Arnold: A Compiler Capable of Learning

INSTRUCTIONS

C A PROGRAM TO MULTIPLY TWO MATRICES AND SUBSTITUTE PLUS ZERO FOR
C EACH ZERO ELEMENT, PLUS ONE FOR EACH POSITIVE ELEMENT, AND MINUS
C ONE FOR EACH NEGATIVE ELEMENT.

10 READ 200 ((M(I, J), I=1, 3), J=I, 4), ((N(J, K), J=I, 4), K=I, 5)
20 DO 140 I=1, 3
30 DO 130 K=1, 5
40 L(I, K)=0
50 DO 60 J=1, 4
60 L(I, K)=L(I, K) * M(I, J) * N(J, K)
   (END OF DO AT 50)
70 IF (L(I, K) < 0, 12, 80
80 L(I, K)=+1
90 GO TO 130
100 L(I, K)=0
110 GO TO 130
120 L(I, K)=-1
130 CONTINUE
   (END OF DO AT 30)
140 CONTINUE
   (END OF DO AT 20)
150 PRINT 200 (L(I, K), I=1, 3), K=1, 5)
160 STOP

A Compiler Capable of Learning

RICHARD F. ARNOLD†

INTRODUCTION

WE WOULD like to consider a new approach to the general problem of programming computers. To date, the methods of handling programming problems can be roughly classified into two families, each of which have certain characteristic advantages and disadvantages which seem to complement those of the other.

The first group, developed from the subroutine philosophy, includes all interpretive schemes, as for example the "Bell Labs Interpretive System" for the IBM 650. The advantages of interpretive routines are that they are very versatile in the languages they can interpret and are comparatively easy to write. It is a fairly simple matter to write an interpretive routine to simulate another computer and thus achieve program compatibility between different machines. The crippling drawback is the excessive time needed to execute routines interpretively. Higher order interpretive schemes increase execution time exponentially.

The second group consists of compilers and assembly programs. They are characterized by the fact that, unlike interpretive routines, they produce object programs which may be executed in reasonable amounts of time. Compilers, however, are difficult to write. "Fortran," for example, took twenty-five man years to write. A second difficulty of compilers such as "Fortran," is that although they are becoming more and more versatile, they still fail to express certain types of operations, and it has become necessary to make it possible to adapt the compiler so that the "Fortran" language may be temporarily left and programming done in a language closer to the initial machine language. Of course, this is a desirable feature for a compiler to have, but it does not solve the initial problem for which it was created namely, to avoid machine languages completely. A further disadvantage is that as a compiler system becomes adapted for use on more than one computer, many of the "coding tricks" will have to be avoided. This may be desirable from the point of view of the compiler writer, but

† Michigan State University, East Lansing, Mich. This research was supported in part by a grant from the Natl. Sci. Found. and taken from a thesis written under the direction of G. P. Weeg.
optimum programs will not be written. This will be particularly true as newer generations of computers are built which may have less similarity to present ones than we expect.

Also, as new computers come into existence, it is likely that it will become increasingly difficult for even the trained programmer to use the machines efficiently. Only time can tell whether or not compilers and their improved offspring are the answer for the future.

However, we would like to offer a different approach to the problem which, although it will probably not prove to be a better solution for the present, may indeed offer a much better line of attack for the future. Let us review first what is desired. We would like a scheme that would accept a program, either in the pseudo code of a compiler or in a different machine language, and would produce an object program for a given machine. We would like this program to take full advantage of all the special features of the machine and to be coded in a fashion such that running time and program length are at a minimum. We would also like to avoid too much work in adapting this scheme to a new machine. Since we may not know ourselves just how best to program a new computer, we would like it if the scheme itself could find the optimum procedures and apply them. Finally—and this is the one requirement which will be most difficult to meet—the scheme must produce a program in a reasonable length of time and not require a tremendous amount of memory space.

With the exception of the last requirement, we believe that the following scheme is capable of satisfying all these requirements, and that perhaps this one too may be met. The techniques used have almost all been suggested in other contexts.\(^1\) Freidberg's programs\(^8\) also have some organizational resemblances to the compiler described.

### A Compiler Using a Random Program Generator

#### The Compiler

Let us consider a new kind of compiler. It is similar in function to other compilers in that it accepts a program in language A and produces an equivalent program in a language B. Since the requirement of our languages is that each order must specify exactly an operation, these languages define computers. Therefore, we may sometimes speak of computers A and B. The compiler operates as follows. It first generates a program in language B at random, as will be described below. Although the program must be of specified length, it may be any conceivable combination of orders. It then proceeds to determine whether this candidate program is equivalent to the given program in language A. The method of determining the acceptability of the program involves the use of two interpretive routines, A and B, which are capable of executing the orders in the two languages. The subject program and candidate program are then both executed using identical data to ascertain whether the candidate program B is capable of producing identically the same output. If it is determined that they are equivalent, then this program will be punched out and the translation will be complete. If not, a new candidate program is produced and tested in a similar fashion. The process is repeated until an acceptable program is produced.

#### The Random Program Generator

The random program generator will have the following characteristics. Provided with the length desired, the first order of the program will be chosen at random from all possible orders that might appear there, so that the probabilities associated with the choosing of all orders are identical. The second order will be chosen in the same manner, and then each successive order in the program, until it is complete. This becomes the candidate program. This method could generate any conceivable program of a given number of orders. However, the probability that a particular program is generated is exactly the same for all. The program generator will utilize a random-number generating routine, and the range of the random numbers will be partitioned in such a way that by assigning equal intervals to each order and selecting that order which corresponds to the interval containing the random number, a program may be generated in the desired fashion. It is recognized that the computer can generate only "pseudo" random numbers and that this will introduce difficulties. It is not too much to expect that the numbers be distributed rectangularly. Of more importance, however, is the sequence that the generation of these numbers may take. It will be necessary that the random numbers not appear in a sequence so that the corresponding programs do not contain certain sequences of orders. We must continue to watch for this possibility even if the compiler works, since it may be producing only certain kinds of programs.

#### The Acceptability Criterion

**The Interpretive Routine Simulation of Computers A and B:** The simulation of the A computer will be straightforward. All one really needs to know is what the output is for a given input. However, we shall provide the interpretive routine that executes the A program with one additional feature, an "address stop." That is, the programmer, or in this case perhaps some other part of the compiler, will be able to specify, before transferring control to the A interpreter, a particular location which, if appearing in the control counter will cause the routine to jump to some previously designated location outside the interpretive routine itself. Any order which would otherwise cause the computer to stop would be treated in a similar fashion.

---

The B computer will be simulated in a like manner with the address stop feature. The B interpreter will also be equipped with a "clock" that will keep track of the running time of the program it is executing as if it were being executed directly by a real computer B. Thus, it will be possible to discriminate between two acceptable programs which have different running times.

Also, by placing a limit on the length of time a program may run on interpreter B, it will have a means of discovering and stopping endless looping. The "clock" would be checked before each order was executed to see if the maximum time permitted had been exceeded.

Our final specification will be that these interpretive routines be in some standard form as it is hoped that to adapt the compiler to any other two languages, it will only be necessary to put in the interpretive routines for those languages and to provide the random program generator with a list of the orders comprising the object language.

The Use of Randomly Generated Data in Acceptability Tests: A first definition of equivalent programs only required identical outputs given identical inputs. It would be possible of course to require that for two programs to be considered equivalent, they must use the same algorithm. However, given a program, it is not possible to recover the algorithm used, and the only alternatives are either that the routine itself be treated as the algorithm or that we consider only results. Acceptance is therefore determined by a statistical criterion and the probability of correct answers on a production run may be made arbitrarily close to 1 by increasing the number of randomly selected sets of input data for which the candidate program must successfully compute the correct answers. It will, however, only be necessary to use one set of data until a candidate program succeeds in producing the correct output for that one. It will also be necessary that the range of the data variables be specified along with the subject routine.

The procedure will be to examine the output and compare it with that of computer A at the completion of the running of a program. If the output is different, the candidate program will be rejected as it would also be if a specified time on the "clock" were exceeded or if the computer "hung up." If the output is found to be identical to that of computer A, then a new set of data would be generated and run through computer A with the subject program to determine the correct output; and computer B would be given the same data and the candidate program executed again. When an acceptable program is found, it may either be punched out on cards or tape or else retained and a search made for a better program in terms of number of words and running time.

We would be fortunate indeed if the compiler, as it has been described, could produce a program of moderate size in the life of the machine, much less within the hour or so maximum that might be allowable. Nevertheless, aside from this difficulty, this compiler would do everything else that was desired. If the compiler were allowed to run long enough so that it could choose among the best of many acceptable programs, it would not only tailor the program to the machine involved but also in fact find many new "coding tricks" that a programmer might never stumble upon. How then can the expected time involved in a translation be cut down? Two methods will be discussed. The possibility of breaking the subject program into parts and translating one part at a time will first be considered. Then we shall consider how the program generating part of the compiler may be replaced by a unit which will gradually modify the probabilities associated with the generation of programs in a manner such that they will produce acceptable programs more frequently.

Sectioning the Subject Program

Criteria for Sectioning: If it is desired to break the subject program into sections, it will be necessary that for a given section there be a criterion for the acceptability of that section. Also, it would be required that this criterion be such that if all the sections were correct, the entire program would also be correct. If a method of sectioning can be found and these requirements met, the compiler could then handle one section at a time in the same way it previously handled the whole program, and such a procedure might be much quicker.

The acceptance or rejection of the program described in the previous section was based on the fact that both computers had input and output devices and, furthermore, that these input and output devices were enough alike in the form in which they handled the data to make comparison easy. However, when considering a section of a program, we can no longer define correctness in these terms because it is unlikely that input and output operations even occur in that section. The input and output devices were treated as equivalent parts of the computer. However, other parts may also be treated as equivalent, and this will make possible the development of adequate criteria for the testing of sections. Since it will be the procedure to test sections by executing them, it will be necessary to specify a set of states, one of which computer B should be in at the termination of the execution of a section. Since this set must be determined by the state of computer A at the same stage, there must be a one-to-many mapping of the states of computer A onto B. This is obtained by defining equivalent parts. One natural equivalence is that the contents of the two memories should be in some sense analogous. It might be said that the state of A is equivalent to the state of B if their memories contain identical information. If it were required that every part of computer A have an equivalent part in B, then the normal operations of B would have to be abandoned and B made to mimic A's every move. This indeed would be undesirable since, for example, if it were required that B be in an equivalent state to A after every order, then even if it were a decimal machine, B would have to find a way in
which it could do operations such as forming logical products, which are extremely awkward in any system other than a binary one. Therefore, equivalent parts will only be defined when the equivalence is natural. Initially, this will mean that the memories, control counters, and input and output devices must be made to correspond. If only equivalent parts may be examined, it is necessary that only those parts of the computer may contain information relevant to the program. This then requires that our sections be constructed so that if the state of all other parts of computer A were altered at the time when tests for equivalent content were being made, there would be no interference with the correct operation of the program. This requirement will be used in sectioning the subject program. The subject program will be "cut" between those orders where the state of all nonequivalent parts of computer A may be altered without affecting the running of the program. The procedure will be that all possible "cuts" be made, the states of those nonequivalent parts altered on the basis of a random number, and the program continued. Those cases where no errors are introduced will then be recorded and the program divided at those points.

The comparison of the inputs and outputs, if any, and the control counters is fairly straightforward. The addresses specified in the control counter would have to be compared by ascertaining whether or not the sections referred to by the two addresses are equivalent.

The comparison of the two memories is more difficult. Regarding memory itself, it is clear that locations which have in them numbers that are computed functions of the data should be compared for equality. Orders which have been modified by a section must somehow be examined to see if they will perform correctly in their modified form. Also, the contents of other locations may have information which depends for its form on the order code of the particular computer, for example, "binary switches" and "dummy" orders. It is necessary then to establish the correctness of such locations by linking them with other sections. Thus we can conclude that if these sections are accepted and executed throughout the entire program, then the location in question must be acceptable. A similar procedure is used for sections whose function is to modify instructions. For each section this entails making a list of sections whose acceptance must come prior to the acceptance of that section. In some cases it may be necessary to work backwards from sections which only operate on data and may be checked immediately through several different sections, the acceptance of each one being a necessary requisite for the acceptance of a prior one.

Admittedly, the above discussion is not complete and there may be situations arising which we have not considered. Nevertheless, it is felt that the method itself is powerful and can probably be adapted to new difficulties as they arise.

**Expected Running Times:** The purpose of sectioning the program is, of course, that it is desirable not to throw out a program completely when only a part of it is incorrect. However, though it might seem obvious that the expected running time would be reduced, this does not follow just because the program is being compiled in sections, as will be seen.

Consider first the probability that an entire program generated by the random program generator is exactly a given program. If the program contains \( l \) orders, each of which might be any of \( k \) different orders, this probability will be \( 1/k^l \). The expected number of programs until it is constructed will be \( k^l \) trials. Now, if the program is sectioned into \( g \) parts of lengths \( m_1, m_2, \ldots, m_g \), then the probability of a particular candidate section being a particular one is \( 1/k^{m_i} \), the expected number of trials is \( k^{m_i} \), and the expected number of trials for the entire program is

\[
\sum_{i=1}^{g} k^{m_i}.
\]

However, this is not the situation. In fact, there are many acceptable programs. If there are \( c \) acceptable programs of length \( l \), then the expected number of trials to hit one of them will be \( k^l/c \), and if the program is divided into \( g \) sections with lengths \( m_1, m_2, \ldots, m_g \) and there are \( c_i \) acceptable ways of writing the \( i \)th section, then the expected number of trials to translate the entire program will be

\[
\sum_{i=1}^{g} k^{m_i}/c_i.
\]

If

\[
\prod_{i=1}^{g} c_i = c,
\]

then it could be concluded that the expected number of trials for the procedure using sectioning is much less than the other one provided the minimum expected number of runs for any section is greater than \( 1/g^{c-1} \). For \( g > 1 \), \( 1 < (1/g^{2c-1}) \leq 2 \) and the expected number of runs for any section will never become that low. However, by requiring the program to achieve certain criteria at many points during its execution, many programs that would satisfy the criteria of identical outputs will be rejected. Therefore, in general

\[
\prod_{i=1}^{g} c < c.
\]

There is no *a priori* method then of saying conclusively that sectioning will lower the expected number of runs, although it seems that it would except in unusual circumstances. However, it might at some point be possible to pool some of the sections and reduce expected translation time, while at the same time increasing the probability of producing a better program because of the fewer restrictions.
The Adaptation of Learning Models to the Compiler

The Alteration of the Random Program Generator to Permit Learning

So far, the random program generator has selected each of the \( k \) possible alternatives at each step with equal probabilities. It is entirely possible to allow the orders to be selected with other probabilities. Given a set \( \{P\} \) of probability vectors \( P = (p_1, p_2, \ldots, p_k) \) such that

\[
\sum_{i=1}^{k} p_i = 1
\]

where \( p_i \) is to be associated with order \( O_i \), the selection of an order may be made in the following manner. Given a pseudo random number \( R \) such that \( 0 \leq R \leq 1 \), and a vector \( P \), then \( O_j \) will be selected when

\[
\sum_{i=1}^{j-1} p_i \leq R < \sum_{i=1}^{j} p_i.
\]

If \( R \) has a rectangular distribution as it should have, then \( O_j \) will be selected with probability \( p_j \). A specific selection of an order will constitute a response, and the set of probability numbers \( (p_1, p_2, \ldots, p_k) \) will be abbreviated \( P \).

Intuitively, it is felt that some \( P \)'s will be more satisfactory than others, and a method must be found to arrive at some “best” one. First, some measure of performance is needed. The measure might include not only how often a particular \( P \) produces an acceptable program, but also whether the program is concise and has a short running time. However, it will be more convenient at present only to consider a \( U(P) = Pr \). (A candidate program using \( P \) is acceptable.) It would then be desirable to find the maximum of this function where \( P \) ranges over the set \( \{P\} \) and use the corresponding \( P \). At present there is no information about this function. A guess might be made that it is continuous but has several modes. It will be assumed, however, that it has but one, the hope being that if any of them are discovered, a fairly satisfactory selection of orders will ensue.

One method that might be used would be to modify the \( p_i \)'s whenever an acceptable program is achieved, or if sections are being dealt with, an acceptable section. Given that a particular order \( O_i \) were in the acceptable section, the current set of \( p_i \)'s would be replaced by

\[
\left[ \left( p_1 - \frac{p_1}{d} \right), \left( p_2 - \frac{p_2}{d} \right), \ldots, \left( p_j + \frac{(1 - p_j)}{d} \right), \ldots, \left( p_k - \frac{p_k}{d} \right) \right]
\]

where \( d \) is a parameter that would reflect the magnitude of the desired change. This procedure would be repeated for each order in the acceptable section. The expected value of \( P \) can be shown to approach in expectation the value at which \( U(P) \) has its mode. It is desirable to let \( d \) be a function of the number of modifications already made, so that as a modal value is approached, the variance about it will be decreased.

This procedure might be called learning because it permits an increment in performance to occur as a result of the “experience” of the compiler.

The Stimulus Situation

Consider now the situation of the response selector at a given point in its selection of orders for a candidate section. Previously, the \( P \)'s used for the selection of each order in the program have been identical. However, if \( P_y \) is the set of probability vectors that might be associated with the selection of a particular response \( y \), it would seem that max \( U(P_y) \) will occur at different values of \( P \) for different values of \( y \). In order to devise a method for obtaining these different values of \( P \), there must first be a method of classifying the \( y \)'s. It may seem at first that \( y \) could be classified according to its location alone in the program, but this would not help a great deal. This seems to have been one of the difficulties of Friedberg's programs. It is clear that most of the deviation of \( P_y \) from \( P \) may be explained in terms of \( y \)'s relation to the orders around it in the program. For example, \( P_2 \) should certainly depend on the selection that has already been made for \( (y-1) \). Since max \( P_y \) is conditional upon the value of \( (y-1) \), a set of \( k \) \( P \)'s may be used, one for each value of \( (y-1) \). In general, not only will max \( P \) be dependent on \( (y-1) \) but also on \( (y-2), (y-3), \) and in fact on many other variables that might not even be defined in the candidate program.

Other variables that should be considered are the particular orders that make up the subject section being translated and the contents of certain locations in both the interpretive routines used. While the relationship of the subject orders might be obvious, that of the interpretive routines is probably not. The interpretive routines do contain variables in the sense that \( (y-1) \) is a variable, for example, the “left-right counter” necessary in an interpretive routine simulating Princeton-type machines having two orders stored in one location in memory. In fact, it might be suspected that since a human being can know all there is to know of a computer's importance to programming by examining an interpretive routine simulating it, indeed a great deal might be gained by considering the relationship between max \( P_y \) and some of these variables. The reason for the concern with the dependence of all these variables with max \( P_y \) is that prior to selecting the value of \( y \), the particular values that these other variables have taken will be known; and if the relationship is also recognized, a better \( P_y \) may be used. Certain definitions follow naturally. If \( X = (x_1, x_2, \ldots, x_n) \) is a set of variables whose values may be determined prior to the selection of order \( y \) and \( S = (s_1, s_2, \ldots, s_n) \), a set of particular values for \( X \), then max \( P_{y,s} \) will mean that value of
\[ \{ P \} \] for which \( U(P_{y,s}) = P \) (a response chosen using \( P_{y,s} \) is acceptable given \( X = S \) ) is maximized. \( S \) is called the stimulus vector, and \( x_1, x_2, \ldots, x_n \), stimulus variables.

The Selection of the Response

Considering again the selection of the value of \( y \), it is desired to find \( \text{max} \ U(P_{y,s}) \). If the best \( P_{y,s} \) were determined independently for each possible \( S \) and if \( d = 1 \), that is, if \( P \) is the correct response for the specified \( S \), and if all possible stimulus variables were included, the scheme would have these characteristics. It would classify completely all the correct responses corresponding to the given stimulus vectors. But the same situation, that is, \( S \) would never be encountered twice unless the scheme were again asked to translate the same program. Therefore, it would never be any better than a scheme which took no cognizance of the stimulus situation. Of course, this method is impossible because of the space requirements. Restricting the number of stimulus variables might be one way out. However, if one such variable could take on the average 40 values, the \( p \)'s expressed to five binary digits and \( k = 40 \), the 512,000,000 words of 40 bits necessary to store the \( p \)'s for just five variables would be prohibitive.

A Linear Modal

Clearly the trouble lies at least partly in allowing the \( \text{max} \ P_{y,s} \) to be freely determined for every \( S \). While it is true that in general the effect of one stimulus variable \( x_i \) on \( \text{max} \ P_{y,s} \) will depend upon the values of the other stimulus variables, it is not necessarily the case that the effect of \( x_i \) on \( \text{max} \ P_{y,s} \) will change for every change in the value of any stimulus variable.

A procedure that would take advantage of this fact might be developed along the lines of linear regression theory. The modal assumes that \( \text{max} \ P_{y,s} \) is a linear combination of \( \text{max} \ P_{y-x_i} \), where \( P_{y-x_i} \) represents the \( i \)th probability vector associated with the stimulus variable \( x_i \). Given then an \( S = (x_1, x_2, \ldots, x_n) \), the value \( P_{y,s} \), to be used in the selection of \( y \) would be

\[
P_{y,s} = b_1(\text{max} \ P_{y-x_1}) + b_2(\text{max} \ P_{y-x_2}) + \cdots + b_n(\text{max} \ P_{y-x_n}).
\]

The \( P_{y-x_i} \) 's would be estimated in the standard way; \( P_{y-x_i} \) would be modified to increase \( p \) when \( O \) was an acceptable order occurring at \( y \) when \( x_i \) had value \( i \).

\( b_i \) should be increased as all the individual probabilities within \( \{ P_{y-x} \} \) tend away from \( \text{max} \ P \) (max \( P \) taken in the sense of the last section). This could be accomplished by letting \( b_i \) increase in increments according to a sample taken from a random variable highly correlated with

\[
\sum_{i=1}^{k} (P_{y-x_i} - P_i)^2
\]

where \( P_{y-x_i} \) is the \( h \)th component of \( \text{max} P_{y-x_i} \) and \( P_i \) is the \( k \)th component of \( \text{max} P \). Likewise, \( b_j \) should be decreased as \( x_j \) shows increasing correlation with the other variables of \( X \).

It is suggested that a measure of how much \( b_j \) should be lowered because of its correlation with other variables might be some function of the similarity of the particular \( P_{y-x_i} \) to the other \( n-1 \) vectors determined by the particular \( S \).

A method for the selection and rejection of stimulus variables would follow naturally. At periodic intervals that \( x_j \) whose \( b_j \) were lowest would be discarded, and a new one selected at random from the remaining variables not currently represented in the stimulus variable set.

The model may be made even more flexible by permitting variables which are the products of two or more stimulus variables.

A dubious future would seem to be ahead for the model described above if required directly to "learn" how to translate programs for an 1103A to 704 language. The unfortunate situation is, however, that almost nothing is known of the joint distributions of what have been called the stimulus and response variables in such a translating endeavor, and therefore our judgment of any model, prior to its incorporation in a translator program, must be intuitive.

An Example

For the purpose of exhibiting some of the general characteristics of the type of translator proposed, a program has been written that does in fact compile complete programs for an imaginary one-address machine given those for an imaginary three-address machine. To be sure the two machines used are neither of the size nor complexity of current machines, nor are they as different from each other as current machines. Nevertheless they are sufficiently powerful to be able to compute transcendental functions, invert small matrices, etc.

Our experience with this compiler is limited and a full report on its performance is not yet ready; however, our results to date have proved both educational and encouraging.

Our estimations of the initial expected time of translation was of the order of one hour per instruction of the subject program. After gaining experience, the translator should reduce this to about 90 seconds per subject instruction. In the first programs translated, the translator did considerably better than anticipated. Upon examining the programs produced, the reason became evident. We had overlooked a large class of acceptable programs, namely, those in which numbers were left in the arithmetic register of the one-address machine at the termination of a section; those same numbers could be of use in the succeeding section. The translator had promptly taken advantage of this.

The principal difficulty encountered so far has come with conditional transfers of control. The trouble lies
in that the acceptance of a sequence containing a conditional transfer is contingent on its correct operation at several points in the program. Therefore, a large portion of the program must be executed before a candidate unit may be rejected or accepted. This implies that for very long programs the probability of success for a particular candidate unit containing a conditional transfer is going to have to be much higher than would be necessary for a purely arithmetic unit. This same difficulty will arise in the situation of units whose function is to modify instructions.

The translations produced have been far from optimal, primarily because the length of unit translated at one time was one subject order. If two at a time had been taken, the initial expected running times would have been much too great. However, the present translator may now be modified to use its experience on one-at-a-time translation in translating two at a time, and the initial expected running times will be reasonable. This suggests that eventually the translator should choose the size of the unit it attempts to translate according to the subject orders involved. Thus the translator would reduce the size of the unit attempted if it were recognized as one with too high an expected translation time, and increase the size as it gained experience.

CONCLUSION

It was suggested at the beginning of this paper that the type of compiler described might not be of immediate use. In the author's opinion, this is not because our machines are too small and too slow, although certainly larger random access memories would be helpful. The cause seems to lie more in our ignorance of machine design and programming, which amounts to essentially the same thing. Any increases in speed are quickly absorbed in the realm of combinatorics.

It is hoped that the ideas expressed here will have some value in finding the solution to this large class of problems, which includes not only compilers but human-language translation, game playing, and other problems where the initial complexities suggest a solution that is self-improving. Development of information processing theory will certainly make these modified British Museum techniques obsolete. Nevertheless, they may be of value in the interim and may be able to contribute to the more rapid development of that theory.

Special-Purpose, Electronic Data Systems—The Solution to Industrial and Commercial Automation

WILLIAM V. CROWLEY†

INTRODUCTION

THE concept that any “standard” electronic data processing or computing system should be adapted solely through programming to suit a particular business or industry is no longer supportable, except as an interim step for testing the planning and design of the data system through simulation. Nor does the answer to making “standard” electronic data systems more easily adaptable to various business applications lie in the area of so-called automatic programming. This is not to imply that this endeavor has not or will not be valuable and useful.

Its main contribution, however, will be to force data definition discipline in business systems so that more powerful electronic data processing equipment, with more pertinent and functional instructions and commands, will be built.

A careful study and comparison of the current data systems of many different types of business enterprises will show considerable similarity in the general structure and media of the data systems, such as files, action documents, information documents, reports, etc., but will disclose numerous differences in the composition and arrangement of the data itself, as well as management's philosophy of and concern with data handling. Some of these differences are degree of variability of the length of the data items; relative occurrence of alphabetic and numeric information; language idioms or usage peculiar to the business or industry; attitudes and practices concerning the coding of information; government regulations; accounting customs, etc. Because of these existing variable factors and the social and economic resistance to absolute standardization, industrial and commercial data systems require equipment designed, not adapted, for their particular situation.

NEGATIVE INFLUENCES

Let us review some of the important factors which tend to inhibit this inevitable trend toward the building of electronic data systems tailored to a particular in-