QUARTERLY OF APPLIED MATHEMATICS
APRIL, 1972

SPECIAL ISSUE: SYMPOSIUM ON
"THE FUTURE OF APPLIED MATHEMATICS"

AUTOMATIC PROGRAMMING

BY

ALAN J. PERLIS

Yale University

Since the development of FORTRAN fifteen years ago we have observed a steady growth in the development of that part of computer science that deals with programming. In particular, there has been an outstanding development in programming languages: we can say more—and in a more natural fashion—to the computer and its attendant service programs to get our tasks accomplished. Our programming languages have grown more sophisticated—more attuned to the classes of algorithmic tasks we have set ourselves—and more selective, as we have come to be able to define classes of computations we will often do. Thus FORTRAN has spawned both PL/I and BASIC. The former has generalized the primitive concepts of FORTRAN's data and control and the latter has distilled from FORTRAN its simplest and most important essentials. Not only have the tasks influenced the languages, but also the hardware and the modes of use thereby engendered have influenced the languages. In PL/I the issues of programmer management of program execution are raised, e.g., in ON conditions and parallel execution. However, the extensions notwithstanding, PL/I is still directed at the same style of programming algorithms in existence as FORTRAN: PL/I is a summation of a decade of enormous experience.

This style of programming and the languages used have opened up a number of interesting areas for study and research:

1. Syntax analysis and parsing algorithms for mechanical languages.
2. Properties of program representation: recursion, iteration and backtracking.
3. Data structures: definition, analysis and computer representation.
4. Semantic models of programming languages.
5. Consequences of program execution: termination, correctness and efficiency.
6. Program equivalence.
7. Design and construction of compilers and interpreters.

Much of computer science research and education in the past seven to ten years has been concerned with the study of these issues which, regardless of the theoretic framework in which they are posed, analyzed and solved, are the responses to an applied problem: how to improve the characteristics of the communication channel between man and computer. The improvements have revealed themselves by permitting an increase in computation while saying less, and by permitting more people of limited computer literacy to communicate with the machine at all.

Those of us in the field of software have learned to be humble before the hardware engineer—we are aware that we are pushed into major new programming problems

arising out of the increased capabilities of the devices, while the conversely directed force is very rare: few major devices are created to solve vexing programming problems. Hardware drives the field! An examination of some growth measures will support this view. There has been an increase by a factor of 1000 in the 20 years from 1950 to 1970 in

1. Machine speed in operations/second.
2. Primary computer storage of a random access type.
3. Cheapness in operations/second-dollar.
4. Storage density in bytes/cu. ft.

The resultant increase in traffic has also caused the same factor increase in the number of lines of system code necessary to support the usage patterns on these larger and faster machines.

The enormous increase in traffic between man and his computer has prompted the development of multi-processing, time-sharing, parallel-processing and networking of computers. The programming systems managing these complex entities are called operating systems and their development has opened up still other areas of study and research in computer science:

1. Statistical models of software-hardware system performance.
2. Parallel processing algorithms.
3. Deadlock prevention and resource allocation.

and a class of management-system problems which arise purely from the complexity of the emerging systems which we deal with:

1. The rules by which a complex is decomposed into sets of simpler modules.
2. The delegation and distribution of design authority.
3. Documentation, testing and modification.
4. Model alteration, improvement and generalization.

Of course these same issues have already arisen in other guises and have been treated there by engineers and management scientists. However, in computer science these issues are new and of interest because of what appears to be the lack of physical constraints which dominate so many other large systems.

All of this ferment has occurred in about fifteen years. Already there are programs in existence with over a million instructions. Surely we may expect there to be many more of them and programs an order of magnitude larger than these in the next decade.

Somewhat separate from the above developments are those in artificial intelligence—another important component of computer science. Here one finds research coupled to a variety of purposes attached to a set of problems from which combination a collection of important techniques has emerged. Artificial intellegence is concerned both with modeling human thought processes and investing the computer, through its programs, with more human-like capabilities. Of course, in the last analysis these probably amount to the same thing. Some of the important purposes are:

1. The study of heuristics, e.g. as in applications arising in chemistry (molecular structure from spectral data) and formal mathematical manipulation by programs.

2. The investigation of human thought processes through the study of modeling programs.

3. Human-like extensions of the machines so that they will better serve us: research in speech, vision, natural language processing, motion and musculature.

Some of the important problems to which these purposes have been attached are:

1. Organic chemistry synthesis and molecular structure analysis.
2. Formula manipulation: integration, differential equations, Laplace transforms, etc.
3. Natural language processing: program understanding of speech and text.
4. Games, such as go and chess and checkers.
5. Robotics.
6. Mechanical theorem proving.

As programs have become larger, modes of use more stylized, and as numbers of machines increased in variety, the movement of programs from one computer environment to another has become an issue of some importance: how does one move a system of programs from one environment to another? Three approaches have been used:

1. Boot-strapping.
2. Program abstraction followed by re-programming.
3. Standardization.

Boot-strapping depends on the generation of programs in an environment-independent way from a programmed kernel which, while environment-dependent, is sufficiently simple that a sufficiently accurate environment-independent description can be given, thus permitting the kernel to be easily rebuilt in a new environment. The remainder of the system is presumably unaffected by the transition. This technique is now widely used and is limited mostly by the ability to describe adequately the data-processing functions of kernels. The method of abstracting and reprogramming maps an environment-dependent program into one which has the same function or purpose but is more abstractly specified—by which one means that it is less environment-dependent! An example would be mapping an efficient linear equation solver which utilizes the available core, disk and tape characteristics of its current environment into a program which is efficient in another different environment. Abstraction is very difficult to mechanize and only a very few algorithms, e.g. sorting, are understood well enough to yield to this method. Standardization, of course, legislates the problem away by insisting that all environments be, if not identical, at least common in a useful sub-environment so that restriction to it eliminates the problem almost completely.

By and large artificial intelligence has not concerned itself with the problems arising from the programming process per se. However, here is a human problem-solving activity of ever-increasing importance to which the tools of artificial intelligence can be applied. Already some activity exists and some progress can be reported:

1. A heuristic program to design operating systems has been attempted.
2. A heuristic program to design instruction codes for a computer has been written.
3. A heuristic program to design sorting programs is being built.
4. An approach to automatic program production from statements of input-output predicates, utilizing mechanical theorem proving, is under development.

Heuristics is associated with design. Consequently we may expect the techniques of artificial intelligence to be influential wherever programs are being designed. This

is particularly true if the goal includes mechanical design of programs. If one may prognosticate, automatic programming research will become absolutely intertwined with artificial intelligence work in this ensuing decade. If that will be so, to what problems in automatic programming will we turn so as to take maximum advantage of what artificial intelligence has to offer? I believe these problems will be:

1. The programming problem itself: how may we create programs that write detailed programs from little information?
2. The re-programming problem: how may we transfer a collection of programs from one environment to another?
3. The program-understanding problem: how may we create programs which "understand" other programs so that they can convey information about a program to anyone who requests such information? It is probably through such programs that the issues of education, documentation, monitoring and improvement of large programs can be brought under manageable control. However, it may well turn out that programs which understand can themselves be understood only with great difficulty. We are already capable, however, of taking some important first steps with the translators we are now accustomed to use. With very little extra effort these same translators could optionally produce an auditing program which could monitor the use of resources by an object program in its successive executions. With somewhat more effort a flow path analysis could be produced which itemizes the paths and conditions under which they were followed.

A good translator is many-one; i.e., it attempts to find the "best" object program for a set of functionally equivalent source language programs, and it attempts to determine this object program from a lexicographic analysis of the source program. Once the translation has been achieved, answers to questions about the original source program are often so difficult to obtain from the object program that re-analysis of the source becomes necessary. Alas, it often happens that the source has long since been lost, or no longer matches the object program, etc.

Thus it seems reasonable often to produce some coded version of the "reverse" translation process as well as to have a program available which can answer questions about the triplet: source program, source to object translation and object program.

In a sense, progress in programming language design can be measured by the ratio of program text we must write which says *what* is to be done to that which says *how* it is to be done.

Of course, we all know that this is a layered issue: "what" at one level must be "how" at another, presumably lower, level in the language processing hierarchy. It is precisely the increase in this ratio which is the source of the difficulty in answering questions about a source program given the object program.

The central idea of automatic programming is precisely that of defining program specification formats which, for an interesting set of tasks, is very high on "what" and very low on "how". The assumptions on which this is based are:

1. It is simpler to state what is to be done than how it is to be done.
2. For most uses of the computer the traffic between people, programs, and computers is of greatest value when it deals with "what" rather than "how".

However, these assumptions are not always valid. It is often much easier for pro-

grammers to write at least parts of correct programs than to state unequivocally what these parts do in some larger context. Programs are often their own best explanations.

An approach which has been suggested by several investigators is to phrase the programming problem as the production of a program as the by-product of the proof of a theorem. The conditions on the input data are expressed as a predicate and the output is similarly expressed. The theorem to be proved is then phrased as follows: Let the input be an n-tuple of objects X satisfying an input predicate $\phi(X)$. The desired output predicate is $\psi(X, Z)$ where Z is an m-tuple of output data objects. The program is a (partial) function F such that $Z = F(X)$, F is defined for the data X and $\psi(X, F(X))$ is true. The function is constructed through the constructive proof of the theorem

$$(\forall\ X)[\phi(X) \supset (\exists\ Z)\psi(X, Z)]$$

by a theorem-proving program. Here "what" is the theorem and "how" is the sequence of substitutions made during the proof and leading to a definition of Z. Programs with loops are constructed by appealing to various forms of mathematical induction. The language of the predicate calculus is used for expressing "what", while the "how" is obtained mechanically by a theorem-proving program. Certainly the above approach focuses attention on the major issues: the definition of linguistic mechanisms which permit us to

1. State what programming task is to be done.
2. Construct the mechanisms for accomplishing this task using a reduction program.

There seems little doubt that, in the early stages of this work, the linguistic mechanism will be a combination of the predicate calculus and the algorithmic languages of the FORTRAN-ALGOL-APL-LISP variety. Furthermore, the "reduction" program will operate on a mixture of standard translation, heuristic search and formal theorem-proving techniques.

It seems reasonable that we cannot confine our serious attention to mechanical theorem-proving techniques, since the problems for which we seek to find programs are far beyond their present or near-future capabilities. Indeed, we must remember that the entire development of programming languages has been focused on the development of mechanisms for expressing "what", e.g. loop-control, macros, procedures, and data structures. Probably we must begin to add to our languages statements expressing problem-solving techniques which are to be applied to statements expressing input-output relations. As a rather trivial example, consider the bucket problem wherein one has $2N$ empty buckets whose gallon volumes $P_1, P_2 \cdots P_{2N}$ are pairwise relatively prime and a reservoir of $M = \sum_{k=1}^{2N} P_k$ gallons of liquid. By emptying or filling only one bucket from the reservoir at each stage one is to generate a sequence of M volumes x_k held by the $2N$ buckets such that each volume from 1 to M is attained. The variable identifiers and the input-output predicates are easy to write down:

1. All variables take on positive integer values.
2. X is $P_1, P_2, \cdots, P_{2N}, 2N, M$.
3. Z is $x_1, x_2, \cdots, x_M$.
4. $\phi(X)$ is $(M = \sum_{k=1}^{2N} P_k) \wedge (\forall^{2N}\ i)(\forall^{2N}\ j)[\gcd(P_i, P_j) = 1]$
5. $\psi(X, Z)$ is $(\forall^{k}\ i)(\forall^{M}\ j)[x_i \leq M \wedge i \neq j \supset x_i \neq x_j$

$$\wedge\ i > 1 \supset (\exists^{2N}\ k)[x_{i+1} = x_i + P_k]].$$

If a back-tracking algorithm were available, a good automatic programming system would presumably use it, since the problem can be solved by enumerating the integers from 1 to M and trying permutations. Thus a program could be created forthwith. However, we know that an algorithm can be easily specified that involves no backtracking or search. Whereas for a problem which seems semantically quite similar, that of obtaining a particular distribution of liquid among the buckets, there doesn't always exist a solution and when there does the only known method for finding it is by a backtracking technique. To find the non-backtracking algorithm for the former problem, the program generator would have to search for a strategy which takes advantage of cycles and then apply induction. The problem becomes trivial by observing that the buckets may be taken two at a time and each pair of buckets once only in the outer cycle of two.

But it is just this kind of problem-solving analysis that is the domain of artificial intelligence: program writing by programs is a fundamental problem in artificial intelligence and hence in computer science. This then points to the task for programming research in the next decade.

References

[1] Frederick M. Haney, *Using a computer to design computer instruction sets*, Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, 1968

[2] B. G. Buchanan, G. L. Sutherland and E. A. Feigenbaum, *Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry, Machine intelligence* **4** (B. Meltzer and D. Michie, eds.), Edinburgh University Press, Edinburgh, Scotland, 1969

[3] Zohar Manna and Richard Waldinger, *Towards automatic program synthesis*, Comm. ACM, pp. 151–165, (1971)