AUTOMATIC CODING FOR DIGITAL COMPUTERS

Dr. Grace Murray Hopper

Remington Rand

AUTOMATIC CODING FOR DIGITAL COMPUTERS

A TALK PRESENTED AT
THE HIGH SPEED COMPUTER
CONFERENCE, LOUISIANA STATE
UNIVERSITY, 16 FEBRUARY 1955

BY

DR. GRACE MURRAY HOPPER

DIRECTOR OF PROGRAMMING RESEARCH
ECKERT MAUCHLY DIVISION

Remington Rand

Automatic coding is a means for reducing problem costs and is one of the answers to a programmer's prayer. Since every problem must be reduced to a series of elementary steps and transformed into computer instructions, any method which will speed up and reduce the cost of this process is of importance.

Each and every problem must go through the same stages:

Analysis,
Programming,
Coding,
Debugging,
Production Running,
Evaluation

The process of analysis cannot be assisted by the computer itself. For scientific problems, mathematical or engineering, the analysis includes selecting the method of approximation, setting up specifications for accuracy of sub-routines, determining the influence of round-off errors, and finally presenting a list of equations supplemented by definition of tolerances and a diagram of the operations. For the commercial problem, again a detailed statement describing the procedure and covering every eventuality is required. This will usually be presented in English words and accompanied again by a flow diagram.

The analysis is the responsibility of the mathematician or engineer, the methods or systems man. It defines the problem and no attempt should be made to use a computer until such an analysis is complete.

The job of the programmer is that of adapting the problem definition to the abilities and idiosyncracies of the particular computer. He will be vitally concerned with input and output and with the flow of operations through the computer. He must have a thorough knowledge of the computer components and their relative speeds and virtues. Receiving diagrams and equations or statements from the analysts, he will produce detailed flow charts for transmission to the coders. These will differ from the charts produced by the analysts in that they will be suited to a particular computer and will contain more detail. In some cases, the analyst and programmer will be the same person.

It is then the job of the coder to reduce the flow charts to the detailed list of computer instructions. At this point, an exact and comprehensive knowledge of the computer, its code, coding tricks, details of sentinels and of pulse code are required. The computer is an extremely fast moron. It will, at the speed of light, do exactly what it is told to do--no more, no less.

After the coder has completed the instructions, it must be "debugged". Few and far between and very rare are those coders, human beings, who can write programs, perhaps consisting of several hundred instructions, perfectly, the first time. The analyzers, automonitors, and other mistake hunting routines that have been developed and reported on bear witness to the need of assistance in this area. When the program has been finally debugged, it is ready for production running and thereafter for evaluation or for use of the results.

Automatic coding enters these operations at four points. First, it supplies to the analysts, information about existing chunks of program, subroutines already tested and debugged, which he may choose to use in his problem. Second, it

supplies the programmer with similar facilities not only with respect to the mathematics or processing used, but also with respect to using the equipment. For example, a generator may be provided to make editing routines to prepare data for printing, or a generator may be supplied to produce sorting routines.

It is in the third phase that automatic coding comes into its own, for here it can release the coder from most of the routine and drudgery of producing the instruction code. It may, someday, replace the coder or release him to become a programmer. Master or executive routines can be designed which will withdraw subroutines and generators from a library of such routines and link them together to form a running program.

If a routine is produced by a master routine from library components, it does not require the fourth phase--debugging--from the point of view of the coding. Since the library routines will all have been checked and the compiler checked, no errors in coding can be introduced into the program (all of which presupposes a completely checked computer). The only bugs that can remain to be detected and exposed are those in the logic of the original statement of the problem.

Thus, one advantage of automatic coding appears, the reduction of the computer time required for debugging. A still greater advantage, however, is the replacement of the coder by the computer. It is here that the significant time reduction appears. The computer processes the units of coding as it does any other units of data --accurately and rapidly. The elapsed time from a programmer's flow chart to a running routine may be reduced from a matter of weeks to a matter of minutes. Thus, the need for some type of automatic coding is clear.

Actually, it has been evident ever since the first digital computers first ran. Anyone who has been coding for more than a month has found himself wanting to use pieces of one problem in another. Every programmer has detected like sequences of operations. There is a ten year history of attemps to meet these needs.

The subroutine, the piece of coding, required to calculate a particular function can be wired into the computer and an instruction added to the computer code. However, this construction in hardware is costly and only the most frequently used routines can be treated in this manner. Mark I at Harvard included several such routines -- sin x, $\log_{10}x$, 10^x. However, they had one fault, they were inflexible. Always, they delivered complete accuracy to twenty-two digits. Always, they treated the most general case. Other computers, Mark II and SEAC have included square roots and other subroutines partially or wholly built in. But such subroutines are costly and invariant and have come to be used only when speed without regard to cost is the primary consideration.

It was in the ENIAC that the first use of programmed subroutines appeared. When a certain series of operations was completed, a test could be made to see whether or not it was necessary to repeat them and sequencing control could be transferred on the basis of this test, either to repeat the operations or go on to another set.

At Harvard, Dr. Aiken had envisioned libraries of subroutines. At Pennsylvania, Dr. Mauchly had

discussed the techniques of instructing the computer to program itself. At Princeton, Dr. von Neumman had pointed out that if the instructions were stored in the same fashion as the data, the computer could then operate on these instructions. However, it was not until 1951 that Wheeler, Wilkes, and Gill in England, preparing to run the EDSAC, first set up standards, created a library, and the required satellite routines and wrote a book about it, "The Preparation of Programs for Electronic Digital Computers". In this country, comprehensive automatic techniques first appeared at MIT, where routines to facilitate the use of Whirlwind I by students of computers and programming were developed.

Many different automatic coding systems have been developed - Seesaw, Dual, Speed-Code, the Boeing Assembly, and others for the 701, the A--series of compilers for the UNIVAC, the Summer Session Computer for Whirlwind, MAGIC for the MIDAC and Transcode for the Ferranti Computer at Toronto. The list is long and rapidly growing longer. In the process of development are Fortran for the 704, BIOR and GP for the UNIVAC, a system for the 705, and many more. In fact, all manufacturers now seem to be including an announcement of the form,

"a library of subroutines for standard mathematical analysis operations is available to users",

"interpretive subroutines, easy program debugging - ... - automatic program assembly techniques can be used."

The automatic routines fall into three major classes. Though some may exhibit characteristics of one or more, the classes may be so defined as to distinguish them.

(1) Interpretive routines which translate a machine-like pseudo-code into machine code, refer to stored subroutines and execute them as the computation proceeds -- the MIT Summer Session Computer, 701 Speed-Code, UNIVAC Short-Code are examples.

(2) Compiling routines, which also read a pseudo-code, but which withdraw subroutines from a library and operate upon them, finally linking the pieces together to deliver, as output, a complete specific program for future running -- UNIVAC A -- compilers, BIOR, and the NYU Compiler System.

(3) Generative routines may be called for by compilers, or may be independent routines. Thus, a compiler may call upon a generator to produce a specific input routine. Or, as in the sort-generator, the submission of the specifications such as item-size, position of keyword, etc. instructs the generator to produce a routine to perform the desired operation. The UNIVAC sort-generator, the work of Betty Holberton, was the first major automatic routine to be completed. It was finished in 1951 and has been in constant use ever since. At the University of California Radiation Laboratory, Livermore, an editing generator was developed by Merrit Ellmore--later a routine was added to even generate the pseudo-code.

The type of automatic coding used for a particular computer is to some extent dependent upon the facilities of the computer itself. The early computers usually had but a single input-output device, sometimes even manually operated. It was customary to load the computer with program and data, permit it to "cook" on them, and when it signalled completion, the results were unloaded. This pro-

cedure led to the development of the interpretive type of routine. Subroutines were stored in closed form and a main program referred to them as they were required. Such a procedure conserved valuable internal storage space and speeded the problem solution.

With the production of computer systems, like the UNIVAC, having, for all practical purposes, infinite storage under the computers own direction, new techniques became possible. A library of subroutines could be stored on tape, readily available to the computer. Instead of looking up a subroutine everytime its operation was needed, it was possible to assemble the required subroutines into a program for a specific problem. Since most problems contain some repetitive elements, this was desirable in order to make the interpretive process a one-time operation.

Among the earliest such routines were the A--series of compilers of which A-0 first ran in May 1952. The A-2 compiler, as it stands at the moment, commands a library of mathematical and logical subroutines of floating decimal operations. It has been successfully applied to many different mathematical problems. In some cases, it has produced finished, checked, and debugged programs in three minutes. Some problems have taken as long as eighteen minutes to code. It is, however, limited by its library which is not as complete as it should be and by the fact that since it produces a program entirely in floating decimal, it is sometimes wasteful of computer time. However, mathematicians have been able rapidly to learn to use it. The elapsed time for problems--the programming time plus the running time--has been materially reduced. Improvements and techniques now known, derived from experience with the A--series, will make it possible to produce better compiling systems. Currently, under the direction of Dr. Herbert F. Mitchell, Jr., the BIOR compiler is being checked out. This is the pioneer--the first of the true data-processing compilers.

At present, the interpretive and compiling systems are as many and as different as were the computers five years ago. This is natural in the early stages of a development. It will be some time before anyone can say this is the way to produce automatic coding.

Even the pseudo-codes vary widely. For mathematical problems, Laning and Zeirler at MIT have modified a Flexowriter and the pseudo-code in which they state problems clings very closely to the usual mathematical notation. Faced with the problem of coding for ENIAC, EDVAC and/or ORDVAC, Dr. Gorn at Aberdeen has been developing a "universal code". A problem stated in this universal pseudo-code can then be presented to an executive routine pertaining to the particular computer to be used to produce coding for that computer. Of the Bureau of Standards, Dr. Wegstein in Washington and Dr. Huskey on the West Coast have developed techniques and codes for describing a flow chart to a compiler.

In each case, the effort has been three-fold:

1) to expand the computer's vocabulary in the direction required by its users.

2) to simplify the preparation of programs both in order to reduce the amount of information about a computer a user needed to learn, and to reduce the amount he needed to write.

3) to make it easy, to avoid mis-

takes, to check for them, and to detect them.

The ultimate pseudo-code is not yet in sight. There probably will be at least two in common use; one for the scientific, mathematical and engineering problems using a pseudo-code closely approximating mathematical symbolism; and a second, for the data-processing, commercial, business and accounting problems. In all likelihood, the latter will approximate plain English.

The standardization of pseudo-code and corresponding subroutine is simple for mathematical problems. As a pseudo-code "sin x" is practical and suitable for "compute the sine of x", "PWT" is equally obvious for "compute Philadelphia Wage Tax", but very few commercial subroutines can be standardized in such a fashion. It seems likely that a pseudo-code "gross-pay" will call for a different subroutine in every installation. In some cases, not even the vocabulary will be common since one computer will be producing pay checks and another maintaining an inventory.

Thus, future compiling routines must be independent of the type of program to be produced. Just as there are now general-purpose computers, there will have to be general-purpose compilers. Auxiliary to the compilers will be vocabularies of pseudo-codes and corresponding volumes of subroutines. These volumes may differ from one installation to another and even within an installation. Thus, a compiler of the future will have a volume of floating-decimal mathematical subroutines, a volume of inventory routines, and a volume of payroll routines. While gross-pay may appear in the payroll volume both at installation A and at installation B, the corresponding subroutine or standard input item may be completely different in the two volumes. Certain more general routines, such as input-output, editing, and sorting generators will remain common and therefore are the first that are being developed.

There is little doubt that the development of automatic coding will influence the design of computers. In fact, it is already happening. Instructions will be added to facilitate such coding. Instructions added only for the convenience of the programmer will be omitted since the computer, rather than the programmer, will write the detailed coding. However, all this will not be completed tomorrow. There is much to be learned. So far as each group has completed an interpreter or compiler, they have discovered in using it "what they really wanted to do". Each executive routine produced has lead to the writing of specifications for a better routine.

1955 will mark the completion of several ambitious executive routines. It will also see the specifications prepared by each group for much better and more efficient routines since testing in use is necessary to discover these specifications. However, the routines now being completed will materially reduce the time required for problem preparation; that is, the programming, coding, and debugging time. One warning must be sounded, these routines cannot define a problem nor adapt it to a computer. They only eliminate the clerical part of the job.

Analysis, programming, definition of a problem required 85%, coding and debugging 15%, of the preparation time. Automatic coding materially reduces the latter time. It assists the programmer by defining standard procedures which can be frequently used. Please remember, however, that automatic programming does not imply that it is now possible to walk up to a computer, say "write my payroll checks", and push a button. Such efficiency is still in the science-fiction future.