Title
ONR
Aut. Prog.
Prog. - Ipt.
Proc.

SYMPOSIUM ON

AUTOMATIC PROGRAMMING FOR DIGITAL COMPUTERS

NAVY MATHEMATICAL COMPUTING
ADVISORY PANEL
13 - 14 MAY 1954

DEPARTMENT OF THE NAVY
OFFICE OF NAVAL RESEARCH

OFFICE OF NAVAL RESEARCH
DEPARTMENT OF THE NAVY
WASHINGTON, D. C.

For
Jan Lee

IBM 701 SPEEDCODING AND OTHER AUTOMATIC-PROGRAMMING SYSTEMS

John W. Backus and Harlan Herrick
International Business Machines Corporation, New York, New York

Before the discussion of IBM's 701 Speedcoding System, it is interesting to consider the entire area of automatic programming for large-scale calculators to obtain a perspective as to the position of such a system in this field. The basic purposes of automatic-programming systems are primarily the following:

1. to provide operations which are not part of a machine's repertoire as operations of the system,
2. to provide operations which are part of a machine's repertoire in a more convenient form.

In other words, it is desired to supply the programmer with a more convenient language in which to designate a problem than that now used by the machine.

The economic reason for providing such a "foreign" language is that usually the programmer can specify a problem in such a language and have the machine translate it into its own language at less cost than would be required to specify the problem in machine language. In turn this economic fact holds true because very often the machine will have to do little more work, if any, in translating from the foreign language to its own and executing the resulting instructions than it would have to do in executing the program written originally in machine language. Any extra work the machine must do in translating from the language in which the program is written to its own (in addition to executing the resulting machine instructions) is more than compensated economically by the additional work the programmer would be required to do to state the problem in machine language rather than the nonmachine language. This small difference between the work a machine does in translating and executing a program in a foreign language and the work it must do to carry out the same job specified in its own language is obtained only if a large percentage of the operations which the programmer needs to specify are not part of the machine's repertoire. As a result some programming systems permit some operations to be designated in machine language while others which are not machine operations are interpreted as belonging to a nonmachine language.

The most common nonmachine operations which automatic-programming systems seek to provide for scientific and engineering calculations are floating-point operations and various elementary functions: square root, sine, exponential, and so on. In a nonfloating-point machine such systems require the use of a subroutine for every arithmetic operation. There are two principal methods by which automatic programming systems make these nonmachine operations available to the programmer: the interpretive method and the compiling method. An interpretive system translates and executes each nonmachine order every time such an order is encountered during the execution of the program, but leaves it untranslated in memory. Thus, each order will be translated as many times as it is encountered during the course of the problem.

A compiling system, on the other hand, translates each nonmachine order into a sequence of machine orders before the problem solution is begun. Each nonmachine order is therefore translated only once. When a comparison is made in this way it appears that compiler systems are more efficient than interpretive systems. However, the relative efficiency of the two systems depends on the type of machine being used.

In a calculator whose computing speed is great relative to the speed at which information may be brought from auxiliary storage into high-speed storage, it may be possible to execute ten or more orders in the time required to bring one order into high speed storage. In such a machine it is of course desirable to limit the number of instructions in a program, even though it may be necessary for the machine to execute more instructions while carrying out a compact program than in a less compact one. If the necessity for bringing a thousand instructions of a lengthy program from auxiliary storage into high speed storage can be eliminated by executing five thousand extra instructions in a more compact program, the more compact program will be executed in less time than the lengthy one plus the time to read the thousand instructions into the main memory. Thus, in such a calculator, saving instruction space is usually equivalent to saving time.

If we now consider how to produce a good machine-language program for a given problem to be done with floating-point arithmetic, we require that it should be compact. Thus, we rule out a program made up of floating-point subroutines written over and over again, once for each floating-point operation, only changing addresses referring to operands. Such a program would occupy many times the amount of space required by a program consisting of a calling sequence for each desired floating-point operation. Each calling sequence designates an operand or operands and passes control to the appropriate floating-point subroutine which is represented only once in the machine.

Thus, if a calculator with high computing speed relative to input-output speed were to employ a compiling system, the compiler would have to produce a program of calling sequences in order to solve the specified problem efficiently. However, the number of steps that a subroutine must carry out in order to obtain the information given by a compact calling sequence is fairly large unless use can be made of a machine facility for automatic address modification. Furthermore, these are almost identically the steps by which an interpretive program could obtain similar information from a suitable nonmachine-language instruction. Thus, a program of machine-language calling sequences (compiled from nonmachine-language instructions) could be executed no faster than the nonmachine-language program could be interpreted and executed by an interpretive system. There is also the time needed to compile the program of calling sequences in addition to the time to execute such a program: it is clear, therefore, that compiling systems are less efficient for this type of calculator than interpretive systems when the majority of operations specified are not machine operations.

In a machine whose computing speed is slow relative to the rate at which information may be transmitted into the main membry, it may happen that the time required to execute an instruction is comparable to the time required to obtain an instruction from auxiliary storage. For such machines the effectiveness of compiling systems as opposed to interpretive systems is evidently reversed. In this case saving instruction space at the expense of executing more instructions usually will not result in saving time. Very large programs consisting of fixed routines repeated many times and differing only in certain addresses are therefore practical, since the process of modifying many instructions in storage becomes more time consuming than bringing new instructions into storage. Consequently, for this type of machine, compiling systems which write large programs as a translation of short, nonmachine-language programs are more efficient than interpretive programs.

This, then, is some of the background of automatic programming as it stands today and some of the apparent reasons for the difference in approach which various groups have taken. Automatic-programming systems have been designed to lessen the enormous burdens of the programmer by providing a larger and more convenient instruction repertoire than a given machine provides. Groups operating machines which execute instructions considerably faster than they can be brought into main storage have developed interpretive systems as the most efficient means of achieving this; other groups have developed compiling systems as the best method for the machines they operate.

Speedcoding was designed to be used in what might be called an "open shop" type of computing installation, that is to say, one in which most of the programming would be done by the scientists or engineers with whom the various problems originated rather than by the group operating the machine. Thus, it seemed necessary to place the utmost emphasis on making the system as easy as possible to use and to learn, even at the expense of some efficiency in actual machine operation.

We felt that the following specific objectives in Speedcoding would contribute to this general aim.

1. As few instructions as possible should be required to specify a program. Programming time should be minimized.
2. Programs should be easy to check out.
3. Scaling should be unnecessary.
4. Address modification should be made very convenient.
5. Transfer of arbitrary blocks of information to and from high-speed storage should be easy to specify.
6. Common functions should be readily available.
7. Optional automatic checking of calculations should be provided.

The following description of the Speedcoding system shows how these objectives were achieved.

Speedcoding is an interpretive system which provides floating-point arithmetic. In a floating-point system data and instructions have completely different forms and are treated differently. Therefore, it was thought desirable to have separate methods of dealing with each of these two types of information. Thus, each Speedcoding instruction has two operation codes in it called OP_1 and OP_2. Code OP_1 has three addresses A, B, and C associated with it and is always an arithmetic or an input-output operation. Code OP_2 has one address, D, associated with it and is always a logical operation. Code OP_1 deals with floating-point numbers; code OP_2 deals with instructions. This arrangement was also adopted because it makes efficient use of the space available for an instruction and because it often speeds up operation by reducing the number of instructions which must be interpreted.

Code OP_1 operations consist of the usual arithmetic operations plus square root, sine, arctangent, exponential, and logarithm. There are also orders for transferring arbitrary blocks of information between electrostatic storage and tapes, drums or printer. These input-output orders have built-in automatic checks for correct transmission. Accompanying the OP_1 operation code is a code to specify that any or all of the three addresses, A, B, C, should be modified during interpretation by the contents of three associated special registers (B tubes) labeled R_A, R_B, R_C. This feature often enables one to reduce the number of instructions in a loop by a factor of 1/2.

The OP_2 operation in an instruction is executed after the OP_1. By means of this operation one can obtain conditional or unconditional transfer of control. One can initialize the contents of any of the R-registers or one can, in one operation, increment any or all of the R-registers and transfer control. Another OP_2 operation allows one to compare the contents of an R-register with the given D-address and skip the next instruction if they are equal. The OP_2 also provides a set of operations for using a fixed point accumulator for computations with addresses and for comparing the contents of this accumulator with the D address. Finally, OP_2 provides a convenient means of incorporating checking in a problem if desired. This feature consists mainly of two operations, START CHECK and END CHECK; all instructions between these two orders may be automatically repeated as a block and at the end of the second repetition two separate check sums which have been accumulated during the two cycles are compared and the instruction following the END CHECK skipped if they agree.

Instructions or data may be stored anywhere in electrostatic or auxiliary storage as single Speedcoding words. Average execution times for various Speedcoding operations are as follows:

Add:	4.2 milliseconds
Multiply:	3.5 milliseconds
Read Tape:	14 milliseconds access plus 1.6 milliseconds per word
Transfer Control:	0.77 milliseconds

Electrostatic storage space available is about 700 words.

Let us follow a problem from its coded form on programming sheets and data sheets until it is checked out and ready to run. First the instructions and data are punched on decimal cards whose formats are identical to those of the sheets. If there are any data or instructions which the program requires from tapes or drums, loading-control cards are punched (one for each block of information) which will cause the loading system to put this information in the proper places in auxiliary storage. The deck of binary cards for Speedcoding is placed in front of this decimal deck consisting of instruction cards, data cards, and, possibly, loading-control cards, and the entire deck is put in the 701 card reader. When the load button is pressed, the information will be stored in electrostatic storage, on tapes or on drums as indicated by locations on the cards. When the last card is read, execution of the program will begin automatically.

In checking out the program, use will be made of a feature of Speedcoding which has not been mentioned yet. Each Speedcoding instruction includes a list code which may be assigned one of three possible values. Associated with each of these values is a switch on the operator's panel of the 701. During execution of a program all instructions will be printed which have list codes corresponding to switches which are on. If one has properly assigned list codes, one may then check out a problem in the following way: One begins execution of the program with all three switches on, after seeing the most repetitive portions of the program printed once or twice, one of the switches is turned off, after which only moderately repetitive parts of the program are listed. Finally, the second switch is turned off and only the least repetitive instructions are seen. If trouble is encountered in the last cycle of a much repeated loop, one can approach this point rapidly with a minimum of printing and just before reaching it one can turn on all three switches and see all details of the program. Each instruction is printed with alphabetic operation codes just as it was originally written on the programming sheet. The floating-point numbers at A, B, and C, the contents of the R-registers and the address accumulator, are also printed with each instruction.

Extensive experience in using Speedcoding at several 701 installations has verified its usefulness and practicality. And of course this experience has resulted in several additions and improvements to the original system as described. Several 701 installations use the system extensively. Although the original system was developed by IBM's New York Scientific Computing Service, two large aircraft companies have contributed important additions to the system which have been made available to all the other 701 installations throughout the country. Some of these outstanding new features are:

1. A program to print out memory which automatically distinguishes between instructions and data and prints accordingly.
2. A facility for automatically doing extended precision floating point calculations.
3. A facility to permit adding the result of an operation to the contents of the specified result register, C, and placing the sum in C.
4. A facility to allow alternating between machine language and Speedcoding language.
5. An automatic table look-up operation which provides quadratic interpolation for bivariate functions.

Now let us again return to the subject of automatic programming in general. Up to now the usual purpose of automatic programming systems such as Speedcoding has been to make available to the programmer nonmachine instructions, each of which usually represents a fairly long sequence of machine instructions when translated. Even though the machine automatically expands the instructions of the programmer into the necessary machine operations and thereby saves him considerable labor, the work of the programmer still contains a lot of drudgery. He must still decide what actual or symbolic addresses he will assign to his data and to his instructions and must keep track of these decisions once made. He must still write hundreds or thousands of instructions to specify how the machine is to do his problem. And as he debugs his program he must be intimately familiar with the relationships between all of these assignments and the quantities involved in his equations and between the multitude of instructions and the operations designated by the equations.

Obviously the programmer would like to write "X" instead of some specific address, and if he wants to add X and Y he would like to write "X + Y" instead of, perhaps:

```
CLEAR AND ADD   100
ADD             106
```

or some such gibberish. To go a step further he would like to write $\Sigma a_{ij} \cdot b_{jk}$ instead of the fairly involved set of instructions corresponding to this expression. In fact a programmer might not be considered too unreasonable if he were willing only to produce the formulas for the numerical solution of his problem and perhaps a plan showing how the data was to be moved from one storage hierarchy to another and then demand that the machine produce the results for his problem. No doubt if he were too insistent next week about this sort of thing he would be subject to psychiatric observation. However, next year he might be taken more seriously.

Few people will dispute the worthiness of such objectives: the question is, can a machine translate a sufficiently rich mathematical language into a sufficiently economical machine program at a sufficiently low cost to make the whole affair feasible? It seems that this is going to be the vital question for automatic programming in the future since the majority of present automatic programming systems seem to be primarily concerned with reducing the number of instructions needed to specify nonmachine operations. As machines of the future acquire larger repertoires of built-in operations these problems will diminish. This trend has already made its appearance in the recent announcement of IBM's new type 704 calculator. This machine has all the features of the 701 plus many new features. Outstanding among these for the purpose of this discussion are automatic floating-point arithmetic and powerful automatic address-modification facilities. Since these are the two most important features which contribute to Speedcoding's economic usefulness, it seems evident that Speedcoding-type systems will be of little value for the 704. The programmer will rather write programs in a language much less removed from the machine language.

Consider the advantages of being able to state the calculations, decisions, operations on ordered arrays of data, and manipulations of blocks of data necessary for a problem solution in a concise, fairly natural mathematical language.

1. Such a language would mean that there was a close resemblance of the problem statement acceptable to the machine and the original formulation of the problem in a form for numerical solution. Thus, the programmer would be largely relieved of the necessity of making and remembering the extensive translation from the symbols of the original formulation to the addresses and operations of the program.

2. The tremendous increase in conciseness of the problem statement for machine solution and its closer relationship to the original statement will make it possible to check the material presented to the machine much more accurately.

3. Once the concise problem statement has been checked carefully and presented to the machine, debugging might virtually be eliminated.

4. The amount of knowledge the programmer would have to have about the machine and about programming conventions and library

programs would be greatly reduced. He would have to know only the rules of the language in which the problem is to be stated and the facts about the memory hierarchy of the machine. This of course would be a tremendous advantage to the open-shop type of computing installation. The programmer would not have to become familiar with dozens of codes, special machine features, timing conditions, addresses of various units, and a host of other facts which are presently necessary to know.

5. Much wasteful duplication of programming effort may eventually be eliminated. In programming many practical problems, programming situations and difficulties are often encountered which require considerable analysis to obtain a good procedure. In a great many instances similar situations have arisen before and a similar analysis has been made. In the rush of completing a problem it is not usually possible to make a good record of the results of such work. Therefore, analysis tends to be repeated again and again and is often never done as well as it could be. Programming techniques which arise in this way could be thoroughly analyzed and incorporated as programming capabilities of an automatic-programming system.

Whether such an elaborate automatic-programming system is possible or feasible has yet to be determined. If it is possible, an enormous amount of work will undoubtedly be involved in producing such a system. The tremendous strides the computing field has already made in developing automatic-programming systems indicate that if such systems are possible they will be produced. If they are produced it seems likely that such an application of the ability of modern calculators to do an enormous amount of work for a small cost may represent one more area in which machines can do a job more economically than human beings.