AUTOMATIC PROGRAMMING AND ITS DEVELOPMENT ON THE MIDAC

J. H. Brown and John W. Carr, III
Willow Run Research Center, University of Michigan
Ypsilanti, Michigan

A new term has developed recently in the automatic digital computer field. "Automatic programming," a concept which ten years ago would have been worthy only of the science-fiction writers, is now within view, if not reach, of the users of high-speed digital computers.

In the decade since the design and construction of the first automatic, high-speed, electronic digital computer, the emphasis has been on increasing speed of the machines; little attention has been paid to increasing automaticity. The efforts of users and designers were on the speed and the electronic portions of these new machines. The fact that these machines could make complicated decisions was generally not used.

The first automatic digital machines were designed for scientific calculations, where the basic outside human language was well-formulated, complicated, and generally understood—the language of mathematics and formal logic. These machines had simple, crude, but less familiar and less understandable interior instruction languages. The gap between such languages was initially crossed only by hard, complicated, routine, clerical human labor. Scientists and mathematicians who preferred more imaginative jobs rebelled at this routine work; later they noticed its similarity to other problems they were solving by means of these machines. When they began to use the computers to do this work, automatic programming was born.

Automatic programming can thus be defined as all those methods which attempt to shift the burden of formulation and programming of problems for automatic computers onto the machines themselves. In its final, unreached stages, it should attempt to maximize some measure of effectiveness for a combination of machines and associated human beings working together. Such a measure of effectiveness would probably approach a minimum of routine effort on the part of humans.

A sequence of concrete steps toward such a goal has now been taken. In many cases, this sequence has come about only as the result of seeking some immediate, short-range goal.

The present discussion is divided into two parts. The first outlines the problem of automatic programming as it has grown over the

past few years, with an attempt to list the ideas and short-range goals which have led investigators along the directions they have taken. The second lists some of the present actual and proposed tools that have been developed and proposed for use with the MIDAC (Michigan Digital Automatic Computer) at the University of Michigan as they fit into the general categories that are discussed in the first part.

Many of the ideas listed here were first proposed by persons in other installations; among those particularly to be mentioned are M. V. Wilkes, D. J. Wheeler, and S. Gill of Cambridge University, C. W. Adams of M.I.T., Grace Hopper of Remington Rand, and N. Rochester of I.B.M. Many others, including numerous members of the Michigan group, have contributed to the general ideas and techniques.

THE GENERAL MOVEMENT TOWARD AUTOMATIC PROGRAMMING

While the everyday important job of solving problems has gone on, users of automatic computing machines have paralleled this operation by taking steps which tended, although not directly, toward the goal of automaticity in the use of digital computers. In the paragraphs that follow some of the most important of these steps will be listed to show how early disconnected progress has brought the use of these machines toward a final, still not exactly defined, goal.

Development of An Understandable Input Language

When automatic digital computers were first constructed, several immediate difficulties confronted the user. The ambiguity of instructions and numbers inside machines—a powerful tool in the induction processes in programming—makes instructions appear only as numerical sequences. In binary machines, therefore, not only numbers but also instructions were unintelligible to those machine users unfamiliar with nondecimal number systems. On the other hand, many "professional" machine users strongly opposed the use of decimal numbers for computer inputs or outputs as being unnecessary. Often they felt that direct binary (or associated octal or hexadecimal) coding of instructions was the only possible method. To this group, the process of machine instruction was one that could not be turned over to the uninitiated.

However, other groups devised programs which translated both numbers and instructions from an external decimal language more useful to human beings, into an internal binary language more useful to the machine. These early programs were often crude and rudimentary More important was the recognition of the principal of a dual language system and a programmed translation between the two languages, using the computer to perform the translation. Along with the numerical translations came the use of "mnemonic"—easy to remember—instruction operations, such as standard algebraic notations, or abbreviations of the corresponding English words. These early schemes developed into the systems now known as "Input Translation Programs."

The general idea of translation was developed in two directions, pre-translation or "compiling," and running-translation or "interpretation." Compiling was developed on those computers which from the first had a large amount of medium-access storage (generally magnetic tape) available; this system was more efficient because translation usually occurred only once. Interpretation, on the other hand, was the only feasible translation method in machines with small amounts of storage; this process is less efficient since the same translation may occur over and over again during performance of a problem, instead of once at the beginning.

As machines get more available storage, compiling techniques will become more and more important. Today, most of the automatic programming schemes of any promise under development (with a few notable exceptions) make use of the pretranslation idea.

Development of Easy-to-Correct and Easy-to-Use Input Languages

The use of external languages that were easily understandable did not, obviously, solve the problem. Despite the use of mnemonic codes, decimal rather than binary addresses, and other such devices, it was found that programming mistakes—arithmetic, logical, or clerical—were not prevented. In the ordinarily sequenced machines, where instructions follow in numerical sequence in memory locations (standard one- or three-address codes), insertion or deletion of instructions which most corrections entail often required renumbering of absolute addresses. The translation process, developed to handle the requirement of the foregoing section, offered a ready-made answer to this: the use of so-called "symbolic" or "floating" addresses, which have no permanent absolute machine internal counterparts. With these addresses, which are almost completely equivalent to an algebraic notation, assignment of addresses can be made completely automatic by the computers themselves.

The use of such automatic address assignment, it turns out, requires two "passes" or traverses through the input information in order to find out, first, what algebraic addresses are present and what their internal equivalent absolute addresses are, and second, to assign these absolute addresses wherever the floating addresses occur. The existence of these two passes-through the input information immediately presented opportunities to perform all sorts of other transformations on the input information.

In order to control the processes of correction, reassignment, and deletion, and to handle the process of translation, a new kind of "instruction" was developed for the computers. So-called "control combination," which told the translation program, rather than the computer hardware, what was to be done gave another dimension of latitude of expression between the programmer and machine. These new "tag words" allowed the instruction process on input to be expanded

indefinitely. This very flexibility provided for easy experimentation that led to newer and even more powerful ideas.

Elimination of Repetitious Coding

From the start it appeared to many machine users to be imperative to eliminate duplication of effort that occurs when similar problems are repeated often. Routines were developed which performed standard operations and which could be called on by any programmer without the necessity for being rewritten from the beginning. To make the most of such routines, complete generality was required. The floating addresses developed for correction purposes helped provide this feature; similar so-called "preset" and "program" parameters were devised to make such routines flexible. Such schemes even prompted built-in equivalents, like the built-in "present-address-relative" coding scheme for the MIDAC.

The combination of these standard subroutines, as they came to be called, with the input translation (compiling) techniques brought forth the concept of "automatic assembly" of programs. Here code words of some sort ("synthetic instructions") are used to call in and store precoded subroutines in a main program. Such "open" subroutines, without automatic entry and exit, worked best with compiling techniques; "closed" subroutines, which act in the same unitized manner as an ordinary instruction, fitted most easily into the interpretive schemes.

Easy Mistake Discovery

Relatively straight-forward methods of correcting mistakes did not speed up program check-out as much as would be expected, since before programming mistakes can be corrected, they must be found. The opponents of translation procedures here put up one of their strongest arguments. Unless complete retranslation of stored information is made available to a programmer, he is forced to know and use both external and internal languages, which nullifies some of the advantage discussed in the first section of this paper. Such retranslation is possible, but adds to the complexity of the translation procedure, as well as causing possible ambiguities because of many-to-one input translations. Nevertheless, such retranslation can contribute to aiding the discovery of human programming mistakes.

Such a procedure can be adapted to two types of mistake diagnosis: static and dynamic. Static procedures give results only at specific points during solution of a problem, usually only at the beginning, end, or both. "Sieved" or "changed-word post mortems," or storage printouts, have proved a welcome use of the principle of machine screening of unnecessary information.

Dynamic mistake diagnosis, on the other hand, has often suffered from the fact that it has of necessity been interpretive, or else that it

has put out information in an unretranslated form. A new concept in dynamic procedures, which give results as the problem is being performed, combines built-in automatic switching with programmed retranslation to speed up this process and make it competitive in time.

Prevention of Mistakes Before They Occur

The use of mistake-location techniques, however, was only a palliative, not a cure. The widespread use of precoded subroutines, which had been checked thoroughly for both arithmetic, logical, and clerical mistakes, offered an example of a preventive technique. Instead of storing complete programs, however, a more efficient method appears to store programs that can generate large classes of programs—"generators," as they have come to be known. If a correct algorithm can be developed by which a computer can generate a general class of small problems, mistake-free codes can be produced directly. Similarly, automatic assembly and automatic subroutine call-in techniques, if correct themselves, will generate mistake-free programs.

An important part of development of a mistake-free coding process is a complete statistical knowledge of the arithmetic, logical, and clerical mistakes which are likely to occur. One such study by a working computer staff is discussed later; such investigations open up further avenues of investigation. An understanding of the possible mistakes aids in a better understanding of the algorithms that are encompassed in the coding procedure itself.

Unification of Techniques

The present knowledge of the techniques mentioned above presents a challenge to the machine user: Combine these into a self-sufficient method of machine operation which will respond properly to the external human stimulus with as little human intervention as possible. Systems using portions of the techniques listed in the preceding sections are today under development in numerous laboratories. It is apparent, however, that certain of the categories are directly opposed; a programming system which generates correct programs, for example, does not need to encompass easy mistake diagnosis or mistake correction procedures. If the computer is to generate large blocks of its own coding, the need for an understandable external counterpart to its individual internal language is no longer so important. If automatic machine program generation is to be available, automatic round-off and truncation error analysis may be required, or else a complete step-by-step copy of the operations involved must indeed be returned to the programmer.

Despite these obvious complications and difficulties, it still appears that given an automatic computer that is large enough and fast enough, the limits on its abilities are few indeed. A "generating" process is still a long way from a "learning" process, but the seeds of creativity—at least in a mechanized sense—are there. What then, in the present

state of the art of automatic programming, should be the goal at which machine users should aim?

Certainly, the side-roads of easy mistake diagnosis or mistake correction are off the main path. It is exactly the human being's imagination, creativity, and dislike for rote that produce human mistakes in the process of machine instruction. Automatic commuters, which are made for the routine, do not make mistakes—at least with comparable probability. The generated program, standardized and somewhat less efficient, perhaps, but certainly mistake-free, is the goal for which to aim. Such programs, even if they take considerable computer time to be produced, would prove more efficient in the end. They must, of course, produce equally satisfactory error analyses and equally satisfactory output formats if they are to compare with step-by-step human efforts. Certainly, algorithms should be able to be developed that will provide as much minimization of storage, or time, or both as is desired under a given over-all measure of effectiveness. Most machine users know intuitively "how to program"; now must come the stage where this intuition is formalized and transferred into the heart of the machine itself. What are the steps by which a code is developed?

Such investigations would appear to lead into the regions of metamathematics, where the problems deal with the generation of systems rather than the systems themselves. Today, very little theoretical treatment has been made of these, at present, empirical problems. Future work may depend very strongly on an understanding of just these problems arising in the "generation" of algorithms by other algorithms.

Development of a Universal Computer Language

The progress through the steps listed above leads, of necessity, towards some sort of standardization of the basic input language of all computers as it looks to the users, before it is fed through the "black box" which contains the integrated system which matches the internal language to the external human language. Certainly, present-day lack of compatibility between algorithms or programs developed on one computer and another is causing as much undue duplication of effort in space, as occurred in time before the advent of subroutines. Universal languages, more or less standardized, exist for certain types of problems, those that can be expressed entirely algebraically, for example. Combinatorial and logical languages, on the other hand, are not as familiar and standard, and are often not sufficient to formulate a problem. Present computer inputs, limited to certain standard input characters, are not able to assimilate the more unusual symbols needed by these external languages. No one is quite certain whether a program formulated for a system with internal binary structure can be efficiently handled by a second system with mainly internal decimal arithmetic. Nevertheless, as an alternative to the commercial capture of the

computer and data processing field by one make of machine, or arbitrary ruling on machine specifications by government fiat, one now has the interesting possibility of a common, universal, external language arrived at by mutual agreement and persuasion, which can be matched to the internal structure of numerous computers by "black boxes" which translate and generate the required computer internal programs.

APPLICATIONS ON THE MIDAC

At the University of Michigan, the available digital computer is made for experimentation of the type just discussed. The computer is a completely packaged device with an internal "building block" structure that allows internal logical changes to be made almost as fast as engineers and logical designers can decide on them. Such an opportunity is, of course, a dangerous one for users of a machine, unless some sort of buffer can be placed between the written programs and the internal structure, which is so open to change. An over-all system of computation, called MAGIC (Michigan Automatic General Integrated Computation), provides just such a buffer. Present plans provide, as internal machine changes are made, that this programmed system itself will be changed to match the different internal logic, rather than necessitating a complete change of all programs each time a change in logical design, number structure, or instruction components is made. Such a system, if handled wisely, allows purposeful experimentation with the hardware, and at the same time provides continuous productive use of the computer as a whole.

The automatic programming procedures that have grown up so far on the MIDAC have been developed for the various, sometimes not connected, reasons enumerated in the first half of this paper. It was only after the concept of an integrated system was arrived at that attempts were made to mesh them into a unified whole. Even today, although portions of the system have been written to be used one with the other, they have not all necessarily been checked out because of unavailability of magnetic-drum equipment and of magnetic-tape equipment which is under test and design.

Development of An Understandable Input Language

The interim MIDAC Input Translation Program (ITP) has control of the computer during the input process so that the automatic translation of a program is accomplished. The MIDAC input language has a direct relationship to the internal binary operation. The computer may take in either eleven-digit, binary-coded hexadecimal words with sign attached, or single, six-bit alphanumerical characters.

The outside language is an arbitrary language designed with two purposes in mind: To make the routine clerical work of programming easier, and to provide an easy method for changing a program. Since this language was to be used on an interim system, restricted in its

capabilities, the physical characteristics of the computer strongly influenced its form. Although the interim ITP may not be as flexible as desired, it is felt that its purpose has been accomplished satisfactorily.

The outstanding features of the interim ITP are:

1. Mnemonic symbols are used for the 19 MIDAC operations.
2. Floating addresses may be used.
3. Numbers may be entered in their decimal form.

The instruction word in the outside language follows closely the structure of the instruction word in the MIDAC input language. The MIDAC has a three-address operation code; therefore, the input language instruction word contains the three addresses involved and the type of operation. The outside language instruction word presents this information in the form of four separate sections, or "components," in the same order as for the input language. From this is derived the phrase, "component-coding," in outside language. The first three components of an instruction contain the three addresses: α, β, and γ. The addresses of the two operands are α and β, and γ is the address of the result. Any of these three addresses may be written as decimal machine addresses, or floating addresses may be used; also, the two types of addresses may be intermixed if desired.

The fourth component is a two-letter symbol representing the operation in mnemonic form. Thus, for example, the operation "add" is written as "ad"; "su" for "subtract," and "dv" for "divide" are other examples. The function of the ITP, as regards instruction words, is to transform the four components into their proper binary equivalent, and to collect them together into one internal MIDAC word.

Number words are written in the form of three components for the outside language. The first component contains the decimal part of the number, the second component contains the power-of-ten exponent, and the third component contains the power-of-two exponent. The power-of-two exponent is generally used only for scaling purposes since the MIDAC operates internally in the binary system. It is the function of the ITP to combine these three components into the proper internal binary equivalent. In this case there are two possibilities for number forms in both the internal and external languages. Floating-point numbers are represented externally by a decimal part which is converted into a standardized "digital number" of magnitude less than one, along with a power-of-two exponent, representing the sum of any original power of two scale factor and the internal power of two required to standardize the number. This floating-point number, when converted, is represented by 44 binary digits. The first 36 binary digits and the sign represent the converted decimal part; the last 8 binary digits represent the power-of-two exponent, including the sign of the exponent.

A number convention fixing the radix point at the extreme left of a MIDAC word was built into the machine, therefore, when fixed-point arithmetic is used the numbers must be less than one in magnitude, either in actual size or by adjusting with power-of-two exponents. Numbers in this form are called "decimal fraction" numbers which are the second of the two number forms. Decimal fraction numbers are converted by the ITP into their 44 binary digit equivalent.

Double precision conversion methods are used within the ITP so that the converted number will be exact to 44 binary digits. Certain limitations must be imposed on the number being converted. Since only double precision conversion methods are used, the decimal part of the number can contain at most eleven decimal digits, and the power-of-ten exponent can range only between the limits -11 and +11.

Development of Easy-to-Correct and Easy-to-Use Input Languages

An early development in the ITP was the use of a number of "control combinations." Control combinations are words which are interpreted by the ITP but are not translated. Certain control combinations are used to designate what type of a word is to be translated: a component-coded instruction word, a decimal fraction or floating point number word, or a pure hexadecimal word. The latter type of word is already in computer input form, and therefore its translation consists of a direct read-in. The ITP is arranged so that the control combination indicating what type of word is to be translated only need precede the first word of a group of the same type. Thus, only on a change of word-type is it necessary to include a new control combination. Other control combinations indicate where the translated words are to appear in the memory and where computation is to start.

In order to present a useful format to the programmer, all control combinations appear in the first column of a coding page; the components of a number or instruction word then appear in succeeding columns of the page. In preparing the tape from the code, the typist follows the same format so that the programmer gets back a neat copy of his code.

One of the most useful of the control combinations is that which, in the ITP, designates a "floating address." A floating address, as mentioned before, is a symbolic address by which the programmer locates a word of a program. A floating address remains with a particular word, even though that word is moved around within a program. This is in contrast to the "fixed," or "machine," addresses of the computer memory in which a program is placed. In this case, as a word is moved around its machine address will change.

In order to assign machine addresses to the floating addresses, the ITP contains within itself a "directory" in which all the floating addresses are listed in a known sequence. When a machine address is

assigned to a floating address, the machine address is stored in the directory in the position corresponding to that floating address. Then, when a floating address is referred to in the body of the program, the ITP locates the floating-address position in the directory, extracts the machine address stored therein, and replaces the floating address with that particular machine address.

The machine addresses are assigned to the floating addresses by making one "pass" through the untranslated program. On this first pass, the ITP reads in the program, only assigning machine addresses. Then a second pass is required to perform the actual translation. At the present time, due to the lack of the necessary storage, both passes are made from external read-in via the Ferranti photoelectric readers. During the first pass, the directory is printed out as the floating addresses are assigned. This gives the programmer information useful during check-out tests. The translated program is punched out on paper tape in a standard form during the second pass; this tape is then ready to be read directly into the computer whenever computation is desired.

Provision has been made for 200 floating addresses to be used. This is an adequate number for most programs. No error check has as yet been provided to prevent the use of floating addresses which have not been assigned machine addresses.

As soon as the magnetic-drum storage is available the immediate extension of the translation program will be to provide a "translate and compute" ITP. Instead of translating and punching out the translation on tape, the translated copy would be stored on the drum. After the translation has been completed, computation could proceed immediately. During the first pass of the translation, the untranslated program would be stored on the drum, and the second pass would be made from this copy.

Elimination of Repetitious Coding

The MIDAC users benefited by the efforts other groups had made toward satisfactory subroutine libraries. The MIDAC library of subroutines consists of those routines which are most commonly used within larger programs. These routines evaluate trigonometric, transcendental, and algebraic functions such as sine-cosine, exponential, and square root functions. Other useful routines such as output routines are also available.

One built-in feature of the MIDAC is the possibility of making addresses of operands in the instruction words relative to the instruction counter. If this is done, the true machine address of the operand is the sum of the instruction counter and the "relative" address of the operand occurring in the instruction. The MIDAC subroutines are coded in this "present-address-relative" form; this permits the subroutine to

be oriented in any desired part of the computer memory without the necessity of re-coding for each new orientation.

Subroutines are written in a standard form so that only the location of the exit need be known, this exit always being the first word of the subroutine (the entrance is always the second word). The arguments of a subroutine are stored in a fixed location of the memory, and the answer appears in a standard location; therefore, the programmer does not need to refer to any register contained within the subroutine.

In using a subroutine the programmer must accomplish three things in his main program: The arguments of the subroutine must be transferred to the standard location; control must be transferred to the subroutine, and after the subroutine is used, the programmer must use the answer. In general this requires two to three separate instructions for each use of the subroutine.

At present the programmer indicates in his program where he wishes the subroutine to be located. When the program is prepared on paper tape, the typist reproduces, in the proper place, the appropriate subroutine from the subroutine library. Thus the subroutine becomes a permanent part of this program.

When the magnetic drum storage becomes available for the MIDAC, many subroutines will be stored on the drum. "Synthetic" operations, furnishing information to the translation program, will then be used to call in the required subroutine and automatically perform the transfers of arguments and answers.

Depending upon storage considerations, the subroutines used in this process will be either open or closed. As the MIDAC magnetic tape units become available to take over from the drum, more open subroutines, "compiled" and filed temporarily on the tape units, will be used instead of the closed subroutines which now require call-in from the drum.

Easy Mistake Discovery

Although the MIDAC had originally rather crude built-in facilities for the determination of coding mistakes (such as, for example, built-in switches for printing out key machine contents) these were early found to be unsatisfactory because they were not automatic, and because they were not compatible with other parts of the system. As a result, a dynamic procedure, the "automonitor," and a static procedure, the "changed-word post mortem" were developed.

The automonitor program is a dynamic program in that it is used while computation is in progress. This program provides a method for tracing the path of computation and it also prints out the results of the operation. Two modes of automonitoring are possible: An instruction-by-

instruction automonitor and a "breakpoint" automonitor. If the first mode is used, the automonitor program performs the instruction, and prints out the contents of the instruction and base counters,* the instruction just performed, and the results of the instruction; then it proceeds to the next instruction and repeats the operations as above. This continues until halted.

The second mode, or "breakpoint" automonitor, provides the same automonitor results as above, but now for only those instructions preselected by the addition of "breakpoints" to the instruction. For the MIDAC a breakpoint is merely a negative sign on an instruction.

The automonitor program is a very useful device for removing the "bugs" which always occur in a new program. The programmer can use the breakpoint automonitor to obtain a quick, over-all picture of the path of computation. Then, if sections need be examined in more detail, the instruction-by-instruction automonitor is used on that section, showing exactly what occurs.

The changed-word post morten routine is a static program since it is used after the computation is "dead." This routine provides a method for obtaining those words of a program which have changed during the process of computation, thus obviating the necessity for "dumping" the dead program and comparing by hand with the original. The MIDAC changed-word post morten program prints out the instruction counter, the original word, and the changed word. For the interim system this process is accomplished by comparing the words on the original input tape with the present contents of the memory; if the two agree no print-out occurs.

The changed-word post morten routine has proven very useful in aiding in the location of program errors. It has become standard practice for the computer operator to perform a changed-word post mortem on any program in which trouble occurs during the computation process.

At present during a program check-out the print-outs of the various checking routines (such as the automonitor and post-morten routines) are in the machine language form. Thus, in checking with the original program, a certain amount of hand translation must be done. This is not as difficult as it may sound; the programmer has the directory print-out and a copy of the translated program. The programmer can make the necessary hand translations quickly with this information and can check with the original to see if it is correct.

Prevention of Mistakes Before They Occur

At this step in the sequence toward automatic programming, development of the MIDAC system is temporarily halted. At present, each

*The base counter is used primarily in cyclic operations.

program that is written and checked out on the machine has complete records of the steps through the process. With each correction made, a mistake analysis of the reasons for correction is handed in, to be assembled into a thorough survey of why MIDAC users make errors. Upon the accumulation of enough data, the present and future MAGIC system will be examined with the objective of providing built-in correctives in two forms: (1) The changing of the input language so that mistakes of a particular kind are impossible, (2) The providing of built-in checks in the ITP portion of the MAGIC system so that mistakes which cannot be prevented will at least be "captured."

Such a survey is only part of a beginning study of just what constitutes the coding process as now practiced by human beings. Can the present coding procedures be mechanized and put in the form of algorithms so that their operation can be turned over to the machine? Just what sequence is followed in the assignment of locations, use of temporaries, use of comparisons, etc.? If the logic and arithmetic aspects of programming can be formalized, much of the mystery of the process will be removed, and machine users can return to the job of mathematics and numerical analysis.

Unification of Techniques

The MAGIC I system, as presently planned, provides a unification of all the present techniques. Beyond it are envisaged other numbers in the series, each one moving the automatic programming process a little nearer for the users of the MIDAC.

The completed MAGIC program will consist of a "master program" and the associated working programs. The master program will determine what working program is to be used. For example, suppose the program being run "dies" of an error, and a post mortem is required. In this case the operator would clear the counters and start the computer. The master program would take over and ask the operator, via a call for input, what is desired. The operator would type in from the supervisory printer a key word asking for the post mortem. The master routine would then call the desired post-morten routine into operation, and it would be performed.

Transfer of control to the master routine can be done automatically. During the running of a program the computer can be made to sense special digits (such as breakpoints) within a word. When such digits have the value "1," control is automatically transferred to the master program. At the same time a key word is set up within the master program depending upon the digit sensed. In this manner the programmer can code automatic call-in of the various checking and translation programs of the MAGIC system.

It is expected that the input translation program of the future will be further changed to permit easier coding. With this future ITP,

instructions will be written more nearly in sentence form, and "synthetic" operations will be used. These synthetic operations will permit the use of one external operation, which must be broken down into several internal MIDAC operations. By the use of such synthetic operations the programming of a problem will be more nearly automatic.

In conjunction with future input translation programs there will be an output translation program which will cause the print-outs from the various error-checking programs to have the same form as the untranslated original. Thus, the programmer will not need to make any hand translations during the program check-out procedure.

Beyond these planned techniques lie (1) the generating and compiling programs which will make use of the libraries of previously developed programs as sources of inspiration, (2) the development of new types of typewriter inputs that will allow algebraic and logical languages to be fed directly into the machine, (3) investigations of new languages that will express a wider range of possible instructions to machines, and (4) studies in the theory of programs which generate lower order programs.

Automatic programming is no longer the speculation of an imaginative mind, it can be accomplished. With the development of the first components of the MAGIC system, the MIDAC has taken the first steps toward the realization of complete automatic programming.