MS-CIS-79-8

University of Pennsylvania

THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

ON PROGRAM SYNTHESIS IN INTELLIGENT CALCULATORS THROUGH CURRENT AUTOPROGRAMMING TECHNIQUES

Frank A. Weber, Jr.

A thesis submitted to the Faculty of The Moore School of Electrical Engineering in partial fulfillment of the requirements for the degree of Master of Science in Engineering (for graduate work in Computer and Information Sciences)

Philadelphia, Pennsylvania

May 1977

University of Pennsylvania

THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

Title of thesis: On program synthesis in intelligent calculators through current autoprogramming techniques

Abstract:

Recent technology has placed powerful computing devices within the grasp of the average layman. The calculator is the most widely utilized such device and its power now compares to that of the first computers. While the users of these calculators are familiar with strict sequential operations, few are familiar with programming techniques. Much work has been done on program and machine inference from example input and output traces. An evaluation and assimilation of current machine synthesis studies and auto-programming techniques has been described. An intelligent calculator capable of automatically constructing or inferring suitable programs from example computations executed by the user, keeping the additional required knowledge of the user at a minimum, has been proposed.

Degree and date of degree: Master of Science and Engineering for graduate work in Computer and Information Sciences.

May, 1977

[illegible]
Frank A. Weber, Jr.

John W. Carr III
Dr. J. W. Carr III

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

"Ever since the early days of electronic computers, man has been using his ingenuity to save himself from the chores of programming."

Alan W. Biermann

<u>Approaches to Automatic Programming</u>

"Declining costs of the [programmable] devices themselves have made them available to an ever widening segment of society; however, true democratization rests on the elimation of the programmer."

Rochelle Fleischmann

<u>A Proposal for Example Induced Programming</u>

CHAPTER I

INTRODUCTION

Program Synthesis

Upon the introduction of the ENIAC (Electronic Numerical Integrator and Computer) in 1947 new professions were born, the computer software designer and the computer hardware designer. Man had invented a tool that would eventually affect almost every segment of his daily life. However, the great majority of mankind not only does not know how to use this tool but also considers the computer to be some form of magic. The hardware designer must learn the characteristics of the target machine and know the technology available in order to create a machine with those features. The software designer must learn the requirements of the desired procedure and know the machine's characteristics in order to create an appropriate program.

It can be argued that the hardware designer and the software designer have essentially the same task. Carr (1976) has shown that a program is a finite state machine which can be represented as a Moore or Mealy model of a sequential machine. Many researchers are currently working on the inference of Turing machines, finite-state acceptors, and computer programs from sample traces or computations. The remainder of this chapter will be devoted to research done in the area of program synthesis from example traces of the program's behavior. Once the computer can "learn" programs from minimal input supplied by the user then it can be stated that this tool serves man.

Research and Problems of Learning Methods

The most fundamental research is in designing systems which actually synthesize programs from very weak information such as input-output pairs or from the specification of desired performance. Amarel (1962,1971), Chang (1974), Lee (1974), and Waldinger (1974) are well known researchers of this method. Their work addresses the very basic problems concerning artifical intelligence; the definition of knowledge, method of representation, utilization of acquired knowledge, and so forth. Amarel (1962) states that construction of a synthesizing algorithm, given this weak information, is incredibly difficult.

A synthesizer of this type could have at its command a set of allowable instructions and then attempt to perform its task by enumerating the set of all possible programs of length N, testing each against the sample input-output pairs to see if it behaves in the desired manner, and increasing N until such time as a suitable program is found. An example of the input-output pairs could be (1,1), (2,1), (3,2), (4,3), and (5,5) where the synthesizer would be expected to produce a program that has any positive integer I, as input and as output would record the I^{th} term of the Fibonacci sequence. Using this example, the synthesizer would probably not construct a program that would find the desired transformation, $(\frac{1}{2}x(1+\sqrt{5})^{I}-\frac{1}{2}x(1-\sqrt{5})^{I})/\sqrt{5}$. One can easily see that this method of research can be frustrating, because even after a program is found it may not be the correct one or it may not halt upon new inputs (Hopcroft and Ullman, 1969).

The major problem area confronting the above research is the formation of a set of instructions that would convert all inputs X_i to the appropriate outputs Y_i. Biermann (1976) suggests subdividing the task. Initially a set of instructions is found for each input X_i that satisfies the transformation to output Y_i. Once this is done all that remains is development of a procedure which will execute each required sequence of instructions when given its associated input X_i. The task of finding a set of instructions for each X_i is the difficult task it was previously. If some form of communication existed between the user and system, such as inputting examples of transformations, only the development of a procedure to execute the correct transformation would be a problem.

The motivation for program synthesis is to relieve the user from the chores of programming. Since man naturally teaches and learns through example, the requirement for the user to supply example transformations to this system does not seem unreasonable. In fact, the user must know examples of the computations he desires before he has the need for the process to be available to him on a computer.

Synthesis from Example Computations

The problem of developing the correct procedure given example computations or traces has been studied considerably, and various reasonable techniques exist for the construction of this procedure. All techniques limit the class of synthesizable programs and provide approaches and restrictions to the human teaching interaction. Bauer (1975) limits procedure representaion to three kinds of instructions and forces the traces to show the failures or conditions not satisfied as terminal left nodes of a trace tree. An example of this kind of trace for a LISP (List Processing Language) program for adding a list of elements, L (possibly empty) and returning the sum, S is shown for L = {1,2,5} in figure 1. Bauer then provides a method for construction of a program from two of these type traces.

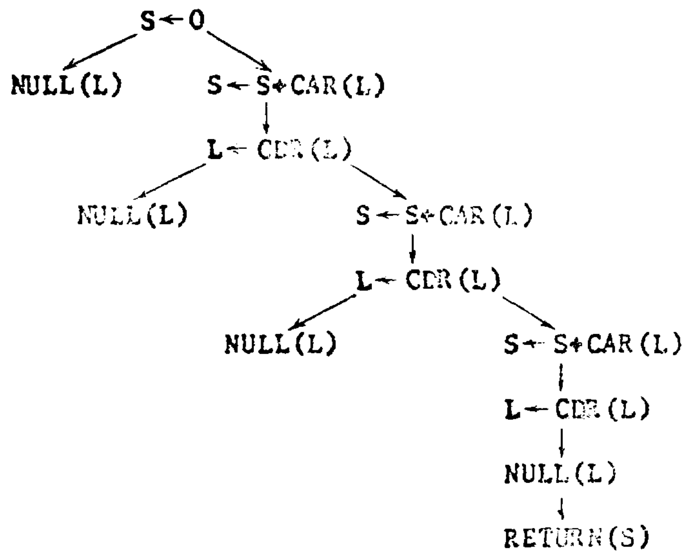


Figure 1. Bauer-type LISP trace for summation of list, L = {1,2,5}.

Biermann (1972) presents an algorithm for finding a Turing machine which is capable of performing the same set of Turing machine computations that were completely described to it. Carr (1976) suggests a desk calculator with a special compare button to input example calculations. He uses state minimization techniques with two such example calculations to infer programs. A model trace for the greatest common divisor for M = 3, N = 2 is as follows (Carr, 1976, p.3):

```
STO 3 in M
STO 2 in N
M÷N→M
Remainder→r
CMP 0:(r)        *False*
r→N
M÷N→M
Remainder→r
CMP 0:(r)        *True*
M→Result.
```

The Autoprogrammer

While these methods are workable, the restrictions regarding the inputting of an acceptable trace to the method may seem as unnatural as programming to the uninformed user. One additional method of confronting this problem deserves special attention and will be the chief topic of this thesis. This method uses more of an interaction between user and machine. Biermann and Krishnaswamy (1974, 1976) have described this process as an "autoprogrammer ... an interactive computer programming system which automatically constructs computer programs from example computations executed by the user" (1976, p.141). They have actually implemented a system which allows the user to display his data structures on a CRT screen and manipulate them via a light pen driven language. In this way example traces are shown to the system with user verification at each step of the computation. Fleischmann (1975) has developed an autoprogrammer type system that asks the user for information that it must know. This system will be described in Chapter Three.

Objectives

The purpose of this study is to examine the modern calculator as a possible medium for program synthesis techniques. A baseline calculator is presented as the basic medium and is used to demonstrate synthesis algorithm that are described. This study does not address the problem of memory space in a calculator or execution speeds for it is believed that technology will continue to find more efficient methods of storage and retrieval. Instead, the logical problems of program synthesis will be analyzed.

Three methods of program synthesis are delineated in subsequent chapters. The first two are extensions and adaptations of two existing methods, Fleischmann's (1975) and Biermann's et al. (1974, 1975, 1976). Each method is described in terms of user requirements, modifications to the baseline calculator, and detailed examples of the synthesis process. It is shown that while Fleischmann's method requires the smallest amount of prior knowledge needed by the user to utilize the system, it is only capable of infering programs with sequential and "If ... then ... else" logic. Biermann's method requires a slight amount more structure from the user in his example calculation. However, his process is capable of producing sequential, "If ... then ... else", and "Do while (given condition)" logic.

The final method of program synthesis in a calculator utilizes concepts from both the previous methods to require the least possible prior knowledge of the user, yet still be able to infer programs that contain all of the previous forms of program logic plus "Do while (condition involving a loop counter)" logic. Examples of this method are given as well as guidelines for the use of this autoprogrammer/calculator in order to insure the correctness of the resulting program. This study concludes with a discussion of the problem and the presented methods.

CHAPTER II

THE CALCULATOR AS THE INTERFACE MEDIUM OF AN AUTOPROGRAMMER

<u>Why the Calculator</u>

While the previous research has given us reasonably effective techniques for dealing with program synthesis, they are limited in many ways. The expense of large scale computers dictates the need for a well informed user (programmer) so that it can be used effectively and efficiently. The synthesizer, at best, can only produce the logic that is implicitly described in the given example; it can not create a procedure where none existed before. In order for the programmer to devise a few examples for the autoprogrammer he essentially develops the program.

A truly complex program would need many examples and be tedious to describe. The user must have a firm grasp of the logic contained in the desired process and thus be well informed. This would seem to indicate less of a need for an autoprogrammer system on large computers and complex programs. The autoprogrammer appears to be a more useful tool to the non-programmer who finds himself with a repetitive task and does not understand how to form a procedure to eliminate such repetition.

In the 1950's and 1960's computers were bulkier, slower, contained less memory, and were made of more costly materials than today's machines. Compared to the high cost of equipment, the cost of software development did not appear to be exorbitant. This caused pressure to be applied to ensure that the software had optimal speed and efficiency, thus producing specialized users. The average person was not likely to see many computers, let alone use one to balance his checkbook or calculate his income tax. Times have changed, technology has advanced, and now more people than ever come in contact with programmable devices. The most common of these is the new programmable hand calculator.

These programmable machines generally are priced out of the range of the average household, as opposed to those calculators which simply add, subtract, multiply, and divide. Programmable calculators continue to decrease in cost at an amazing rate. For example the Hewlett-Packard HP-55 listed at $595 in 1973 and is now available for $149.95. The comparably featured Texas Instruments model SR-56 (makers of electronic chips for Hewlett-Packard) is currently selling for $79.95. The SR-56 has ten general purpose registers, 100 memory locations, and is supplied with an owner's manual and an applications library booklet which are essentially a "how to program" course.

What of the individual who does not feel the need or wish to bother about becoming a programmer but desires some help in a repetitive task that he must perform? An "autoprogramulator" could be very useful for this type of user. This device, a calculator with a built in autoprogrammer, could allow the user to input a few example calculations and then construct a program that would push the keys for him.

Advantages/Disadvantages

The availability of the calculator has a distinct advantage over other types of interface -- familiarity. Any interface used in an autoprogrammer system should not be foreign to the user, otherwise training would be required. By using the calculator we can further assume that the user is capable of executing examples of his problem. Instead of using a new language, a trace may be keyed in directly. The system could be used by any person familiar with a standard pocket calculator.

Unfortunately, by choosing the calculator, we have placed a restriction on the display area. A large area such as a CRT display could be used conveniently to depict data structures and graphically manipulate their contents like blocks on a table. However, even the size of the CRT display limits the number of different data structures that may be visually manipulated. The calculator's display can simply be viewed as a one data structure display.

A very strong point in the calculator's favor is that it dictates a limit on the class of computations that are involved. This class is the one desired in order to insure the synthesis of a program from example computations. The inferred procedure will have a single start instruction and will be a functional procedure, that is, a procedure which returns a single value. During the computation several variables and constants may exist but the resulting value will be in the data structure display. This should not present any difficulty. Take for example a calculator trace solving a quadratic equation, the first root can be thought of as a temporary variable of the computation and the second root (if any) will be the final result of the procedure.

Currently, the physical size of an autoprogramulator may not permit enough logical memory to contain the autoprogrammer, sample traces, and other work areas. For the sake of argument, this paper assumes that there will be enough memory for the autoprogrammer, two sample traces, and a certain number of compartments for previously inferred programs. In this presentation the length of any resulting program will be arbitrarily limited to one hundred instructions. In this way it can be determined if a reasonable program exists for assimilation of the example computations.

The Baseline Calculator

Each of the various autoprogrammer systems under discussion uses the calculator as the interface between user and machine. These autoprogrammer systems require additional features not included in the usual hand calculator. An example baseline calculator is described in order to provide a better understanding of the deviations between the proposed systems.

This baseline calculator is very similar to the average scientific calculator. In fact, most of its functions are modeled after the Texas Instruments SR-56 calculator. Physical size and weight are such that it can be hand held or carried in a shirt pocket. The keyboard is comprised of thirty-four keys which serve as the vehicle for user input. A ten digit display serves as the top element of an eight element operand stack. Included in the display is an error indicator which will light on overflow, divide by zero and other illegal operations. Also, internal to the calculator are ten storage locations and a seven element operator stack. These two stacks allow for mathematical expressions to a maximum of seven pending operations.

The keyboard is divided into four classes of keys: (1) data entry/storage keys; (2) unary operator keys; (3) binary operator keys; and (4) a terminate last pending operation key (Table 1).

TABLE 1

THE BASELINE CALCULATOR KEYBOARD

Data Entry/ Storage Keys (15)		Unary Operator Keys* (12)		Binary Operator Keys* (6)		Terminate Last Operation Key (1)	
0		$\lvert X \rvert$	change sign	+	add	=	equals
1		X^2	square	-	subtract		
2		1/X	reciprocal	x	multiply		
3		$\sqrt{X}$	square root	÷	divide		
4		SIN	trigonometric functions	$\sqrt[X]{Y}$	root of		
5		COS	"	Y^X	power to		
6		TAN	"				
7		10^X	power of 10				
8		e^X	power of e				
9		LogX	log				
π	PI	LnX	natural log				
.	Decimal	RadX	X→radians				
CLR	0→X						
STO	X→location n						
RCL	location n→X						

* X is the top element of the operand stack, Y is the next to top element.

The data entry keys are used when the user wishes to place a value on top of the operand stack. Initially the top of the operand stack (the display) is equal to zero. Unless the user depresses a data entry key it will be assigned as zero. The user may depress any combination of up to 10 arabic numbers with at most one decimal point, recall a storage location, or depress the special Pi key to set the value of Pi onto the stack. Ten storage locations (0-9) are reserved for temporary storage during a computation and are initially set to zero. Depression of the STO key and an arabic number causes the current display to be stored at the specified location. The RCL key followed by an arabic number sets the display to the current value of the location. A clear display key is included in case of a mistaken entry. Entering over ten digits causes an error condition.

Upon the depression of any unary operator, the depressed operator is immediately applied to the value on top of the operand stack and the resulting value replaces the previous top of the operand stack. If the calculator display contained a value of three and the square key was depressed the top of the operand stack (display) would be replaced with the value of nine.

Selecting a binary operator causes a different type of action. A zero is "pushed" down on top of the operand stack (providing the stack is not full), thus replacing the previous display. The operator chosen is then "pushed" down on top of the operator stack. If the operand stack is full the error indicator will light and no operation will be performed. The display, now equal to zero, is ready for another operand to be entered.

Finally, the equal sign or terminator causes the top operator to be "popped" from the operator stack and applied to the top two operands. The resulting value will replace those operands and appear in the display. If the operator stack is empty, the error condition will occur.

In addition to the above ways of entering the error condition some operators restrict the types of operands that can be used, such as the reciprocal key being depressed when the current display equals zero. The error condition is cleared by turning the calculator off and on which also returns the calculator to its initial state. Use of the baseline calculator to solve the equation 2x((-3x4)+5) is shown in Table 2. Notice the way in which the equal sign is being

TABLE 2

USE OF BASELINE CALCULATOR

TO SOLVE 2x((-3x4)+5)

User Function	Resulting Operand Stack	Resulting Operator Stack
calculator on	$\underline{0}$	empty
2	$\underline{2}$	empty
x	$\underline{0}$,2	x
3	$\underline{3}$,2	x
|X|	$\underline{-3}$,2	x
x	$\underline{0}$,-3,2	x,x
4	$\underline{4}$,-3,2	x,x
=	$\underline{-12}$,2	x
+	$\underline{0}$,-12,2	+,x
5	$\underline{5}$,-12,2	+,x
=	$\underline{-7}$,2	x
=	$\underline{-14}$	empty

$\underline{n}$ indicates displayed operand used. This method was chosen in order to eliminate the need for parentheses, but still allow the user to input the calculation in a normal form instead of Polish notation or similar non-parenthetical methods.

The calculator has been chosen because of its familiarity and practicality as a medium for an autoprogrammer system. Inherent in this choice is a limit on the number of different instructions that may comprise an example computation. Also, there are limits imposed on the number of instructions that can comprise an example trace and an inferred process. These limitations are such that the algorithms presented can be described fully and are adequate for most applications performed on a calculator. The succeeding chapters discuss three approaches to creating an autoprogrammer system using the baseline calculator.

CHAPTER III

THE FLEISCHMANN PROCESS ADAPTED TO THE BASELINE CALCULATOR

Approach

In A Proposal for Example Induced Programming, Fleischmann (1975) analyzed a program as a formal structure of a user's concept. It is the programmer-analyst's task, she stated, to interface between user and machine by comprehending the user's concept and constructing a program that is acceptable to the machine. Her plan was to incorporate the programmer-analyst into the programmable device as a "student", whose job it was to learn the concept through user (teacher) supplied examples and induce an appropriate program.

In order for the calculator (student) to learn the concept from the uninformed user (non-programmer), the "student" must take an active role by asking questions of the user when the example is ambiguous. Fleischmann envisioned the sample computation as a path in a binary tree from the root to a terminal node. The synthesizing process would accept the first user supplied sample computation as a tree, and subsequent sample calculations as branches to be integrated with the tree. Where the subsequent examples differed from the original tree in terms of operands or operators, the "student" would ask the user for additional information. This information would make it possible for the synthesizer to modify the original tree to allow variable operands or concatenation of a new branch to the tree.

<u>Modifications Required to Basic Calculator</u>

Implementation of the Fleischmann induction process in the autoprogramulator will require several additions to the baseline calculator. Modifications are necessary in order that the user can notify the process that he is starting or ending an example. He must also be able to signal when an example calculation is to be used to extend a synthesized process, or when execution of a pre-existing process is desired. These needs are satisfied by the signal keys (Table 3). The number depressed after these keys indicates in which compartment the synthesized program is to be stored and also serves as the program name. The equal sign is used as the terminator for these commands.

TABLE 3

PROCESS SIGNAL KEYS

Signal Keys		Meaning
START	n	Clear compartment n and prepare to start synthesis
SAVE	n	End of current example, store program
EXTEND	n	Prepare for additional example of program n
EXECUTE	n	Execute pre-synthesized program n

The autoprogramulator will take an active role in the synthesis process when it needs information to modify the program (or tree in this case) and during the execution of a program. The display will be able to show questions that the Fleischmann process needs to ask when interrupting a user example. The needed questions are "C OR V", "V", and "REASON". For the user to answer these questions, six additional keys are needed. Keys that denote relational operators, {EQ (equal), NE (not equal), LT (less than), GTEQ (greater than or equal)} and keys which answer the "C OR V" question, {CONSTANT, VARIABLE}, must be included.

Methodology

The synthesis process is divided into two phases. The first phase consists of the user depressing the START key, entering a sample calculation, and depressing the SAVE key. The autoprogrammer is passive during this phase since this is a new example and there are no previous ones with which to compare. It simply accepts the sample calculation as a sequence of constants and operations then stores it away as the induced program.

The second phase begins with the user depressing the EXTEND key. This signals the autoprogrammer that an additional example is to be entered which may affect the state of the previously stored program. As the user enters the second (or subsequent) trace the autoprogrammer starts comparing this trace to the stored program. As long as they are exactly the same the autoprogrammer remains passive and allows the user to continue. If, however, the operator or operands change during the trace, the autoprogrammer "freezes" the keys of the calculator and questions the user in order to find the reason for the change.

If the operator (or both the operator and operand) have changed from the stored program tree then a new branch must "grow" from the previous instruction. The autoprogrammer will insert a branch on condition immediately after the previous instruction, but first it must know the condition involved. In order to do this it causes "REASON" to be displayed to the user and waits for the specific user input. The user is then obligated to name a storage location, one of the relation operators, a value or second storage location, and the terminator symbol. Omission of the first storage location indicates that the value of the display is to be used.

The autoprogrammer will receive the condition as keyed-in by the user and check for the proper form. It must then test that the condition described actually seperates this example from the previous tree. If either of these tests are not met, the autoprogrammer will loop and repeat the question, "REASON", otherwise it will permit the user to continue the example.

If, on the other hand, the operand differs it may be that the operand is a variable, or that it is a constant and a new branch must "grow" from the previous instruction. To determine which cause is the correct one, the autoprogrammer displays the question "C OR V" whichs means constant or variable. If the user enters "CONSTANT" a new branch must "grow" and the autoprogrammer enters the "REASON" questioning previously described. If "VARIABLE" is entered the autoprogrammer marks this instruction so it will know to ask the user for a value during the execution process.

The execution process involves retrieving a stored program and executing each instruction starting from the root of the tree. When it encounters a branch on condition statement it tests the condition and control continues accordingly. No user interaction is required during this process except when the autoprogrammer finds an instruction marked as containing a variable. The autoprogrammer will display "V" and wait for the user to input the value followed by the terminator. An example of these processes is presented in Appendix A.

Restrictions of the Fleischmann Process

While this method presents a workable solution of program synthesis with the least amount of user interaction and training it has several problem areas. One of these deficiencies is that the user is expected to identify all of the variables and present them before they are needed. For instance, he may want a process that can compute the average of two, three, or four numbers. His first trace may be (2+13)/2 and his next trace may be (5+8+1)/3. When he entered the second "+" in the last trace, the autoprogrammer would display "REASON". The user would have had to realize that the variable 3 needed to be declared before it was even used. The autoprogrammer needs a way of learning this information and unfortunately no other way seems practical.

A second problem inherent in this method is that the interrogation of the user's actions requires afterthought on the part of the user. The user has already made a decision but the autoprogrammer system does not know it needs to ask a question until after it has observed an inconsistency.

The remaining problems are directly or indirectly related to the tree approach. Due to this approach only the trivial sequential process with "If ... then ... else" statements can be realized. The length of the induced program will be at least as long as the original trace since the autoprogrammer is passive during the first trace. For each branch on condition the program under synthesis will "grow" directly proportional to the remaining number of instructions. In Appendix B the example of inducing a process that will compute the average of two, three, or four numbers is used again. It shows that the three traces required contain eleven, fourteen, and seventeen instruction respectively whereas the resulting program contains twenty-five instructions excluding the needed condition logic.

Since it is not possible to infer loops, the user has not been efectively relieved from his tedious task. If he wanted to create a program that would find the first N primes he would be required to enter N seperate traces. Finally, due to the propor-tional growth of the synthesized program it is hard to place a limit on the space necessary to contain the resulting program.

The method which Fleischmann used to pull together the user transformations of input to output is very basic and simple. It requires only that the user be able to tell the autoprogrammer system why the process is being transferred to a new branch. The system then "remembers" the reason by including it as a "branch on condition" instruction in the synthesized program and concatenates the remainder of this trace to the pre-existing one. However, Fleischmann's method has restrictions and methods requiring more user interaction will be examined to see if some of these restictions can be avoided.

CHAPTER IV

THE BIERMANN PROCESS ADAPTED TO THE BASELINE CALCULATOR

Approach

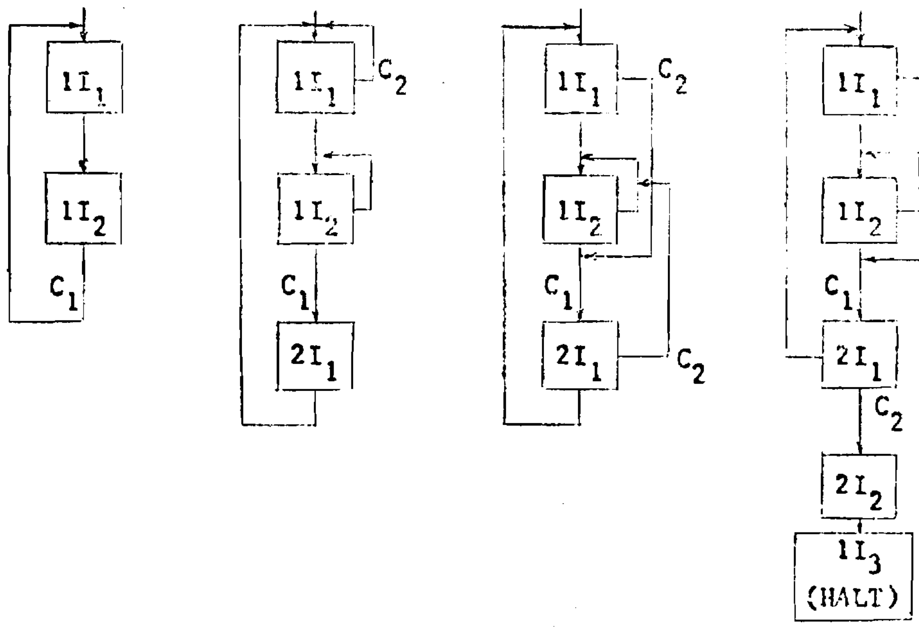
In a number of articles by Biermann et al. a different approach to the synthesis problem has been taken. Throughout the remainder of this paper this approach is called the Biermann process. The goal of this process is to automatically synthesize the shortest possible program that could execute the user supplied examples and still provide an easy method for transmitting the user's concept to the machine. In order to develop a synthesis system that is sound, complete, and able to infer the shortest possible program, it is necessary to require more information from the user's example than was done in the previous method.

In the Biermann process, an example calculation begins with the user naming the program to be created and declaring all input variables to be used in this example. The computation trace is then composed of the user selected instructions and their order of execution. Additionally, the user is required to indicate when a condition occurs which will affect the execution of the example. Notice that the synthesis algorithm now requires the user to indicate a controlling condition before the trace deviates from a baseline trace, without prompting by the algorithm. The requirements of declaring inputs and control conditions are necessary to make this approach possible.

Biermann et al. envisioned this trace as a Moore-type representation of a finite state machine. Each occurrence of an instruction can be thought of as a state. When an instruction is executed the next instruction (state) that will be transferred to is dependent on the current program instruction (machine state) and the condition (input). State minimization processes are then used to synthesize the shortest possible correct program. A large portion of the notation used in this chapter is from Speeding Up the Synthesis of Programs From Traces, Biermann, Baum, and Petry (1975).

As the user enters an example calculation, the set of distinct instructions will be formed and denoted as I_1, I_2, I_3, etc. Also, at the same time the set of distinct conditions will be formed and denoted as C_1, C_2, C_3, etc. The absence of a condition preceding an instruction indicates that the transition is not dependent on a condition or that control conditions were not met. Each condition (if any) and the following instruction denotes a level of the trace. For example, level four of the trace shown in figure 2 is the set $\{C_1, I_1\}$.

After the declaration of the program name and its inputs, a sample calculation might appear as: I_1, I_2, I_2, C_2, I_1, I_1, C_1, I_1, I_1, C_1, I_1, C_1, I_2, I_3; where I_1 and I_2 are instructions, I_3 is the halt instruction, and C_1 and C_2 are conditions that affect the execution of the example (refer to figure 2). Notice that specific instructions and control conditions may occur several times in an example trace. Multiple occurrences of an instruction are caused by "looping" through part of the previously traversed logic contained in the computation and/or because the instruction occurs in logically separate parts of the computation. In the sample calculation, instruction I_1 occurs six times, however in the minimized Moore-type representation of the target program (figure 2(d)) it is shown that only two distinct I_1's occur. The first occurrence of an instruction in a program will be labled $1I_j$, the second distinct occurrence $2I_j$, and so on.



(a) (b) (c) (d)

Figure 2. The Biermann synthesis process for the sample calculation: I_1, I_2, I_2, C_1, I_1, I_1, C_2, I_1, I_1, C_2, I_1, C_2, I_2, I_3.

Continuing with the comparison of machines, observe that an example trace is an incompletely specified finite state machine comprised of the finite set of triples M_1, M_2, M_3, etc. each of the form (nI_i, C_j, mI_k). This states that for the nth occurrence of I_i (specific state) under condition C_j (input) a transformation will occur to the m^{th} occurrence of I_k (next state). The Biermann process is an algorithm which determines the number of distinct occurrences per each instruction such that the summation of these numbers ($\sum_{i=1}^{n}$ number of distinct I_i, where n is the number of different instructions used in the trace) will be minimal, thus producing the shortest possible program. As in the Fleischmann process, a synthesized program can be extended by additional sample calculations.

Modifications Required to Basic Calculator

Implementation of the Biermann process in the autoprogramulator will require few additions to the basic calculator. Fewer modifications are necessitated by this process because it does not take an active role in questioning the user, as was done in the Fleischmann process. Instead it places the restriction on the user to declare all inputs before starting an example and to declare conditions that affect the execution of the trace.

The process signal keys (Table 3) are again needed in order for the user to declare the start of a new example, the termination of an example, the start of an example to extend a previously synthesized program, or the execution of a previously synthesized program. The four relational operators, {EQ (equal), NE (not equal), LT (less than), GTEQ (greater than or equal)}, will again be needed to state conditions. However, the autoprogramulator does not initiate the questioning of the condition in this process so it must be told that a comparison is to be made. This is done by the addition of the compare key, CMP, as suggested by Carr (1976). For example, the sequence: CMP, RCL, 2, EQ, 176, = indicates the fact that storage location two equals 176 is affecting the computation.

Because the types of program logic that the Biermann process can infer include loops, it is felt that two additional functions should be added to the basic calculator primarily for convenience. These functions are READ and RECORD. As opposed to a process without loops, there may be many more instances where a great deal of output is produced, as with a prime number generator. The RECORD key is an autoprogramulator command that, when executed as part of a synthesized program, will cause the current top of the operand stack (the display in a user example) to be displayed for ½ second. This would be similar to the PAUSE instruction on the SR-56 (Texas Instruments, 1976) which causes a ½ second delay before the next instruction is executed. If the autoprogramulator was compatible with a printing unit the RECORD key could cause the top of the operand stack to be output.

Instead of declaring all inputs to this process before the example actually started, the READ key could be used as an autoprogramulator command that would indicate the next entry in the sample calculation is a variable. For example, the sequence: READ, 12, STO, 2 would indicate that the autoprogrammer should expect user input every time this occurrence of the example entry was executed. In this example the READ can be thought of as replacing the 12 as a remembered instruction by the autoprogramulator. The calculator will ignore both the READ and RECORD instructions as they are autoprogramulator commands and are not executed until the execution of the synthesized program is requested.

synthesis process assumes that each instruction it encounters is the first distinct occurrence. The process of associating an encountered instruction I_j with a particular distinct program occurrence kI_j is called a "move". Considering the first four levels of the sample trace, $\{I_1, I_2, I_2, C_1, I_1\}$, the process would associate them with $\{1I_1, 1I_2, 1I_2, 1I_1\}$ producing the flow diagram of figure 2(a). These associations imply that instruction $1I_2$ will follow instruction $1I_1$ under the null condition, instruction $1I_2$ loops to itself under the null condition, and a transition to $1I_1$ takes place from instruction $1I_2$ when condition C_1 occurs. From this point on, when the last instruction is $1I_1$ and no condition occurs, a transition to $1I_2$ will be a "forced" move.

In other words, the process first attempts the move of associating $1I_j$ with the encountered instruction I_j unless this move has been forced by a previous move. If this attempt is incorrect, a contradiction will occur later in the process. A contradiction occurs when a forced move dictates kI_j as the next instruction and the next instruction is not a j-type instruction. It can also occur when, during a move, the process must add another distinct j-type instruction, since all known j-type instructions cause contradictions, and this addition would cause the guess of instructions, L, to be exceeded. When a contradiction occurs the process must backup to the last non-forced move it made and increment k at that level. If the backup continues as far back as the first instruction, L must be increased if possible, otherwise the synthesis can not occur.

Continuing with the example, the fifth level of the trace has no condition, thus the previous moves force the next instruction to be $1I_2$. However, the next instruction is of the I_1-type, so a contradiction has occurred. The last non-forced move was in assigning the second $1I_1$ so that move is changed to $2I_1$. The move associating the new instruction to $1I_1$ now occurs. The sixth level states condition C_2 has occurred. Since no previous move dealt with being at $1I_1$ and having condition C_2 occur, the move of associating $1I_1$ with the sixth level instruction is made to obtain the flow diagram of figure 2(b). The entire synthesis process for the example trace depicted in figure 1 is presented in table 4.

TABLE 4

THE SYNTHESIS PROCESS USING THE EXAMPLE TRACE

Level	Condition	Instruction	Initial Synthesis	First Backup	Second Backup	Third Backup
1		I_1	$1I_1$			
2		I_2	$1I_2$			
3		I_2	$1I_2$			
4	C_1	I_1	$1I_1$	$2I_1$		
5		I_1	X	$1I_1$		
6	C_2	I_1		$1I_1$	$2I_1$	
7		I_1		X	$(1I_1)$	
8	C_2	I_1			$(1I_1)$	
9	C_2	I_2			$1I_2$	$2I_2$
10		I_3			X	$1I_3$

X indicates a contradiction
() indicates a forced move

A more detailed example of this process involving a prime number generator implemented on the baseline calculator is presented in Appendix C. It appears that the number of conditions are generally small compared to the number of sequential instructions in a normal calculation. A flowchart of the synthesis algorithm has been taken from Speeding Up the Synthesis of Programs from Traces, Biermann, Baum, and Petry (1975) and is depicted in figure 3.

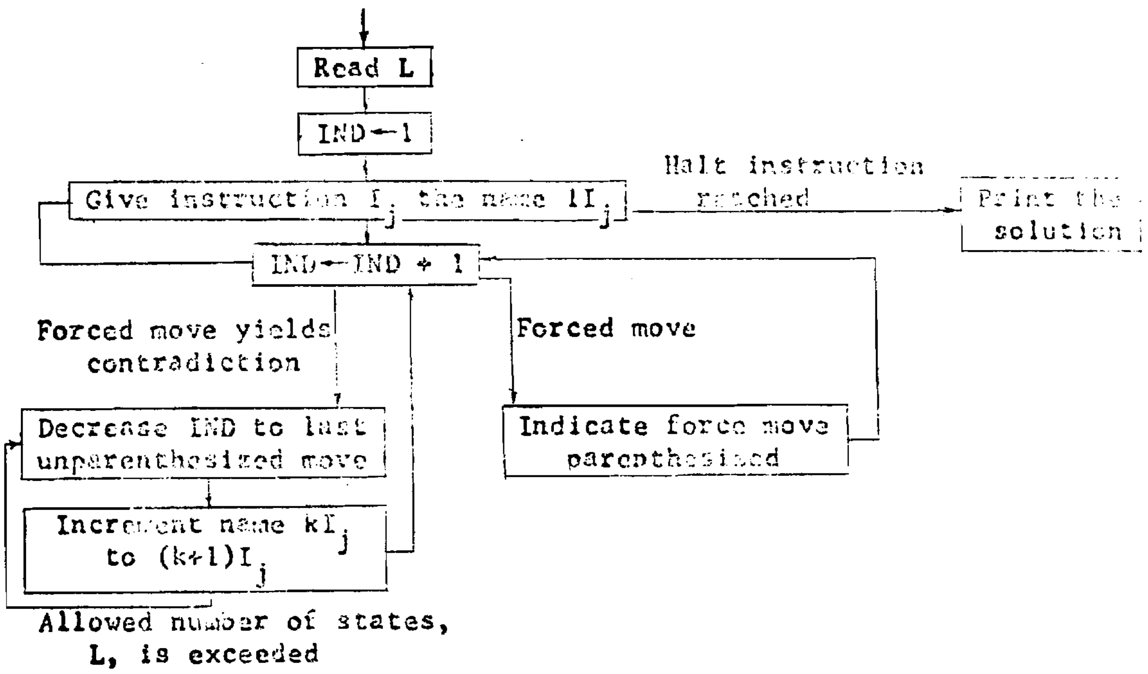

Read L
IND ← 1
Give instruction I_j the name $1I_j$
Halt instruction reached
Print the solution
IND ← IND + 1
Forced move yields contradiction
Forced move
Decrease IND to last unparenthesized move
Indicate force move parenthesized
Increment name kI_j to $(k+1)I_j$
Allowed number of states, L, is exceeded

Figure 3. Flowchart for the synthesis algorithm. (Biermann, Baum, and Petry 1975, p. 126)

The stored program will be comprised of the determined distinct instructions along with comparison and branching logic. When the user wishes to extend a previously stored program, the requested program will be returned and used to build the proper transition tables before the new example is input.

Enhancements

A strictly enumerative technique has been shown in order to demonstrate that the Biermann process could be adapted to the calculator in a simplistic manner. It is recognized that it is always possible to enumerate the set of all possible programs of increasing length until a solution which satisfies the sample trace is found. This is essentially what was done in the synthesis algorithm except that the trace included not only input/output pairs but instructions and conditions that comprise the program.

Biermann noted in the algorithm, when L becomes larger, "the search space grows exponentially and the processing time for the synthesis becomes prohibitive" (1975, p. 125). He proposes methods of static and dynamic pruning techniques and parallel trace processing to speed up the process. On the autoprogramulator the user is not required to interact with the process other than inputting the sample trace so multiple "passes" on the sample trace by the synthesis algorithm is transparent to the user.

Briefly, the static or preprocessing pruning technique is an initial scan of the sample calculation to obtain a "difference set" for each level. The difference set of a level is comprised of a set of nonequivalent occurrences for the instruction at the level. This is used to determine a lower bound for L, and to prevent instructions to be associated with occurrences that they can not be equivalent with. The dynamic processing consists mainly of the creation and use of a "failure memory". It prevents the process from making an assignment that is known to lead to a contradiction more than once, as can happen when backups occur. These pruning techniques should be utilized in an implementation of the synthesis algorithm for speed considerations, but they do not affect the method's capabilities in any other way. The interested reader is directed to the referenced Biermann et al. articles.

An additional enhancement can relieve the user from a repetitious task in some cases. When the next move is forced, the autoprogrammer could display the forced move and allow the user to verify the move via depression of a "CONTINUE" key. This could come in handy during a very repetitious example since the user need only verify repetition instead of entering it. However, the addition of this feature precludes the preprocessing pruning techniques which would be needed in an actual implementation.

Restrictions of the Biermann Process

Apart from the guidelines set for the way in which a sample calculation is entered, the major area of concern is whether a suitable program can be synthesized after a reasonable amount of computation. This problem can be alleviated by the previously mentioned techniques and the availability of faster memories such as the new bubble-type memory. The guideline forcing the user to declare all variables before the calculation seems restrictive but, with the addition of the READ key, this does not appear to be a problem.

An inherent problem with either of the presented systems, and more so with Biermann's guidelines, is the ability of the user to produce correct and sufficient sample calculations for the program synthesis. The form of communication to the autoprogramulator has been solved through the use of the calculator as a medium but the domain of examples remain.

The sample calculation must exhibit instructions that the correct program would exhibit. The user must refrain from skipping steps in a calculation (ie. no mental arithmetic such as adding two to eleven to find the next prime). The sufficient condition is also difficult to enforce. All "paths" through the desired program must be presented in examples, and for logic that "loops" an example should be included which cycles the minimal amount of times.

CHAPTER V

THE INTEGRATION OF METHODS

Objective

It is desirable to have a method which requires the least dependence on the user's prior knowledge yet is still able to synthesize a class of programs which contains sequential, "If... then...else", and "Do while (condition)" logic. The Fleischmann method requires only the sample traces, asks the user for any further information it needs, and successfully synthesizes the class of programs containing sequential and "If...then...else" logic. The Biermann method requires input of the sample trace and conditions used for logic flow determination. It then synthesizes a class of programs containing the logic found in the Fleischmann class and "Do while (user given condition)" logic that is limited to relying upon the user given conditions.

The aim of the integration of these methods is to require only a small user demand as that in the Fleischmann process and be able to synthesize a class of programs equal to, in fact greater than, that of the Biermann process

Approach

From an external view the integrated method first appears as the Fleischmann method. The user enters a number of traces to the autoprogramulator and also enters any reasons for trace deviations as requested. It has been shown that this allows the autoprogramulator to form a tree representation of the sample traces. The user then instructs the autoprogramulator to synthesize this tree by means of a new control key. The tree is traversed and a chain from each terminal node to the root is parsed to a Biermann-like process. The integrated method is then a merger of methods: the Fleischmann method as a preprocessor to the Biermann process.

This merger satisfies the objective of user demands and can synthesize a class of programs equal to those synthesizable by the Biermann process. However, a severe restriction remains. The loops that are produced by the Biermann process are controlled solely by the stated reasons for trace deviation as supplied by the user. As long as the reason is constant over traces a reasonable program will be developed and the loop will be able to handle general cases. For instance, the conditions stated for program control in the prime number generator of Appendix C would be constant over any size trace. However, if the reason for program control changes with each trace the produced loop will only be able to handle the specific cases that were samples and extensions to more generalized cases are lost.

This problem is exemplified in figure 4. The three chains that are passed to the Biermann process are shown. Any instructions denoted as VARIABLE by the Fleischmann process are replaced by a READ command during this process. Note that the synthesized program (figure 5) contains a loop formed by two distinct conditions. The synthesized program only reflects the three sample traces and does not extend to the generalized averaging problem. A method which starts to alleviate this problem by creation of a loop counter is presented.

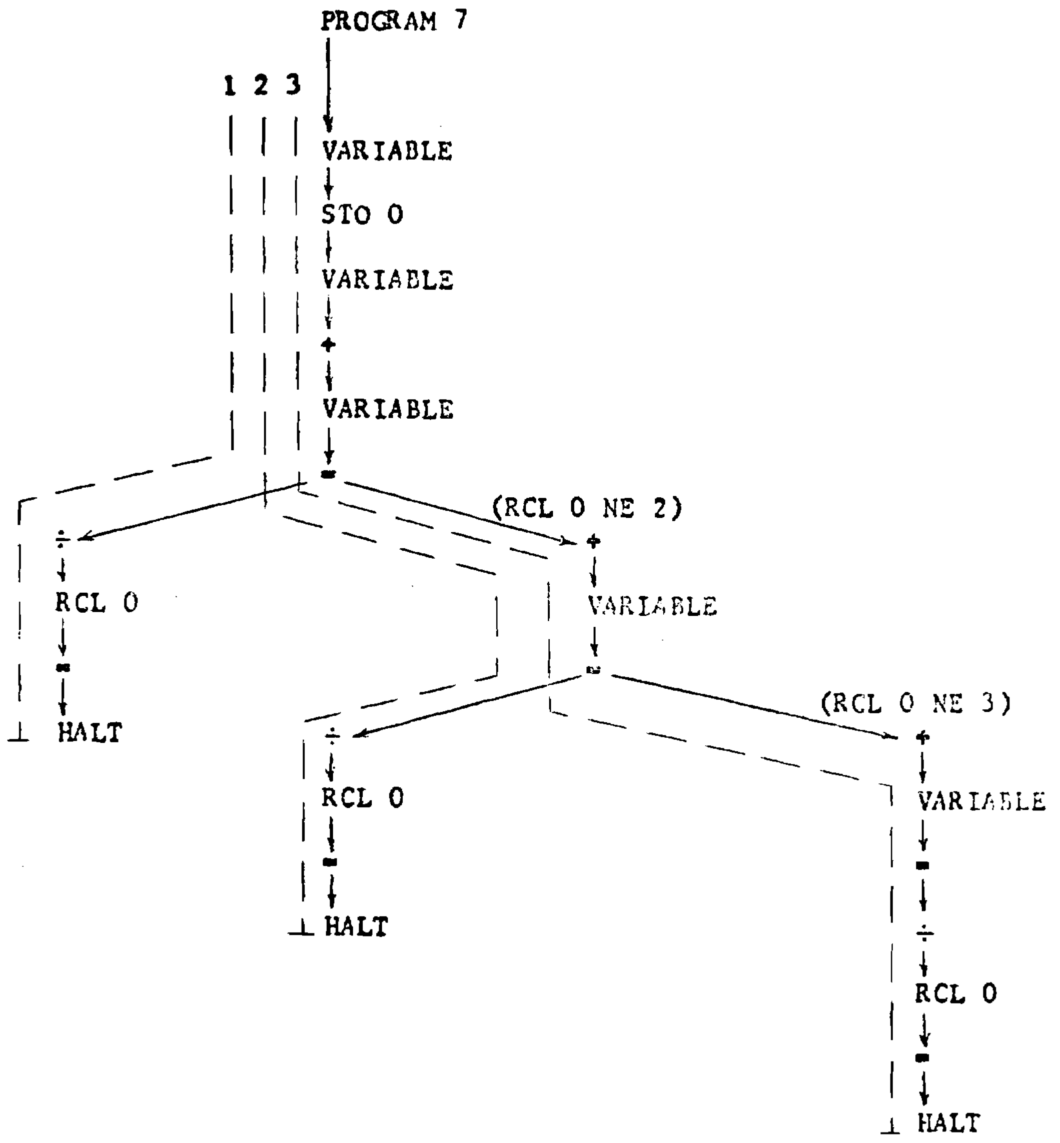


Figure 4. Tree structure and parses of averaging program synthesized in Appendix C.

<u>Modifications Required to Basic Calculator</u>

Nearly all of the modifications previously proposed to the basic calculator described in Chapter 2 will be used in the integrated synthesis method. There is a need to signal the autoprogrammer that the Fleischmann tree representation is to be parsed and sent to the second part of the integrated process. This is done by the addition of the SYNTHESIZE (n) key to the process signal keys (table 3). It will cause the actual program synthesis with loop inference and storing of the inferred program. When the synthesis is complete the resulting program could be flashed on the display (½ second per instruction) or, if a compatible printing unit existed, could be outputted to allow for user v[illegible]ification/review. In order to exhibit the synthesized program a [illegible]ethod could be devised for depiction of the control and branching [illegible]ogic which the autoprogrammer system created. The SAVE and EXEC[illegible] (n) keys are modified slightly. SAVE simply halts the current [illegible]mple and does not store the current program tree. EXECUTE (n) wil[illegible]use either the current program tree or the stored program to be c[illegible]uted depending on whether SYNTHESIZE (n) has been depressed.

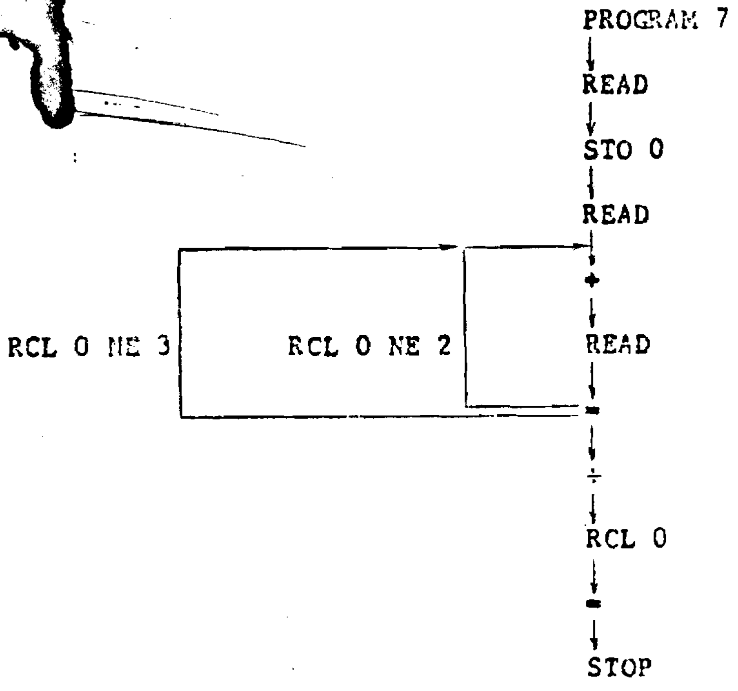


Figure 5. Unaltered Biermann synthesis of averaging program.

Since the autoprogramulator again assumes an active role, the questions: "C OR V", and "REASON" together with the means of answering them {EQ, NE, LT, LTEQ, CONSTANT, VARIABLE} are included. The READ option proposed under the Biermann process can be included as an unnecessary but alternate way in which to declare the entrance of a variable. (Recall the Fleischmann method "finds" variables). The READ is used to replace any found variable for passing to the Biermann process during the parsing phase.

The RECORD key is included as a convenient way in which to handle multiple output. The CMP key is no longer needed since the user will not enter conditions unless asked specifically by the integrated synthesis method. Table 5 shows the entire keyboard that will be used in the integrated method for the autoprogramulator.

Methodology

It has been discussed that by using the resulting program tree from the Fleischmann process, by changing the variable references to the READ statement, and by parsing the tree into the Biermann process, a program can be synthesized. While this process may induce some loops, a method which induces loops in such a way that extensions to more generalized cases occur is desired. Recall that a loop was identified by a triple, $\{nI_i, C_j, mI_k\}$ such that if C_j is true and the present state is nI_i the next instruction will be mI_k.

In the Biermann process described in Chapter 4 each occurrence of an instruction was identified. Also identified, but not formally stated, was each occurrence of a condition. A condition C_j which occurs after instruction nI_i will be denoted $n.iC_j$, meaning this occurrence of C_j occurs after the nth occurrence of an i-type instruction.

TABLE 5

CALCULATOR KEYBOARD FOR INTEGRATED METHOD

Keys	Keys (continued)	Keys (continued)
ON/OFF	$\sqrt{X}$	START (n)
0-9	10^X	SAVE
π	e^X	EXTEND (n)
.	Log X	SYNTHESIZE (n)
CLR	Ln X	EXECUTE (n)
STO (n)	Rad X	EQ
RCL (n)	+	NE
$\vert X \vert$	-	LT
1/X	÷	CONSTANT
SIN	$\sqrt[X]{Y}$	VARIABLE
COS	Y^X	READ
TAN	=	RECORD

The number of times a specific occurrence of a condition is found to be true in a user trace is essentially a loop counter. In the integrated method, during the course of inputting the Fleischmann chain to the Biermann process there are two additional tasks performed. First, a condition counter is kept for each distinct occurrence of a condition. It is incremented each time the distinct occurrence of that condition is traversed and found to be true during input of a chain from the completed Fleischmann process. In addition to incrementi the counter, when encountering a true condition, the expressions of the condition are examined. Recall that the REASON was comprised of storage location/display, relation, and value/storage location. If a value was used the integrated method will attempt to substitute the loop counter (plus or minus a constant if necessary). For example, note that in the second parse of figure 4 the loop counter plus one could be substituted for the value two, thus producing the flow in figure 6. The loop counter is depicted as LC.

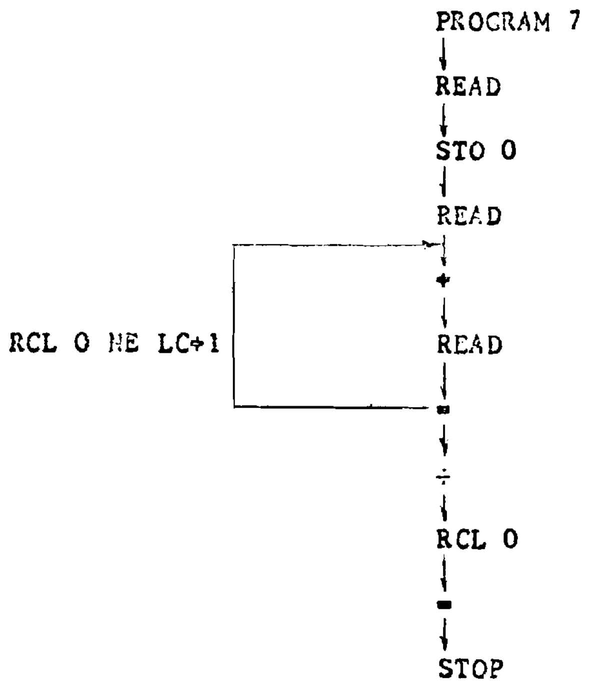


Figure 6. Integrated synthesis after input of two parses of averaging program.

The second task performed in this interface section is to find equivalent conditions. Two conditions are equivalent if they define the same loop and they become identical when an expression involving this loop's counter is substituted for their value portion. In terms of our notation, C_j is equivalent to C_h when they both occur after instruction nI_i (ie. $n.iC_j$ and $n.iC_h$ are equivalent when...), they transfer to instruction mI_k, and differ only in their values which can be substituted by a single expression involving the loop counter.

When two or more conditions are equivalent they control the same loop counter. Returning to the example in figure 6, the loop counter is equal to one after the second parse. When, in the third and final parse, the condition RCL 0 NE 3 is encountered it is found to be equivalent to the previous condition. The previously modified condition, RCL 0 NE LC+1, will suffice. Therefore the program shown in figure 6 is the program capable of averaging any amount of numbers. By using the loop counter as part of the encountered conditions the number of cases that the synthesized program can perform has been extended.

Appendix D demonstrates the integrated method to synthesize a program that will compute factorials. It is shown that two traces, one for two-factorial and one for three-factorial, are sufficient to infer a program capable of computing N-factorial.

Enhancements

The integrated method, as presented so far, can only discover loops that are brought to its attention via trace discrepancies. The user is still responsible for demonstrating the logic of the reiterative concept through proper traces. For loops, a proper trace is composed of a minimal looping example and then a number of more involved examples until the trace discrepancies in these examples have forced the autoprogrammer to ask for all the controlling reasons.

In some cases this method may not be practical or even feasible. For instance, in the prime number generator there is no way to force the autoprogrammer to ask why or how the next prime has been chosen (since, after any trace deviation the remaining trace is taken in as a single branch in the Fleischmann process). For these cases the CMP key as described in Chapter 4 should be added to the integrated method keyboard. This will combine the loop-inference capability of both the Fleischmann and Biermann processes. Therefore this enhancement is effective for inter-example traces whereas the Fleischmann-type questioning is intra-example oriented.

Another enhancement to the method presented would be extending the search for expressions involving the loop counter that replace the value portion of the REASON condition. While using the calculator as the interface medium for an autoprogrammer system it appears that expressions involving the loop counter plus or minus a constant are sufficient.

CHAPTER VI

SUMMARY AND CONCLUSIONS

The purpose of this study was to analyze current autoprogramming techniques and to present feasible methods for implementation of an autoprogrammer system on a calculator or similar programmable device. Three such methods have been described. It has been shown that three forms of program logic (sequential, "If...then...else", and "Do while") are synthesizable using the presented methods. Böhm and Jacopini (1966) have demonstrated that these three basic building blocks are sufficient to form every Turing machine or program.

While developing the methods, characteristics of the calculator were observed to be somewhat deficient for an autoprogrammer system. A discussion of the presented methods, the guidelines for utilization of the proposed autoprogramulators, and a discussion of other approaches to program synthesis using an autoprogrammer-type system follow.

Discussion of Methods

The methods described are approaches to automatic program writing through user supplied examples. In order to develop or synthesize a program from a sequential trace it is necessary to obtain knowledge on conditions affecting the example. The methods differ in how they accept the user supplied examples, in how they obtain additional knowledge needed to form the different types of program logic, and in the types of programs that they can produce.

In Fleischmann's system synthesis is performed by comparing multiple user supplied traces to find inconsistencies. When discrepancies are found, the user is questioned regarding the motivation for such trace deviations. This allows for "If...then...else" logic to be synthesized in addition to sequential logic. However, the condition portion of that logic is obtainable only if the user creates at least two traces to show a logical branch in the concept being conveyed to the machine.

Assuming that the user did supply correct examples, this method presents two major drawbacks. The length of the resulting program grows proportionately to the number of examples put into the synthesis process. The second drawback compounds the first in that this process does not create loops or extend the user supplied examples to more generalized cases. This causes the user to enter examples for every case he wishes the resulting program to be able to handle.

In an attempt to cause loop inferrence and minimize the size of synthesized programs, the Biermann process was adapted to the basic calculator. It provides a means for user stipulation of affecting conditions. This enables the synthesis of all of the basic forms of program logic. The codition portion of the "If...then...else" and "Do while..." logic is user specified, so again the autoprogrammer is dependent on the user declaring all controlling conditions and providing examples of all logic flow.

While Biermann's method is able to produce the types of programs that are desired a price has been paid. The user must state conditions affecting the flow of the program. He is no longer simply supplying examples but instead he treads a fine line between being an uninformed user and a full fledged programmer. All the loops that this process creates are actually user defined. It is desired that submission of simple examples to the system will cause the machine to develop all control structures (or at least question the user in order to obtain them).

The Integrated method was an attempt to relieve the user of the burden of declaring control conditions yet still be able to synthesize all of the basic program logic forms. Because of the "blind" acceptance of new traces it was found that pre-declaring control conditions was still necessary in some cases. Both the Biermann and Integrated systems find minimal programs to execute the supplied examples and also provide extensions to generalized programs. To further decrease program size, a subroutine scanner could be added to scan the finalized program for groups of re-occurring instructions and replace them with "subroutine" calls.

In addition to not completely relieving the need for user declarations of control conditions the Integrated system still can not find all types of conditions that could be desired for the "Do while..." construct. Conditions were either user supplied or involved a compare against the loop-counter plus or minus a constant.

What are the guidelines for use of the autoprogramultor? To effectively utilize the presented methods the user must know his concept and all branches of it. To cause synthesis of a sufficient program he must transmit those aspects of his concept through the example traces. In order to cause loop generation he must show the smallest or simplest possible loop, use an extendable condition for program flow control, and then show increasingly difficult looping until the generalization can take place. For extensions to the general concept the conditions supplied by the user must be consistent with each other and a pattern for the extension must be shown.

It can be argued that the user then has not been successfully relieved from the task of developing programs. On the other hand, once the user learns how to become a teacher for the autoprogramming systems, the systems can be an effective program development tool.

Future Approaches to Autoprogramming

The calculator, although readily available, is limited in its form and methods of communication with the user. Note that for every enhancement to the baseline calculator, additional calculator modifications and devices were required (i.e. english-type questions, keys, and a printer). Other devices should be considered for automatic program synthesis where the user can transmit the desired concept more explicitly and precisely. Requirements-specification languages appear to step in that direction but they still require training of the user.

For implementation of the Integrated system major advances should be developed in two areas (in addition to speed enhancements as suggested in Chapter V). These areas are functions of the loop-counter and trace acceptance in the Fleischmann phase of the user example.

The proposed loop-counter creation to be used in program control is limited to an add or subtract function. The conditions that are under scrutiny for possible equivalence can be thought of as simultaneous equations or incompletely specified functions. Work in these areas regarding the finding or synthesis of suitable functions could possibly be utilized to expand the loop-counter control capability.

After determining that a new branch has been taken in the first (Fleischmann) phase of the Integrated method, all further user input is accepted as sequential constructs. This caused the neccessity of continuing the input of controlling conditions without prompting from the autoprogramming system. A possible solution to this problem would involve using alarger display such as a CRT. When a new branch is taken a loop-watcher could be added to this first phase. At any time the loop-watcher has reason to believe that this new branch may be looping it could display the full instruction loop and ask for user verification. To terminate the looping the user will then be prompted to supply the controlling reason.

Work continues to be needed in the interaction process between user and machine. It has been shown that by asking the proper questions a machine can begin to grasp or learn a concept from the user. Some user interactions have been presented here, and it is hoped that more will follow.

APPENDIX A

ADAPTATION OF FLEISCHMANN'S EXAMPLE OF "HERO'S ALGORITHM FOR THE AREA OF A TRIANGLE"

Adaptation of Fleischmann's (1975, p.73-6) example of "Hero's Algorithm for the Area of a Triangle" to the baseline calculator. Note that the area of a triangle is equal to $\sqrt{px(p-a)x(p-b)x(p-c)}$ where $p=\frac{1}{2}(a+b+c)$.

TABLE 6

FIRST EXAMPLE

Autoprogrammer	User Actions	Display	Comments
	START 9	0	program name & location
	3	3	side a length
(autoprogrammer	STO 0	3	loc 0 = side a length
remains	5	5	side b length
passive	STO 1	5	loc 1 = side b length
during the	7	7	side c length
first example)	STO 2	7	loc 2 = side c length
	+	0	add to c
	RCL 1	5	b
	=	12	b+c
	+	0	add to b+c
	RCL 0	3	a
	=	15	a+b+c
	÷	0	divide a+b+c
	2	2	2
	=	7.5	(a+b+c)/2 (p)
	STO 3	7.5	loc 3 = p
	x	0	multiply p
	RCL 3	7.5	p
	-	0	subtract from p
	RCL 0	3	a
	=	4.5	p-a
	=	33.75	px(p-a)

TABLE 6--Continued

Autoprogrammer	User Actions	Display	Comments
	x	0	multiply px(p-a)
	RCL 3	7.5	p
	-	0	subtract from p
	RCL 1	5	b
	=	2.5	p-b
	=	84.375	px(p-a)x(p-b)
	x	0	multiply px(p-a)x(p-b)
	RCL 3	7.5	p
	-	0	subtract from p
	RCL 2	7	c
	=	0.5	p-c
	=	42.1875	px(p-a)x(p-b)x(p-c)
	$\sqrt{X}$	6.495190528	$\sqrt{px(p-a)x(p-b)x(p-c)}$
	SAVE		store program no. 9

TABLE 7

TRY OF PROGRAM AFTER FIRST EXAMPLE

Autoprogrammer	User Actions	Display	Comments
	EXECUTE 9	6.495190528	system not aware of any variables or branches

TABLE 8

SECOND EXAMPLE

Autoprogrammer	User Actions	Display	Comments
	EXTEND 9	0	extend program no. 9
	5	5	side a length
C OR V		C OR V	constant or variable?
	V	5	user states variable
	STO 0	5	
	12	12	side b length
C OR V		C OR V	
	V	12	
	STO 1	12	
	13	13	side c length
C OR V		C OR V	
	V	13	
	STO 2	13	
	+	0	
	RCL 1	12	
	=	25	
	+	0	
	RCL 0	5	
	=	30	
	÷	0	
	2	2	
	=	15	p
	STO 3	15	
	x	0	
	RCL 3	15	
	-	0	
	RCL 0	5	
	=	10	
	=	150	px(p-a)
	x	0	

TABLE 8--Continued

Autoprogrammer	User Actions	Display	Comments
	RCL 3	15	
	-	0	
	RCL 1	12	
	=	3	
	=	450	px(p-a)x(p-b)
	x	0	
	RCL 3	15	
	-	0	
	RCL 2	13	
	=	2	
	=	900	px(p-a)x(p-b)x(p-c)
	$\sqrt{X}$	30	$\sqrt{px(p-a)x(p-b)x(p-c)}$
	SAVE		store program no. 9

TABLE 9

TRY OF PROGRAM AFTER SECOND EXAMPLE

Autoprogrammer	User Actions	Display	Comments
	EXECUTE 9		execute program no. 9
V		V	asking for variable 1 (side a)
	3	3	
	=		
V		V	asking for variable 2 (side b)
	4	4	
	=		
V		V	asking for variable 3 (side c)
	5	5	
	=		
		6	answer

TABLE 10

EXTENSION OF PROGRAM TO INCLUDE CALCULATION

FOR AREA OF RECTANGLE

Autoprogrammer	User Actions	Display	Comments
	EXTEND 9	0	extend program no. 9
	4	4	side a length
	STO 0	4	
	6	6	side b length
	STO 1	6	
	0	0	side c does not exist
	STO 2	0	
	RCL 0	REASON	
REASON	RCL 2	0	autoprogrammer wants to know why branch taken,
	EQ	.	user indicates that
	0		side c = 0 is the reason
	=	4	so RCL 0 takes effect
(no further autoprogrammer	x	0	multiply a
interaction	RCL 1	6	b
since new branch similar to the	=	24	axb
first example)	SAVE		store program no. 9

TABLE 11

TRY OF PROGRAM AFTER LATEST EXTENSION

Autoprogrammer	User Actions	Display	Comments
	EXECUTE 9		execute program no. 9
V		V	asking for variable 1 (side a)
	10	10	
	=		
V		V	asking for variable 2 (side b)
	50	50	
	=		
V		V	asking for variable 3 (side c)
	0		
	=		
		500	answer

APPENDIX B

EXAMPLES CONCERNING RELATIONSHIP OF TRACES TO THE RESULTING PROGRAM IN THE FLEISCHMANN PROCESS

At least three traces are needed to cause the autoprogrammer to infer a process that is able to compute the average of two, three, or four numbers. Table 12 shows three such sample traces, the resulting program, and their respective lengths.

TABLE 12

SYNTHESIS OF AN AVERAGING PROGRAM

Trace 1	Trace 2	Trace 3	Synthesized Program
START 7	START 7	START 7	EXECUTE 7
2	3	4	*V
STO 0	STO 0	STO 0	STO 0
1	1	1	*V
+	+	+	+
2	2	2	*V
=	=	=	=
÷	+	+	*RCL 0 : 2 NE
RCL 0	3	3	÷
=	=	=	RCL 0
SAVE	÷	+	=
	RCL 0	4	*STOP
	=	=	+
	SAVE	÷	*V
		RCL 0	=
		=	*RCL 0 : 3 NE
		SAVE	÷
			RCL 0
			=
			*STOP
			+
			*V

TABLE 12--Continued

Trace 1	Trace 2	Trace 3	Synthesized Program
			-
			÷
			RCL 0
			-
			*STOP
11 steps	14 steps	17 steps	25 steps

APPENDIX C

AN EXAMPLE OF THE BIERMANN SYNTHESIS METHOD

The following examples demonstrate the Biermann synthesis method. Table 13 shows the sample computation that records all prime numbers up to twelve. Note that the user instructions are labled by type and order of occurrence. For the purposes of this example the baseline calculator includes the unary operator, "FRAC", which strips off the integer portion of the current display.

The second table (Table 14) shows the actual synthesis process in which the occurrence of the labeled instructions are identified, thus creating a flow diagram. Figure 7 is the Moore-type representation or flowchart of the synthesized program. Note that the minimal sample trace needed for this computation was the one shown for N = 12. Inspection of the synthesized program's flowchart shows that this was necessary to traverse the $\{C_2, 3I_{11}, \ldots\}$ logic. The minimal trace, then, was comprised of eighty-one key strokes where the resulting program contains twenty-eight distinct instructions with control and branching logic.

TABLE 13

TRACE OF COMPUTATION THAT RECORDS ALL

PRIME NUMBERS UP TO N, WHERE N=12

Condition	Instruction	User Actions	Display	Comments
		START 8		program name & location
		READ		indicate parameter input
	I_1	12	12	
	I_2	STO 2	12	initialize N to 12
	I_3	1	1	
	I_4	RECORD	1	record automatically
	I_5	2	2	primes 1, 2, & 3
	I_4	RECORD	2	
	I_6	3	3	
	I_4	RECORD	3	
	I_7	+	0	increment last prime by
	I_5	2	2	2, skip even numbers
	I_8	=	5	
	I_9	STO 0	5	
	I_6	3	3	set number to begin divide
	I_{10}	STO 1	3	with initial value of 3
	I_{11}	RCL 0	5	
	I_{12}	÷	0	
	I_{13}	RCL 1	3	
	I_8	=	$1.66\bar{6}$	check for even divisibility
	I_{14}	FRAC	$.66\bar{6}$	
	I_{13}	RCL 1	3	
	I_{15}	X^2	9	are all checks done?
C_1	I_{11}	DISPLAY>RCL 0;RCL 0	5	number is a prime
	I_4	RECORD	5	
	I_7	+	0	increment last prime by
	I_5	2	2	2, skip even numbers
	I_8	=	7	

TABLE 13--Continued

Condition	Instruction	User Actions	Display	Comments
	I_9	STO 0	7	
	I_6	3	3	set number to begin divide
	I_{10}	STO 1	3	with initial value of 3
	I_{11}	RCL 0	7	
	I_{12}	÷	0	
	I_{13}	RCL 1	3	
	I_8	=	$2.33\bar{3}$	check for even divisibility
	I_{14}	FRAC	$.33\bar{3}$	
	I_{13}	RCL 1	3	
	I_{15}	X^2	9	are all checks done?
C_1	I_{11}	DISPLAY>RCL 0;RCL 0	7	number is a prime
	I_4	RECORD	7	
	I_7	+	0	increment last prime by
	I_5	2	2	2, skip even numbers
	I_8	=	9	
	I_9	STO 0	9	
	I_6	3	3	set number to begin divide
	I_{10}	STO 1	3	with initial value of 3
	I_{11}	RCL 0	9	
	I_{12}	÷	0	
	I_{13}	RCL 1	3	
	I_8	=	3	
	I_{14}	FRAC	0	
C_2	I_{11}	DISPLAY=0, RCL 0	9	number not prime,
	I_7	+	0	increment to next
	I_5	2	2	odd number
	I_8	=	11	
	I_9	STO 0	11	
	I_6	3	3	reset number to begin divide
	I_{10}	STO 1	3	with initial value of 3
	I_{11}	RCL 0	11	

TABLE 13--Continued

Condition	Instruction	User Actions	Display	Comments
	I_{12}	÷	0	
	I_{13}	RCL 1	3	
	I_8	=	$3.66\overline{6}$	check for even divisibility
	I_{14}	FRAC	$.66\overline{6}$	
	I_{13}	RCL 1	3	are all checks done?
	I_{15}	x^2	9	
	I_{13}	RCL 1	3	
	I_7	+	0	increment number to divide
	I_5	2	2	with next odd number
	I_8	=	5	
	I_{10}	STO 1	5	
	I_{11}	RCL 0	11	
	I_{12}	÷	0	
	I_{13}	RCL 1	5	
	I_8	=	2.2	check for even divisibility
	I_{14}	FRAC	.2	
	I_{13}	RCL 1	5	are all checks done?
	I_{15}	x^2	25	
C_1	I_{11}	DISPLAY>RCL 0;RCL 0	11	number is a prime
	I_4	RECORD	11	
	I_7	+	0	increment last prime by
	I_5	2	2	2, skip even numbers
	I_8	=	13	
C_3	I_{16}	DISPLAY>RCL 2; SAVE		computation of primes up to 12 complete

TABLE 14

THE SYNTHESIS FLOW FOR THE PRIME NUMBER GENERATING TRACE

Level	Condition	Instruction	Initial Synthesis	Backup Attempt 1st	2nd	3rd	4th	5th	6th	7th	Number of Instructions
1		I_1	$1I_1$								1
2		I_2	$1I_2$								2
3		I_3	$1I_3$								3
4		I_4	$1I_4$								4
5		I_5	$1I_5$								5
6		I_4	$1I_4$	$2I_4$							6
7		I_6	X	$1I_6$							7
8		I_4		$1I_4$	$2I_4$	$3I_4$					8
9		I_7		X	X	$1I_7$					9
10		I_5				$1I_5$	$2I_5$				10
11		I_8				X	$1I_8$				11
12		I_9					$1I_9$				12
13		I_6					$1I_6$	$2I_6$			13
14		I_{10}					X	$1I_{10}$			14
15		I_{11}						$1I_{11}$			15
16		I_{12}						$1I_{12}$			16
17		I_{13}						$1I_{13}$			17
18		I_8						$1I_8$	$2I_8$		18
19		I_{14}						X	$1I_{14}$		19
20		I_{13}							$1I_{13}$	$2I_{13}$	20
21		I_{15}							X	$1I_{15}$	21

TABLE 14 -- Continued

Level	Condition	Instruction	Initial Synthesis	Backup Attempt 7th	8th	9th	10th	11th	Number of Instructions
22	C_1	I_{11}		$1I_{11}$	$2I_{11}$				22
23		I_4		X	$1I_4$	$2I_4$	$3I_4$		
24		I_7			X	X	$(1I_7)$		
25		I_5					$(2I_5)$		
26		I_8					$(1I_8)$		
27		I_9					$(1I_9)$		
28		I_6					$(2I_6)$		
29		I_{10}					$(1I_{10})$		
30		I_{11}					$(1I_{11})$		
31		I_{12}					$(1I_{12})$		
32		I_{13}					$(1I_{13})$		
33		I_8					$(2I_8)$		
34		I_{14}					$(1I_{14})$		
35		I_{13}					$(2I_{13})$		
36		I_{15}					$(1I_{15})$		
37	C_1	I_{11}					$(2I_{11})$		
38		I_4					$(3I_4)$		
39		I_7					$(1I_7)$		
40		I_5					$(2I_5)$		
41		I_8					$(1I_8)$		
42		I_9					$(1I_9)$		
43		[illegible]					[illegible]		

TABLE 14 -- Continued

Level	Condition	Instruction	Initial Synthesis	Backup Attempt 10th	11th	12th	13th	14th	15th	Number of Instructions
44		I_{10}		$(1I_{10})$						
45		I_{11}		$(1I_{11})$						
46		I_{12}		$(1I_{12})$						
47		I_{13}		$(1I_{13})$						
48		I_8		$(2I_3)$						
49		I_{14}		$(1I_{14})$						
50	C_2	I_{11}		$1I_{11}$	$2I_{11}$	$3I_{11}$				23
51		I_7		X	X	$1I_7$				
52		I_5				$(2I_5)$				
53		I_8				$(1I_8)$				
54		I_9				$(1I_9)$				
55		I_6				$(2I_6)$				
56		I_{10}				$(1I_{10})$				
57		I_{11}				$(1I_{11})$				
58		I_{12}				$(1I_{12})$				
59		I_{13}				$(1I_{13})$				
60		I_8				$(2I_8)$				
61		I_{14}				$(1I_{14})$				
62		I_{13}				$(2I_{13})$				
63		I_{15}				$(1I_{15})$				
64		I_{13}				$1I_{13}$	$2I_{13}$	$3I_{13}$		24
65		I_7				X	X	$1I_7$	$2I_7$	25

TABLE 14 -- Continued

Level	Condition	Instruction	Initial Synthesis	Backup Attempt 14th	15th	16th	17th	18th	19th	Number of Instructions
66		I_5		$(2I_5)$	$1I_5$	$2I_5$	$3I_5$			26
67		I_8		$(1I_8)$	X	$(1I_8)$	$1I_8$	$2I_8$	$3I_8$	27
68		I_{10}		X		X	X	X	$1I_{10}$	
69		I_{11}							$(1I_{11})$	
70		I_{12}							$(1I_{12})$	
71		I_{13}							$(1I_{13})$	
72		I_8							$(2I_8)$	
73		I_4							$(1I_{14})$	
74		I_{13}							$(2I_{13})$	
75		I_{15}							$(1I_{15})$	
76	C_1	I_{11}							$(2I_{11})$	
77		I_4							$(3I_4)$	
78		I_7							$(1I_7)$	
79		I_5							$(2I_5)$	
80		I_8							$(1I_8)$	
81	C_3	I_{16}							$1I_{16}$	28

X indicates a contradiction
() indicates a forced move

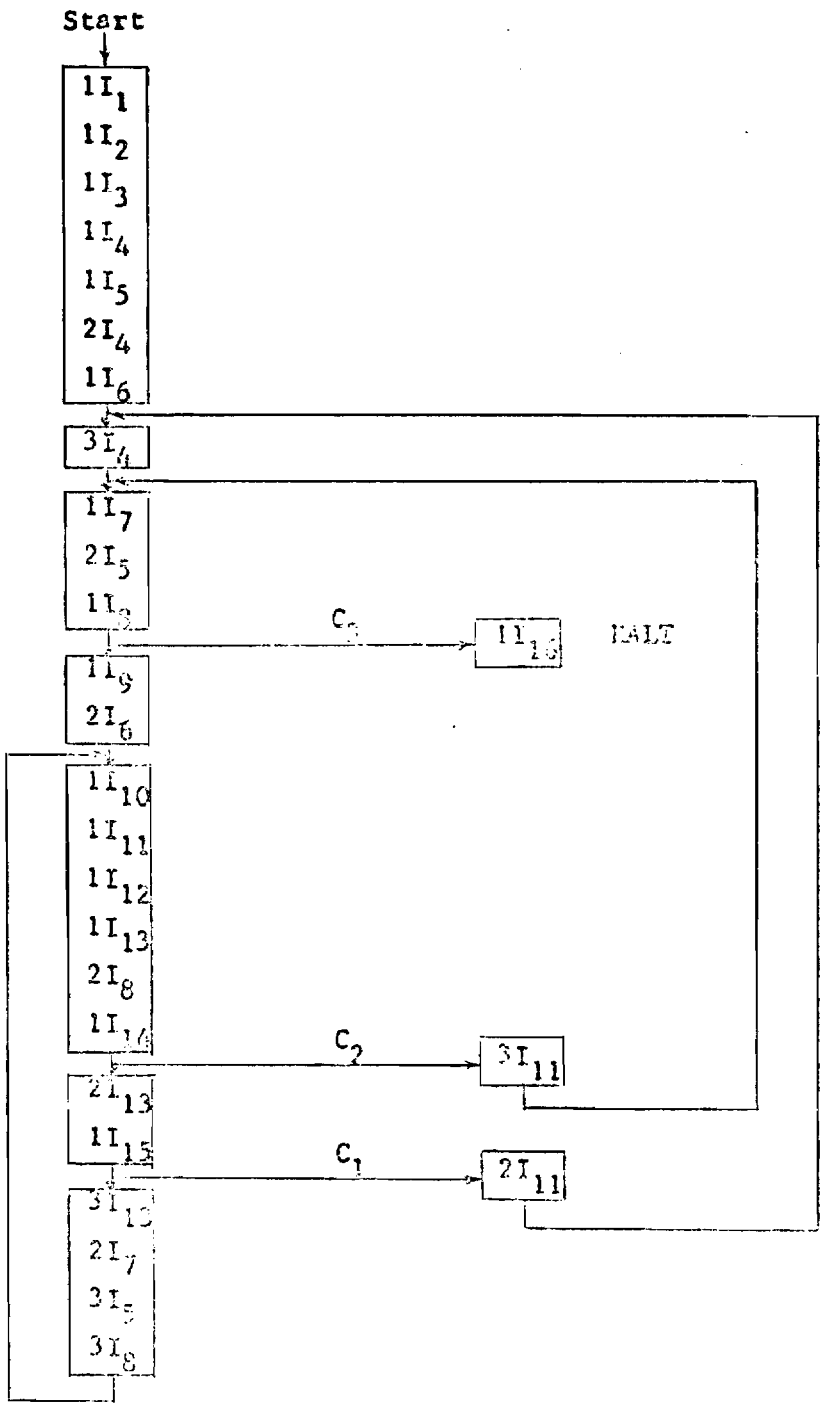


Figure 7. Moore-type representation of synthesized prime number generator

APPENDIX D

AN EXAMPLE OF THE INTEGRATED SYNTHESIS METHOD

FOR A FACTORIAL PROGAM

TABLE 15

PHASE ONE OF FACTORIAL PROGRAM SYNTHESIS (FLEISCHMANN PHASE)

Trace 1	Trace 2	Synthesized Tree	Instruction
START 4	EXTEND 4	(before synthesis)	
2	3	*V	I_1
STO 0	STO 0	STO 0	I_2
-	-	-	I_3
1	1	1	I_4
=	=	=	I_5
STO 1	STO 1	STO 1	I_6
x	x	x	I_7
RCL 0	RCL 0	RCL 0	I_8
=	=	=	I_5
HALT	STO 0	*RCL 1 : 1 $\xrightarrow{NE}$	
	RCL 1	HALT	I_9
	-	STO 0	I_2
	1	RCL 1	I_{10}
	=	-	I_3
	STO 0	1	I_4
	x	=	I_5
	RCL 0	STO 1	I_6
	=	x	I_7
	HALT	RCL 0	I_8
		=	I_5
		HALT	I_9

TABLE 16

PHASE TWO OF FACTORIAL PROGRAM SYNTHESIS (BIERMANN PHASE)

Level	Condition	Instruction	Initial Synthesis	1st Backup Attempt	2nd Backup Attempt
1		I_1	$1I_1$		
2		I_2	$1I_2$		
3		I_3	$1I_3$		
4		I_4	$1I_4$		
5		I_5	$1I_5$		
6		I_6	$1I_6$		
7		I_7	$1I_7$		
8		I_8	$1I_8$		
9		I_5	$1I_5$	$2I_5$	
10	none	I_9	X	$1I_9$	
10'	$1.9C_1$	I_2		$1I_2$	$2I_2$
11		I_{10}		X	$1I_{10}$
12		I_3			$1I_3$
13		I_4			$(1I_4)$
14		I_5			$(1I_5)$
15		I_6			$(1I_6)$
16		I_7			$(1I_7)$
17		I_8			$(1I_8)$
18		I_5			$(2I_5)$
19		I_9			$(1I_9)$

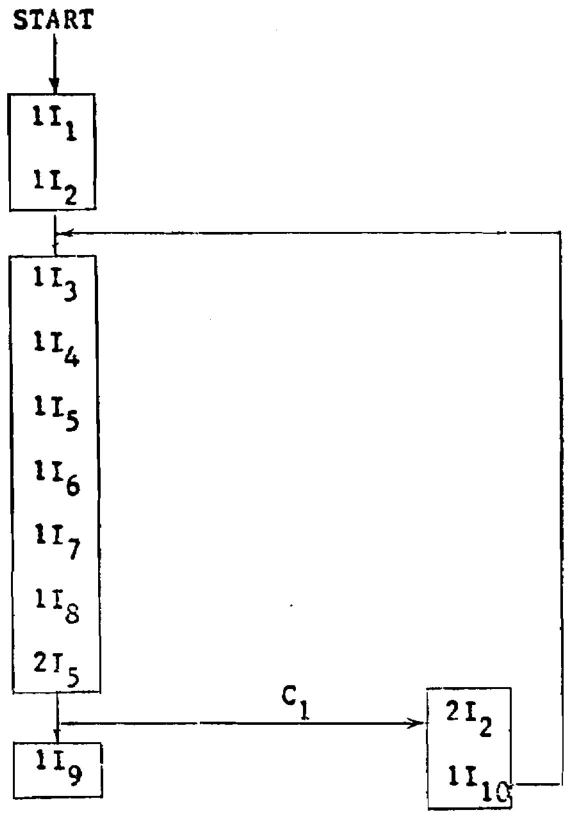


Figure 8. Moore-type representation of factorial program

BIBLIOGRAPHY

Amarel, S, "On the Automatic Formation of a Computer Program which Represents a Theory." Self Organizing Systems Edited by Jacobi, Yovits, and Goldstein. New York: Spartan, 1962, pp. 107-75.

________, "Representations and Modeling in Problems of Program Formation." Machine Intelligence Edited by Meltzer and Michie. New York: American Elsevier 6 (1971):441-66.

Bauer, Michael A., "A Basis for Aquisition of Procedures from Protocols." Advanced Papers of the 4th International Joint Conference on Artificial Intelligence Tbtlisc, Georgia, U.S.S.R., (September 1976):226-31.

Biermann, A. W., "Approaches to Automatic Programming." Advances in Computers Edited by Yovits and Rubinoff. New York:Academic, 15 (1976):1-63.

________, "On the Inferrence of Turing Machines from Sample Computations." Artificial Intelligence 3,(1972):181-98.

Biermann, A. W.; Baum. R. I.; and Petry, F. E., "Speeding Up the Synthesis of Programs from Traces." IEEE Transactions on Computers c-24, (February 1975):122-36.

Biermann, A. W.; Feldman, J. A., "On the Synthesis of Finite State Acceptors." Stanford Artificial Intelligence Project Memo AIM-114 (April 1970).

Biermann, A. W. and Krishnaswamy, R., "A System for Program Synthesis From Examples." 1974 Proceedings IEEE System, Man and Cybernetics Conference Dallas, Texas (October 1974):514-18.

________, "Constructing Programs from Example Computations." IEEE Transactions on Software Engineering SE-2, no. 3, (September 1976): 141-53.

Bohm, Corrado and Jacopini, Guiseppe, "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules." Communications of the ACM 9 (May 1966):366-71.

Carr III, J. W., "Inference of Computer Programs from Traces." September 1976. (Typewritten notes).

________, "Programs and Their Traces." September 1976. (Typewritten notes).

Fleischmann, Rochelle, "A Proposal for Example Induced Programming." M. S. E. Thesis, The Moore School of Engineering, University of Pennsylvania, April 1975.

Hopcroft, J. E. and Ullman, J. D., "Formal Languages and Their Relation to Automata." Reading, Massachusetts: Addison-Wesley, 1969.

Lee, R. T. C.; Chang, C. L.; and Waldinger, R. J., "An Improved Program Synthesizing Algorithm and its Correctness." Communications of the ACM 17 (April 1974):211-17.

Texas Instruments, "Programmable Slide-Rule Calculator SR-56." 1976 (owners manual).