THE LOGIC THEORY MACHINE A COMPLEX INFORMATION PROCESSING SYSTEM

Allen Newell and Herbert A. Simon[1]
The RAND Corporation, Santa Monica, Calif.
and the Carnegie Institute of Technology,
Pittsburgh, Pa.

Abstract

In this paper we describe a complex information processing system, which we call the logic theory machine, that is capable of discovering proofs for theorems in symbolic logic. This system, in contrast to the systematic algorithms that are ordinarily employed in computation, relies heavily on heuristic methods similar to those that have been observed in human problem solving activity. The specification is written in a formal language, of the nature of a pseudo-code, that is suitable for coding for digital computers. However, the present paper is concerned exclusively with specification of the system, and not with its realization in a computer.

The logic theory machine is part of a program of research to understand complex information processing systems by specifying and synthesizing a substantial variety of such systems for empirical study.

Introduction

In this paper we shall report some results of a research program directed toward the analysis and understanding of complex information processing systems. The concept of an information processing system is already fairly clear and will be made precise in Section I, below. The term 'complex' is not so easily disposed of; but it is the crucial distinguishing characteristic of the class of systems with which we are concerned.

We may identify certain characteristics of a system that make it complex:

1. There is a large number of different kinds of processes, all of which are important, although not necessarily essential, to the performance of the total system;

2. The uses of the processes are not fixed and invariable, but are highly contingent upon the outcomes of previous processes and on information received from the environment;

3. The same processes are used in many different contexts to accomplish similar functions towards different ends, and this often results in organizations of processes that are hierarchical, iterative, and recursive in nature.

Complexity is to be distinguished sharply from amount of processing. Most current computing programs for high speed digital computers would not be classified as complex according to the above criteria, even though they may involve a vast amount of processing. In general they call for the systematic use of a small number of relatively simple subroutines that are only slightly dependent on conditions. In order to distinguish such systematic computational processes from the processes we regard as complex, we shall call the former algorithms, the latter heuristic methods. The appropriateness of these terms will become clearer as we proceed.

One tactic for exploring the domain of complex systems is to synthesize some and study their structure and behavior empirically. This paper provides an explicit specification for a particular complex information processing system --a system that is capable of discovering proofs for theorems in elementary symbolic logic. We will call the system the logic theorist (LT), and the language in which it is specified the logic language (LL). This system is of interest for a number of reasons. First, it satisfies the criteria of complexity we have listed above. Second, it is not so large but that it can be hand simulated (barely). Third, the tasks it can perform are well-known human problem solving tasks--it is a genuine problem solving system. Fourth, there are available algorithms, and a realization of at least one of these algorithms (the Kalin-Burkhart machine),[2] that can perform these same tasks; hence, the logic theorist provides a contrast between algorithmic and heuristic approaches in performing the same problem solving tasks.

The task of this paper, then, is to specify LT with sufficient rigor to establish precisely the complete set of processes involved and exactly how they interact. This is a lengthy and somewhat arduous undertaking but one that the authors feel is required in the present state of knowledge. As a result, the paper largely abstains both from comment on the more general significance of the ideas and techniques introduced, and from relating these to contemporary work.[3]

The plan of the paper is to give, in Section I, a description of the language, LL, in which LT

---

[1] The authors are indebted to Mr. J.C.Shaw of the RAND Corporation, who has been their partner in many aspects of this enterprise and particularly in undertaking to realize the logic theorist in a computer—work that will be reported in subsequent papers.

[2] See B. V. Bowden[1]

[3] We should like to make general acknowledgment of our indebtedness for many of the ideas incorporated in LL and LT to two areas of vigorous contemporary research activity: (1) to research on automatic programming of digital computers, for the approach to the construction of LL; and (2) to research on human problem solving, for the basic structure of the program of LT. In addition we should like to record a specific indebtedness to the work of O.G.Selfridge and G.P.Dinneen on pattern recognition, which clarified many basic conceptual issues in the specification and realization of complex information processes.

will be specified. In Section II there is given a verbal description of LT, which is closely enough tied in to the formal program to motivate most of the latter. Finally, in Section III, the program is given in full detail.

I

Language for Information Processing Systems

The two major technical problems that have to be solved in studying information processing systems by means of synthesis may be called the specification problem and the realization problem. To study all but the simplest of such systems, it is necessary to make a complete and precise statement of their characteristics. This statement, or specification, must be sufficiently complete to determine the behavior of the system once the initial and boundary conditions are given. An example, familiar to mathematicians, is a system specified by n first order differential equations in n variables.

Once the specification has been given, a second problem is to find or construct a physical system that will behave in the manner specified. This can be a trivial or an insurmountable task. For example, it is relatively easy to find electrical circuitry that will behave like a system of linear differential equations; it is rather difficult to represent by circuitry most kinds of nonlinear systems. We will call the problem of finding or constructing the physical system the realization problem, and the particular physical system that is used the realization.[4]

Although this paper is concerned exclusively with the specification problem, the form of language chosen is dictated also by the requirements of realization. Since an important technique for studying the behavior of complex systems is to realize them and to study their time paths empirically under a range of initial and boundary conditions, they must be specified in terms that make this realization relatively easy.

The high speed digital computer is a physical system that can realize almost any information processing system and our research is oriented toward using it. Its limitations are in overall speed and memory, rather than in the complexity of the processes it can realize. The machine code of the computer is the language in which a system must ultimately be specified if it is to be realized by a computer. Conversely, however, once the system is correctly specified in machine code, the realization problem is essentially solved; for the computer can accept these specifications, and will behave like the system specified.

The machine code, although suitable for communicating with the computer, is not at all suitable for human thinking or communication about complex systems. For these purposes, we need a language that is more comprehensible (to humans), but one that can still be interpreted by the computer by means of a suitable program. Technically, such a language is known as a pseudo-code or interpretive language. Hence the two problems of specification and realization of an information processing system are subsumed under the single task of describing the system in an appropriate pseudo-code.

This paper is concerned solely with specifying the system of LT. The particular language, LL exhibited here has not been coded for a computer. However, one very similar to it, which is less convenient for exposition, is in the process of being coded and will be the subject of later papers. Here, no further mention will be made of the relation of the logic language to computers.

The terms of the language that are undefined--its primitives--determine implicitly a set of information processes that are to be regarded as elementary and not reducible, within the language, to simpler processes. The more complex processes are to be specified by suitable combinations of these elementary processes. Generally speaking, the elementary processes in LL are of the nature of information processes: that is, their inputs and outputs are comprised of symbolized information.

Information Processing Systems: Basic Terms

An information processing system, IPS, consists of a set of memories and a set of information processes, IP's. The memories form the inputs and outputs for the information processes. A memory is a place that holds information over time in the form of symbols. The symbols function as information entirely by virtue of their capacity for making the IP's act differentially. The IP's are, mathematically speaking, functions from the input memories and their contents to the symbols in the output memories. The set of elementary IP's is defined explicitly, and through these definitions all relevant characteristics of symbols and memories are specified.

Particular systems can be constructed from the memories and processes of an IPS that behave in a determinate way once the initial information in the memories is given (initial conditions), along with whatever external information is stored in the memories during the course of the system's operation (boundary conditions). Each such particular system we call a program, IPP. Thus an IPS defines a whole class of particular IPP's, and conversely, an IPP consists of an IPS together with a set of rules that determines when the several information processes will occur. The logic language is an IPS; the logic theorist is an IPP. Many variations of LT could be constructed with the same IPS.

Symbolic Logic

The logic language handles information referring to expressions in the sentential calculus and their properties. This paper assumes some

[4] We prefer "realization" to "simulation," for the latter implies that what is being imitated is another physical system. Since the specification is an abstract set of characteristics, not a physical system, it is not correct to speak of "simulating" the specification.

familiarity with elementary symbolic logic,[5] and only a resume of the notation will be given.

The sentential calculus deals with variables, p, q, ..., A, B, ..., a, b, ..., which are usually interpreted to mean sentences. These variables are combined into expressions by means of connectives. The primitive connectives of Whitehead and Russell (and ours) are "not" (-) and "or" (v). In this paper we shall have occasion to use only one other connective: "implies" (→), which is defined by:[6]

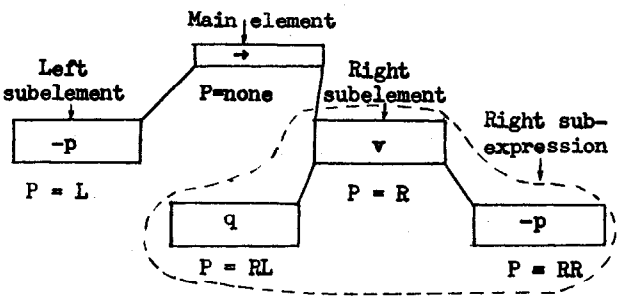
1.01 $p \rightarrow q \ =_{def} -p \vee q$ (Read: (p implies q) is equivalent by definition to (not-p or q).)

Coding

A logic expression, X, is represented in the IPS by a set of elements, E, one corresponding to each variable and to each connective (excluding the punctuation dots and negation symbols) in the logic expression. Each element holds a number of symbols that refer to the various properties of the element. (Note that the term "element" and not the term "symbol" is used in this paper to refer to the variables and connectives in logic expressions. Symbols denote properties of elements, and to each element there correspond a number of symbols.) An example will show what is meant by these terms.[7] Consider the expression 1.7:

1.7 -p .→. q v -p ((not-p) implies (q or not-p))

The entire sequence is the expression, X(1.7). It consists of the elements -p, →, q, v, -p. The expression may be written in "tree" form, as follows, where the rectangles indicate the elements:



[5]For definiteness, we have used the system of A.N. Whitehead and Bertrand Russell.[3] An introduction sufficient for our purposes will be found in D. Hilbert and W.Ackermann.[2]

[6]For ease of reference, we shall use the numbers employed by Whitehead and Russell to identify particular propositions and definitions, only omitting the asterisk (*) that they insert in front of the number.

[7] We follow Whitehead and Russell in using dots in place of parentheses as punctuation. It is unnecessary here to give exact rules for numbers of punctuation dots.

The main connective at the top is called the main element, EM (1.7). The other elements are reached through a series of Left and Right branches from the main element. With each element there is associated a subexpression, namely, the subtree of which that element is the top element.

The symbols in each element provide the following information, which will be explained more fully as we proceed.

Symbol

- G The number of negation signs (-) before the expression. In the figure above, two elements--those containing the variable p--have G = 1; all the rest have G = 0. If a negation applies to a whole expression it appears in the element associated with that expression.
- V Whether the element is a variable or not.
- F Whether the element is free, i.e., available for substitution. This is relevant only if E is a variable.
- C The connective (v or→). This is relevant only if E is not a variable.
- N The name of the variable or expression. In X(1.7), there are variables named "p" and "q".
- P The position of the element in the tree. This is represented by a sequence of L's and R's, counting branches from the main element. In the figure, the P for each element is shown beneath the element.
- A The location of the whole expression (not the element) in storage memory.
- U Whether the element is to be viewed as a unit or not. The term "unit" will be explained later.

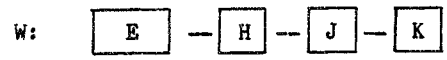
The eight symbols defined above characterize completely each element and the expression in which it occurs. For many purposes, however, it is convenient to define additional symbols ("descriptive symbols") that correspond to interesting or important properties of expressions. In LL, three such descriptive symbols, represented as small positive integers, are defined. These are:

- H The number of variable places in an expression. Thus X(1.7) has three variable places: P = L, RL, and RR; hence, H(1.7) = 3.
- J The number of distinct variables (i.e., distinct names) in the expression, ignoring negation signs. Since X(1.7) contains the names "p" and "q", J(1.7) = 2.
- K The number of levels in the expression. The number of levels corresponds to one plus the maximum number of letters in P for any element in the expression. Hence, K(1.7) = 3.

Memory Structure

There are two kinds of memories, working memories and storage memories. The major distinction --that all information to be processed must be

brought in from the storage memories to the working memories and then returned—will be brought out clearly when we define the elementary IP's. Structurally, the working memories hold single elements, E, with additional spaces for the symbols H, J, and K. Hence, we can picture a working memory unit as:



The storage memories consist of lists. A list holds either a whole logic expression or some set of elements generated during a process, such as a set of elements having certain properties. Each list of logic expressions has a location, symbolized by A. The elements are placed in the list in arbitrary order, since the information in each element is sufficient to locate it unequivocally in the tree of the logic expression. (The ordering of the list is used only to carry out searches.) For example, X(1.7) might be listed in the storage memory thus:

No limitations are imposed here on number of memories, either working or storage. In actual fact, the number used is not large.

Three particular lists have special locations in storage memory that can be referred to directly in IP's: (1) the theorem list, T, of all axioms and theorems that have previously been proved; (2) the active problem list, P; and (3) the inactive problem list, Q. Each list consists of the main elements of the appropriate expressions (theorems or problems, respectively) in arbitrary order. For the rest, the storage memory is entirely unspecialized.

Information Processes

A term that specifies an IP is called an instruction, by analogy with computer terminology. As Figure 1 shows, an instruction consists

	REFERENCE PLACES			
OPERATION	LEFT	CENTER	RIGHT	BRANCH LOCATION

Fig. 1

of an operation part, three reference places (left, L, center, C, and right, R), and a branch location, B. The kinds of operations that can be performed by an IPS will depend, first, on what elementary IP's are postulated, and second, on what restrictions are placed on how they can be combined. For the moment, the exact nature of the elementary processes is unimportant; for concreteness, the reader may think of the following as typical: transferring information from memory x to memory y, or adding the number in memory x to the number in memory y.

The reference places refer to the working memories, so that the same operation may operate on different memories at different times and under different circumstances. The working memories will be designated by small integers, 1, 2, ..., and by the letters x, y, z.

No direct reference is made in an instruction to any storage memory, except T, P, and Q. Lists are located by the A stored within elements belonging to the lists; and elements within a list are located by their relation to known elements. An example will make this clear. A typical operation involving the storage memory is:

OPER	L	C	R	B
FR	x		y	

which reads: Find the element that is the right subelement of E(x)—i.e., of the element in working memory x—and put it in working memory y. The operation is executed thus: Working memory x contains the A(x) that is the location of the expression in which E(x) occurs. Memory x also contains the symbol P(x). Since we wish to put in y the right subelement of E(x), P(y) is by definition obtained by appending an R to P(x). Hence, we can determine P(y), and locate E(y) by going to storage memory A(x) and searching the list of its elements in order until we find the element with the correct P. We then transfer this element, which is the one we want, to working memory y.

Programs and Routines

The rule of combination of IP's is simple: any one IP may follow another. We shall consider time to be discrete, using it essentially as an index, and shall assume that only one process occurs at a time. We say that a particular IP has control when it is occurring. Thus, when a sequence of IP's occurs one after the other in consecutive time intervals, there occurs a series of transfers of control from each IP to the next in the sequence.

The operation of any IP includes a processing component and a control component. The processing component changes the memory content of the IPS; the control component transfers control to another IP. In some IP's, processing is the significant component. In these the transfer of control is independent of the memory contents at the time the IP occurs. In certain other IP's, control is the significant component. These do not alter memory contents, but transfer control to various IP's depending on the memory contents when they occur. In other IP's both processing and control components are significant.

Control. We allow only a binary branch in control at any one instruction. Normally, control passes in a linear sequence through a set of IP's. We write this sequence vertically. Each instruction is considered to have a location in the sequence. For branch instructions (those in which the control component transfers control to one of two IP's depending on memory content), control transfers either (1) to the next instruction in the sequence or (2) to the instruction named in the branch location, B. These locations are designated by letters A, B, C, ... In

Figure 2, Instruction #1 transfers controls to #2; #2 transfers control to #3 or branches to A (which is #4) depending on memory content; #3 transfers control to #4; #4 transfers control to #5 or branches back to B, which is #1.

Each control operation can be reversed in sense by putting a minus sign in front of the operation name. The effect of the minus sign is simply to reverse the condition of transfer. That is, if CC-A transfers to A when two specified numbers are equal, then -CC-A transfers to A when these numbers are unequal.

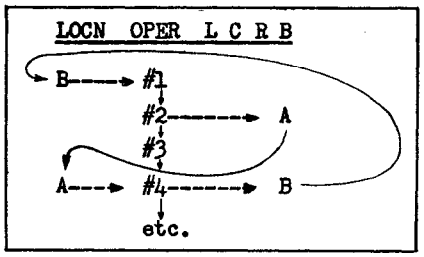


Fig. 2

Routines. We will call such a list of instructions with a control network a routine, again, in direct analogy to computer terminology. Notice that a routine satisfies our definition of a program (IPP): if all the memories referred to have specified initial contents, the routine determines their contents at all later times covered by its duration.

If we postulate a set of elementary information processes, each specified by an instruction, it might be supposed that each routine would define a new (non-elementary) information process. This is not the case, for in LL the format of an instruction (Figure 1) allows reference to not more than three working memories and to not more than one branch. Hence, only those routines may be regarded as definitions of IP's which satisfy the following conditions:

1. The routine contains branches to not more than two instructions outside the routine;
2. Not more than three working memories that are to be referred to subsequently are changed by the routine. This means that even though other working memories are changed, there is no way to refer to these memories in subsequent routines.

Within these restrictions we can define a set of new IP's in terms of the elementary IP's, then another set of IP's in terms of both the elementary and defined IP's, and so on; thus creating a whole hierarchy of IP's and their corresponding routines. The elementary IP's and the hierarchy of defined IP's for LT are given in Section III, and its structure as explained in some detail in Section II.

The restrictions imposed above on numbers of branches and working memories in IP's have the following two consequences for the structure of the routines that are used to define IP's:

1. A working memory can be used only within the routine in which it is introduced. That is, working memories introduced in a particular routine cannot be referred to when control is in any other routine, except as noted in rule 2. For this reason, no ambiguity arises from using the same names, 1, 2, . . ., for different memories in distinct routines.

2. Within the routine that defines a particular IP, reference may be made to the working memories that are designated in the reference places of that IP. Let I_1 be an instruction that appears in the routine defining I_2. The symbols L, C, R in I_1 refer, respectively, to the working memories in the left, center, and right reference places of instruction I_2, in whose definition I_1 occurs. (See, for example, the first instruction, FEF, in the routine given in full at the end of the section.) Some such arrangement is obviously required if the defining routine is to have any connection with the instruction it defines.

Elementary Processes

In LT there are forty-four different elementary processes. These represent variations on eight types of operations. The remainder of this section will be devoted to a description of these types, and an enumeration of the elementary processes that belong to each type. Separate, explicit definitions for each elementary IP are given in Section III. The first letter in the name of an operation designates the type to which it belongs: A for assign, B for branch, C for compare, F for find, N for numerical, P for put, S for store, and T for test.

Find instructions obtain information from storage memory on the basis of stated relationships, and put it in specified working memories. An example, FR-x-y (Find the right subelement of E(x) and put it in y), has already been described. Two other Find instructions are very similar: FL (Find the left subelement) and FM (Find the main element).

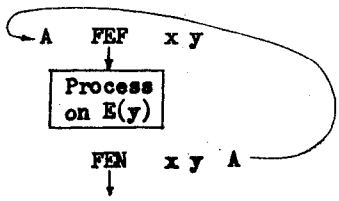
Other Find instructions involve the ordering relation on the lists. An example is:

OPER	L	C	R	B
FEF	x	y		A

This reads: Find the first element in X(x)—the expression associated with E(x)—in the list A(x), and put this element in y. Then go to next instruction, but if no element is found, branch to instruction A. Here the order of elements is essential since there may be many elements in X(x). This kind of operation is used to start a search, and it is always combined with an instruction, FEN, for continuing and terminating the search:

OPER	L	C	R	B
FEN	x	y		A

This reads: Find the element in X(x) that is next in order after E(y) and put it in y. When such an element is found, branch to A; if none is found, transfer control to the next instruction in sequence. FEF and FEN together allow the familiar cycling or iteration that is a common feature of computing routines:



(after all elements of X(x) have been processed)

The complete list of elementary Find instructions is:

FEF	FL	FM
FEN	FR	

Store instructions transfer information from working memory back to storage memory. An example is:

OPER	L	C	R	B
S		x		

This simply reads: Store E(x) in the storage memory. If the element in x is one that was previously withdrawn from storage, it will be replaced in its original location within A(x); if it is a new element in List A, it will be placed at the end of the list.

Another elementary Store instruction is SEN, which puts E(x) into storage memory at the end of the list A(y). A third is *SX, which simply stores a copy of X(x) in memory location A(y).[8]

The complete list of elementary store instructions is:

S	*SX	*SXL	*SXM
SEN	*SXE	*SXR	

Instructions belonging to the remaining six types are concerned only with working memory (See Figure 3). No complex processing may take place in storage memory, and conversely, as we have seen, no information may be stored in working memory except on a temporary basis.

Put instructions transfer information and symbols around the working memory. A typical Put instruction is:

OPER	L	C	R	B
PE		x	y	

This reads: Put E(x) in E(y). The operation leaves E(x) unchanged and duplicates it in E(y). The variations on this instruction correspond to the different symbols in an element that may need to be transferred. The list of Put instructions is:

PE	PCv	PU
PK	PC→	PUB

[8]Certain of the Store instructions are marked with an asterisk. These are treated as elementary operations in the present section and in Part I of Section III, but in Part II of Section III they are defined in terms of simpler operations.

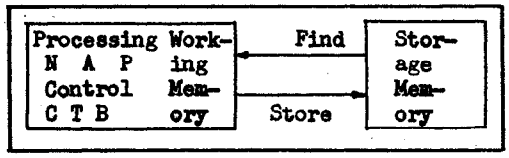


Fig. 3

Numerical instructions carry out the arithmetic operations. An example is:

OPER	L	C	R	B
NAG		x		

This reads: Add 1 to G(x). Operations are required to permit addition and subtraction for symbols G, H, J, K, and W. The list of Numerical instructions is:

NAG	NAH	NAJ	NSG
NAGG	NAK	NAW	NSGG

Assign instructions write in new names and locations in elements that are in working memory. One Assign instruction is:

OPER	L	C	R	B
AN		x		

This reads: Assign an unused name to E(x). The other Assign instruction, AA, assigns new list locations. There are, then only two Assign instructions:

AA AN

Compare instructions belong to a class of pure control instructions. They compare two symbols for equality (or, if appropriate, for the relation "greater"); then transfer to the branch location if the condition is satisfied or to the next instruction in sequence if the condition is not satisfied. The sense of the branch on these and all other branch instructions can be reversed by a minus sign preceding the operation. A typical example is:

OPER	L	C	R	B
CC		x	y	A

This reads: If C(x) = C(y), branch control to location A; if not, go to the next instruction in sequence. That is, if the connective in x is identical with the connective of y, we branch to A. Notice that there is no change in memory content; only a transfer of control has occurred. The compare instructions are:

CC	CGG	CWG
CN	CKG	CPS

Test instructions are also control instructions. They test the properties of a single element, and transfer control accordingly. The variations of the type deal with different properties. An example is:

OPER	L	C	R	B
TU		x		A

This reads: If E(x) is a unit, transfer control to A; if not, go to the next instruction in sequence. TC→ transfers control if C(x) is "implies"; goes to the next instruction if C(x) is "or". The Test instructions are:

TV	TB	TU	TF
TC→	TN	TGG	

Branch instructions are unconditional control instructions that cause the program to branch to the indicated address instead of going to the next instruction in sequence. The simplest example is:

OPER	L	C	R	P
B				b

When this instruction is reached, the program simply branches to instruction b in the same routine.

When the instructions BHB or BHN occur in a routine, they cause the program to branch to an address determined by the higher-level instruction that the routine defines. For example, suppose BHB appears as one of the defining instructions within the routine:

OPER	L	C	R	B
MSb	x			b

Then, the occurrence of BHB will cause control to branch to the address b of MSb.

Suppose, further, that MSb appears as one of the instructions in the routine Ex, and that the instruction MDt appears immediately after MSb in Ex. Then, if BHN is one of the instructions in the routine MSb, its occurrence will cause control to branch to the next instruction after MSb in the higher routine, Ex, i.e., to MDt. Thus BHB and BHN are the instructions that terminate control by a particular routine, and cause control to transfer, respectively, to the branch designated in the higher-level instruction defined by the routine, or to the higher-level instruction that follows the routine. Instruction BHB produces the former transfer, BHN, the latter. The three Branch operations are:

B BHB BHN

Example. It will clarify matters and provide some introduction to the complete program given in Section III if we set forth in detail one of the simpler defined routines, the routine NH. This routine consists of six instructions, all of them primitives included in the list we have already given:

	A	OPER	L	C	R	B
		NH	x			
		FEF	L	1		C
A		-CPS	1	L		B
		-TU	1			B
		NAH	L			
B		FEN	L	1		A
C		BHN				

Count the number of variable places in X(x), and record the result in H(x).

(1) FEF finds the first element in X(x) and puts it in working memory 1. If there is no element, it branches to C. (2) -CPS (note the negative sense) determines whether E(1) is a subelement of E(x). If it is not, control transfers to B; if it is, control transfers to the next instruction in sequence. (Henceforth we will abbreviate these transfers as →B and →next, respectively.) (3) -TU determines whether E(1) is a unit (i.e., is to be viewed as a variable). If it is not (negative sense), →B, if it is, →next. (4) NAH increases by 1 the number H(x). (Because of the previous branches, NAH will occur only if the element in 1 is viewed as a variable and is a subelement of the element in x.) (5) FEN finds the next element in X(x), puts it in working memory 1, and returns control to instruction A, whereupon the cycle is repeated from step (2). If there are no more elements, →next. (6) BHN terminates the routine after all elements in X(x) have been examined, and transfers control to the instruction that follows NH at the next higher level of the hierarchy of routines.

Conclusion

We have now completed our description of the language LL. We have outlined the coding system, the memory structure, the structure of the information processes, the routines, and the types of elementary processes. Further detail can be found by consulting Section III. In Section II we shall construct in this language a program, LT, that will permit the information processing system to solve problems in symbolic logic.

II

The Logic Theory Machine

In the language we have constructed, we have variables (atomic sentences): p, q, r, A, B, C, ... and connectives: - (not), v (or), → (implies). The connectives are used to combine the variables into expressions (molecular sentences). We have already considered one example of an expression:

1.7 $\quad -p .\rightarrow. q \vee -p$

The task set for LT will be to prove that certain expressions are theorems--that is, that they can be derived by application of specified rules of inference from a set of primitive sentences or axioms.

The two connectives, - and v, are taken as primitives. The third connective, →, is defined in terms of the other two, thus:

1.01 $\quad p \rightarrow q \ =_{def} -p \vee q$

The five axioms that are postulated to be true are:

1.2 $\quad p \vee p .\rightarrow. p$

1.3 $\quad p .\rightarrow. q \vee p$

1.4 $\quad p \vee q .\rightarrow. q \vee p$

1.5 $\quad p .v. q \vee r :\rightarrow: q .v. p \vee r$

1.6 $\quad p \rightarrow q .\rightarrow: r \vee p .\rightarrow. r \vee q$

Each of these axioms is stored as a list in the theorem memory, T, with all its variables marked free, F, in their respective elements.

From the axioms other true expressions can be derived as theorems. In the system of Principia Mathematica, there are two rules of inference by means of which new theorems can be derived from true expressions (theorems and axioms). These are:

Rule of Substitution: If A(p) is any true expression containing the variable p, and B any expression, then A(B) is also a true expression.

Rule of Detachment: If A is any true expression, and the expression A → B is also true, then B is a true expression.

To these two rules of inference is added the rule of replacement, which states that an expression may be replaced by its definition. In the present context, the only definition is 1.01, hence the rule of replacement permits any occurrence of (-pvq) in an expression to be replaced with (p→q), and any occurrence of (p→q) to be replaced with (-pvq).[9]

In this system, then, a proof is a sequence of expressions, the first of which are accepted as axioms or as theorems, and each of the remainder of which is obtained from one or two of the preceding by the operations of substitution, detachment, or replacement.

Example: prove 2.01, p → -p .→. -p:

(1) ⊦ p v p .→. p (axiom 1.2)[10]
(2) ⊦ -p v -p .→. -p (by subst. of -p for p)
(3) ⊦ p → -p .→. -p (by replacement on left)

The problem now is to specify a program for LT such that, when a problem is proposed in the form of a theorem to be proved (like 2.01 above), a proof will be discovered and constructed. First, it should be observed that there is a systematic algorithm for constructing such a proof should one exist. Starting with the five axioms, we construct all the theorems that can be obtained from them by a single application of the rules of substitution, detachment, or replacement.[11] We thus obtain the set of all theorems that can be obtained from the axioms by proofs not more than one step in length. Repeating this process with the enlarged set of theorems, we obtain the set of all theorems that can be derived from the axioms by proofs not more than two steps in length. Continuing, we finally obtain the set of theorems that can be derived by proofs not more than n steps in length.

Now, if the theorem in which we are interested possesses a proof k steps in length, we can, in principle, discover it by constructing all valid proof chains of length not more than k, and selecting any one of these that terminates in the theorem in question. This "in principle" possibility is in fact computationally infeasible because of the very large number of valid chains of length k that can be constructed, even when k is a number of moderate size. Under these circumstances, the rules of inference do not give us sufficient guidance to permit us to construct the proof we are seeking; and we need additional help from some system of heuristic.

The problem will be solved if we can devise a program for constructing chains of theorems, not at random, but in response to cues that make discovery of a proof probable within a reasonable computing time. For example, suppose the rules of inference were such as to permit any given proof chain to be continued, on the average, in ten different ways. Then there would be ten thousand proofs chains four steps in length (10^4). The expected number of proof chains that would have to be examined to find any particular proof by random search is five thousand. Suppose, however, that LT responded to cues that permitted eight of the ten continuations at each step to be eliminated from consideration. Then the number of proof chains four steps in length that would have to be examined in full would be only sixteen (2^4), and the expected number would be only eight.

The Program of LT

We wish now to describe the program of LT, which is given in full in Section III; hence, in the text we shall refer frequently to Section III for detail. We shall refer to each routine by its name (e.g., LMc for the matching routing), but we shall need some additional notation to refer to the main segments of routines that do not themselves have names. The names of these segments are given in Section III in the column marked "Seg." In each segment there is generally one main operation to be performed; and this main operation, or sub-routine, is usually surrounded by a number of procedural and control operations that fit it into the larger routine. In ordinary language, we would say that the "function" of the segment is to perform the main operation that is contained in it. For example, the main operation in the third segment of LMc is LSby, a substitution. The function of this segment in the matching program is to substitute one sub-expression for another in one of the expressions being matched. Hence, we will name the segment after the main operation: LMc(Sby). Similar designations will be used for the other segments of routines. This notation emphasizes the fact that each routine consists in a sequence (or branching tree) of main operations that are connected by procedural and test operations. Thus, an abbreviated description of the matching routine might be given as:

---

[9] As we shall see, 1.01 is not held in storage memory, but is represented, instead, by two routines for actually performing the replacements.

[10] The exclamation point in front of an expression indicates that the expression in question is asserted to be true. To designate an expression whose truth has not been demonstrated, we will use a question mark preceding the expression.

[11] A technical difficulty arises from the fact that there is an infinite number of valid substitutions. This difficulty can be removed rather easily, but the question is irrelevant for the purposes of this paper.

LMc
T Perform diagnostic tests
LMc Recursion of matching with next elements in logic expression
Sby Substitute the element y for the element x
Sbx Substitute the element x for the element y
CN Compare variables in x and y
Rp Replace connectives, if required and possible

The Substitution Method

Let us take as our first example the very simple expression, 2.01, for which we have already given a proof. We suppose that, when the problem is proposed, LT has in its theorem memory only the axioms, 1.2 to 1.6. We wish to construct a proof (the one given above, or any other valid proof) for 2.01.

As the simplest possibility, let us consider proofs that involve only the rules of substitution and replacement. We are now able to state the problem thus: how can we search for a proof of the expression by substitution without considering all the valid substitutions in the five axioms? We use two devices to focus the search. Both of these involve "working backward" from the expression we wish to prove--for by taking account of the characteristics of that expression, we can obtain cues as to the most promising lines to follow:

1. In attempting substitutions, we will limit ourselves to axioms (or other true theorems, if any have already been proved) that are in some sense "similar" in structure to the theorem to be proved. The routine that accomplishes this will be called the test similarity routine, CSm.

2. In selecting the particular substitutions to be made in a theorem that has been chosen for trial, we will attempt to match the variables in that theorem to the variables in the expression to be proved. Similarly, we will try to use the rule of replacement to match connectives. The routine in which these various operations occur is called the matching routine, LMc.

Using these devices, the proposed routine for proving theorems—the method of substitution, MSb--works as follows. MSb(Sm): search for an axiom or theorem that is similar to the expression to be proved. MSb(Mc): when one is found, try to match it with the expression to be proved; if a match is successful, the expression is proved; if the list of axioms and theorems is exhausted without producing a match, the method has failed. (Reference to Section III will show that there is another segment, MSb(NAW), that we have not mentioned. The function of this segment will be discussed later in connection with the executive routine.)

To see in detail how the method operates, we next examine the main operations, CSm and LMc, of the two segments of the substitution method. For concreteness, we will carry out these operations explicitly for the proof of the expression 2.01.

2.01 ? $p \rightarrow -p .\rightarrow. -p$

Test for Similarity, CSm. We must state what we mean by similarity. We start from a common-sense viewpoint and regard two propositions as similar if they "look" similar to the eye of a logician. In Section I we have already defined three characteristics of an expression that can be used as criteria of similarity. These are: K, the number of levels in the expression; J, the number of distinct variables in the expression; and H, the number of variables in the expression.[12]

Applying these definitions to 2.01 (routines NK, NJ, and NH, respectively), we find that K = 3, J = 1, and H = 3. That is, 2.01 has three levels, one distinct variable (p), and three variable places. We may write this:

$$D(2.01) = (3,1,3)$$

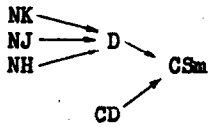
In the same way, we can write descriptions for the various sub-expressions contained in 2.01--in particular, the sub-expressions to the left and to the right of the main connective, respectively. We have for these:

$$DL(2.01) = (2,1,2); \text{ and } DR(2.01) = (1,1,1)$$

Now, we say that two expressions, x and y, are similar if they have identical left and right descriptions; i.e., if DL(x) = DL(y) and DR(x) = DR(y). The routine for determining whether two theorems are similar, CSm, consists of two segments: (1) CSm(D), a description segment, and (2) CSm(CD), a comparison of descriptions. The description segment is made up of four description routines, D, one each to compute DL(x), DR(x), DL(y), and DR(y). The comparison segment is made up of two compare description routines, CD, one of which compares DL(x) with DL(y), the other DR(x) with DR(y).

A diagram of the hierarchy of principal subroutines in testing similarity will look like this:

[12] The assertion is that two expressions having the same description "look alike" in some undefined sense; and hence if we are seeking to prove one of them as a theorem, while the other is an axiom or theorem already proved, then the latter is likely construction material for the proof of the former. Empirically, it turns out that with the particular definition of similarity introduced here, in proving the theorems of Chapter 2 of Principia Mathematica about one theorem in five that is stored in the theorem memory turns out to be similar to the expression we are seeking to prove. It is easy to suggest a number of alternative, and quite different criteria that would be equally symptomatic of "similarity." Uniqueness is of no account here; all we are concerned with is that we have some criteria that "work"— that select theorems suitable for matching.



In the case of 2.01, the segment MSb(Sm) will search the list of axioms and theorems and will find that axiom 1.2 is similar to 2.01:

1.2 ! p ∨ p .→. p

for it, too, has the descriptions: DL(1.2) = (2,1,2); DR(1.2) = (1,1,1). Moreover, 1.2 is the only axiom that has this description.

Matching Expressions, LMc. Next we carry out a point-by-point comparison between 2.01, the expression to be proved, and 1.2, the axiom that is similar to it. We start with the main connectives, and work systematically down the tree of the logic expressions—always as far as possible to the left. In the present case the order in which we will match is: main connective (P = none), connective of left sub-expression (P=L), left variable of sub-expression, (P=LL), right variable of sub-expression (P=LR), and right sub-expression (P=R).

The matching routine is fairly complicated, consisting of six segments, but not all segments are employed each time two elements are matched. The first segment, LMc(T), and the initial operations of most of the other segments consist of tests that determine whether the two elements to be matched are already identical, whether they can be made identical by substitution (if one is a free variable) or by replacement (if both are connectives), or—finally—whether matching is impossible. The second segment, LMc(LMc), is a recursion of the matching routine with each of the next lower pair of elements in the tree of the expression. This recursion segment operates only if the elements to be matched in LMc are identical connectives (or have been made so).

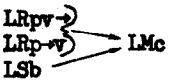
The third and fourth segments, LMc(Sby) and LMc(Sbx), apply the rule of substitution when the tests have shown this to be appropriate. LMc(Sby), which is executed whenever E(x) is a free variable,[13] simply substitutes the expression X(y) for E(x). LMc(Sbx), which is executed whenever E(y) is a free variable, substitutes the expression X(x) for E(y). In both cases, of course, substitution must take place throughout the whole expression in which the free variable occurs. This is taken care of automatically by the process LSb. Also, since LMc matches X(x) to X(y), LMc(Sby) has priority over LMc(Sbx), as a careful examination of the test network will reveal.

The fifth segment, LMc(CN), reports the successful termination of the matching program if E(x) and E(y) are identical variables, its failure if they cannot be made identical by substitution.

The sixth segment, LMc(Rp), operates when E(x) and E(y) have different connectives. The segment replaces the connective in x by the connective in y whenever this replacement is legitimate, and then returns control to the recursion segment.

By virtue of the recursion segment, the matching routine will attempt to match each pair of elements; if successful, will proceed to the next pair; if unsuccessful, will report failure. Hence, the routine will continue until it makes the theorem that is being matched identical with the expression to be proved, or until the matching fails.

The hierarchy of principal routines looks like this:



Returning to our specific example of the two similar expressions, 1.2 and 2.01, we carry out the matching routine as follows:

2.01 ? p → -p .→. -p
1.2 ! A ∨ A .→. A

(We use A instead of p in 1.2 to indicate that the variable is free (F).)

a. The main connectives agree: both are →.
b. Proceeding downward to the left, the connective is → in 2.01, but ∨ in 1.2. To change the ∨ to →, we must have (because of the definition, 1.01, a - before the left-hand A in 1.2. This we can obtain by making the substitution of -B for A in 1.2. Having carried out this substitution, and having then replaced (-B ∨ -B) with (B→ -B), we have the following situation:

2.01 ? p → -p .→. -p
1.2' ! B → -B .→. -B

c. Proceeding again to the left, we find B in 1.2', but p in 2.01. We therefore substitute p for B in 1.2', and now find (after recursion through the remaining two elements) that we have a complete match:

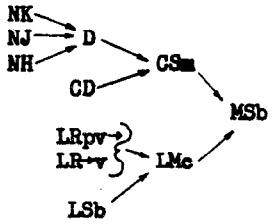
2.01 ? p→ -p .→. -p
1.2'' ! p→ -p .→. -p

Thus, we have discovered a proof of 2.01 (in fact, precisely the proof we gave before), which consists in substituting the variable -p for the variable in 1.2, and replacing the connective ∨ in 1.2 with →.

This completes our outline of the method of substitution as a routine for discovering proofs in symbolic logic. The method may be viewed as an information process that is composed of a considerable number of more elementary in-

[13] Essentially, a variable is free when no substitution has yet been made for it. After any substitution it is bound and no longer available for subsequent substitutions. As previously noted, all variables in expressions stored in the theorem memory are free.

formation processes arranged to operate in highly conditional sequences. Each of the main components—the test for similarity routine, and the matching routine—is made up, in turn, of subroutines. The test conditions that control the branchings of the sequences depend in a number of instances upon the outcomes of searches through the theorem memory. Hence, the method of substitution represents a complex information process in the sense in which we have defined the term. Combining the two diagrams depicted above, we can illustrate the hierarchy of the main operations that enter into the substitution method:



The method is a heuristic one, for it employs cues, based on the characteristics of the theorem to be proved, to limit the range of its search; it does not systematically enumerate all proofs. This use of cues represents a great saving in search, but carries the penalty that a proof may not in fact be found. The test of a heuristic is empirical: does it work?

Moreover, the cues that are used in the method are not without cost. For example, in order to limit matching attempts to "similar" theorems, theorems must be described and compared. The net saving in computing time, as compared with random search, is measured by the reduction in the number of theorems that have to be matched less the cost of carrying out the search and compare for similarity routines. Stated otherwise, cues are economical only if it is cheaper to obtain them than to obtain directly the information for which they serve as cues.

To be sure, we have found a proof for one proposition in Principia; but how general is the substitution method? On examination of the 67 propositions in Chapter 2 of Principia, it appears that some 21 can be proved by the method of substitution, including for example: 2.01, 2.02, 2.03, 2.04, 2.05, 2.10, 2.12, 2.21, 2.26, 2.27. The remaining propositions evidently require more powerful techniques of discovery and proof. It is evident, for instance, that we must employ the rule of detachment.

The Method of Detachment

We will describe next the method of detachment, MDt, which, as its name implies, incorporates the rule of detachment. The method, of course, is not synonymous with the rule, but includes also heuristic devices that select particular theorems to which the rule is applied.

Let us review the principle of logic that underlies the method. Suppose LT must prove that expression A is a theorem; and assume that there are in the theorem memory two theorems, B and B→A. Then, by application of the rule of detachment to B and B→A, A is derivable immediately.

We can generalize this procedure by combining matching (substitution and replacement) with detachment. Assume that the theorem memory contains B" and B'→A'; that A is obtainable from A' by matching; and that B' is obtainable from B" by matching. Then we can construct a proof of A as follows: (1) By matching with B", B' is a theorem. (2) Since B'→A' is also a theorem, it follows by detachment that A' is a theorem. (3) By matching with A', A is a theorem.

This settles the problem of constructing a valid proof by the method of detachment. From the standpoint of the discovery of a proof employing this method, the trick lies again in narrowing down the search for B'→A' and B", so that these do not have to be sought through a very large scale trial-and-error search and substitution program.

Structure of the Detachment Method. The basic structure of the detachment method is quite similar to that of the substitution method, for both methods utilize the same basic operations. The first two segments of the detachment method, MDt(SmV) and MDt(SmCt), carry out searches for similar expressions, in a way that will be indicated more precisely below. The next segment, MDt(Mc), carries out a matching of any expression so found with the theorem to be proved. If the matching is successful, a new problem is created by the segment MDt(F). This problem is then attacked, in the final segment, MDt(MSb), by the method of substitution.

Again, designate by A the expression to be proved. In MDt(SmV) we search the theorem memory for theorems whose right sides are similar (by the test, CSm, described previously) to the whole expression A. If we find such a theorem (call it T), we go to segment MDt(Mc), and apply the matching operation to the right side of T and to A. If we are successful in the matching, we find the left side of T, MDt(P); and seek to prove by the method of substitution that it is a theorem, MDt(MSb). For if the left side of T is a theorem and T is a theorem, then by detachment, the right side of T is a theorem. But A can be obtained from the right side of T by substitution, hence is a theorem. (Note that a check is made to see that T has → for a connective.)

Contraction. If the detachment method fails to find a proof in the manner just described, a new attempt is made by means of the second segment, MDt(SmCt), employing a different criterion of similarity from the one we have used thus far. If the theorem is similar, the method proceeds with the matching segment exactly as before.

To see what is involved in this generalized notion of similarity, let us consider two expressions, A and A', with different descriptions. If A has more levels and variable places than A', it is still possible that A is derivable from A' by substitution—specifically, by substituting appropriate molecular expressions for the variables of A. For example, take as A the expression:

2.06 ? p→q .→: q→r .→. p→r

for which we have DL(2.06) = (2,2,2), DR(2.06) = (3,3,4); and take as A' the expression:

A' ? a .→. b→c

for which we have DL(A') = (1,1,1), DR(A') = (2,2,2).

If in A' we substitute p→q for a, q→r for b, and p→r for c, we obtain 2.06. Operating in the reverse direction, if we contract 2.06 by making the inverse substitutions, we obtain A'. We can therefore refer to A' as "2.06 viewed as contracted."

Since the purpose in searching for similar theorems is to find appropriate materials to which to apply the matching routine, there is no reason why we should not use this more general notion of similarity if it proves effective in finding materials that are useful.

In general, what parts of an expression should be considered as units in the search for proofs is not a "given" for the problem solver. LT makes an explicit decision each time it looks for similar expressions as to what subexpressions will be taken as units. In contracting 2.06, a decision has been made that the elements p, q, and r are too small, and that more aggregative elements, e.g., (p→q) = a, should be perceived as units.

Examination of the routines for describing expressions (NH, NK, NJ) will reveal that these routines in fact count units rather than variables. Normally, the variables are the units used in description, for VV precedes CSm in every program except MDt. In the latter program, however, it is sometimes useful to view expressions as contracted, by means of VCt.

Example of Proof by Detachment. To illustrate the method of detachment, let us carry out explicitly the proof of 2.06:

2.06 ? p→q .→: q→r .→. p→r

The reader may verify that this theorem cannot be proved by substitution in the axioms and earlier theorems. Moreover the detachment method without contraction will also fail, for there is no theorem whose right side is similar to 2.06. However, we have already seen that when we contract 2.06, we obtain:

A' ? a .→. b → c

where p→q has been contracted to a, q→r to b, and p→r to c. We now have DL(A') = (1,1,1) and DR(A') = (2,2,2), descriptions that are identical with the descriptions of the sub-expressions of the right side of 2.04.

2.04 ! A →. B → C : → : B → . A → C
A' a → . b → c

Having selected 2.04 by use of the routine MDt(SmCt), we now proceed to match its right side with 2.06 in segment MDt(Mc):

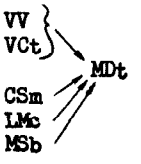
2.04 ! A .→. B → C :→: B .→. A → C
2.06 ? p→q .→: q→r.→.p→r
2.04' ! q→r .→: p→q.→.p→r .:→:. p→q .→: q→r.→.p→r

We have now created a new problem to replace the original one: to prove that the left side of 2.04' (the part underscored) is a theorem. We apply the method of substitution, MDt(MSb). The search of the theorem memory discloses 2.05 to be similar to the left side of 2.04', and we proceed to match them:

2.04'L ? q→r .→: p→q .→. p→r
2.05 ! A→B .→: C→A .→. C→B

It is easy to see that with the substitution of q for A, r for B, and p for C, the matching will be successful. Hence we have B (2.05 with the indicated substitution), and B→A (2.04'), from which A (2.06) follows by the rule of detachment.

The diagram below summarizes the principal routines incorporated in the method of detachment. A comparison of this diagram with the one for the substitution method shows clearly that both methods rest on the same component processes, with minor modifications and new combinations and conditions. The sole new process involved in detachment is the viewing of theorems as contracted.



The Chaining Method

A number of expressions that do not yield to the method of substitution can be proved by the method of detachment. We shall add an additional method, however, to the repertoire available to LT. We shall call this method chaining, MCh. Like the methods previously described, chaining involves heuristic procedures which we shall consider first.

Theorem 2.06, which we have just proved, embodies one form of the principle of the syllogism (2.05 is another form of this principle). Now suppose T_1, (p→q) is a true theorem, and T_2, (q→r) is another true theorem. Theorem 2.06 is of the form:

$$T_1 .\rightarrow. T_2 \rightarrow E$$

where E is (p→r), an expression not known to be true. By detachment, from ! T_1 and ! $T_1.\rightarrow.T_2\rightarrow E$, we get ! $T_2\rightarrow E$. By a second detachment, from ! T_2 and ! $T_2\rightarrow E$, we get ! E. Hence, if we know p→q and q→r to be true, we can construct a proof of p→r by means of two detachments with the use of 2.06. Instead of carrying through this derivation explicitly in each instance, we simply construct a program that makes direct use of the transitivity of syllogism. This proof method is the basis for chaining.

Suppose that we wish to prove A→C. We search for a theorem, T (with → for a connective) whose left side is similar to A, using the segment MCh(SmF). We match the left side of T with A, MCh(McF), and if we are successful, we have then proved a theorem of the form A→B, for T, as modified by matching, is of this form. We check first, in segment MCh(McR) whether we can simply match B to C. If we succeed, we have proved the theorem. If we fail, we now construct, by segment MCh(S), the expression B→C, and attempt to prove this expression by substitution, MCh(MSb). If we are successful, we now have a chain: A→B, B→C. Then by syllogism, as indicated above, we obtain A→C, the expression we wished to prove.

The procedure just described is chaining forward. Alternatively, we may chain backward. That is, to prove A→C, we may search for a theorem of the form B→C; then try to prove A→B by substitution.

Proof by the chaining method is illustrated by:

2.08 ? p → p

A search for theorems that have left sides similar to 2.08 yields 1.3, 2.02, and 2.07. The latter is:

2.07 ! p .→. pvp

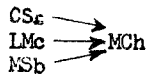
If we take 2.07 as the (A→B) of the schema given above, then B is (pvp). Two theorems have left sides similar to B: 1.2 and 2.01. An attempt to match the left side of 2.01 to the right side of 2.07 will be unsuccessful, but the matching is immediate with 1.2:

2.07 ! p .→. pvp
1.2 ! pvp .→. p

Hence we can take 1.2 as the (B→C) of the chaining method. We now form (A→C) by joining the left side of 2.07 to the right side of 1.2 by →. The result is 2.08:

2.08 ! p .→. p

The chaining method is summarized by the following diagram:



The Executive Routine

It remains to complete the specification of LT in two directions; first, to assemble the three methods that have been described into a coherent program; and second, to show how the information processes in terms of which LT has been described here can be specified precisely in terms of the elementary processes listed in Section I. The latter task is carried out in detail in Section III. We will turn our attention here to the former, which is embodied in the executive routine, Ex.

In its first segment, Ex(R), the executive routine reads a new expression that is presented to it for proof, and places it in a working memory.[14]

In the next three segments, Ex(MSb), Ex(MDt), and ExMCh), successive attempts are made to prove the expression by the methods of substitution, detachment, and chaining, respectively. If a proof is obtained by one of these methods, the executive routine writes the proof, Ex(WP); and stores the newly-proved theorem (changing all its variables to free variables) in the theorem memory, Ex(ST).

To explain what happens if the three methods are unsuccessful, we have to take up some details that were omitted above. These have to do with the creation of subsidiary problems and with stop rules.

Subsidiary problems. Both detachment and chaining are two-step methods. Suppose we wish to prove A. In detachment, we try to find a theorem, B→A, and if we are successful, we then try to prove B. The task of proving B we may call a subsidiary problem.

Suppose we wish to prove a→b. In chaining, we try to find a theorem, a→c, and if we are successful, we then try to prove c→b. The task of proving c→b is also a subsidiary problem.

Within both the detachment and chaining methods, only the method of substitution is applied to the subsidiary problem. If that method fails, failure is reported for the main problem. But before control is shifted back to the executive routine, the main element of the subsidiary problem is stored in the problem list, P, in the storage memory. (The operation that stores the problem in the problem list is the operation SEN that can be found in segment MDt(P) and segment MCh(P).)

When the three methods have failed for a given problem, the executive routine stores it in the inactive problem list, Q. It then selects from the problem list, P, an expression that is, in a certain sense, the simplest--specifically, an expression with the smallest possible number of levels, K, Ex(CK). It erases this new subsidiary problem from P; checks to make certain it does not duplicate one previously attempted, Ex(CX); and then tries to solve this subsidiary problem by the methods of detachment and chaining.[15] This sequence is repeated until some subsidiary problem is solved (in which case the main problem is also solved), or until no problems remain on the problem list, or until the other stop rule, to be described, comes into operation. In the latter two cases, the routine reports that it is unable to prove the theorem, Ex(WNP).

[14]Certain segments of Ex, in particular Ex(R), Ex(WP), Ex(ST) and Ex(WNP), are not written in Section III in terms of the primitives but are simply indicated by parentheses. It would be rather simple to formalize them, but this would further lengthen the description of the program.

[15]There is no need to attempt to prove the subsidiary problem by substitution, since an unsuccessful substitution attempt was made immediately before the expression was stored in the subsidiary problem list.

The check to prevent duplication of subsidiary problems, Ex(CX), is handled as follows: for each problem that is selected from list P by Ex(CK), a check is made, by Ex(CX), against all expressions in the inactive problem list, Q, and if the new problem duplicates any expression found there, it is dropped. The main operation of this segment, CX, applies the same basic tests of identity of elements that are applied in the matching program, but does not modify the expressions to make them match.

Stop Rules. Since all proof methods may fail, even if the expression given to LT is a genuine theorem, the executive routine needs a stop rule. One stop rule is provided by the exhaustion of list P, but there is no guarantee that the list will ever be exhausted. A second stop rule is provided by an operation that measures the total amount of "work" that has been done in attempting to prove a theorem, and that terminates the program with a "no proof" report when the total work exceeds a specified amount. The first operation in the substitution routine, NAW, tallies one for each time the routine is used. This tally is kept in a special location, W, in the storage memory. The executive routine, just before it seeks a new subsidiary problem, checks the cumulative tally in this register, Ex(CW), and if the tally exceeds a given limit, terminates the program. Since the substitution routine is used in each of the methods, the number of substitutions attempted seems to be one reasonable index of the amount of work that has been done.

This stop rule operates as a global constraint on the total work applied in trying to prove a single theorem. The rule does not govern the direction in which this effort is expended. The latter is determined by the priority rule previously described for selecting subsidiary problems from the problem memory and by the other elements of LT's program.

Learning Processes

The program we have described is primarily a performance program rather than a learning program. But, although the program of LT does not change as it accumulates experience in solving problems, learning does take place in one very important respect. The program stores the new theorems it proves, and these theorems are then available as building blocks for the proofs of subsequent theorems. Thus, in the theorems used as examples in this paper, 2.06 was proved with the aid of 2.05 and 2.04, and 2.08 was proved with the aid of 2.07. Without this form of learning it is doubtful whether the program would prove any but the first few theorems in Chapter 2 in a reasonable number of steps.

III

The Complete Program for the Logic Theorist

This Section is divided into two parts. The first part constitutes the program as described in the text, including the following routines: Ex; MCh, MDt, MSb; LMc, LSb, LRp→v, LRpv→, VV, VCt; CX; CSm, CD, D, NK, NH, NJ. These routines are preceded by a list of the most important primitive IP's—those that are used in several routines. Following each routine is a supplementary list of primitive IP's used in the definition of that routine.

The second part of this Section consists of routines for five IP's—those Store instructions that are marked with asterisks (*)—which up to this point have been treated as primitives.

Principal Primitive Instructions

A	OPER	L	G	R	B	
	B				b	Branch to b (→b).
	BHB					In higher instruction, →b.
	BHN					In higher instruction, →next.
	FEF	x	y		b	Find the first E in A(x) and put in y; if none, →b.
	FEN	x	y		b	Find the E in A(x) next after E(y), put in y; then →b. If none (end of list), →next.
	FL	x	y			Find EL(x) and put in y; if none, leave y blank.
	FR	x	y			Find ER(x) and put in y; if none, leave y blank.
	PE	x	y			Put E(x) in E(y); E(x) remains.
	S	x				Store E(x) back in A(x) (match on P); if not there, store E(x) at end of A(x).
	SEN	x	y			Store E(x) as next E in A(y); E(x) now last item in A(y).
	*SX	x	y			Store a copy of X(x) at (new) A(y). E(x) = M.
	TC	x			b	If C(x) = → (implies), →b.
	TV	x			b	If E(x) = V, →b.

A	OPER	L	C	R	B	Seg.	Executive routine
	Ex						
	(Read problem X)					R	
	(Put EM(X) in 1)						
	-MSb	1		C		MSb	
A	-MDt	1		G		MDt	
	-MCh	1		G		MCh	
	SEN	1	Q				X(1) is finished.
	CWG			H		CW	
B	FEF	P	1	H		CK	Find problem with
	NK	1					lowest K.
C	-FEN	P	2	D			
	NK	2					
	CKG	2	1	C			
	PE	2	1				
	PK	2	1				
	B			C			
D	E	1	P			CX	Remove duplicates
	FEF	Q	3	F			of previous problems.
E	CX	1	3	B			
	FEN	Q	3	E			
F	B			A			
G	(Write proof.)					WP	Succeeds in proving P.
	(X(1) a theorem)					ST	
	(Stop)						

```
H  (Write:no proof) WNP   Fails to find proof.
   (Stop)
```

Primitives

```
   CKG   x y   b    If K(x)>K(y), →b.
   CWG         b    If W (work done) > limit, →b.
   E     x y        Erase E(x) in A(y).
```

Note: There are six IP's in the executive routine that are not formally defined in LT. These are written in parentheses above: read problem, find problem and put in working memory 1, write proof, store expression as theorem, write "no proof", and stop.

```
A  OPER   L C R B    Seg.
```

Substitution method
If can't prove X(x) by substitution, →b.

```
   MSb    x      b
```

```
   NAW                NAW  Count one unit of work.
   VV     L           Sm
   FEF    T 1    C
A  VV     1
   CSm    L 1    D
B  FEN    T 1    A        Find next T and repeat.
C  BHB

   SX     1 2        Mc
   LMc    2 L
   BHN
```

Primitives

```
   NAW                Add one to W (work done).
```

```
A  OPER   L C R B    Seg.
```

Matching routine
Match X(x) to X(y); if can't, →b.

```
   LMc    x y    b
```

```
   CGG    C L    A    T
   CGG    L C    C        Now G(x) = G(y).
   TV     L      E
   TV     C      D
  -CC     L C    F
   FL     L 1         LMc
   FL     C 2
   LMc    1 2    H        Mc left subexpression.
   FR     L 3
   FR     C 4
   LMc    3 4    H        Mc right subexpression.
   BHN

A  TV     L      H   Sby
  -TF     L      H
B  NSGG   L C
   FM     L 5             Assures Sb everywhere.
   LSb    C L 5
   BHN

C  TV     C      H   Sbx
D -TF     C      H
   NSGG   C L
   FM     C 5             Assures Sb everywhere.
   LSb    L C 5
   BHN
```

```
E  TF     L      B   CN
  -TV     C      H
  -CN     L C    D
   BHN

F -LRp→v  L      G   Rp   LRp's are self-testing.
   LRpv→  L      H
G  LMc    L C    H
   BHN
H  BHB
```

Primitives

```
   CC     x y    b    If C(x) = C(y), →b.
   CGG    x y    b    If G(x) > G(y), →b.
   CN     x y    b    If N(x) = N(y), →b.
   FM     x y         Find EM(x) and put in y.
   NSGG   x y         Subtract G(x) from G(y).
   TF     x      b    If E(x) is free, →b.
```

```
A  OPER   L C R B    Seg.
```

Substitution routine
Substitute X(x) for E(y) (=V) in X(z) (=M).

```
   LSb    x y z
```

```
   FEF    L 1    F   F
A  CPS    1 1    B        E(1) must belong to X(x).
   CN     1 C    G
B  FEN    L 1    A
C  FEF    R 2    F   Sb   Search through X(z).
D -CN     2 C    E
   PE     L 3
   NAGG   2 3         G's add in Sb.
   SXE    3 2
E  FEN    R 2    D        Find next E(z), repeat.
F  BHN

G  AN     4          LSb
   LSb    4 C R
   B           C
```

Primitives

```
   AN     x           Assign an unused name to E(r).
   CN     x      b    If N(x) = N(y) →b.
   CPS    x y    b    If E(x) subelement of E(y)→b
                      (P(x) ⊃ P(y)).
   NAGG   x y         Add G(x) to G(y); result in
                      G(y).
  *SXE    x y         Store X(x) in A(y) in place
                      of E(y) (=V).
```

```
A  OPER   L C R B    Seg.
```

Detachment method
If can't prove X(x) by detachment, →b. Store new problems in P.

```
   MDt    x      b
```

```
   FEF    T 1    C   T
A  TC→    1      B        T must have C = →.
   VV     1
   FR     1 2
   VV     L          SmV
   CSm    L 2    D
   VCt    L          SmCt  Change view.
   CSm    L 2    D
B  FEN    T 1    A        Find next T and repeat.
C  BHB
```

```
D  SX    1 3          Copy, to work on T.
   FR    3 4
   LMc   4 L   B  Mc
   FL    3 5      P
   SXM   5 6          Create new X.
   S     6            Stored away fixed ME.
   SEN   6 P
   MSb   6     B  MSb
   BHN
```

Primitives

```
   *SXM   x y        Store X(x) at (new) A(y) as
                      main expression.
```

```
A  OPER   L C R B  Seg.
                          Chaining Method
                           If can't prove X(x) by
                           chaining, →b; Store new
   MCh    x     b          problems in P.
   -TC→   L     D  T    C(x) must be →.
    VV    L
    FL    L 1
    FR    L 2
    FEF   T 3    D
A  -TC→   3      C        T must have C = →.
    VV    3
    SX    3 4             Copy, to work on T.
    FL    4 5
    FR    4 6
   -CSm   1 5   B  SmF
   -LMc   5 1   E  McF
B  -CSm   2 6   C  SmB
   -LMc   6 2   F  McB
C   FEN   T 3   A          Find next T and repeat.
D   BHB
E   PE    2 5              Put E(2) and E(6) in
    PE    6 1               proper wkg. memory.
   -LMc   1 5   G  McR
F   AM    7         S      Create EM for new X.
    PC→   7                Fix connective.
    S     7                Store parts.
    SEN   7 P
    SXL   1 7
    SXR   5 7
    MSb   7     C  MSb
G   BHN
```

Primitives

```
   PC→   x          Put C(x) = → (implies).
  *SXL   x y        Store X(x) in A(y) as XL(y).
  *SXR   x y        Store X(x) in A(y) as XR(y).
```

```
A  OPER   L C R B  Seg.
                          Replacement of → with v.
                           If C(x) = →, replace
   LRp→v  x     b          with v; if not →b.
   TC→    L     A  T
   BHB
A  PCv    L        Pv     Fix E(x).
   S      L
   FL     L 1             Fix EL(x).
   NAG    1
   S      1
   BHN
```

Primitives

```
   NAG   x          Add one to G(x).
   PCv   x          Put C(x) = v.
```

```
A  OPER   L C R B  Seg.
   VV     x                 View variables as units.
   FEF    L 1       T
A  PUB    1                 Erase old unit.
  -TV     1    B
   PU     1         P
B  S      1
   FEN    L 1  A            Find next E and repeat.
   BHN
```

Primitives

```
   PU    x          Make E(x) a unit, (U).
   PUB   x          Make U(x) blank.
```

```
A  OPER   L C R B  Seg.
                          View as contracted
                           Make units of binary
                           expressions and
   VCt    x                 isolated variables.
   TV     L     C  T
   FL     L 1       VCt
   FR     L 2
   TV     1     B
   VCt    1                 Recursion
   TV     2     E
A  VCt    2                 Recursion
   PUB    L
   S      L
   BHN

B  -TV    2     D
   PUB    1         Ct      Blank V's of Ct unit.
   S      1
   PUB    2
   S      2
   TN     L     C           Give X(x) a name if
   AN     L                  one needed.
C  PU     L
   S      L
   BHN

D  PU     1         VV      Make left (isolated)
   S      1                  variable a unit.
   B            A           XR(x) still to be done

E  PU     2
   S      2                 Make right (isolated)
   BHN                       variable a unit.
```

Primitives

```
   AN    x        Assign E(x) an unused name.
(See VV for PU and PUB)
   TN    x    b   If E(x) has a name →b.
```

```
A  OPER   L C R B  Seg.
                          Replacement of v with →.
                           If C(x)=v and G(EL(x))
                           >0, replace v with →;
```

```
   LRpv→ x      b          if not →b.

  -TCv   L      A  T
   FL    L 1
   TGG   1       C
  -TV    1      A
  -TSb   1      B
A  BHB

B  PE    1 2        Sb
   NAG   2
   FM    2 3
   LSb   2 1 3
   FL    L 1
C  PC→   L          P     Fix x.
   S     L
   NSG   1
   S     1
   BHN
```

Primitives

```
   FM    x y              Find EM(x) and put in y.
   NAG   x                Add one to G(x).
   NSG   x                Subtract one from G(x).
   PC→   x                Put C(x) = →.
   TGG   x      b         If G(x) >0 →b.
```

```
A  OPER  L C R B  Seg.
                          Compare expressions
                          Compare X(x) with X(y); if
   CX    x y    b         they match, →b.

   CGG   L C    B  T
   CGG   C L    B         G(L) = G(R), otherwise →B.
   TV    L      A
   TV    C      B
  -CC    L C    B         C(L) = C(R)
   FL    L 1       CX     Recursion down tree of
   FL    C 2              expressions.
  -CX    1 2    B
   FR    L 3
   FR    C 4
  -CX    3 4    B
   BHB

A -TV    C      B  CN     L and C both variables;
  -CN    L C    B         with identical names.
   BHB

B  BHN
```

Primitives

(For CC, CGG, and CN, see LMc)

```
A  OPER  L C R B  Seg.
                          Similar expressions test
                          If DL(x) = DL(y) and
   CSm   x y    b         DR(x) = DR(y), →b.

   FL    L 1       D
   FR    L 2
   D     1
   D     2
   FL    C 3
   FR    C 4
   D     3
   D     4
  -CD    1 3    A  CD
  -CD    2 4    A
   BHB

A  BHN
```

```
A  OPER  L C R B  Seg.
                          Compare descriptions
                          If K(x) = K(y), J(x) = J(y),
   CD    x y    b         and H(x) = H(y) →b.

  -CK    L C    A         Def: If K(x) = K(y) →b.
  -CJ    L C    A         Def: If J(x) = J(y) →b.
  -CH    L C    A         Def: If H(x) = H(y) →b.
   BHB

A  BHN
```

```
A  OPER  L C R B  Seg.

   D     x                Describe

   NK    x
   NJ    x
   NH    x
   BHN
```

```
A  OPER  L C R B  Seg.

   NH    x                Count variable places

   FEF   L 1    C
A -CPS   1 L    B
  -TU    1      B
   NAH   L
B  FEN   L 1    A
C  BHN
```

Primitives

```
   CPS   x y    b   If E(x) subelement of E(y) →b.
                    (P(x) ⊃ P(y)).
   NAH   x          Add one to H(x)
   TU    x      b   If E(x) is a unit, →b.
```

```
A  OPER  L C R B  Seg.

   NJ    x                Count distinct variables

   AA    1                List for counted-V.
   FEF   L 2    E   F     Find first E of X(x).
A -CPS   2 L    D
  -TU    2      D
   FEF   1 3    C         Find first V of list.
B  CN    2 3    D   CN
   FEN   1 3    B         Find next V of list.
C  SEN   2 1
   NAJ   L          A
   FEN   L 2    A         Find next E of X(x).
E  BHN
```

Primitives

```
   AA    x          Assign an unused list of A(x).
   CN    x y    b   If N(x) = N(y), →b.
```

CPS	x y	b	If E(x) subelement of E(y), →b. (P(x)⊃ P(y)).
NAJ	x		Add one to J(x).
TU	x	b	If E(x) is a unit, →b.

A	OPER	L	C	R	B	Seg.
	NK	x				Count levels
	TU	L		A		T
	TB	L		B		
	FL	L	1			NK
	NK	1				
	FR	L	2			
	NK	2				
	CKG	2	1	C		CK
	PK	1	L			KL
A	NAK	L				
B	BHN					
C	PK	2	L			KR
	B			A		

Primitives

CKG	x y	b	If K(x) > K(y), →b.
NAK	x		Add one to K(x).
PK	x y		Put K(x) in K(y).
TB	x	b	If E(x) is blank →b.
TU	x	b	If E(x) is a unit →b.

PART 2: Reduction of procedural processes [*S]

The Store instructions that rewrite expressions in various ways can be reduced to processes more like the rest of the primitive set. The new primitives required are (a) two (PA and CP) which belong to types of operations already considered, and (b) four of a new type to manipulate the P sequences. The latter operations insert and delete subsequences from the front end of a given sequence. Thus if P = LRRL and P' = LRRLRLR, then P'' = P' - P = RLR and P'' + P = LRRLRLR. Observe that subtraction can only be performed when the subtrahend is an initial segment of the minuend, and also that addition is not commutative All these routines involve bringing in the elements, one by one, modifying them and storing them in the new list.

Store a copy of X(x) at (new) A(y) (E(x) = M).

A	OPER	L	C	R	B
	SX	x	y		
	AA	C			
	FEF	L	1		B
A	PE	1	2		
	PM	C	2		
	S	2			
	FEN	L	1		A
A	BHN				

Store X(x) in A(y) in place of E(y) (E(y) =V) (take E(x) from w.m.)

A	OPER	L	C	R	B
	SXE	x	y		
	FEF	L	1		D
A	CP	L	1		E
	CPS	1	L		C
	PE	1	2		
B	PM	C	2		
	HSPP	L	2		
	HAPP	C	2		
	S	2			

Store X(x) at (new) A(y) as main expression

A	OPER	L	C	R	B
	SXM	x	y		
	AA	C			
	FEF	L	1		C
A	CPS	1	L		B
	PE	1	2		
	PM	C	2		
	HSPP	L	2		
	S	2			
B	FEN	L	1		A
C	BHN				
C	FEN	L	1		A
D	BHN				
E	PE	L	2		
	B				B

Store X(x) in A(y) as XL(y).

A	OPER	L	C	R	B
	SXL	x	y		
	FEF	L	1		C
A	CPS	1	L		B
	PE	1	2		
	PM	C	2		
	HSPP	L	2		
	HAPL	2			
	HAPP	C	2		
	S	2			
B	FEN	L	1		A
C	BHN				

Store X(x) in A(y) as XR(y).

A	OPER	L	C	R	B
	SXR	x	y		
	FEF	L	1		C
	CPS	1	L		B
	PE	1	2		
	PM	C	2		
	HSPP	L	2		
	HAPR	2			
	HAPP	C	2		
	S	2			
	FEN	L	1		
	BHN				

Primitives

AA	x		Assign an unused list to A(x).
CP	x y	b	If P(x) = P(y) →b locates "same" element even though V, G, etc. have been modified).
CPS	x y	b	If E(x) subelement of E(y), →b (P(x)⊃ P(y)).
HAPL	x		Add a Left to front of P(x).
HAPR	x		Add a Right to front of P(x).
HAPP	x y		Add P(x) to front of P(y).
HSPP	x y		Subtract P(x) from front of P(y).
PA	x y		Put A(x) in A(y).

Conclusion

In this paper we have specified in detail an information processing system that is able to discover, using heuristic methods, proofs for theorems in symbolic logic. We have confined ourselves to description, and have not attempted to generalize in abstract form about complex information processing. Because of the nature of the description, involving considerable rigor and detail, it may be useful to set out in conclusion the main features of LT, especially as these appear to reflect basic characteristics of complex systems.

First of all, LT can be specified at all only because its structure is basically hierarchical, and makes repeated use of both iteration and recursion. So true is this, that one of LT's

main features, the use of a problem-subproblem hierarchy, is hardly visible in the program at all.

LT offers no guarantee of finding a proof; on the other hand, it brings to its task a number of different heuristic methods for achieving its goals. All of these methods are important in making LT sufficiently powerful to find proofs in most cases, and to find them with a reasonable amount of computation, but not all of them are essential. Without chaining, for instance, LT could still function. The methods MSb and MDt still provide it with ways to prove theorems—and even some theorems more easily provable by MCh would yield to the more directly "brute force" approach of the other two.

LT is still a very simple process compared, for instance, with the array of methods, techniques, and concepts used by a human logician. For example, the concepts of commutativity and associativity are nowhere to be found in LT. The analysis of LT and its variations is a subject for later papers. However, the following facts, based on hand simulation, may help put LT in perspective. LT will prove in sequence most of the 60 odd theorems in Chapter 2 of Principia Mathematica. With some extension in the variety of methods and cues employed, it will prove most of the theorems in Chapter 3, in which another connective, "and," is introduced.[16] We know nothing, as yet, about what will be required for an extension to the predicate calculus or to other types of problem solving.

LT uses similarity-testing and matching as a multi-stage search and selection process. The questions of efficiency involved in such processes have already been commented upon in Section II. Additional variation and complexity enters the program through the alternative modes, VV and VCt, for perceiving the logic expressions in the course of testing similarity and of matching.

In these and other ways, the logic theorist is an instructive instance of a complex information process. We expect to learn more about such processes when we have realized the logic theorist in a computer and studied its operations empirically; and when the logic theorist will have been joined by similar systems capable of performing other complex information processing tasks.

References

1. Bowden, B. V. (ed.), Faster Than Thought (London: Pitman, 1953), pp. 181-198.

2. Hilbert, D., and W. Ackermann, Principles of Mathematical Logic (New York: Chelsea, 1950), Chapter 1.

3. Whitehead, A. N., and Bertrand Russell, Principia Mathematica, vol. 1, 2nd ed. (Cambridge: Cambridge U. Press, 1925).

[16] A program to do this has been developed and hand simulated by Mr. Kalman Cohen.