A CRITICAL REVIEW OF THE STATE OF THE PROGRAMMING ART

R. S. Barton
Altadena, California

... When once the engine shall have been constructed, the difficulty shall have been reduced to the making out of cards; but as these are merely the translation of algebraical formulae, it will, by means of some simple notations, be easy to consign the execution of them to a workman. Thus, the whole intellectual labour will be limited to the preparation of the formulae, which must be adapted for calculation by the engine.

L. F. Menabrea, 1842, from Sketch of the Analytical Engine Invented by Charles Babbage

INTRODUCTION

What follows is intended not as a scholarly review of the programming art, but as a personal appraisal. The bias is that of one interested in the subject of machine organization and its relation to programming.

Surveying the still small body of computer literature, one is struck by the scarcity of substantial material on programming as such, and might wonder if the subject of programming is inherently trivial, or whether its study is being neglected as a separate discipline.

The writer thinks that while programming has developed as a practical art rather than as a science, present trends suggest that a formal theory of programming will emerge and become an important branch of mathematics. It is not clear why those with formal training in mathematics and logic were not attracted in larger numbers to the subject in the preceding two decades; and it is now difficult to predict how quickly a formal theory of programming will affect existing technology in the computer sciences.

THE PRACTICAL ASPECTS OF PROGRAMMING

Superficially, programming is an easy skill to acquire. A few hours of instruction on a particular machine or language is enough for a start. Aptitude varies and successful programmers, in the past, acquired much lore which became part of their stock in trade. Today, however, many of the machine peculiarities and operational technicalities are handled in the workings of system programs so that the practical difficulties of programming have been alleviated for the average machine user. The professional programmer designs and constructs these programming systems.

The practical aspects of programming today consist of (1) the design of suitable languages to express problems of interest, (2) the instruction of people in the use of these languages, (3) the maintenance of a library of programs to economize on effort, (4) education of users in techniques of efficient program organization, and (5) the designing of the
translators and systems for the machines. All other considerations are extraneous.

Language Design

A programming language must meet, ideally, these criteria: (1) it must enable precise description of any representative of the class of problems for which it was intended; (2) it must be mechanically translatable to a common standard; (3) it should be concise; and (4) it should take into account the preferences of the average user.

The user's preferences are influenced by his training and experience. Within a field of application, programmers are likely to have similar educational backgrounds, and hence, familiarity with certain notations and language forms. Then, too, the nature of the human mind, eye, and nervous systems may predispose people towards certain notational forms.

Matching the human language to the machine language is the task of the system program. Compromises tending to favor machine requirements have been often detrimental, and, as an intermediate measure, hand transcription—as a separate clerical task—might be preferable to the use of a restrictive notation in the case of certain problem-oriented languages.

Teaching the Language

A language is characterized by its vocabulary and a grammar, and recently, beginning with the description of ALGOL in Backus notation, precise and concise descriptions of syntax have appeared. The value of such a precise description is not yet fully appreciated as an aid to learning and subsequent use.

Instructional texts have been prepared both in linear and branching programmed text form. Practice in the writing of programs can be made more convenient in some cases through use of an automated programming laboratory.

These developments in methods of precise language description, techniques of presentation, and economical methods for providing machine practice are all potentially very important in teaching fluent and widespread use of programming language. It is desirable to teach both reading and writing of programs rather than, as in the past, only program writing. As an aid to encouraging correct grammar, the existence of a precise syntax for a language permits mechanical checking of correct usage as an additional feature of automated instruction.

It would now seem both feasible and desirable to delegate all programming language instruction to a computer program provided with each computer.

The Library Problem

The importance of a library to provide a repository for programs was recognized early, but full exploitation has been impeded by poor program documentation, lack of interest on the part of programmers, and language problems. Many program libraries now consist of programs in languages either dead or destined for an early demise. Limitations result from the dearth of program "readers" and the serious practical difficulties in translation between machine languages.

Libraries will, of course, be stratified, ranging from the personal through the semi-private to the public; and classification and abstracting systems become increasingly important as programs enter the public domain. Part of the education of programmers must be directed to the construction of the most general routines likely to be of use, since such routines rather than specialized ones, are most likely to be useful items in a library.

A standard language is needed to ensure permanence and maximum utility for programs in the public domain. This need not be directly readable, and hence, can be a universal machine language.

Automatic documentation techniques of flow diagram synthesis and language expansion are relevant to library maintenance and convenience of library usage. Library space is less of a problem than in libraries containing books. Programs can be in hierarchical form, it no longer being necessary to accept the constraints of serially accessible storage to allow efficient assembly.

It is important to note that in addition to explicitly referencing the library in the program, an ever-increasing amount of implicit referencing will result from the use of problem-oriented (as distinguished from procedural) languages. Thus, the geometric languages cause the selection of library programs
to carry out geometric solutions implied rather than stated procedurally. Use of the pertinent part of the classification system is built into the translator.

Teaching Programmers the Techniques of Program Organization

Program organization is, in some respects, the heart of programming, and should be independent of the processor characteristics; though, in the past, these have intruded prematurely in most analyses. Perlis has summed up the matter in seven words: definition, sequencing, selection, substitution, binding, replication, and evaluation. It is unfortunate that, as yet, there is no sufficiently general standard notation to represent these programming concepts. Fragments appear in various languages; examples are the block, conditional, and for statement of ALGOL. The flow diagram notation is the most common method of representing structure, but is a cumbersome device for use with complex programs unless a hierarchy of diagrams is used.

Iverson, in his recent book, “A Programming Language,” defines general operations on structured operands such as vectors, matrices, and trees at one extreme and descends to a level of detail pertinent to the logic designer at the other.

The writer believes that the representation of structure is the most important aspect of programming for purposes of formalization. With this accomplished, suitable basic education would be feasible in the schools, and a mathematics of program transformations might follow. An eventual consequence would be the mechanical resolution of the problem of determining the best arrangement of a program subject to given constraints.

Designing Translators and Systems for the Machines

The systems programmer now has the responsibility of satisfying the user-programmer and making-do with a machine that is already in existence, or, equivalently, completely specified. The second requirement which now presents serious difficulties need not be a permanent obstacle, since the system programmer will enjoy increased participation in the design of future machines. Of course, this poses the problem of educating system programmers in the subject of computer organization.

A machine language need not be suited for direct use by human beings. This is convenient only if one allows for a translation in both directions between person and machine. If the fact is accepted that a machine language need never be seen or directly manipulated by programmers, some artificial constraints in the language design are removed, and its representation in the machine insofar as both format and codes are concerned is a problem that need concern only the logic designer.

For the immediate future machine language will be of procedural type, whereas the convenience of nonprocedural languages for the programmer is widely recognized. The syntax of the machine language seems likely to be simpler than that of the usual programming language. Most important, the essential part of the machine language will be descriptive of program and data structure and the choice of evaluative operations is of secondary importance, since in principle as few as one logical operation is sufficient to synthesize higher level operations, provided the structure of the programs can be represented. Putting this last point in a somewhat different way: The evaluation-type operations required can always be effected by subroutines, and these can be made as efficient as desired by use of high-speed storage, concurrent operation, and compact coding.

HISTORY AND THE PERIOD OF COMMERCIALISM

The Pioneers

The recently published book, “Charles Babage and his Calculating Engines,” makes fascinating reading and helps to provide some perspective. The book contains notes by Ada Augusta, Countess of Lovelace, who could be justly described as the first programmer.

After the work of Aiken and Stibitz independently led to the first automatic computers in the early 1940's, the first programs comparable to those conceived for the Babbage Analytical Engine were prepared and executed, though prior to this, astronomers had used punched card machines in a manner analogous

---


2 Dover, 1961.
to Babbage's proposed Difference Engine. It is interesting to note that Aiken, realizing that the preparation of computer codes was a demanding and fatiguing task, designed a coding machine with keyboard labelled with appropriately arranged mathematical symbols. Aiken also foresaw the coming shortage of adequately trained personnel, developed some of the earliest academic courses, and taught many of today's leading theorists in the computer field.

In the mid-forties, when the stored-program computer had been conceived, Goldstine and von Neumann prepared a report which was fairly widely distributed, but not formally published, entitled "Planning and Coding of Problems for an Electronic Computing Instrument." This report contains one of the earliest discussions of program structure and its representation by flow diagrams and is worth reading today. During this same period, Mauchly developed coding methods for the new stored-program machines.

The Commercial Era

The present period began when Univac, which was to be the first commercially available large computer, became, as a result of an airplane crash, a "business machine," and its inventors, Eckert and Mauchly, were no longer in control of the manner of its introduction to the market. The business machine industry, managed by executives of the pre-electronic era, absorbed the few scientist/engineer-run computer firms, and imposed on a new, struggling technology the ideas and practices of the punched card and office machine industry. The promotional and sales effort that followed has had, in little more than a decade, the effect of making computers a familiar part of our industrial, business, and government operations, but the competition has had some unhealthy side effects pertinent to the development of programming as a profession.

The intense competition of the early fifties quickly turned instead into a quasi-monopolistic situation dominated by IBM, with a semblance of competition maintained through fear of anti-trust action, some equity in distribution of government orders, and the rapid growth of the market. The programmer, during this period, was in the paradoxical position of being much in demand and the recipient of ever-larger paychecks and yet was often regarded as being of subprofessional status. This followed, in part, from the efforts of the marketing man, who, with his semi-technical sales-support and advertising staff, has promulgated so much nonsense about computers that the programmer, in the position of having to make good the claims, has been often only partially successful, and sometimes foredoomed to failure.

During this time, the universities were relatively inactive in computer design and programming theory. Since the manufacturers allocated funds mainly to the development of new components and hastily-conceived designs for the market, technical progress in machine organization and programming was erratic. The relation between the ever-growing programming problem and machine organization was overlooked. The programming problem, finally reaching massive proportions, was attacked via programming systems, also hastily designed.

The professional general-purpose applications programmer was soon obsoleted by an abundance of cheap machine time and the delegation of nuisance aspects of programming and machine operation to the inside of system programs. Somewhat amusingly, the machines became their own best customers with FORTRAN, a leading contender for machine time, not because of quality of compiled program, but rather because of its own lengthy compilation process.

There was a hasty exodus of the more skillful into the new field of system programming, a specialty in which it was quite possible to forget the purpose of computers in the face of challenging problems to solve. Emphasis on the old goal of efficient machine utilization diminished and attention was re-focused on systems. User organizations, dedicated to sharing and good fellowship, the salesmen selling "software," and the language standardization committees were typical of the new period.

The user organizations, pioneered to share developmental cost and programmer talent in support of new machines, soon degenerated into pressure groups intended to persuade the manufacturers to provide system programs and other support for their products.

The former self-reliance of many of the major computer users was replaced by dependence on the manufacturer, and many of the
most capable programmers from the user installations gravitated to the manufacturer's support activities.

One may wonder where the foolish term, "software," originated, but need not doubt that it was eagerly seized upon by the gimmick-hungry salesmen. There is something conveniently intangible about "software," and an agile, alert salesman can make promises as his needs dictate. Irresponsible marketing and unreasonable customers combined to furnish continual harassment of the programmers supposed to produce the systems, and who were already handicapped by poorly defined machines and languages and the lack of a sound technical basis for the work being attempted. Subject to whimsical sales pressures and working, as is often the case, directly under nontechnical marketing management, the results were seldom satisfactory, often near-fraudulent.

The language standardization issue, by committee or de facto, had three main episodes: FORTRAN, ALGOL, and COBOL.

FORTRAN was sufficiently useful to achieve widespread use and was known to be a program of great size and sparsely documented complexity. It was the first compiler to be widely advertised and aggressively sold to the market, and not surprisingly, a user-cult formed which is now quite effectively hampering progress in the adoption of improved scientific languages. For some time, superior methods for implementing a language of the comparative simplicity of FORTRAN have existed. Useful translators have demonstrated this fact. But still the usual FORTRAN compiler is little improved over the earlier systems and the few exceptions to the rule are not widely used. The user remains satisfied with a limited language and absurdly inefficient translation. The user excused indifference to improvement by pointing out the investment in program libraries, the difficulties of conversion, and the cost of retraining.

ALGOL was an attempt to make a step beyond FORTRAN and achieve, at the same time, a standard. Manufacturers' representatives, members of the academic computer world, and prominent computer theorists from abroad made contributions. Programmers in this country seemed uninterested in ALGOL, in contrast to the Europeans and Soviets, criticizing it in the superficial way that has its origins in ignorance. This opportunity to introduce some professional standards in place of crass commercialism was rejected because of such factors as mental laziness in the face of the Backus notation, awe of recursion, and the use of a larger character set than was then available for existing equipment.

COBOL was a rather different phenomenon. Bureaucratically inspired, industry politics ridden, and representative of an eccentric view of programming, it and its forerunners seem in retrospect to have been predicated on the notion that inability to learn concise notations could be circumvented by providing a kind of pidgin English. The possibility was overlooked of making programs intelligible to the nonspecialist by language expansion during translation. It is unfortunate that the real contribution of the business application programmers, the notion of data descriptions logically separate from the procedural description, was obscured by association with narrative-style language.

Machine Organization and the Programming Problem

It is true that the earliest machines were, from a programming standpoint, often logically more elegant than their successors. Anxiety about circuit reliability was a motivating factor behind their simplicity, but it should also be remembered that the logical organization was usually the concept of one, or at most a few, closely associated inventors. By the time that computers had become more reliable, the commercial organizations had taken over and the committees and other foibles of human organization did violence to the logic as the demands for greater speed and ease of use increased and more elaborate computers were built. The clumsy designs which followed inspired application of the word "kludge" and programmers, guilty more by omission than commission, were wryly amused.

It is notable that even as early as 1947, Stibitz, at the Harvard Symposium on Large-Scale Digital Calculating Machinery, pointed out the intimate relation between machine organization and the practical usefulness of the machine in terms entirely appropriate in 1963. A difficulty is that both designers and programmers are now overspecialized. A programmer to be professional must not ignore
the subject of machine organization, since pro-
gramming and machine organization are inex-
tricably intertwined. The machine designer, on
the other hand, cannot remain ignorant of pro-
gramming in the general sense; he must, for
example, have some knowledge of the problems
of translation of mechanical languages and an
appreciation of the control problems arising in
the preparation and use of very large programs.

The designer is still under marketing influ-
ence, defined here as what the salesman thinks
the customer wants or what he has convinced
him that he wants, and speed measured in terms
of time for individual memory accesses, arith-
metic operations, or simple loops provide easy,
if inadequate measures. Marketing, of course,
wants the latest glamour component since the
user has learned, with some reason, to equate
the new devices with easily observed improve-
ments in performance. The result of these pres-
sures is a neglect of improvements in machine
organization, improvements which could en-
hance over-all performance of systems and re-
move an obstacle to substantial advances in
programming system design.

Programming and Education

The computer sciences are beginning to find
a place in the university. Still overshadowed
by matters of computer procurement and de-
mands for internal services, there is now con-
siderable serious research going on in program-
ming and machine organization, automata
theory, switching theory, and mathematical
linguistics.

Programming is not trivial, and to quote
from the previously cited report by Goldstine
and von Neumann: “Since coding is not a static
process of translation, but rather the technique
of providing a dynamic background to control
the automatic evolution of meaning, it has to
be viewed as a logical problem and one that
represents a new branch of formal logics.”
For some reason, formal treatments of pro-
gramming have been both specialized and rare.
The idea that the construction and description
of algorithms is a legitimate concern of mathe-
matical instruction has its advocates. In view
of the major revision now taking place in the
mathematics curriculum, it would seem timely
to put programming on a formal basis for mass
education. Neither existing machine languages
nor the one-level-removed programming lan-
guages will do for this purpose. Considering
the enormous impact that the computer is cer-
tain to have in education, the necessity of avoid-
ing excessive influence from the computer
manufacturer must be recognized by all con-
cerned.

THE DESIGN AND CONSTRUCTION OF
PROGRAMMING SYSTEMS

Some of the important ideas that have filtered
down into current practice are reviewed in the
following sections in roughly the order of occur-
rence. Since the development of programming
techniques and concepts is poorly documented,
it is difficult to assign credit for origination of
the ideas or their successful application. Work-
ers in the field know that most of the useful
concepts had multiple origin. Significant ideas
have been embodied in useful programs, but not
otherwise documented, long before publication.
In view of the scarcity of program readers and
the lack of standards in language for descrip-
tion, one might say that the work in point was
not “published,” though its existence is implied
by the operating programs.

Combining Programs, Assembly

The importance of combining programs from
different sources was recognized early and a
program to accomplish this was described by
Goldstine and von Neumann.

Today, the assembly process is often a sub-
ordinate function in a compiler and symbolic
forms of machine language are rapidly declin-
ing in use in the technical application area.

Interpretation

Interpretive programs were important for a
number of years in programming technical ap-
lications, and it is interesting that one of the
first, developed at MIT, accepted an algebraic
language. Most of the later programs simu-
lated multiple-address machines with floating-
point arithmetic, index registers, and built-in
mathematical functions. Interpreters were suc-
cessful in spite of their inefficiency because
many of the deficiencies of the machine could
be easily corrected in the simulation and, thus,
in addition to more convenient and compact
codes, checkout was greatly simplified. The
success of these interpreters helped to ensure
the incorporation of more powerful operations
in the popular machines of the middle fifties which change, together with the development of compilers, caused the interpreter to pass out of use.

Compilation

The most promising solution to the programming problem was clearly compilation and, because of work volume, was conceived first for business data processing use. The first attempts were not outstandingly successful from the use standpoint. Excessive compilation time, difficulties in checkout that encouraged patching in machine language, and relatively bulky generated codes tended to discourage users.

The first languages were compromises of programmer convenience, complication requirements, and machine characteristics. Now that the compilation process is better understood, and machine designs less restricted, the programmer should soon have the use of the more powerful languages now practical.

The deficiencies of some of the first compilers stemmed from neglect of integrated design, use of existing programs for the assembly process, and storage limitations in the machines.

In the scientific systems, translation and code generation were conceptually separated into two problems through use of an intermediate language describable as that of an "ideal" object machine for the translator writer. The use of an intermediate language (often three-address or Polish) was one of the first important advances in compiler design.

Many improvements of technique were discovered only in recent years: the incorporation of a simplified assembly process as an integral part of the compilation, the use of linked lists for storage pools, stacks for analyzing tree structures in the input language, improved symbol dictionary techniques to eliminate bulky sorting passes unnecessary for the symbol-handling requirements of a compiler, and the coupling of sequential processes via buffer areas in storage to minimize intermediate storage on magnetic tape are notable examples. These techniques have made possible compact, fast, and inexpensively constructed compilers.

Later ideas applied successfully include the organization of a translator using recursive subroutines to handle control; and the writing of a system in its own language, followed by hand translation of a sufficient part of the code to allow compilation of the entire compiler program.

Syntax-Directed Compilers

Syntax-directed compilers are representative of the latest thinking in the field. In one plan (see the diagram) a translator is constructed for a machine A which for some class of languages described will produce as output a machine-independent intermediate language from input in language L. Statements in the intermediate language are then used as input to a machine-code generator, which produces actual machine code for machine B. Then, if translator and generator sections for use with ma-
chine $B$ are written in language $L$, the system can be transferred to machine $B$. The machine-code generator might itself be of a general nature in that specifications for the object machine $B$ might determine its output. In another plan, a compiler-generator on Machine $A$ produces the code for a compiler to run on machine $B$ given the syntax of the language $L$ and the machine-code specifications for the object machine $B$. In practice, the intermediate language has only a transitory existence within the machine.

Just how close these schemes are to practical application in their entirety is not known as published material is scanty: possibly the techniques are regarded as proprietary. There has been much speculation about and occasional partial demonstration of these ideas during the past few years. Both notions need clarification.

Theoretical investigations by Chomsky and others on formal languages are beginning to influence practical work. The least clear aspects remain in the areas of machine-code specification, and the writer suspects that the general machine-code generator of Scheme I is ill-defined in a practical sense, and perhaps presents the same order of difficulty as a machine-language-to-machine-language translator. The first part of Scheme II presents a similar difficulty, but by properly associating the syntactical elements of the language with desired machine code, a relatively simple program might serve for the first step, although it is not obvious what restrictions would have to be placed on the syntax and machine code.

**A Standard Machine Language**

A simple, and the writer believes effective, solution to the problem is in standardization on the transitional language of Scheme I as a symbolic *machine language*. The machine-code generator is then a straightforward program which handles differences of representation and makes substitutions for evaluative operations not available in the object machine. The description of the object machine relative to the standard language would consist of code representations for standard operators, code bodies in the machine language equivalent to omitted evaluative operators in the standard language, and *with limitations*, operator symbols in the machine language *not* corresponding to evaluative operations in the standard language, together with their equivalent symbol strings in the standard language. In the foregoing, the term, “evaluative operator,” is used to distinguish from control and descriptive operators since the standard machine language would be complete and definitive in that regard.

**Storage Allocation**

Storage allocation was the primary function of the early assembly programs and continues to be an important part of a modern programming system. A resident program is given the responsibility of handling the grosser storage allocation problems because many machine environments require this to be done dynamically.

Since multi-level storage is an economic necessity, its efficient utilization has long been a prime objective, though the difficulties appear formidable. In this regard, the machine designers have largely ignored the problem; extensive programmed control, very costly in terms of storage, is usually the consequence. It now appears that things can be done to alleviate this problem by providing adequate interrupt signals indicating states of tape, disc, and drum storage components and, through use of simple adaptive schemes for handling information transfers between storage levels.

The incorporation in procedural programming languages of notations for describing data structures such as arrays, files, and trees, and the provision to use these structures recursively together with indications of the scope of definition, will help greatly with the storage allocation problem and assist the programmer organizationally, and yet not burden him with need to cater to special machine characteristics. The structuring of programs should be, of course, an important part of the education of machine users.

**Operating Systems**

If operating systems are defined as a means of automatically accomplishing gross storage allocation, the ordering of programs, and the assigning of processor subsystems, it is reasonable to expect that these functions be mainly built into the equipment as a judicious combination of circuit logic and fixed program. That bulk of present-day systems is an implicit criticism of the machine designs. In the allocation of tape control, processor, storage, or input-
output subsystems, experience shows that if a choice must be made between equivalent units, the hardware should make the assignment. Automatic interrupt logic should make available error signals and state indications for the units involved, and should cause the automatic preservation and restoration of machine states, and thus, is closely related to mechanisms for the handling of subroutines.

**Levels of Language**

If machine languages are thought of as first-level languages, then procedural languages are second-level and problem-oriented languages third-level. ALGOL is an example of a second level, and the geometric language, APT, the third level.

Often a problem-oriented language will contain a procedural-type language as a sub-language, and ideally, that procedural language should be that in which the system is written and extended.

A procedural language should not contain a machine language as a subset, since this, in effect, is tantamount to admitting that the procedural language is incomplete, or that the machine is of such peculiar characteristics that rules for its use cannot be subordinated algorithmically within the translator for the procedural language. The fact is, however, that procedural languages, as defined at present, tend to be incomplete, and this is presumably compensated for in a nonstandard fashion by augmenting the language for each compiler, or including machine language. Machine pathologies are still with us, and this will be the case until the design philosophy in fashion is replaced and the machines now in use are obsolete.

A wealth of problem-oriented languages must be expected, since these will be the most powerful means of communication with machines, and their translators will manage the program libraries which grow up about the specialties for which the languages are conceived.

Variations of taste will ensure a number of procedural languages, but these are likely to be very similar. Standardization at level two might be desirable, but the difficulties seem tremendous and the cost of not standardizing will not turn out to be forbidding if some rationale is developed for level-one languages.

The reasons for rigid standardization at level one have been mentioned elsewhere in this paper, but are worth repeating. Machine languages need not take account of the questions of taste or humanly suitable representations, since virtually instantaneous translations in both directions between person and machine are clearly practical. All programs are expressed in terms of the same structural elements, and thus the designer has a standard which is completely independent of the proposed class of applications for the machine. In a given design, some characteristics could be emphasized at the expense of others (though probably not in a true general-purpose machine), but all control and data accessing functions would be present. The particular representation of elements in the machine language employed is also at the option of the designer, but each element must be available separately and be combinable in all meaningful ways.

In the sense just described, there can be a standard machine language acceptable to both machine designers and system programmers, and this language would be the logical and economically most satisfactory level for standardization. By developing a standard for machine language, both programming and machine design can proceed in a more orderly fashion with better use of valuable technical manpower.