DOCUMENT RESUME

ED 152 327 IR 005 874

AUTHOR	Bierman, A. W.; Krishnaswamy, R.
TITLE	Constructing Programs from Example Computations.
INSTITUTION	Ohio State Univ., Columbus. Computer and Information Science Research Center.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	OSU-CISRC-TR-74-5
PUB DATE	Aug 74
GRANT	GJ-34739X
NOTE	46p.
EDRS PRICE	MF-$0.83 HC-$2.06 Plus Postage.
DESCRIPTORS	*Computer Programs; *Display Systems; Electromechanical Aids; *Input Output Devices; *Man Machine Systems; On Line Systems; *Programing; Programing Languages
IDENTIFIERS	*Autoprogrammer

ABSTRACT

This paper describes the construction and implementation of an autoprogramming system. An autoprogrammer is an interactive computer programming system which automatically constructs computer programs from example computations executed by the user. The example calculations are done in a scratch pad fashion at a computer display, and the system stores a detailed history of all of the steps executed in the process. The system then automatically synthesizes the shortest possible program which is capable of executing the observed examples. Various sections of the report describe (1) the system, (2) its users, (3) the computational environment, (4) basic formalisms, (5) the program synthesis system, (6) convenience features, shortest possible and (7) programming details. (Author/DAG)

DOCUMENT RESUME

ED 152 327 IR 005 874

AUTHOR	Bierman, A. W.; Krishnaswamy, R.
TITLE	Constructing Programs from Example Computations.
INSTITUTION	Ohio State Univ., Columbus. Computer and Information Science Research Center.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	OSU-CISRC-TR-74-5
PUB DATE	Aug 74
GRANT	GJ-34739X
NOTE	46p.
EDRS PRICE	MF-$0.83 HC-$2.06 Plus Postage.
DESCRIPTORS	*Computer Programs; *Display Systems; Electromechanical Aids; *Input Output Devices; *Man Machine Systems; On Line Systems; *Programing; Programing Languages
IDENTIFIERS	*Autoprogrammer

ABSTRACT

This paper describes the construction and implementation of an autoprogramming system. An autoprogrammer is an interactive computer programming system which automatically constructs computer programs from example computations executed by the user. The example calculations are done in a scratch pad fashion at a computer display, and the system stores a detailed history of all of the steps executed in the process. The system then automatically synthesizes the shortest possible program which is capable of executing the observed examples. Various sections of the report describe (1) the system, (2) its users, (3) the computational environment, (4) basic formalisms, (5) the program synthesis system, (6) convenience features, shortest possible and (7) programming details. (Author/DAG)

ABSTRACT

This paper describes the construction and implementation of an autoprogramming system. An autoprogrammer is an interactive computer system which accepts as input example calculations, and which yields computer programs for doing these calculations.

PREFACE

This paper describes an autoprogramming system which constructs executable computer programs from example computations. Messrs. Richard Baum and Frederick Petry have been in charge of developing our synthesis algorithms. We are greatly indebted to Mr. Serge Fournier of our PDP-10 staff for help with our systems programming problems. This work was done at the Computer and Information Science Research Center and was supported by the National Science Foundation Grant No. GJ-34739X and by the Department of Computer and Information Science, The Ohio State University.

The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report presents research accomplished in cooperation with the Department of Computer and Information Science. The research was administered and monitored by The Ohio State University Research Foundation.

3

ABSTRACT

This paper describes the construction and implementation of an autoprogramming system. An autoprogrammer is an interactive computer system which accepts as input example calculations, and which yields computer programs for doing these calculations.

TABLE OF CONTENTS

1. INTRODUCTION

An autoprogrammer is an interactive computer programming system which automatically constructs computer programs from example computations executed by the user. The example calculations are done in a scratch pad fashion at a computer display using a light pen or other graphic input device, and the system stores a detailed history of all of the steps executed in the process. Then the system automatically synthesizes the shortest possible program which is capable of executing the observed examples.

The autoprogramming concept as a program construction technique attempts to divide the responsibilities of man and machine as optimally as possible giving the man the creative tasks of choosing the data structures and furnishing the algorithm while the machine produces the actual code of the program. The user works in the familiar domain of concrete examples as he pushes the information around in the data structures by hand. He does not need to mentally visualize the effects of his instructions since they take place on the screen before his eyes. The code created by the machine is guaranteed to precisely mimic the actions of the user in his examples. Language syntax in the traditional sense is totally absent from the user's point of view except for the correct ordering of the graphic inputs.

This work is aimed at the development of a simple, reliable, effective, and convenient program synthesizer. Features will be described here which help the user correctly complete his examples, which enable him to be somewhat carefree about the style of his inputs, and which enable him to find and correct program errors by dealing with the effects of the program rather than the program itself. It is assumed that the user will change his mind often during the synthesis process, that he will want to add and delete data structures at unpredictable times, that he will make mistakes

6

in his examples that must be corrected, that he will want to call subroutines that have not yet been created, and that he may provide information in a fragmentary manner. The system described here allows for all of these possibilities without losing its basic simplicity of design.

The next section describes the computational environment provided by an autoprogrammer within which the user can execute his examples. Section 3 will introduce the basic formalisms to be used in this paper. In Section 4, we will describe the program synthesis system and show that it is both sound and complete in the following senses: we can guarantee that a synthesized program will correctly execute the given examples (soundness) and that every possible program (or its equivalent) can be created by our system (completeness). Section 5 will give some of the features which can be built into an autoprogrammer to increase its convenience and will discuss how they are incorporated into the total design. Section 6 will discuss some programming details of our current system and will give some programs that it was used to create.

2. DOING THE EXAMPLES

An example calculation begins with a declaration of the name of the routine to be created and any parameter inputs to be included with its call. Then the data structures which are to appear on the screen are declared. On our current system, these declarations are made at the teletype although they could be input graphically as are most other commands. The declarations include not only arrays and variables but also pointers into arrays. That is, if I has value 3 and is listed as a pointer into linear array A, then an arrow labeled I will point to the third location in A. We can refer graphically to the location A(I) by touching the pointer and to location A(3) by touching the actual location A(3). This usage will become clear in the example to follow.

Once the data structures have appeared on the screen, one may begin the sample calculation using the graphic input device. Probably the best such device for autoprogramming would be a touch sensitive surface on the display screen, but on our current system we have used a light pen. We suspect that a touch sensitive screen would yield a much better system because it would accept inputs at a much higher rate and would allow the programmer to use both hands. In any case, we will refer to one graphic input, one pair of x,y-coordinates, as a <u>touch</u> or a <u>hit</u>.

The commands to the system are indicated by a touch sequentially of the command name which appears on the screen and each of its operands. Let p_i stand for the i-th graphical hit after the command is designated. That is, suppose we touch sequentially the instruction <u>move</u> followed by I and J. Then p_1=I, p_2=J, and by the definition given below, the contents of I will be loaded into J.

We will need eight commands in our forthcoming example:

start - the first instruction in any program.

move - $p_2 \leftarrow p_1$

$+$ - $p_2 \leftarrow p_1+p_2$ or if there are three operands: $p_3 \leftarrow p_1+p_2$

$-$ - $p_2 \leftarrow p_2-p_1$ or if there are three operands: $p_3 \leftarrow p_2-p_1$

subst - this is a special operator invented for the purpose of this example: apply the grammatical rule with left hand side at p_1 and right hand side at p_2 to string p_3 at location p_4 and put the result into p_5. Thus if p_1 references BC, p_2 references XYZ, p_3 references ABCDE, and $p_4 = 2$, then AXYZDE will be entered into location p_5. (Typically, a subroutine would be synthesized to do this task rather than creating such an operator.)

length - yields the length of string p_1.

print - types out on the teletype the string p_1.

halt - ends execution of the routine and returns control to the calling routine.

It is also necessary to indicate to the system when a condition is being checked. For example, in a sorting routine we note that two items are out of order before we exchange them: <u>note</u> A(I)>A(J). So for checking conditions, we have the relations =, >, and < available with the usual definitions* and <u>terminal</u> which is defined for the purposes of the following example. The predicate <u>terminal</u> yields a value of true if and only if its operand p_1 has all terminal symbols as defined below.

Let us suppose that we wish to create a program called GENERATOR which generates and prints all of the terminal strings that can be produced by N or less applications of rules of an arbitrary grammar starting from a given initial string. The algorithm will be to generate all possible immediate

*If X and Y are strings of different length and the shortest one has length i, then X=Y if the first i characters of X and Y are identical.

successors of the initial string and to store them on a stack. Then it will load the top string from the stack, generate all of its immediate successors and add them to the stack, and so forth. Terminal strings will be immediately printed and deleted from the stack as they are generated, and nonterminal strings resulting from N applications of rules will be deleted to insure termination of the computation.

The autoprogrammer will need an example computation from which to construct the program and we will choose the grammar {BA→BBA, ABA→a} using initial string ABA and searching to a depth of N = 2. The nonterminals in this grammar are A and B, and the only terminal symbol is a. The data structures will be:

STRING which holds the current string being processed,

LEVEL which gives the depth of generation of STRING,

N as defined above,

LEFT and RIGHT to hold the left and right sides of the grammatical rules,

NORULES to hold the number of rules in the grammar,

STRSTACK and LEVSTACK to hold the stored strings and their levels, and the pointers P, I, and J.

Figure 1 shows how these structures will appear on the screen after they are declared. P is a special substring-pointer which references all of the contents of STRING from the p-th character onwards. Thus, if P = 3 and STRING = "ABCDE", then STRING(P) = "CDE".

The calculation proceeds as shown in Figure 2 where the commands are given in the leftmost column and their results in the major data structures are indicated to the right. Thus, the first hit is the start instruction, the second is move, the third is the literal 0 at the bottom of the screen, the fourth is J, and so forth. Scanning down the figure, one can see the pointer P being advanced across STRING searching for an application of rule 1 of the

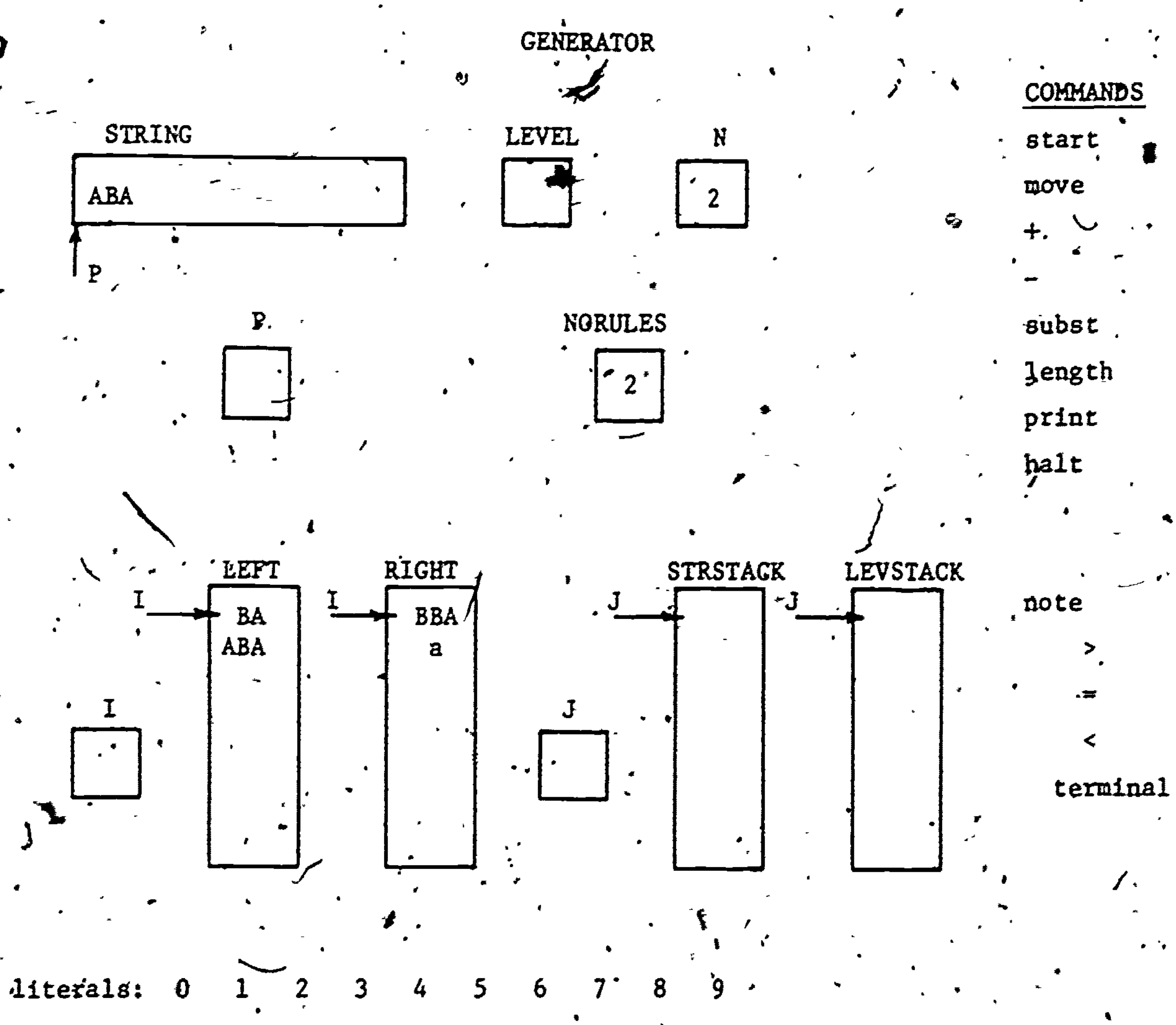


FIGURE 1. The autoprogrammer screen before the sample calculation begins.

	Instruction	STRING	LEVEL	I	STRSTACK	LEVSTACK	
1	start	ABA	0		ϕ	ϕ	
2	move 0 LEVEL						
3	move 0 J						
4	move 1 I			1			
5	move 0 P	_ABA					
6	+ 1 P	ABA					
7	+ 1 P	ABA					
8	note LEFT(I) = STRING(P)						
9	+ 1 J						
10	subst LEFT(I) RIGHT(I) STRING P STRSTACK(J)				ABBA		
11	+ 1 LEVEL LEVSTACK(J)					1	
12	+ 1 P	ABA					
13	note length(LEFT(I))>length(STRING(P))						
14	+ 1 I			2			
15	move 0 P	_ABA					
16	+ 1 P	ABA					
17	+ 1 J						*
18	subst LEFT(I) RIGHT(I) STRING P STRSTACK(J)				a,ABBA		*
19	+ 1 LEVEL LEVSTACK					1,1	*
20	note terminal STRSTACK(J)						*
21	print STRSTACK(J)						
22	- 1 J				ABBA	1	
23	+ 1 P	ABA					
24	+ 1 I			3			
25	note I>NORULES						*
26	move STRSTACK(J) STRING	ABBA					
27	move LEVSTACK (J) LEVEL		1				
28	- 1 J				ϕ	ϕ	
29	move 1 I			1			
30	move 0 P	_ABBA					
31	+ 1 P	ABBA					*
32	+ 1 P	ABBA					*
33	+ 1 P	ABBA					*
34	+ 1 J						*
35	subst LEFT(I) RIGHT(I) STRING P STRSTACK(J)				ABBBA		*
36	+ 1 LEVEL LEVSTACK(J)					2	*
37	note LEVSTACK(J) = N						*
38	- 1 J				ϕ	ϕ	
39	+ 1 P	ABBA					
40	+ 1 I			2			
41	move 0 P	_ABBA					*
42	+ 1 P	ABBA					*
43	+ 1 P	ABBA					*
44	+ 1 P	ABBA					*
45	+ 1 I			3			*
46	note J = 0						*
47	halt						

FIGURE 2. The steps of an example calculation: generating terminal strings from a grammar.

* Instruction is inserted automatically using the continue feature of Section 5.

grammar. In step 7, we discover rule 1 can be applied which yields the string ABBA in step 9. Then the second rule of the grammar is applied yielding string a which is printed out. Finally string ABBA is brought in from the stack and its successors are generated in the search for a terminal string. The halt instruction terminates the calculation.

Of course, in actual practice, the user never sees anything like Figure 2, and his total experience is with the display of Figure 1 and the movement of information from place to place. We have found that a programmer can execute a surprisingly long sequence of steps without er[illegible]has the method well in mind. However, such long sequences are almost never necessary as will be shown in later sections.

The careful reader will observe that the condition (LEFT(I) = STRING(P)) of step 7 should have also been noted immediately after step 15 and immediately after step 32. In fact, there are other places in the calculation where conditions were omitted. The rule is that if every condition is properly inserted at least once in the calculation, the synthesis technique properly constructs the program.

After one or several example calculations are complete, the program is synthesized as described in the following sections.

3. BASIC DEFINITIONS

Before it is possible to define the synthesis method and study its properties, it is necessary to introduce some notation. A computation will be thought of as a sequence of steps with the instructions i_t being executed at the discrete times $t = 1,2,3,\ldots$. m_t for $t = 0,1,2,\ldots$ will designate a complete description of the computer memory immediately before instruction i_{t+1} has been performed. Thus, instruction i_t will operate on memory contents m_{t-1} to yield m_t which may be written in functional notation as $m_t = i_t\ (m_{t-1})$. Actually $i_t(m_{t-1})$ may yield many different results since i_t might be, for example, a read instruction so we prefer to write $m_t \varepsilon\ i_t\ (m_{t-1})$. Referring to the above example in Figure 2, i_1 = start, i_2 = move δ J, and so forth. m_o, m_1, ... may be thought of as sequential photographs of the displayed data structures as the computation progresses.

The symbol a will designate an atomic predicate or <u>atom</u> with value true or false which is measurable by the machine for the purpose of making branching decisions. "A(I) > A(J)" and "LEFT(I) = STRING(K)" are examples of atoms taken from the previous section. A <u>signed</u> atom will be either an atom or or a negated atom $\neg$ a. A condition c_t is a predicate which is a (possibly empty) conjunction of atoms and/or their negations. c_t will be represented as a set of signed atoms but we will also use a functional notation $c_t(m_{t-1})$ which will have value true if and only if all of its unnegated atoms applied to m_{t-1} are true and all of its negated atoms applied to m_{t-1} are false. If c_t is the empty set ϕ, its value is true.

$r_t = (c_t, i_t)$ is a condition-instruction pair executed at time t. That is, at time t condition c_t was observed to be true and then instruction i_t was executed. A computation may thus be visualized as a sequence of memory snapshots separated by condition-instruction pairs:

$$m_o \xrightarrow{r_1} m_1 \xrightarrow{r_2} m_2 \xrightarrow{r_3} \cdots\cdots \xrightarrow{r_n} m_n$$

Of course, many of the conditions c_t will be the trivial empty condition.

A partial trace T of a computation will be defined as the (2n+1)-tuple

$T = (m_o, r_1, m_1, r_2, m_2, \ldots, r_n, m_n)$

where for each t = 1,2,3,...n we have

$r_t = (c_t, i_t)$,

$m_t \in i_t (m_{t-1})$,

$c_t(m_{t-1})$ is true, and $c_1 = \phi$.

The instructions available in the autoprogramming language will be denoted $I_o, I_1, I_2, \ldots, I_z$, and I_H where I_o is a do-nothing start instruction and I_H is the halt instruction. Every program will have exactly one occurrence of I_o and usually one occurrence of I_H. A trace will be a partial trace $T = (m_o, r_1, m_1, r_1, \ldots, r_n, m_n)$ with the additional requirements that $r_1 = (\phi, I_o)$ and $r_n = (c_n, I_H)$. A particular instruction, say I_6 = move R S, may occur many times in the same program so that it will be necessary to label each such occurrence separately. We will do this by concatenating an integer prefix to the instruction name so that, for example, three occurrences of I_6 would be designated $1I_6$, $2I_6$, and $3I_6$. These will be called labeled instructions and the positive integer prefix k will be called the label.

An incomplete program P is a finite set of triples of the form (q_j, c_k, q_e) where each q_j and q_e is a labeled instruction and c_k is a condition and where the following restriction holds:

If $(q, c, q') \in P$ and $(q, c', q'') \in P$ and there exists m such that

$c(m) = c'(m)$ = true, then $c = c'$ and $q' = q''$.

Thus an incomplete program is a finite set of labeled instructions connected by triples or transitions which are each associated with a particular condition.

A transition is taken if its condition is true, and no two applicable transitions can ever be simultaneously satisfied. An example of this Moore machine type representation appears in Figure 3. This program is called incomplete because there is no start instruction I_o and because the transition $\{\neg a\}$ out of state $2I_1$ is missing.

Now we define an operator B which takes as arguments an incomplete program P and an instruction I:

$$B(P,I)=\{a \mid (jI,c,q)\varepsilon P \text{ for some } j,c, \text{ and } q \text{ and } a\varepsilon c \text{ or } \neg a\varepsilon c\}$$

B(P,I) is the set of all atoms which are observed on transitions leading away from I in program P. Assuming that $B(P,I) = \{a_1,a_2,\ldots,a_k\}$, then another operator B' is defined as the set of all <u>minterms</u> that can be constructed from these atoms:

$$B'(P,I) = \{\{a_1,a_2,a_3,\ldots,a_k\},\{a_1,a_2,\ldots,\neg a_k\},\ldots,\{\neg a_1,\neg a_2,\ldots,\neg a_k\}\}.$$

Note that B'(P,I) may be empty.

A <u>program</u> P will be an incomplete program with the additional requirements that

(1) for all I (where $I \neq I_H$) such that $(jI,c,q)\varepsilon P$ for some j, c, and q, $\cup\{c \mid (jI,c,q)\varepsilon P\} = B'(P,I)$ for each such j, and

(2) there is exactly one start instruction, namely $1I_o$, and $(1I_o,c,q)\varepsilon P$ for some c and q.

The first requirement means that every minterm in B'(P,I) must be represented in a transition out of every occurrence of I. Therefore, after any instruction I in the program is executed, there will be exactly one transition condition satisfied to a next instruction until the halt is reached. The second requirement asserts that there must be exactly one start instruction. An example of a program can be constructed if the transitions $(1I_o,\phi,1I_1)$ and $(2I_1,\{\neg a\},1I_2)$, are added to the incomplete program of Figure 3.

Before introducing the synthesis algorithm, it will be helpful to broaden the above definition of B so that it can operate on a set S of partial traces.

$B(S,I) = \{a \mid T \epsilon S,\ I = i_t$ for some i_t in trace T and $a \epsilon c_{t+1}$ or $\neg\, a \epsilon c_{t+1}$ in T$\}$.

Here B(S,I) is the set of all atoms which are observed in conditions following I in a trace T in S. Consistent with the previous definitions, if $B(S,I) = \{a_1, a_2, \ldots, a_k\}$ then define $B'(S,I) = \{\{a_1, a_2, \ldots, a_k\}, \{a_1, a_2, \ldots, \neg\, a_k\}, \ldots, \{\neg\, a_1, \neg\, a_2, \ldots, \neg\, a_k\}\}$.

4. THE PROGRAM SYNTHESIS ALGORITHM

The synthesis algorithm will be defined in terms of four operators, Q_1, Q_2, Q_3, and Q_4. Let S be a set of partial traces; we will define $Q_1(S)$ to be another set of partial traces as follows: if $T=(m_o,(c_1,i_1),m_1,\ldots,(c_n,i_n),m_n)$ is in S, then $T'=(m_o,(c_1',i_1),m_1,\ldots,(c_n',i_n),m_n)$ is in $Q_1(S)$ where $c_1' = \phi$ and for $t = 2,3,\ldots,n$, $c_t' \epsilon B'(S,i_{t-1})$ and $c_t'(m_{t-1})$ is true. (If $B'(S,i_{t-1}) = \phi$, then $c_t' = \phi$.) Nothing else is in $Q_1(S)$. Notice that c_t is uniquely defined since there can be only one minterm c_k' in $B'(S,i_{t-1})$ with the property that $c_t'(m_{t-1})$ is true.

Q_1 is the operation which inserts into each trace all conditions which may have been omitted by the user. Examining the trace T of Figure 2, one sees that the atoms I>NORULES and J=0 can immediately follow the instruction + 1 I. Thus

$$B(\{T\}, + 1\ I) = \{I>NORULES,\ J=0\}$$

$$B'(\{T\}, + 1\ I) = \{\{I>NORULES,\ J=0\},\ \{I>NORULES, \neg J=0\},\ \{\neg I>NORULES,\ J=0\},\ \{\neg I>NORULES, \neg J=0\}\}.$$

$Q_1(\{T\})$ is a trace similar to the one in Figure 2 except that one of the four minterms in $B'(\{T\}, + 1\ I)$ will appear after every occurrence of + 1 I, and certain other conditions will be similarly inserted after other instructions.

Let g be a function which puts an order on a set of partial traces. For example, g(S) may be the set S of partial traces ordered into the sequence in which they were received. If $g(S) = T_1,T_2,\ldots,T_k$ is the ordered set of partial traces $T_j = (m_o^{(j)},r_1^{(j)},m_1^{(j)},r_2^{(j)},\ldots,r_{n_j}^{(j)},m_{n_j}^{(j)})$, $j = 1,2,\ldots,k$, then define f(g(S)) to be the $(2(n_1+n_2+n_3+\ldots+n_k)+2k-1)$-tuple

$$f(g(S)) = (m_0^{(1)}, r_1^{(1)}, \ldots\ldots, r_{n_1}^{(1)}, m_{n_1}^{(1)}, d$$

$$m_0^{(2)}, r_1^{(2)}, \ldots\ldots, r_{n_2}^{(2)}, m_{n_2}^{(2)}, d$$

$$- - - - - - - - - - -$$

$$m_0^{(k)}, r_1^{(k)}, \ldots\ldots, r_{n_k}^{(k)}, m_{n_k}^{(k)})$$

where d is called a dummy transition and is distinct from all other symbols in the formalism. Then f(g(S)) is one long partial trace with all of the partial traces of S concatenated together and separated by dummy transitions d.

Let $T = (m_0, r_1, m_1, r_2, \ldots\ldots, r_n, m_n)$ be a partial trace which may be made up of a concatenation of several traces, and let U be an n-tuple of positive integers $U = (u_1, u_2, \ldots\ldots, u_n)$. Then

$$Q_2(T,U) = \{(u_j i_j, c_{j+1}, u_{j+1} i_{j+1}) \mid r_j \neq d,$$
$$r_j = (c_j, i_j),\ r_{j+1} \neq d,$$
$$r_{j+1} = (c_{j+1}, i_{j+1}),\ u_j \text{ and}$$
$$u_{j+1} \text{ are in } U = (u_1, u_2, \ldots\ldots, u_n), \text{ and}$$
$$T = (m_0, r_1, m_1, \ldots\ldots, r_n, m_n)\}$$

$Q_2(T,U)$ is a set of triples which constitute an incomplete program if U is chosen properly. U is the set of labels which will be applied to the instructions in trace T in the synthesis of the program. An example n-tuple that would work is U = (1,2,3,....,n) which yields a linear program with no branching. Using this U and the trace of Figure 2, one can begin constructing $Q_2(T,U)$: (1 start, ϕ, 2 move 0 J), (2 move 0 J, ϕ, 3 move 1 I), etc. The purpose of Q_3 will be to find a program which is more interesting than this linear one.

We will need a function h which counts the number of instances of instructions in a program. Define |S| to be the cardinality of the set S, and let Z be a set of triples.

$h(Z) = \left| \{x \mid \exists y \exists z((x,y,z) \in Z \text{ or } (y,z,x) \in Z)\} \right|$

Thus, if Z is a set of triples representing a program P, then h(Z) is the number of different instances of instructions in P.

If $U = (u_1, u_2, \ldots\ldots, u_n)$ and $U' = (u_1', u_2', \ldots\ldots, u_n')$ are two integer n-tuples then we define $U<U'$ if there is a j, $1 \leq j \leq n$, such that $u_1 = u_1'$, $u_2 = u_2', \ldots\ldots, u_{j-1} = u_{j-1}'$, $u_j < u_j'$. Let k and k' be integers and U and U' be n-tuples, then we define $(k,U)<(k',U')$ if $k<k'$ or if $k=k'$ and $U<U'$. This puts an ordering on a set of such pairs (k,U) and allows us to speak of a minimum.

Define (k_S, U_S) to be the minimum pair (k,U) with the properties that $k = h(Q_2(f(g(Q_1(S))),U))$ and $Q_2(f(g(Q_1(S))),U)$ is an incomplete program. Define

$Q_3(S) = Q_2(f(g(Q_1(S))),U_S)$ which is the desired incomplete program. Intuitively, one enumerates the set of pairs (k,U) in increasing order until one is found such that $Q_2(f(g(Q_1(S))),U)$ is an incomplete program. Most of the possible values for (k,U) will yield a nondeterminism in the flow of control thus violating the definition of an incomplete program. The enumeration will certainly halt somewhere because there always exists a trivial solution (n,(1,2,3,.....,n). A pseudo-program for computing $Q_3(S)$ might look something like this:

```
for k = 1 step 1 until infinity do
    for each U≠(1,2,3,.....,n) such that h(Q2(f(g(Q1(S))),U) = k do
        if Q2(f(g(Q1(S))),U) is an incomplete program then
            halt and return Q3(S)=Q2(f(g(Q1(S))),U);
```

This program will never enter an infinite calculation on any given value of k because there are only a finite number of n-tuples U which satisfy $U \le (1,2,3,\ldots,n)$. The art of performing this calculation efficiently is discussed in some detail in [3] and will not be further considered here. For most programs of the size and complexity considered in this paper, this calculation can be completed in less than one hundred milliseconds.

We will review the above synthesis process by doing a simple example. Suppose a calculation is performed with the instruction sequence $I_1, I_1, I_2, I_1, (\neg a), I_H$. Then the partial trace is

$$T = (m_o, (\phi, I_1), m_1, (\phi, I_1), m_2, (\phi, I_2), m_3, (\phi, I_1), m_4, (\{\neg a\}, I_H), m_5).$$

If $S = \{T\}$, then $B(S,I_1) = \{a\}$ and $B'(S,I_1) = \{\{a\}, \{\neg a\}\}$. Now assume that $a(m_1)$ and $a(m_2)$ are true. Q_1 inserts all applicable minterms into T.

$$Q_1(S) = \{(m_o, (\phi, I_1), m_1, (\{a\}, I_1), m_2, (\{a\}, I_2), m_3, (\phi, I_1), m_4, (\{\neg a\}, I_H), m_5)\}$$

Next it is necessary to find a minimum (k,U) such that $Q_2(f(g(Q_1(S))),U)$ is an incomplete machine. Enumerating each possible (k,U), we find:

k = 1	no U's
k = 2	no U's
k = 3	U = (1,1,1,1,1) nondeterministic
k = 4	U = (1,1,1,2,1) nondeterministic
k = 4	U = (1,2,1,1,1) incomplete program

This terminates the search so $k_S = 4$ and $U_S = (1,2,1,1,1)$. Thus Q_3 can be computed:

$$Q_3(S) = Q_2(f(g(Q_1(S))),U_S)$$

$$= \{(1I_1,\{a\},2I_1),(2I_1,\{a\},1I_2),(1I_2,\phi,1I_1)(1I_1,\{\neg a\},1I_H)\}$$

The resulting incomplete program appears in Figure 3.

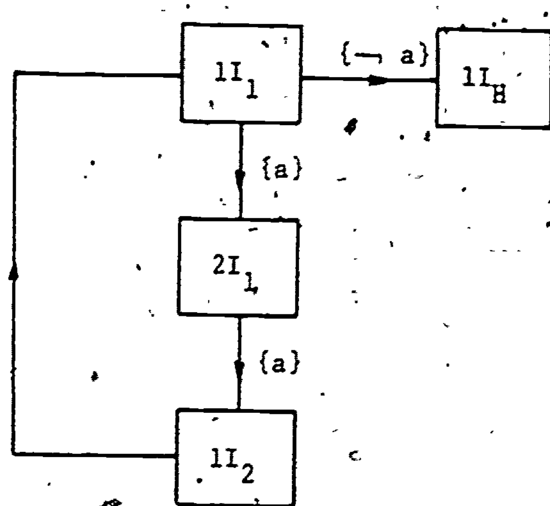


FIGURE 3. Incomplete program $Q_3(S) = \{(1I_1,\{a\},2I_1),(2I_1,\{a\},1I_2),(1I_2,\phi,1I_1),(1I_1,\{\neg a\},1I_H)\}$.

We will define one more operator Q_4 which will convert incomplete programs with initial states into programs. However, $Q_3(S)$ has the desired properties of soundness and completeness, and we will, therefore, prove these two theorems before continuing.

We will say that an incomplete program P can execute a partial trace $T = (m_o, r_1, m_1, r_2, \ldots, r_n, m_n)$ if there exist $u_1, u_2, \ldots, u_n$ and $c_1', c_2', \ldots, c_n'$ such that for each $j = 1, 2, \ldots, n-1$, $(u_j i_j, c_{j+1}', u_{j+1} i_{j+1}) \in P$ where $c_{j+1}'(m_j)$ is true. (We continue to follow the notation $r_j = (c_j, i_j)$ for $j = 1, 2, 3, \ldots, n$.)

Theorem 1. If S is a set of partial traces, then $Q_3(S)$ is an incomplete program which can execute each trace T in S.

The proof follows essentially from the definitions of the various operators. Assume for simplicity that S has only one trace, $S = \{T\}$ where $T = (m_o, r_1, m_1, \ldots, r_n, m_n)$ and each $r_j = (c_j, i_j)$. It is necessary to show that there exist $u_1, u_2, \ldots, u_n$ and $c_1', c_2', \ldots, c_n'$ such that for each $j = 1, \ldots, n-1$ $(u_j i_j, c_{j+1}', u_{j+1}, i_{j+1}) \in Q_3(S)$ where $c_{j+1}'(m_j)$ is true. But $Q_3(S) = Q_2'(f(g(Q_1(S))); U_S)$ and U_S provides the n constants $u_1, u_2, u_3, \ldots u_n$. ($U_S = (u_1, u_2, u_3, \ldots, u_n)$). Furthermore, $f(g(Q_1(S))) = (m_o, (c_1', i_1), m_1, (c_2', i_2), \ldots, (c_n', i_n), m_n)$ where $c_1' = \phi$ and $c_{j+1}'(m_j)$ is true for $j = 1, 2, \ldots, n-1$ by definition of f, g, and Q_1. By definition of Q_2 we note that $(u_j i_j, c_{j+1}', u_{j+1} i_{j+1}) \in Q_3(S)$ for each $j = 1, 2, \ldots, n-1$ which completes the proof. A simple extension of these observations will complete the proof for the case where S has $k > 1$ traces.

Theorem 1 guarantees that the synthesized program $Q_3(S)$ will be able to execute all of the given example traces in S. The next theorem assures us that if a user begins executing example calculations for some program P, the system will synthesize a correct program P_o after only a finite number of examples have been completed. P_o will have the property that it can execute every calculation that P could execute, and this convergence property will hold without regard to the order of presentation of the examples. The corollary will futher assert that if P is complete then P_o will be "equivalent" to P.

Theorem 2. Let P be an incomplete program and let $T_1, T_2, \ldots$ be any enumeration of all of the partial traces executable by P.* Then there exists a finite k and some incomplete program P_o such that

(1) $P_o = Q_3(\{T_1, T_2, \ldots, T_i\})$ for all $i \geq k$,

(2) P_o can execute each T_i, $i = 1,2,3,\ldots$, and

(3) no program with fewer instances of instructions than P_o can execute each T_i, $i = 1,2,3,\ldots$.

This result also has a simple proof. Suppose P has exactly p instances of instructions. Notice that the construction of Q_3 involves a complete search through the space of all possible incomplete programs which could execute the traces and which have i instances of instructions for $i = 1,2,\ldots$. Since P will exist somewhere in the enumeration done by Q_3, the enumeration will be bounded, and $Q_3(\{T_1, T_2, \ldots, T_i\})$ will yield either P or some incomplete program which precedes P in the enumeration. Thus, there exists a finite v such that for all i, $Q_3(\{T_1, T_2, \ldots, T_i\})$ need enumerate no more than v incomplete programs before it can yield its answer. Define $P_i = Q_3(\{T_1, T_2, \ldots, T_i\})$ for each $i = 1,2,3,\ldots$, and we can think of $P_1, P_2, P_3, \ldots$ as a sequence of guesses at the answer P_o over a period of time. Then the set $\{P_i | i=1,2,\ldots\}$ by the above argument has finite cardinality. Also notice that any incomplete program P' that is chosen at some time ($\exists j$ such that $P' = P_j$) and later rejected ($\exists j'$ such that $P' \neq P_{j+j'}$) can never be chosen again (not $\exists j''$ such that $P' = P_{j+j'+j''}$). This is because if P' is rejected when it is found unable to execute $T_1, T_2, \ldots, T_{j+j'}$, then it will certainly be unable to execute $T_1, T_2, \ldots, T_{j+j'+j''}$. So the finiteness

*We assume that P can execute only countably many different partial traces.

of the set $\{P_i | i = 1,2,3,\ldots\}$ and the inability to return to previously rejected guesses implies result (1) of the theorem. P_o can execute every T_i by Theorem 1 and has minimal size by construction which completes the proof.

Programs P_1 and P_2 will be said to be equivalent if for every partial trace T which begins with the start instruction $1I_o$, P_1 can execute T if and only if P_2 can execute T.

Corollary. If P is a (complete) program, then P_o of Theorem 2 is equivalent to P.

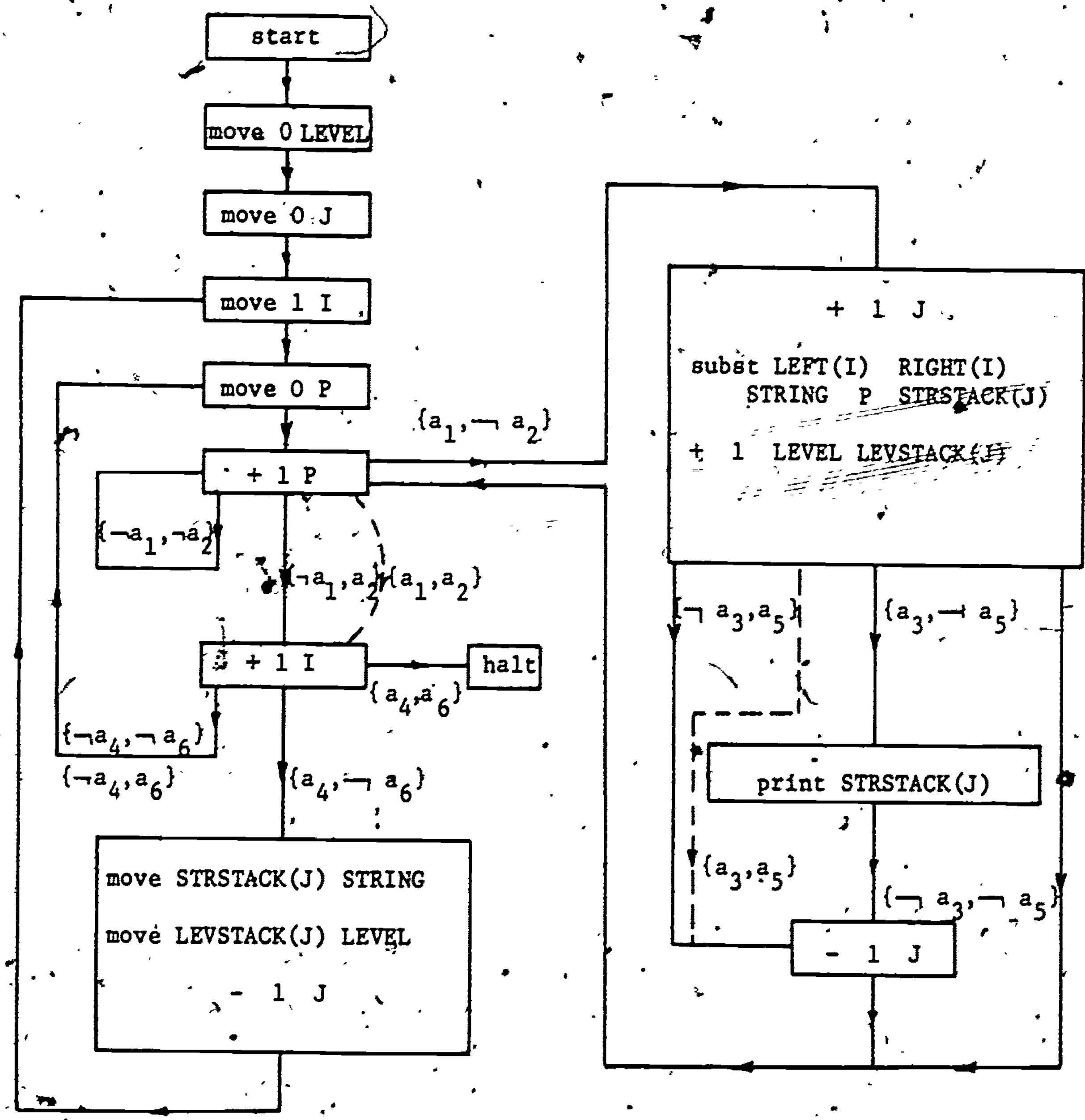
Since Theorem 2 asserts that P_o can execute every partial trace executable by P, it is only necessary to show that P can execute every partial trace executable by P_o which begins with $1I_o$. Assume the contrary that there is a $T = (m_o,(\phi,1I_o), m_1, (c_2,i_2),\ldots,(c_n,i_n),m_n)$ which P_o can execute but P cannot. Then there is a largest prefix of T, say $T' = (m_o,(\phi,1I_o),m_1, (c_2,i_2),\ldots,(c_k,i_k),m_k)$, $0<k<n$, which P can execute. Furthermore, since P is complete, it can validly continue T' and can execute $T'' = (m_o,(\phi,1I_o),m_1, (c_2,i_2),\ldots,(c_k,i_k),m_k,(c',i'),m')$ for some c', i', and m' where $(c',i') \neq (c_{k+1},i_{k+1})$. But P_o cannot execute T'' which contradicts Theorem 2 and completes the proof. (Comment: P_o may not be complete even though it is equivalent to P.)

These results are neither new nor surprising considering earlier papers in grammatical inference [5, 6, 7]. Notice that even though Theorem 2 guarantees that the correct incomplete program P_o will be found after some finite time k, there is no way of knowing at any given time i whether or not P_o has been found. Thus, there is no proof of correctness intrinsically built into the system, and at any time, the next partial trace T_{i+1} may cause the system to discard its current guess of P_o and try a new one. This kind of learning is known elsewhere as identification in the limit [5, 6, 7].

This means that the programmer in debugging his code is theoretically no better off with this system than he was with traditional programming techniques. He still must find errors by running test cases and by studying his code. From a practical point of view, however, we hope that the auto-programmer will provide facilities that will speed this process considerably.

Applying the synthesis technique to the trace of Figure 2 yields U_S = (1,1,1,.....,1,2,1,1,.....,1), twenty-three 1's followed by 2 followed by seventeen 1's. The resulting incomplete program is shown in Figure 4. This would be a correct complete program except that two triples are missing, (+ 1 P,{LEFT(I) = STRING(P), length (LEFT(I)) > length (STRING(P))}, + 1 I) and (+ 1 LEVEL LEVSTACK(J), {terminal (STRSTACK(J)),LEVSTACK(J) = N}, print STRSTACK(J)). Instruction labels are omitted in Figure 4 because all but one of them are 1.

Omitted triples in an incomplete program can often be guessed and filled in correctly to produce a complete program. For example, in Figure 5a, the condition $\{a_1, \neg a_2\}$ has not been observed after instruction $1I_1$ and $\{\neg a_1, \neg a_2\}$ has not been observed after $2I_1$. These omissions can take place either because it is impossible for the associated conditions to occur (such as J>2 and J<0) or because they simply have not yet been observed in the traces. In any case, arbitrary addition of the missing transitions will not destroy the guarantees of Theorems 1 and 2 and can often be done to achieve quicker convergence to the desired program. In the case of Figure 5a, it would seem natural that $1I_1$ followed by $\{a_1, \neg a_2\}$ would lead to the



a_1 = (LEFT(I) = STRING(P))

a_2 = (length(LEFT(I))>length(STRING(P)))

a_3 = (terminal STRSTACK(J))

a_4 = (I>NORULES)

a_5 = (LEVSTACK(J) = N)

a_6 = (J = 0)

FIGURE 4. The program synthesized from the trace of Figure 2. (The dotted transitions, one of which is erroneous, are inserted by Q_4.)

same instruction as $1I_1$ followed by $\{a_1, a_2\}$ and $1I_2$ followed by $\{\neg a_1, \neg a_2\}$ and $\{a_1, \neg a_2\}$ would also lead to the same next instruction. This results in the simplified diagram of Figure 5b. In other words, a reasonable heuristic for completing the program is to add transitions so as to minimize the total complexity of the boolean expressions on the instruction-to-instruction transitions. For the purposes of this paper, it is not important to more clearly define $Q_4(S)$ other than to say that if $Q_3(S)$ is an incomplete program with a start instruction $1I_0$, then $Q_4(S)$ is a complete program constructed by adding triples to $Q_3(S)$. Hopefully $Q_4(S)$ will better approximate the desired program than $Q_3(S)$.

Let us assume that Q_4 operates on the incomplete program of Figure 4 and adds the two missing transitions as shown with the dotted lines. It turns out that one of these additions has introduced an error into the program, and one of the purposes of the next section will be to show how this error can be found and corrected.

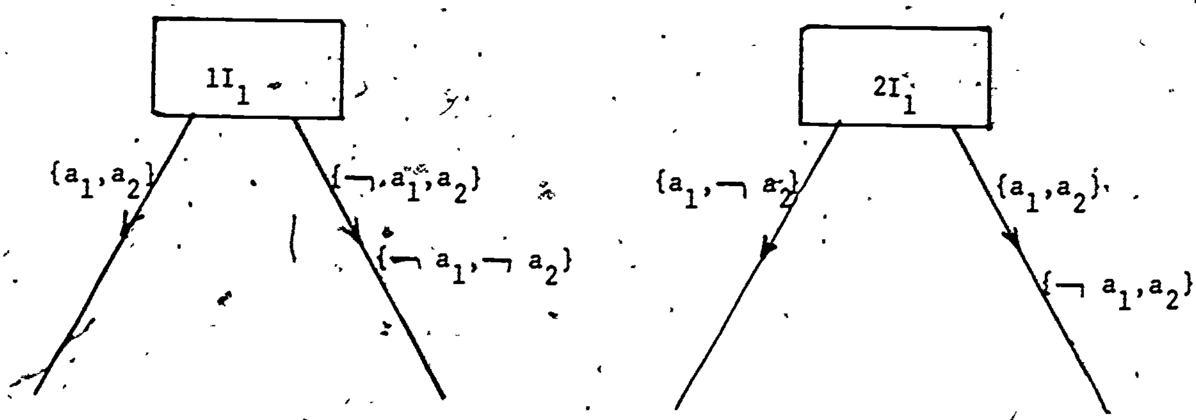


FIGURE 5a. Two instances of instruction I_1 in an incomplete program.

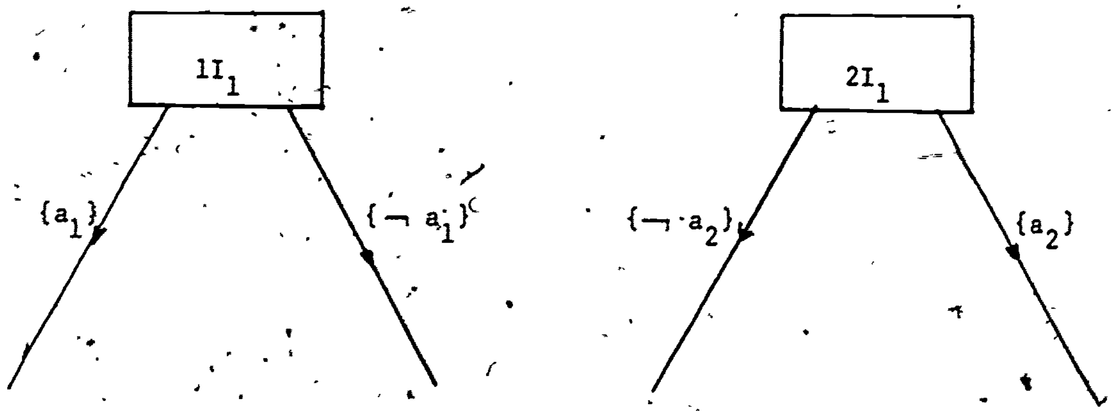


FIGURE 5b. The same two instances after the operation Q_4.

5. SYSTEM DESIGN AND MAJOR FEATURES

The general organization of the autoprogramming system is shown in Figure 6 where the major functional units are

(1) the display and top level routines which interface with the user and which transfer user commands to the rest of the system,

(2) the interpreter which inputs instructions and data structure contents and outputs changes in the data structure contents, and

(3) the synthesizer which inputs sets of partial traces and outputs incomplete or complete programs.

The major storage areas keep the following information for each routine to be synthesized:

(1) Data structure display information including each data structure name, type, dimensions, organization, location on the display, pointer information, etc.

(2) Data structure contents: the actual values currently held in each location.

(3) Computation traces from which the routine is to be created.

(4) The synthesized program.

A typical usage of the system is easy to visualize. The programmer enters the name of the routine to be created; we will call it "routine 2". Then he declares the data structures to be associated with this routine, and their descriptions are entered into the Data Structure Display Information area as shown in Figure 6. Now this information is available to the display routines so that the user will see these structures on the screen. In preparation for doing an example calculation, he switches the system to <u>local mode</u> and enters the example data into the data structures. Local mode insures that the instructions he uses will not become part of the trace and will not be synthesized into the program. He can do any other hand calculation

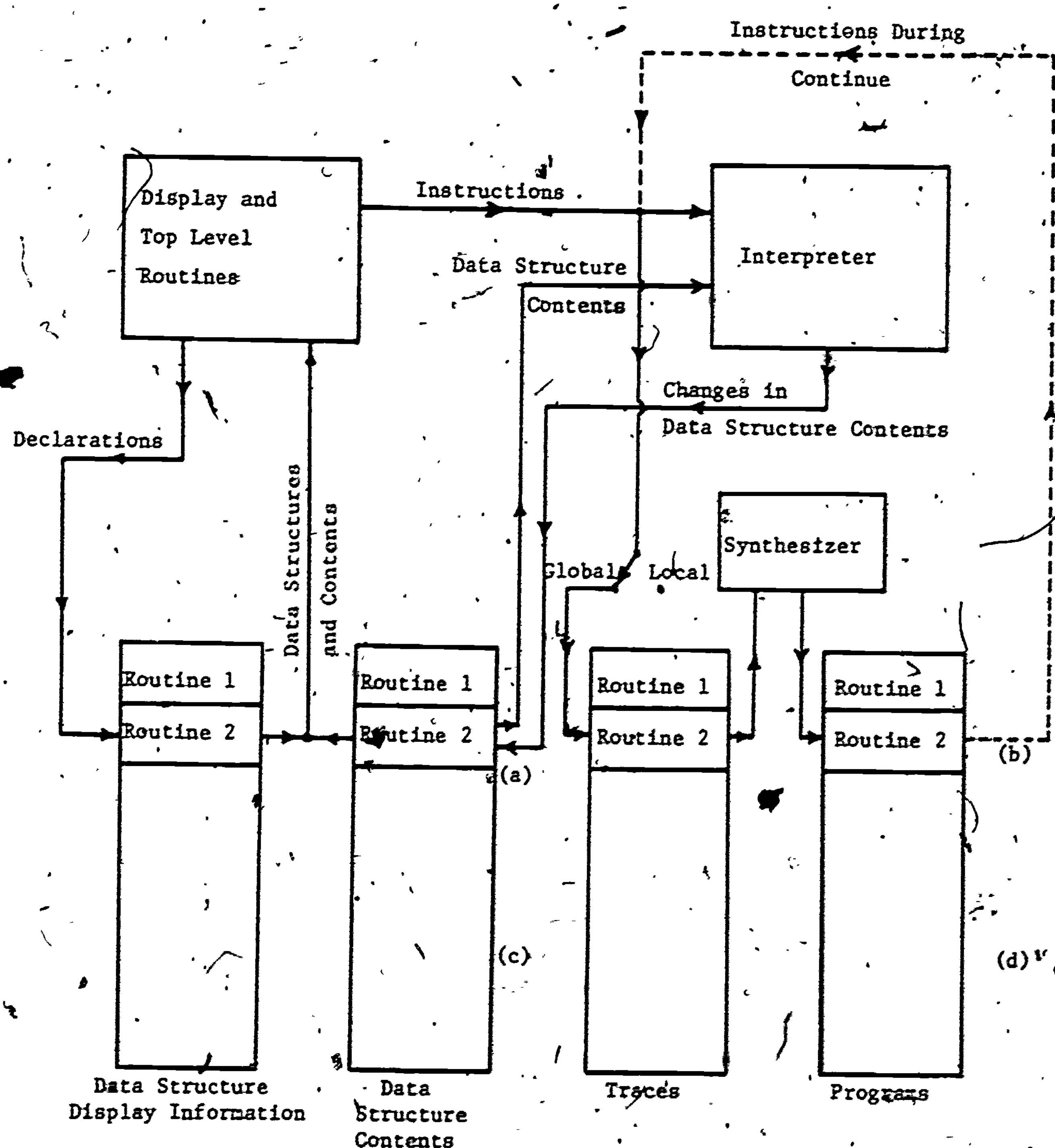


FIGURE 6. Major programs and storage areas.

he wants while in local mode without affecting the traces. Each instruction he performs that causes changes in the data structures is immediately updated on the screen. When he is ready to begin the example, he switches the system to <u>global mode</u> and now all instructions performed are saved in the trace storage area for routine 2. The synthesizer is operative at all times keeping the smallest incomplete program compatible with the traces to date in the program area for routine 2. This incomplete program can be revised after every new trace instruction without significant computational loss and with important benefits to the user to be explained later in this section. After the user completes the partial trace (with or without a halt instruction), the synthesizer applies Q_4 to turn routine 2 into a complete program.

At this point, the user may either begin testing the current version of routine 2 or do another example. It is important to remember that the traces may be partial and need not include either a start instruction or a halt. Thus, the user may want to say: "After reading J, if J = 1 then print A and if J = 2 then print B". This fragmentary information may be input to the system with two partial traces: read J, note J = 1, print A and read J, note J = 2, print B. The synthesized program will always be the smallest program compatible with the given traces and the result of these two new traces will be additional transitions "glued" into the already created program. Usually because of the nature of programs, they will be added at the correct position in the program. If they are later found to be incorrectly inserted, the programmer can do another partial trace increasing the amount of information about these instructions. For example, he might input: "After K is incremented and J is read, then if J = 1, print A". But users quickly learn what they must input to get the desired program and such trial and error revisions are not typical.

The fact that the synthesizer continuously maintains an updated version of the incomplete program during trace creation enables us to add an extremely important feature to the system. It may be that while the user is executing an example, a partial program will be created which is quite capable of continuing or even completing his example for him. If this is true, he should certainly turn control over to this partial program and save himself the trouble of doing the instructions by hand. For example, if he wishes to add a column of numbers, the loop required to do the summing would probably exist in the updated program after he has added the first two or three numbers, and this partial program could sum the rest of the column automatically. The <u>continue</u> feature then works as follows: The system at all times keeps track of which instruction in the current incomplete program corresponds to the last instruction in the current trace. If the given instruction in the incomplete program is followed by a valid transition, the command <u>continue</u> appears on the user's screen along with the other instructions. If the user wishes to let the synthesized incomplete program issue the next instruction rather than doing it himself, he touches the continue command. Then he can observe the results of this continue, and if it is correct and the continue command still remains on the screen, he can repeatedly hit continue to carry on the example. If the continue command produces incorrect results, he can hit the <u>backup</u> command, undo the effect of the last instruction, and insert the correct instruction by hand.

The inclusion of such features means that the experience of doing examples should not be thought of as simply a long string of hand inserted instructions. The programmer pushes the system through new parts of the desired program, uses continue to do other parts of the example, backs up, inserts instructions now and then, returns to continue, and so forth. The reader should examine

Figure 2 again to see how much of that example could be done automatically with the continue feature.

The backup command is available on the system at all times so that the user can undo any instructions that he has executed and decided to erase. The backup can be used repeatedly even to the point of erasing a complete trace.

The process of discovering errors on this system is similar to that using more conventional systems. One may print out and study the synthesized code, and one may run a number of test examples. Suppose the example of Section 2 is run again as a test with N set to value 1. The synthesized program should still find one terminal string, specifically, the string a, but it fails to because Q_4 of the last section inadvertantly inserted an error. Not realizing why the program did not print the correct result, we can display the data structures, initialize to do the example with N = 1, and in local mode use continue to advance the calculation through, step-by-step. It will all go perfectly until the instant string a is put on the stack and is supposed to be printed. Much to our surprise, the synthesized program immediately erases a from the stack and proceeds to the next step. At this point, we can back the calculation up to the point where a was about to be put on the stack, switch to global mode to create a partial trace, use continue to put a on the stack again, insert the print STRSTACK(J) instruction, use continue to check that the calculation is proceeding normally, and terminate the partial trace. The synthesized program will now include a corrected transition which will do this example and all other examples correctly. Notice that the cause of error was discovered by examining the effect of the code in the data structures, and the error was removed by forcing correct action at the point of error. Thus, errors can be found and corrected without direct reference to the code.

The example of Section 2 is now in perfect working order but, as usual, the programmer may wish to change it in some way. This can be done using the override feature while running a new sample calculation. Assume that it is desired to put a counter COUNT into the program which counts the number of terminal strings which have been printed, and then it is desired to print the total count before halting. The programmer first declares the new variable so that it will appear on the screen. (Declarations can be made or deleted at any time.) Then he initializes the data structures to do an example, sets the mode to global, and uses continue to begin advancing automatically through the example. Immediately after the start instruction, he touches the override command, loads zero into COUNT, and then returns to usage of the continue instruction. The effect of the override command is to return to all previous traces and replace the $r_j \leftarrow (c_j, i_j)$ term that would have been executed at this point by the dummy symbol d. Since this symbol d is used as a separator between traces, such an insertaion effectively cuts the trace into two partial traces as well as eliminating the unwanted transition. The programmer now proceeds forward with the continue feature until a terminal string is printed at which time he touches override, increments COUNT, and returns again to continue. As he proceeds, he will be gratified to see COUNT automatically incremented as other terminal strings are generated since the continuously updated program will have already incorporated his change. Finally, just before the halt instruction, the programmer uses override one more time to cause COUNT to be printed. The automatically synthesized program will be identical to the earlier version except that the variable count is now included and will be correctly initialized, incremented, and printed.

The fact that the override feature chops up earlier traces does not affect the convergence guaranteed by Theorem 2. That theorem states that any enumeration of partial traces converges on the desired P_o, and thus an arbitrary amount of chopping on the early traces will not prevent a correct synthesis. Of course, the decision to alter the synthesized program means that the goal program P_o has been changed, and the purpose of the trace deletions made by override is to make the set of traces compatible with the new goal program. Because chopping of the traces does not eliminate convergence, the override feature can be used without limit to make changes to a synthesized program. The only cost in using this feature is a slower convergence to P_o due to the information lost in the deletions.

The subroutine feature enables the programmer to build a large program out of many smaller ones and to properly modularize his task. With an autoprogrammer, it also makes it possible to deal with shorter traces and fewer data structures on the screen. As each new subroutine is created, some of its data structures can be designated as arguments to be supplied at the time of the call. One of the instructions available on the screen is CALL SUBROUTINE which may be used like any other instruction. If CALL SUBROUTINE is hit at any time, the names of all subroutines created to date including the current subroutine appear on the screen and the user can designate which one he wants. Then he touches among the current data structures the arguments for the routine. After the subroutine call is made, the connections at (a) and (b) in Figure 6 are moved to, say, (c) and (d) to reference the called program and its data structure contents. These connections, of course, return to (a) and (b) when the subroutine execution terminates.

Many times a programmer in the process of doing an example suddenly realizes that he would like to call a subroutine to do a task that he has not anticipated. In this case, he can execute CALL SUBROUTINE and type in the name of the desired subroutine even though it does not yet exist. Then he can insert on the screen the results the subroutine would have yielded if it did exist and proceed onward. Thus, he can do top down programming in a fairly convenient manner. If he wishes to execute this routine before creating its supporting subroutines, he, of course, must be willing to fill in by hand the results of every call to every nonexistant subroutine.

6. AN IMPLEMENTED AUTOPROGRAMMER

An autoprogramming system for integer calculations has been implemented and tested extensively by the authors. The system uses a Digital Equipment Corporation Model 340 display with light pen connected to a PDP-10 computer. The implemented instructions are add, subtract, multiply, divide, move, read, write, call subroutine, and note greater than, equal to, or less than. The allowed data structures are individual variables, linear, and rectangular integer arrays.

Because some of the features described in this paper have only recently been developed, they were not incorporated into the original design. The synthesis algorithm in this paper, for example, allows the user to freely omit conditionals during a sample calculation as long as each conditional is properly inserted at least once. The implemented system makes more stringent requirements on the user. Continuous updating of the synthesized program during a computation is not available so the continue and override features are not included. This system does, however, include a convenient subroutine feature with recursion, the backup feature, local and global modes, and the ability to add and remove data structures at will.

The Data Structure Contents array of Figure 6 was implemented using a hash coding scheme with the key computed from a combination of the data structure name, its associated subroutine name, the level of the call (in a hierarchy of calls), and the array indexes, if any. This organization is quite convenient in that it makes the subroutine feature recursive without any additional coding and it effectively increases all arrays to an infinite size as long as the hash table is not full. Thus, an array which is declared to be two-by-two will appear on the screen to be that size at synthesis time. However, at execution time when the subroutine is called, it can reference and use the 100,100-th entry of the array without concern about overflow. This is quite important because the limited size of the display screen prohibits the declaration of large arrays.

An example program synthesized on this system appears in Figure 7, the sorting algorithm known as "quicksort" [8]. The program accepts three arguments, a linear array A to be sorted and the bounds N1 and N2 for the sort. QUICKSORT (A,N1,N2) reorders the entries A(N1+1), A(N1+2),.....,A(N2) into ascending order. One can create this routine by executing the algorithm on the example list (2,7,1,6,3). Set the pointers P1 and P2 to the entries given by N1 and N2:

```
        2    7    1    6    3
 ↑                        ↑
 P1                       P2
```

Advance pointer P1 until we note that A(P1)>A(P2):

```
        2    7    1    6    3
             ↑              ↑
            P1              P2
```

Exchange those entries and then decrease P2 until we again note that A(P1)>A(P2):

```
        2    3    1    6    7
             ↑    ↑
            P1   P2
```

Exchange those entries and increase P1 until P1=P2.

```
        2    1    3    6    7
                  ↑
                P1=P2
```

Decrease P1 by one, call recursively QUICKSORT (A,N1,P1) and QUICKSORT (A,P2,N2) to complete the sort, and halt. Because the program is not synthesized until the trace is completed on this system, the recursive calls to QUICKSORT result in a message from the system: "This routine does not exist". But the trace is correct and the fact that the calls result in no action at the time of the example calculation is of no concern. If it is important to have the results of calls to nonexistant routines updated on the screen during

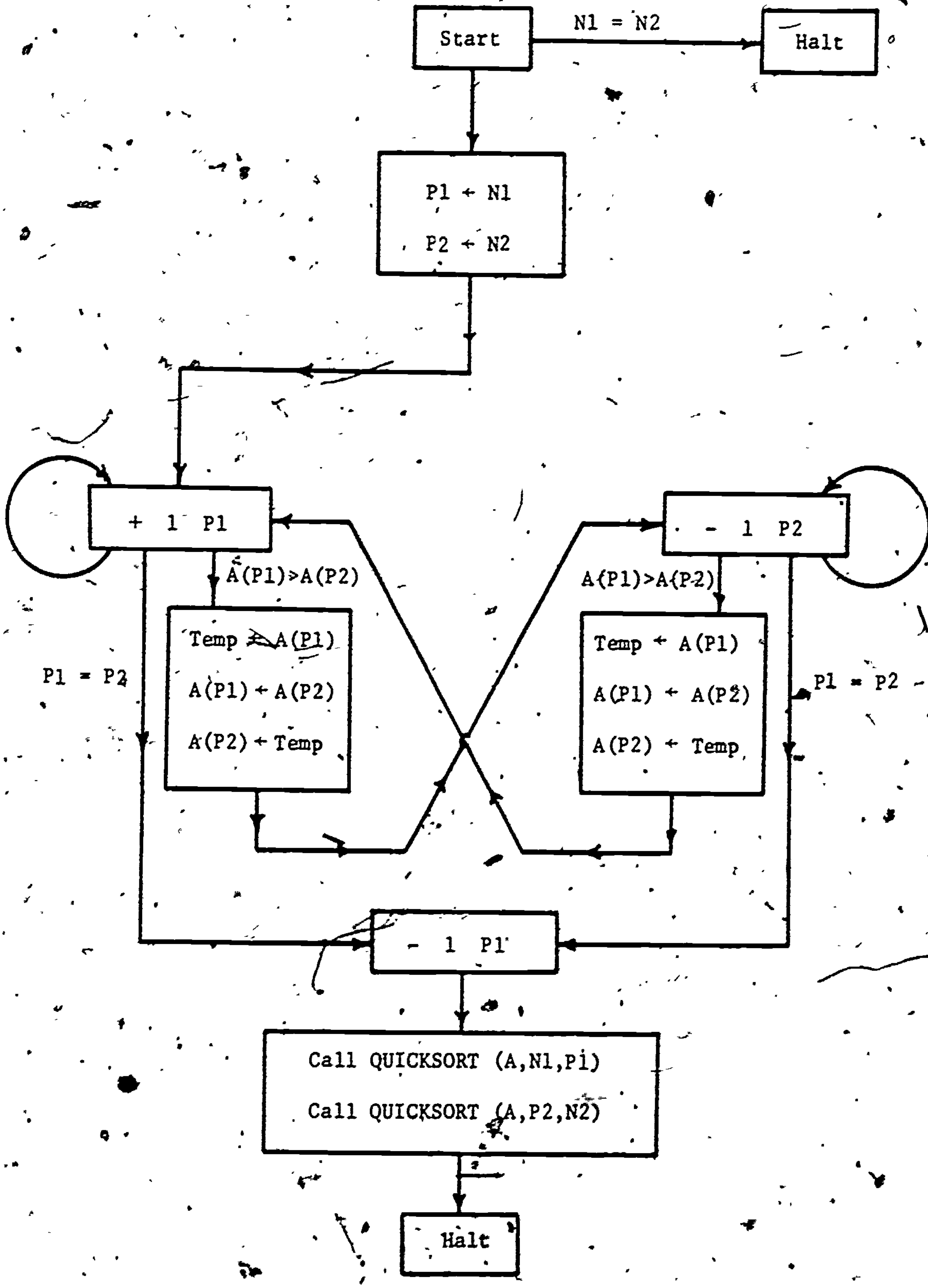


FIGURE 7. A sorting routine created with an autoprogrammer: QUICKSORT (A,N1,N2)

a sample calculation, these results can be inserted by hand using local mode.

Next we execute another example calculation sorting the list (2,1) and an example with arguments N1 = N2 = 0. After completing these three traces, the program of Figure 7 is correctly synthesized.

Careful examination of Figure 7 reveals that this autoprogrammer handles conditionals differently from the algorithm of Section 4. After executing an instruction, the transition with the true condition is taken, and if no condition is true, the unlabelled transition is taken. Unfortunately, this occasionally leads to a nondeterminism with two or more valid transitions which must be resolved either with additional traces or by answering a query from the system.

Another program created on the autoprogrammer was a compiler for a simple ALGOL-like language called Y73. This language has been used as the source language for a compiler writing exercise in programming classes and has only integer mode, no arrays, and no subroutine feature. The available key words in Y73 are READ, WRITE, BEGIN, END, WHILE, POS, and NPOS. The WHILE statement has the form WHILE e .x. p; which means "while arithmetic expression e has the property x, continue repeating program p". x is either POS (positive) or NPOS (not positive) and p is a program bracketed by a BEGIN and an END. A typical program in Y73 appears in Figure 8. The object code for the compiler was IBM 370 machine language.

Of course, both the input and the output for the compiler had to be coded into integers by hand because the current autoprogrammer handles only integers. Thus, the input tokens were coded 1 for +, 2 for -, ..., 8 for ;, ... , 10 for READ, ..., and 17 for BEGIN. Identifiers were coded 21,22,.....,29, and constants were coded 30 for 0, 31 for 1, etc. This means that the input program was a sequence of integers; in the case of the program of Figure 8, it would be 17, 10, 21, 8..... .

```
BEGIN
READ  N;
WRITE  N;
WHILE   N-1  .POS.
     BEGIN
     WHILE    N-N/2*2  .POS.
            BEGIN
            N ⇒1 + 3 * N;
            WRITE   N;
            END;
     N = N/2;
     WRITE  N;
     END;
END;
```

FIGURE 8. A program in the language Y73 which was compiled with an auto-programmer created compiler.

An example output instruction from this compiler would be "load into register 5 from the location addressed by base register 12 with a displacement of 20". This instruction would be 585C0014 in IBM hexadecimal and would be printed out by the autoprogrammed compiler as

```
INST = 88          88 (decimal) = 58 (hexadecimal)
R1   =  5
R2   = 12
DISP = 20
```

Except for this coding problem, the object code was directly executable on the IBM machine. The READ and WRITE instruction were implemented with locally defined supervisor call instructions.

This autoprogramming system has been used to create the above mentioned compiler which involved fifteen subroutines, a program synthesizer similar to the Q_3 function described above, and dozens of other programs. The amount of effort required to produce these programs does not differ greatly from that required using more conventional systems. It is hoped that as all the features discussed in the paper become implemented and as others are developed, autoprogramming will, in fact, become a desirable alternative to conventional systems.

7. DISCUSSION

Autoprogramming is by nature a language independent concept and can provide the context for many different kinds of computing. The approach is designed to put the user in intimate contact with his data structures and the events which affect them. It enables the user to create, debug, and modify his program by working with the effects of the code rather than the code itself. The approach puts the man and machine in a truly interactive relationship at the time when the source code is being created, and it breaks away from the batch mode psychology: write the program, type the code, and compile.

Our research has emphasized simplicity of design both in the autoprogramming language and in the total system. Because the language is without syntax in the traditional sense and because the results of each instruction are updated immediately before the user's eyes, the amount of training required for a new user is minimal. We believe that the special system features such as continue, backup, and override should be few in number and so simple and obvious in their operation that the novice programmer can use them immediately and without hidden dangers.

Our current work is aimed at developing language features and error correction mechanisms which will enable the programmer to be more casual and less detailed in his execution of examples and to still maintain the expectation that a correct program will be created.

8. BIBLIOGRAPHY

1. Amarel, S., "Representations and Modeling in Problems of Program Formation", Machine Intelligence 6 (Meltzer and Michie, eds.) American Elsevier Publishing Company, Inc., New York 1971.

2. Biermann, A. W., "On the Inference of Turing Machines from Sample Computations", Artificial Intelligence 3, 1972.

3. Biermann, A. W., Baum, R. I., and Petry, F. E., "Speeding Up the Synthesis of Programs from Traces", to appear in IEEE Transactions on Computers.

4. Biermann, A. W., Baum, R. I., Krishnaswamy, R., and Petry, F. E., "Autoprogrammer", ten-minute movie.

5. Blum, L., and Blum, M., "Inductive Inference: A Recursion Theoretic Approach", Recursive Function Theory Newsletter, No. 6, July, 1973.

6. Feldman, J. A., "Some Decidability Results on Grammatical Inference and Complexity", Information and Control, 1972

7. Gold, M., "Language Identification in the Limit", Information and Control, 1967

8. Hoare, C. A. R., "Quicksort", Computer Journal 5, 1962.

9. Lee, R. C. T., Chang, C. L., and Waldinger, R. J., "An Improved Program Synthesizing Algorithm and its Correctness", Communications of the ACM, March, 1974.

10. Manna, Z., and Waldinger, R. J., "Toward Automatic Program Synthesis", Communications of the ACM 14, No. 3, 1971.

11. Waldinger, R. J. and Lee, R. C. T., "PROW: A Step Toward Automatic Program Writing", Proceedings of the International Joint Conference on Artificial Intelligence, Washington, D. C. 1969.