COMPILING ROUTINES

By

Richard K. Ridgway
Eckert-Mauchly Division of Remington Rand Inc., Philadelphia

Since the advent of automatic computation, programmers have devoted much of their time and energy to looking up, adjusting, and transcribing material previously programmed. This has proved a most inefficient method of program preparation. Within the experience of the programming staff of Eckert-Mauchly, such manipulation and transcription has been a major source of programming errors.

In an attempt to lighten the load on the programmer, and to eliminate such errors, members of the Computational Analysis Laboratory have devised programs called "compilers". A compiler looks up subroutines, adjusts them, and assembles them, as a complete program. The fruitfulness of the compiler method of program preparation is now clearly evident. One immediate result is a considerable saving in time in the preparation of programs for the solution of mathematical problems. At present, compilers are capable of handling scientific problems, and in the near future, they will be available to treat commercial problems.

Data has been collected on the time required for the solution of simple problems. For example, it was required that a table be prepared of the values of the function $y = e^{-x^2}\sin(x/2)$, for the range $|x|<1$, and the tabular interval $\Delta x = 0.01$.

The program for solving this problem was prepared on UNIVAC by a compiler some three months ago. The component operations were timed. The results permit a comparison between the compiler method, and the conventional method of program preparation.

By the conventional method:

1) The programmer analyzed the problem in twenty minutes.
2) Four hundred and eighty minutes were required to prepare and write the program.
3) Checking the program required 240 man-minutes.
4) Forty-five minutes were required to transcribe the program on tape.
5) Twenty minutes were devoted to typing out the tape.
6) Proofreading the tape required forty minutes.
7) Sixteen minutes were needed to correct the tape on UNIVAC.
8) Fifteen minutes were spent checking the program on UNIVAC.
9) Four minutes were required to solve the problem on UNIVAC.

Thus, 740 programmer-minutes, 35 Univac-minutes, and 105 auxiliary-manpower-and-equipment-minutes, were required to program and solve the problem.

For the compiler method:

1) The programmer analyzed the problem and prepared an information sheet. Time required, twenty minutes.
2) The information sheet was transcribed on tape in ten minutes.
3) The information on the tape was printed in five minutes.
4) Five minutes were required for proofreading.
5) UNIVAC compiled the program in one and one half minutes.

(The output of the compiler was a tape to be used as the instruction, or program, tape for solving the problem.)

6) The problem was solved on UNIVAC with a time expenditure of five minutes.

Thus, twenty programmer-minutes, eight and one-half UNIVAC-minutes, and twenty auxiliary-manpower-and-equipment-minutes were required to prepare and solve the problem.

Mathematically, the problem is trivial and the use of UNIVAC for its solution is analogous to killing a fly with a sledge hammer. Considered as illustrative of time factors, however, the problem yields information.

Minutes	Conventional	Compiler	Ratio
Programmer	740	20	37:1
Auxiliary manpower and equipment	105	20	5.3:1
UNIVAC	35	8.5	4.1:1

Thus, while more UNIVAC time may be required for the numerical solution of a problem as programmed by UNIVAC, more UNIVAC time, *in toto*, is consumed by the conventional method. This remains true until the entire problem including its self-contained repetitions is to be repeated, in this case at least eighteen times.

If the total preparation time is considered, the problem must be repeated some 800 times before the conventional programming method overtakes the compiler method. In this case, the compiler used was the "antique", or A-0, the first to be constructed and the most inefficient. Later compilers not only "squeeze" the coding, but also minimize the latency time.

If the staff of a computer installation expects to process a number of different problems, a set of compiling routines, and of service and diagnostic routines, is essential to the efficient and economical operation of the computer.

The first diagram represents the method of problem solution conventionally employed. In phase one, the programmer analyzes the problem, breaking it down into arithmetic steps which are reduced to the instruction code of the computer. The programmer refers to tables and formulas during the analysis, and to the instruction code during the programming. In phase two, the program is fed to the computer, and instructs the computer to operate upon the "input data" to produce the desired results.

If a compiler is used, there are three phases to the problem solution, Fig. 2. In phase one, the programmer analyzes the problem, and breaks it down into steps, each of which can be performed by a subroutine. The programmer writes each step as an item of information with the aid of a catalogue. In phase two, this information is fed to the computer, which,operating under the direction of the compiler, looks up subroutines in the library and assembles them, properly translated, into a program. In phase three, the program is fed to the computer along with the input data, and the program instructs the computer to operate upon the input data to produce the desired results.

The basic element of the compiled program is the subroutine. Fig. 3 shows an operational diagram of a subroutine. The input consists of arguments and the output of results; control may be received from one or more subroutines and transferred to one or more other subroutines.

A subroutine must be coded so that it requires a minimum both of memory space and of computer time. Hence, it should be expressed in minimum latency coding. Ideally, the kernel of a subroutine should not be required to compute on quantities which exceed computer unity in magnitude. If this is the case, the kernel can be used in subroutines designed for floating decimal and other special calculations, as well as in subroutines designed for fixed or stated decimal calculations. Thus, the subroutine must be a neat, efficient, and compact little package of potential computation. The kernel of the subroutine should be even neater, more compact, and more efficient.

Subroutines are stored on tape in the form shown in Fig. 4. Particular attention should be paid to the section at the beginning of the subroutine, labeled "information about the subroutine". The first word of the information is the "call number" identifying the subroutine. The next information locates the position of the arguments in the subroutine. The next words designate the position of the results which are to be transferred out of the subroutine. Finally, transfers of control are specified. These indicate the destinations of the exits from the subroutines.

After the subroutine has been adjusted and sent to the program by A-0, it is in the form shown in Fig. 5. Sections are reserved for instructions which will transfer arguments from working storage into the subroutine and results from the subroutine into working storage. In more efficient compilers, these transfers, together with much of the temporary storage within the subroutine have been eliminated. However, working storage still constitutes a common pool from which quantities are drawn by the subroutines and into which quantities are placed by the subroutines. This form was defined arbitrarily for subroutine descriptions and can be changed provided the necessary elements are included. However, such a change entails a new compiler.

There are certain logical interconnections among subroutines in a complete program. These are control transfers and are not simple to present. The operations in a given problem are assigned numbers in an increasing sequence. If it is desired to transfer from operation #7 to operation #5 (Fig. 5), and subroutine #7 is being processed, then subroutine #5 has already been processed by the compiler. Hence, the entrance of #5 can be found listed in the record the compiler maintains of its entries. From the record information, a transfer of control from #7 to #5 can be generated and placed in #7.

If, however, subroutine #5 is being processed (Fig. 6) and a transfer of control from operation #5 to operation #7 is indicated, A-0, the antique compiler, does not know where the entrace of #7 is going to be since #7 has not yet been processed. Control is transferred from #5 to a temporary storage location. The compiler records the fact that a transfer to #7 is required. When #7 is processed, a transfer is generated and placed in the temporary storage which will transfer control to #7.

Thus it may be seen that the essential information required to make use of compiling routines and a library of subroutines for UNIVAC, or any large scale, self-checking, automatic computer with a large secondary storage must include:

1) definition of the operation, call number of the subroutine;
2) specification of the input, arguments;
3) destination of the output, results; and
4) statement of logical sequence, controls.

In mathematics, a "function" can be defined as "a law which transforms

one set into another set. If the first set be given, the second set is determined". The functional relation between the two sets is not, in general, symmetrical. It is possible to specify a function and to specify a resulting set, and to be unable to derive the first set from this data. If a function is written: $y = F(x)$, three things are specified,

1) a set, x, which is acted upon by the law,

2) y, a set which is the result of action by the law on 1)

3) a law, F ( ).

Controls may also be inserted; i.e., $y = F(x)$, $x^2 < 1$ if $x^2 > 1$, stop.

The information form used by a compiler contains all four elements: arguments, results, a law, and controls. Hence it follows that, a compiler can deal with operations that fall under the definition previously given. Thus, the compiler manipulates functions in symbolic form rather than numerical data. The implications of this line of thought are not completely apprehended at present. It does seem possible to make a compiler that would deal with problems in a multi-valued propositional calculus.

Also noteworthy, is the fact that compilers can apply linear operators to information. Thus, new information can be generated which specifies the symbolic result of operating upon the old information. For example, one compiler has been constructed which will replace the information specifying a function by information specifying the function and its derivative. This can be extended to the application of "infinite series" of linear operators to the information-functions. For example; if the solution is desired of the differential equation, $y'/y = P(x)$, where P is a polynomial, it can be found by this means:

$y'/y = (D/1)y$

So $y = (1/D)^{-1}p(x) =$

$(1-D/D^2-D^3/D^4-.....)P(x)$.

A point which should be mentioned briefly, is that by using the ability of UNIVAC to write large masses of instructions in a short time it is possible to attain an appreciable reduction in the time required for the solution of certain classes of problems. If it is desired to repeatedly sweep a two- or three-dimensional mesh, performing a set of operations at every point, it is conventional practice to write the instructions once, then increase these instructions to refer to the next point, and again act on the same set of instructions. If the operations are at all complicated, a disproportionately large amount of time may be consumed in altering the instructions. Time can be saved by using UNIVAC to generate the instructions for a suitably large portion of the mesh, or the entire mesh, then transferring the data and the generated instructions to and from secondary storage as required. Compilers providing this service are particularly suited to problems involving relaxation methods, auto-correlation, and vector algebra.

In conclusion, if many problems other than a few base-load problems are to be solved, compilers have a distinct advantage over the conventional method of programming. They open, therefore, a large field of effort, barely scratched by the crude compilers just described.

For those entering this field, a reading of Dr. Wilkes' book on programming for an automatic computer is essential. I wish to express deep indebtedness to Dr. Grace Murray Hopper for her ideas on the "education of a computer". I also wish to express my appreciation to the programming and operational staffs of Eckert-Mauchly for the help I have received from them.

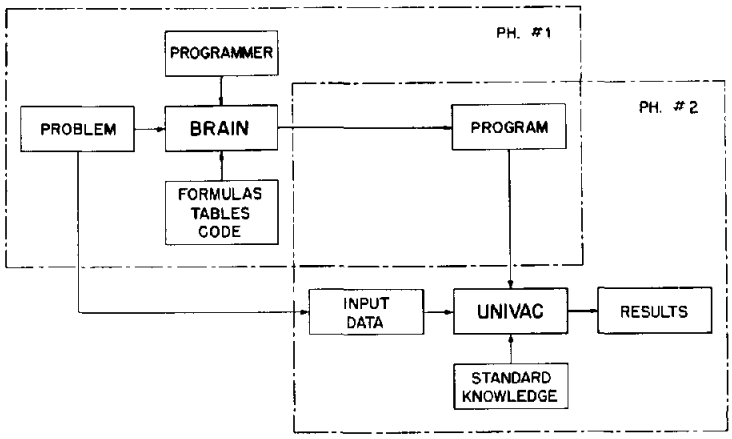

Fig. #1 – CONVENTIONAL METHOD

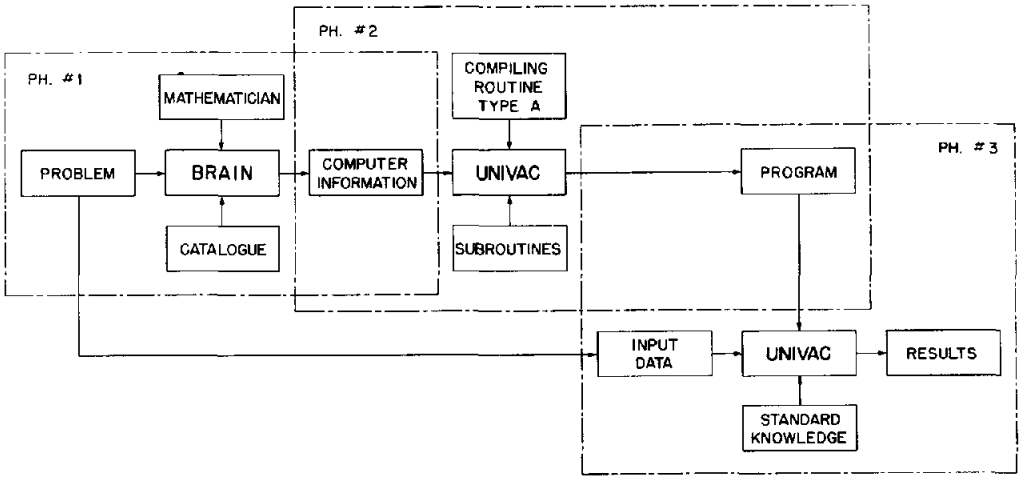

Fig. #2 – COMPILER METHOD

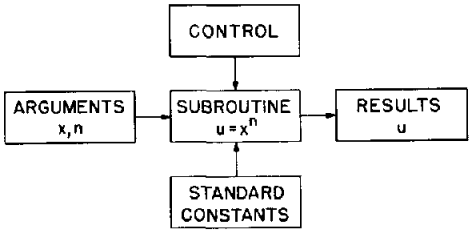

Fig. #3 – SUBROUTINE

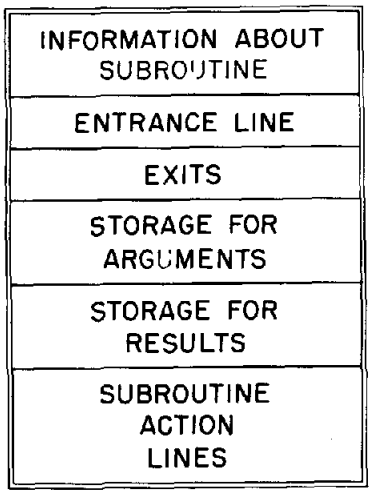

Fig. #4—SUBROUTINE IN STORAGE

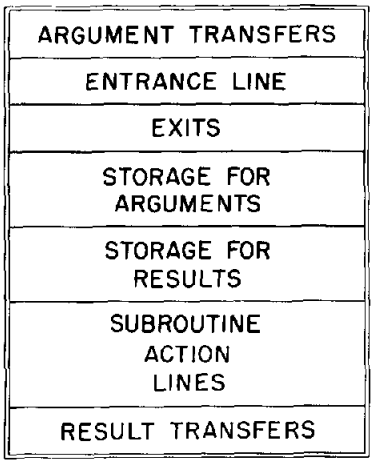

Fig. #5—ADJUSTED SUBROUTINE

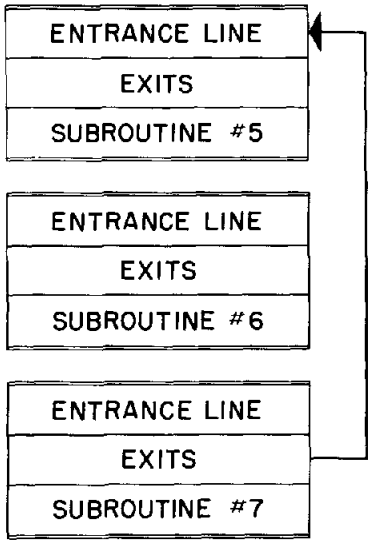

Fig. #6

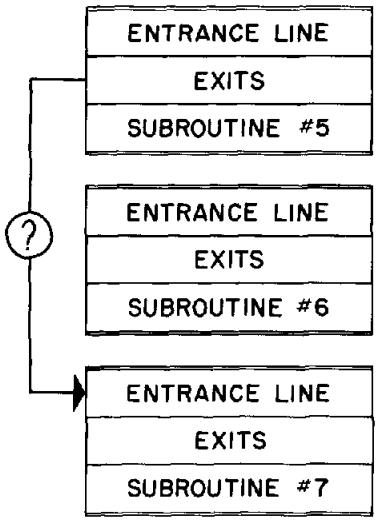

Fig. #7