criteria. Two appendices accompany the paper and each appendix gives a brief explanation on each feature or criterion, respectively. The matrix is a ternary-value matrix where each element is either positive (+), negative (—) or zero (.), depending on whether the possession of a particular feature has an adverse, beneficial or null affect on the corresponding criterion.

Attempts to attach quantitative values to the individual elements of the matrix were abandoned as being too controversial, but the user is encouraged to attempt his own quantitative assessment if he so desires.

Conclusion

Those most immediately concerned with the development of computer languages are, of course, the major commercial users like the government who have a large number of programs distributed across a wide variety of machines and are faced with the problem of transferring these programs from one machine to another, but there are a large number of other users who have been content to stand on the side lines and watch the progress of these languages while taking little active part in their development.

This is a perfectly understandable attitude to take while the situation remains comparatively fluid, but there comes a stage, which has now been reached, when most of the major ground-work has been completed and further development can only come from widespread field trials. It would be encouraging if, in the future, users were prepared to lend support to these techniques which can only be to their eventual benefit.

Reference

Requirements for Programming Languages (1964) ECMA/TC2/64/11. (Obtainable from E.C.M.A., Rue d'Italie 11, Geneva, Switzerland.)

Book Reviews

Progress in Operations Research, Vol. II, edited by DAVID B. HERTZ and ROGER T. EDDISON, 1964; 455 pages. (London and New York: *John Wiley and Sons*, 84s.)

One would not expect this to be a treatise on computers, and yet there are so many points of contact between the worlds of Operational Research and computers that it is surprising to find only infrequent examples referred to. For a student of computer applications, however, these accounts of O.R. in all the major areas of business management and industry have a great deal to offer, not so much because O.R. is itself demanding more and more from computers, but because their use in business can be relatively ineffectual without the O.R. approach to management problems.

Not unexpectedly, simulation by computer appears in several contexts, and is clearly responsible for much of the contact between O.R. practitioners and computers. Unlike the first volume in this series, however, this is not a book on techniques, and the substantial variations which exist among simulation techniques and procedures are not discussed here.

The editors, prominent in O.R. on either side of the Atlantic, present in this volume contributions from the United States, United Kingdom, France and Canada. The early chapters deal with several classes of problem which arise in most organizations, while the remainder of the survey is of applications of O.R. in Government and in several industrial groupings. Inevitably, some authors have found it difficult to produce an international balance in the content of their papers, but there are extensive bibliographies.

Those not already familiar with Operational Research in this country should not assume too readily that the counterpart of all the American work that is mentioned can be found here in Britain, nor *vice versa*. In particular, it is distressing to find in one contribution from the U.S. a tribute to managements who use inventory simulations to "make their inventories a truly effective competitive weapon against labor, vendors, and other industries." Fortunately, evidence abounds elsewhere that O.R. is playing a more responsible role than that.

D. G. OWEN

Perspectives in Programming, edited by R. T. FILEP, 1963; 324 pages. (London: *Collier-Macmillan Ltd.*, 45s.)

This book is a collection of papers presented at seminars in American Universities during 1962. Despite this, the material has been collected together in a commendable way by Mr. Filep so that it falls naturally into sections, and consequently it is possible to read the text more as a book than as isolated descriptions of various aspects.

The first reading of the book left me with a considerable appreciation of the enormous amounts of money and resources which must be ploughed back into education both in the technologically advanced and in the emerging nations (an excellent contribution by Komoski). Any educationally viable system which can increase productivity in education must be welcomed; and it is hard to understand much of the opposition in this country to programmed learning in general, and teaching machines in particular, which appears to be based on very flimsy knowledge. Whilst there is no panacea for this demand for education, anything which can contribute towards its satisfaction is worthy of consideration, and this book can be used to introduce many people to this type of instruction.

The book starts with four papers setting out the basic facts and arguments concerning programmed instruction—and anyone who has never even heard of Pressey or Skinner can start quite happily at the beginning of the book. The

(*Continued on page 20*)