USE OF AUTOMATIC PROGRAMMING *

by

Walter F. Bauer

THE RAMO-WOOLDRIDGE CORPORATION

LOS ANGELES, CALIFORNIA

There are many definitions of the term "automatic programming". One possible definition is as follows: Automatic programming consists of devising and organizing computer programs which will allow the utilization of the computer in the performance of certain clerical tasks which would otherwise need to be done manually by the programmer. Automatic programming can be considered as involving only the organization of subroutines in the library or it can be regarded as the organization of an appropriate compiler or assembly program together with a comprehensive system for using the compiler to prepare and check out problems.

Motivations and History

There are four principle motivations for automatic programming: the high cost of programming, the programming manpower shortage, the desire to reduce elapsed time from presentation until production of a problem, and the desire and necessity to overcome certain computer shortcomings. The first of these, the high cost of programming, is important for it is estimated that the cost of programming and checking a problem for high speed electronic digital computer lies somewhere between $2 and $10 per instruction. In view of the fact that a thousand word program is a relatively short program, one sees this cost is far from trivial. Another commonly accepted statistic applied especially to scientific computation groups, but probably generally applicable, is that the manpower expense (not counting overhead and costs) supporting the computer is as great as the cost of running or renting the computer. Certainly the cost of renting a modern computer is a considerable matter, ranging upwards of $20,000 per month for the more powerful ones.

The programmer manpower shortage has been a serious matter for approximately two years and will continue to plague computer groups. During the year

---

* These notes result from a series of lectures given by the author at the Special Summer Session on Digital Computers at the University of Michigan in 1954, 1955 and 1956.

1956 the amount of computer time available (considering the increased speeds of the modern computers) will probably increase by a factor of at least 4. It is difficult to see how the manpower training problem will keep pace with this expansion rate. The need then is great to make good use of programmer time.

The third item, the elapsed time from problem origination to production of answers, is especially important in a computer laboratory which has, as its main function, the programming of many "one-shot" problems. It is only slightly less important in the case of the business data handling group which continually makes programming changes on its standard bill-of-fare programs. Once the decision has been made as to the logical steps involved, it becomes seriously aggravating to the originator and the programmer alike to accept the long drawn out period until the production state is reached.

The University of Cambridge in England was probably the first group which used automatic programming to any significant degree. The EDSAC computer had a very limited memory size of 256 words and computer word length of 16 binary digits and most problems needed special attention to overcome these difficulties. The clever and imaginative scientists operating the computer built a system of subroutines and a method for handling them which to this date serves as a model for this type of work. The organization and a description of the computer is contained in a book by Wilkes, Wheeler, and Gill.* Similarly the Whirlwind computer at the Massachusetts Institute of Technology had the limited word length of 16 bits and double precision interpretive programs were required to perform almost all numerical operations. Here again the scientists working with the Whirlwind computer continued and expanded upon the work of the Cambridge people and developed a comprehensive programming system for the computer. This system involves a complete scheme for programming problems in many forms (e.g. fixed point or floating point) and for debugging the programs and performing the necessary input and output. The automatic programming performed at M.I.T. has probably had more influence on computer use than the activity of any other group in the United States.

Composition of Automatic Programming

There are many items which can be included under the topic "Composition of Automatic Programming". The first of these and the most elemental is subroutines. Subroutines themselves do not warrant considerable discussion here except to remark that the subroutine is the basic building block of automatic programming. Recently, subroutines have been "glorified" by appending to them, certain prelude routines which "particularize" the routine. As a simple example, a prelude routine may particularize the routine for finding X^k to obtain a routine for finding the square root of a number X. In the same general vein, routines have been prepared which generate subroutines. For example, an output subroutine generating routine would construct a subroutine to provide output in a given format.

Perhaps the most common element of automatic programming is the use of interpretive type programs. These programs take instructions written in extra-machine logic, logic foreign to the internal computer logic, to perform certain desired operations. One of the most frequently used routines of this type is a routine for performing floating point arithmetic operations on a computer which

---

* Wilkes, M.V., Wheeler, D.J., Gill, S., The Preparation of Programs for an Electronic Digital Computer, Addison-Wesley Press, Cambridge, Mass., 1951.

can perform only fixed point arithmetic. Other interpretive programs allow the programmer to write programs in 3-address instruction logic for performance on a single address computer. Still other types can be found such as one programmed for the UNIVAC computer which enables the performance of analytic differentiation automatically by the computer. With the interpretive type program the psuedo instructions to the computer are stored internally and are translated to machine language as the program is run. Each instruction is translated each time the instruction is performed. The ability to perform such a function is the quintessence of the high-speed digital computer.

Currently the point of focus of an automatic programming scheme is the assembly program, alternatively called "compiler" or "executive routine." We shall use the term "assembly program" here for that name emphasizes the central function. By means of such a program, essentially a synthetic machine is created, a machine which is easier to program for and easier to check out prepared programs. As with the interpretive type of program mentioned above, programming is performed in a language foreign to the computer itself, the computer performs the translation into its own language by means of the assembly program. The important difference between the interpretive type program and the compiler is that the translation is done once and for all upon input of the program data. The program information in machine language is then either stored in the computer for immediate computation or it is read out of the computer onto magnetic tape, paper punched tape or punched cards, for later input. The advantage of outputting the data for storage outside the computer is that it may then be placed in a form for read-in into the computer at much higher speeds than the untranslated data. For example, untranslated program data at one instruction per 80-column card is read in at the rate of two instructions per second with the conventional card reader, while "binary cards", cards punched with translated data can be read into the computer at 48 words per second with the 701 and 1103 computers. The fact that the binary card can not be read and printed with conventional tabulating equipment is of little consequence since the original card is on hand and available. Since the assembly and translation function is performed, once-and-for-all, on input, it is sometimes referred to as an "input translation program".

Some further aspects of the detailed form of the assembly program are discussed below. However, an important over-all aspect should be brought forth. The first assembly programs were such that each instruction to be performed by the computer was explicitly written. In other words, a one-to-one correspondence existed between untranslated and machine instructions. As techniques improved, it was seen to be desirable from many points of view to design a scheme to cause one line of untranslated program data to generate many machine instructions. As an example, one psuedo instruction specifying the merging of two sequences could generate a 50-100 word program. This one-to-many correspondence allowed the programmer to write fewer instructions and, as a consequence, reduced the number of programming errors. However, programming errors became harder to find with such a scheme since the translated machine data held little resemblance to the original. Also, compilers with the one-to-many philosophy often produced programs which ran less economical of machine time than the "hand-tailored" ones, much to the distress of the experienced programmer.

The main functions performed in the assembly operation are as follows:

1. Subroutine inclusion
2. Mnemonic devices
3. Number conversion
4. Standardized automatic data read-in
5. Cursory error analysis

The first of these, subroutine inclusion, implies that the assembly program can alter subroutines, usually routines relative to some fixed address so that they can operate in any chosen spot in the memory. In certain cases the assembly program changes or particularizes the subroutine as mentioned above. The subroutines may be read into the computer in a straightforward fashion from punched cards or punched tapes, or they may be compiled into the program in a more automatic manner from a higher volume storage such as magnetic tape or magnetic drum. The second item, mnemonic devices, implies that the assembly program allows the programmer to write in a language easier to use and easier to remember. For example, the symbol W1 could be used for the address of the cell containing the aircraft weight in the first period, and W2 for the aircraft weight during the second period instead of using the hard to remember numerical notation such as 12968 and 12969. Number conversion refers to some method of converting decimal numbers to binary for fixed point or floating point operations. Standardized data read-in implies that all programs are read in in substantially the same fashion, thus allowing a non-professional machine operator to operate the computer and obviating the necessity of the programmer's presence during program check. Cursory error analysis refers to a "quick look" by the assembly program as it translates to ferret out any obvious errors. An example of this is a scaling error which scales the number so high so as to make it "run out" of the left side of the register. In this case, the computer would indicate the error to the programmer.

Recently, the USE organization (Univac Scientific Exchange), a national organization which exists for cooperative programming among 1103A computer users much the same as the SHARE organization exists for IBM-704 users, discussed assembly programs at considerable length. The result of this discussion was a list of 17 features of a compiler deemed desirable by the members present. The 17 features, just as they were written in the minutes of that meeting, are included here in an Appendix.

More recently there has been a trend toward the integrated computation system which involves an assembly program only as an important but rather small part. The 5 items mentioned above refer only to the language, that is the means by which the programmer communicates information to the machine on the detailed level of his program data. In the integrated computation system this amount of information communicated is expanded to include items which otherwise would have to be communicated by word of mouth or by written instructions to the machine operator. In the integrated computation system, the following four broad areas are important:

1. The program language
2. Computing mode selection
3. Program alteration
4. Flexible error analyses

As mentioned above, the language refers to the five items of the above paragraph. Computing mode selection refers to the selection of the various possibilities in the computation system which the programmer has at his disposal. He may choose automatically, for example, the fixed point or the floating point computing mode, he may wish to translate his program data and immediately compute, or he may wish to translate and output the translated information for later input. Program alteration is, of course, necessary after mistakes are found, or in case the problem originator wishes to change certain procedures. Program alteration could involve deletions, corrections, or insertions, or, as is usually the case, various combinations of these. Flexible error analysis implies the selection of the means by which the programmer does a detailed analysis to find programming errors.

He may, for example, signal the computation system to store the data as it comes into the computer for a later "changed word post mortem" analysis. He may, for example, ask the system to monitor the computation on breakpoints or on each instruction. The important concept here is that all items are integrated together to form one computation system to the exclusion of the use of the machine with isolated subsystems. The integrated computation system almost certainly implies a more economical use of the computer, especially for code check operations which normally require about one-third of the computing time. It allows the programmers to use the computer for short lengths of time and receive the data they need for program check in a very short time. Further, it allows "unattended runs" with the resultant greater flexibility of computer operation and scheduling since the programmer does not have to be present.

Present Systems

One of the most important developments along the lines of automatic programming has been that of the Programming Research Group under Dr. Grace Hopper of Remington Rand. This group produced the A-1 compiler and its successor the A-2 compiler for the Univac computer. The A-2 compiler allows the programmer to communicate with the machine in a language much simpler than the machine language, allows him to generate many machine instructions by a relatively few A-2 instructions, allows automatic segmentation of the problem between the magnetic tapes and high-speed storage of the Univac, and allows a number of computing options such as floating point operation. More recently this group has prepared the B-0 compiler which is an extension of the A-2 in two directions: it allows for certain operations which are more frequently used in business or commercial applications, and it is so designed to allow flexibility for future applications, making possible modifications and additions as may be necessary. The philosophy of the B-0 compiler is such that it translates various verbs of the imperative mode into computer language and would allow such translations for verbs such as "merge", "collate", "sort", "find the sine of", "find the nth root of", etc.

As another branch of Remington Rand's activities, Dr. Herbert Mitchell's New York group has prepared the BIOR (Business Input-Output Re-run) compiling system. This system, oriented toward commercial applications, provides the means for the programmer to take various blocks of his own coding and move them around and use them conveniently and simply. The design of the BIOR system took cognizance of the fact that the typical commercial problem involves large records of data which are read in from bulk storage such as magnetic tape, processed, and returned to magnetic tape or read out in another fashion.

When the IBM 701 computer appeared on the scene in 1954, a number of so-called "regional programming" schemes were prepared for the computer. These schemes designated certain regions of the computer memory for subroutines, instructions, and data. They involved mnemonic devices at least to the extent that cells belonging to different regions could be distinguished. Usually they allowed for inserting instructions by means of a type of "Dewey Decimal system" which would allow nine instructions, 129.1 to 129.9, to be inserted between instructions numbered 129.0 and 130.0.

About the same time that the regional programming schemes were being developed, the IBM SPEEDCO system was developed by a group at New York under John Backus. It allowed for easy programming since floating point arithmetic was used but suffered from the fact that the resulting interpretive program ran very slowly on the computer. Meanwhile, a number of customers of IBM were preparing

interpretive schemes which allowed entrance and exit from the interpretive floating point mode. Los Alamos' DUAL, Douglas' (El Segundo) QUICK, and Lockheed's FLOP are examples of these routines. More recently, John Backus' group at IBM has prepared FORTRAN (FORmula TRANslation) for the IBM-704 computer. FORTRAN will translate into computer language a program written very close to the language of the mathematician or scientist. This is an example of the so-called "algebraic coding system" which allows translation of indices, summation signs, parentheses, and arithmetic operation symbols of mathematical language.

In early 1955, a group of IBM 701 users in the Southern California area began the preparation of a compiler called PACT. This assembly program emphasizes index notation for operation on sequences of data and a scheme for simplifying the scale factoring operation. The PACT compiler is being modified to be used with the 704.

The automatic programming developments at M.I.T. and the University of Michigan are notable. Both of these systems include an assembly program or input translation device which uses symbolic notation, free addresses, and pseudo-machine commands. The emphasis, however, remains on the broad scope aspects of the entire system. Both systems place heavy emphasis on unattended computer runs and consequently allow automatic selection of computing modes and error diagnosis devices.

The activities with the integrated computation system at The Ramo-Wooldridge Corporation follow the lines of the M.I.T. and the University of Michigan system. In current use for the 1103 computer is a system which stores all service routines and subroutines (a total of about 8,000 words) on the 16,000-word drum and backed up on magnetic tape in the case of inadvertent destruction of drum data. The assembly program automatically assembles subroutines from the drum into the main program. During program check-out the operator has at his disposal the various service routines (dump routines, input and output routines, etc.) stored on the drum and can call upon them and use them quickly without recourse to any manual operation involving reading the program into the computer. The result is that only programmed data originated by the programmer is stored on punched cards external to the computer. All other routines, service and subroutines, are stored accurately and economically inside the computer where they can be quickly and automatically summoned to use.

Future Systems

Two trends are in evidence in future systems for computer use: the first is the increasing use and development of the automatic, comprehensive computation system referred to above, and the beginning uses of "microprogramming" techniques.

The completely automatic, comprehensive computation system is born of two central motivations: first, the desire to decrease the clerical work of the programmer, up-grade his level of work, and generally increase his effectiveness; and second, the desire to decrease the amount of non-productive computer time. In the face of the development of machines which will be extraordinarily complex and extraordinarily expensive to own and operate by today's standards, this second motivation looms important. This writer believes that the computation system used for most large-scale computers will, within 2-3 years, evolve into one with the following characteristics:

1. The computer will run automatically and without interruption for 3-5 hours, handling many different types of computer runs such as code checks and production computation of many different types.

2. The programmer will handle no computer storage media whatsoever but will deal only with printed pages of programming material he originates or with printed material giving the results of a computer run. He will submit his programming material to a data preparation room and, soon thereafter, receive on his desk the results of a computer run.

3. The computer will be scheduled 4-6 hours in advance by preparing "run tapes", magnetic tapes on which all of the information for running the various problems, in order, is recorded. Deviations from the 4-6 hour schedule to higher priority problems which arise will be possible.

4. Since programmers' run instructions and program alterations will be recorded on tape and handled automatically by the machine, running and checking out the 50-100 current programs of the scientific computer installation will become a file maintenance problem having many of the characteristics of that of the commercial installations today and, in many respects, considerably more difficult.

5. The computation system will find almost all clerical programming errors and, in most cases, will perform the operation the programmer "most likely" meant. In all cases the machine will inform the programmer of errors discovered and interpretations made through print-outs.

6. Machine operators will do nothing but change magnetic tape reels and actuate certain special routines in case of computer error, programmer error in using the computation system, or in exercising options to change the computer schedule to allow interjection of higher priority problems. Essentially, the programmer will operate the computer from his desk.

7. The system will check the operator's actions to a considerable extent. For example, the placing of a wrong tape reel on a tape unit would, in many cases, bring a remark to that effect printed out on the monitor typewriter.

8. Output tape units will be periodically removed from tape reels and placed on high-speed (500-1000 line per minute) printers for outputting information to be returned to the programmer. In many cases, output will be initiated automatically by the computer as needed, with no operator handling necessary.

Certainly much of the onus for the development of a system such as that described above rests with the computer manufacturer. For the system to operate as described, certain design features must be included into computer systems which have not been included to date. In particular, tape unit operation must become much more flexible. Independent tape operation (preferably independent tape search) and the operation of many tape units simultaneously will be mandatory. The ability

to erase and re-record within a block of previously recorded data will be of great importance. Another item is that all computer switches must be controllable by the internal program. Although no computer at present has these capabilities, the IBM has recently announced changes in the IBM-704 along these lines.

In the design of the computation system now in preparation for the 1103A computer to be delivered to The Ramo-Wooldridge Corporation certain steps have been taken toward the "ultimate system". The programmer writes run instructions, program alterations, program error diagnosis procedures, and output instructions on his programming form. This information is transcribed directly to magnetic tape and placed on the computer. The system assembles the program, includes all program changes, runs the program, performs the error analysis, and produces appropriate and complete print-outs for all these occurrences. The programmer submits the instructions for the entire operation in written form and waits until the computer results are returned to him. The file maintenance problem will be avoided during the first design by having an individual tape reel for each program. In a somewhat abridged form, the system will be in operation by the end of 1956.

The next couple of years will see the first uses of a technique in computer design and computer use called "microprogramming". Many computer users have long chafed under the use of instruction logics which were not well suited for their application. Microprogramming would allow the programmer to synthesize his own computer instructions from "microinstructions". Essentially, then, with microprogramming the programmer would "construct" his own control unit, the "construction" taking the form of a plugboard or assimilation at electronic speeds by means of computer program.

As of this date, most work in microprogramming is in the "talking stage". To this writer's knowledge, only one paper on microprogramming has been published, that by Herbert T. Glantz*. In the article Glantz presents a plan for a microprogramming facility which would provide for the computer to leave its normal (conventional) mode and jump to a "Micro Mode". In this mode, the computer would perform instructions made up of microinstructions according to a sequence as indicated in a special magnetic core memory. At the command of the programmer, the computer would jump back to the normal mode, performing arithmetic and logical operations in a conventional fashion.

One of the first meetings on microprogramming was held at Massachusetts Institute of Technology in March, 1956. At this informal meeting Dr. David Wheeler of Cambridge University, England, spoke of Cambridge's activities in the field. The Cambridge group, operating the EDSAC II computer, plans to encode subroutines in a magnetic core control matrix to allow the subroutines to operate at speeds comparable to normal instruction speeds. This technique could be characterized as making certain machine changes which would allow "macroprogramming", the synthesizing of complex instructions from common ones. Their activities along the microprogramming lines apparently will make use of a magnetic core matrix forming the heart of the computer control.

---

* Glantz, H. T., "A Note on Microprogramming", Journal of the Association for Computing Machinery, Vol. 3, No. 2, April 1956.

Two university groups in this country are interested in microprogramming and are formulating plans: M.I.T. and University of California at Los Angeles, the Numerical Analysis Research group which operates the SWAC. M.I.T. has ideas which would make available an additional microinstruction on the Whirlwind I computer as part of its regular instruction repertoire. The address part of this instruction would control 120 subcommand lines of the control matrix of the computer. Presumably, the programmer would contrive his own instruction by using the 120 lines as he wishes.

The activity at the University of California at Los Angeles has resulted in the writing of a master's degree-level thesis by Robert Mercer (to be published) under the direction of C. B. Tompkins. If present plans materialize the Numerical Analysis Research group will make extensive studies of the SWAC control system, followed by extensive changes which will probably result in a plugboard to allow the selection of sub-commands. The group hopes to complete the next step which would allow the choice of sub-commands at electronic speeds under stored program control.

The future possibilities of microprogramming are considerable. With microprogramming one visualizes a three-level hierarchy of programming. At the production level programmers would use subroutines performing operations like "find sine of", "merge", etc., probably by means of elaborate compiling routines. At the next level "subroutine programmers" would prepare the subroutines which would be composed of instructions made up of micro-commands. A number of programmers would program instructions from microinstructions. At each level, the programmers would prescribe the routines or instructions necessary; it would be the responsibility of the next "lower" level group to fashion these tools so that they are in some sense optimum. Most likely programmers at the production level would know nothing of the activity at the microprogramming level, and vice-versa. This is nearly the case at the present time between production programmers and those programmers preparing compiling routines and the computation system. Future computation system design will largely integrate the activities of the three levels.

There seems little doubt that the degree of success of microprogramming will be primarily a function of the speed with which synthesized instructions can be executed. Five years ago, most computer people were satisfied that floating point could be performed interpretively and that the slow speeds were acceptable. As computer usage increased, it became evident that floating point operation in this fashion was much too slow, that floating point as built-in hardware was necessary. This may prove to be the experience again with microprogramming. As soon as an instruction is synthesized, used, and appreciated, users will want it included as hardware so that it will operate faster. Microprogrammed instructions must operate at speeds competitive with their permanent, wired-in counterparts or the whole technique will fall into disfavor and die of atrophy.

APPENDIX

The following is a list of desirable compiler features which the Univac Scientific Exchange (USE) organization recently adopted.

1. Compile subroutines from pseudo-instructions. A pseudo instruction requiring the use of some library subroutine would appear in the main program. The subroutine necessary to carry out the desired function would then be automatically compiled into a so-called compiled region. The line of coding which originally contained the pseudo-instruction would be replaced by the appropriate calling sequence of one or more instructions.

2. Assign cell numbers to otherwise undefined symbolic addresses. Ordinarily these cell numbers will be assigned addresses in a compiled region. This feature allows easy assignment of working storage locations.

3. Use numerical constants as addresses. The compiler should be able to detect that an address section of an instruction is actually a numerical constant. The value of this number would then be stored in an otherwise unused cell in the compiled region--the address of that cell would be filled in as the appropriate address section of the instruction.

4. Symbolic addresses. The compiler should be able to accept symbolic addresses similar to those now accepted as standard for subroutines by the USE Organization. Implicit in the phrase symbolic addresses is the concept of free addressing.

5. Easy method of writing numbers. In this sense a number is a numerical constant which occupies one or more full registers and is ordinarily thought of as a number--this is in contrast to the writing of numerical addresses. It is expected that both stated point and floating point single and double precision decimal numbers will be acceptable to the compiler as well as octal numbers.

6. Ability to generate in-out routines. The thought here is that the programmer could make relatively simple specifications of the form of the numerical output which he desires and that the compiler would generate and assemble automatically the routine necessary to do the particular job specified.

7. Ability to make any type of change easily with both card and tape input. This is obviously a worthwhile and noncontroversial objective. However, the discussion showed that there may be considerable compromise necessary to work out the details of just how such generalized changes would be made.

8. Generate calling sequences. This ability of the compiler was alluded to in point number 1 above. A calling sequence may very well require more than just an RJ--the common compiler should be able to generate automatically these calling sequences in a predictable fashion.

9. Provide binary tape output. In some installations a binary tape output may be the most common form of output, in others, it may be provided as an option. In any case the compiler should have this ability.

10. Provide symbolic side-by-side listing. Some installations have found this type of listing a most useful form of output, particularly during trouble-shooting periods. The symbolic side-by-side listing is to be contrasted with the present method at some installations where the original keypunched cards are listed on one piece of paper; subsequently a related listing showing the translated code (usually in octal) is produced on another piece of paper.

11. Detect errors during input conversion. Clearly typing or syntactical errors may be made in preparing the code and the input cards or tape. A good compiler should be able to detect such errors, make a list of them for use by the programmer, and still continue the conversion if at all possible. This then allows the programmer to study the list of errors and correct as many as possible before returning to the machine.

12. The compiler must be able to handle symbolic programs which are input on either cards or tape--in other words both forms of input must be possible and convenient.

13. Compatibility with mistake diagnostic routines. The form of the input and the provisions made for the programmers use of mistake diagnostic routines must be completely compatible. That is, information which the programmer must specify in order to diagnose coding errors, must be in a form which can be handled by the compiler and is compatible with the ordinary form of input.

14. Can incorporate USE subroutines. The compiler should be able to handle a USE subroutine unchanged from its original symbolic form and incorporate such a subroutine into the main body of a program when that symbolic subroutine is included as part of the original manuscript.

15. Identification of Output. Any material which the compiler produces as output should be completely identified as a matter of routine. For example, the symbolic side-by-side listing should be identified--this would include the programmer's name, the date if possible, program number, etc.

16. Direct input. After the compiler has completed the read-in and compiling of a program, along with any changes which might have been incorporated, the translated program will finally be stored in its operating position so that the program may be executed immediately after compilation has been completed without any intermediate steps being necessary.

17. Compatibility with operational procedures. There should be built into the compiler some provision for handling simple operational instructions having to do with the sequencing and button pushing which are necessary to complete a run on the machine. It is not intended that

the inclusion of this point requires that the common compiler shall have built into it automatic operational features--rather the compiler should be planned so that when a particular installation decides that they want to incorporate automatic operational procedures, the compiler will be able to accept these changes without any major modifications.