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.

Summary

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.

---

[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.

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 use of the processes is not fixed and invariable, but is 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 hierarchical, iterative, and recursive organizations of processes.

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 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 under-

---

[2] See B. V. Bowden (ed.), Faster Than Thought (London: Pitman, 1953), pp. 181-198.

taking, but one that the authors feel is required in the present state of knowledge. As a result, the paper abstains largely 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 will be specified. In Section II there is given a verbal description of LT, 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.

---

[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.

This can be a trivial or an unsurmountable 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 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

---

[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.

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--will 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 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 familiarity with elementary symbolic logic,[5] and only a

[5] For definiteness, we have used the system of A.N.Whitehead and Bertrand Russell, Principia Mathematica, vol. 1, 2nd edition (Cambridge: 1925). An introduction sufficient for our purposes will be found in D.Hilbert and W. Ackermann, Principles of Mathematical Logic (New York: Chelsea, 1950), Chapter 1.

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 two connectives taken as primitive by Whitehead and Russell (and by us) 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 corresponds a number of symbols.) An example will show what is meant by these terms.[7] Consider the expression 1.7:

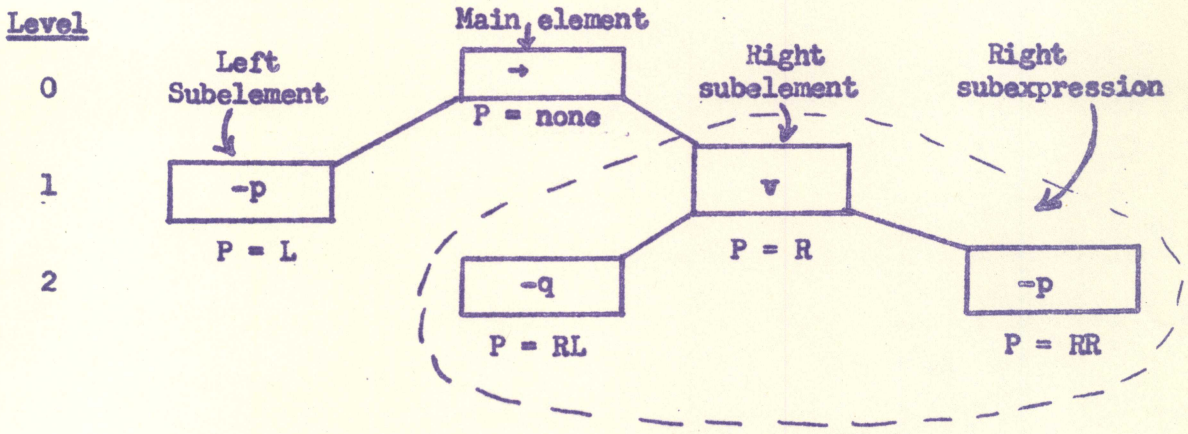
1.7 $-p .\rightarrow. q \vee -p$ ((not-p) implies (q or not-p).)

---

[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 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:



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 sub-expression, to wit, the sub-tree of which that element is the top element.

The symbols in each element provide the following information, some of which will be explained more fully later on.

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.

Symbol

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:

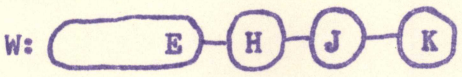
H — The number of variable places in an expression. Thus X(1.7) has three variable places: P = L, RL, and RR; hence, H(1.7) = 3.

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.

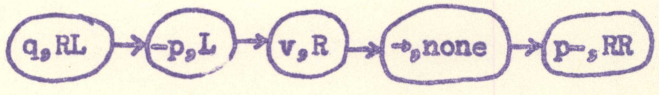
K The number of levels in the expression. The number of levels corresponds to one plus the maximum number of letters in P for any element in the expression. Hence, K(1.7) = 3.

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, and also have space for the symbols H, J, and K. Hence, we can picture a working memory unit as:



The storage memories consist of lists. Each such 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 very 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

Figure 1

of an operation part, three reference places (left (L), center (C), and right (R)), and a branch location, (B). What kinds of operations can be performed by an IPS will depend, first, on what elementary IP's are postulated, and second, on what restrictions are placed on the ways in which 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" elementary processes: 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 with small integers, 1, 2, ..., and with the letters x, y, z.

No direct reference is made in an instruction to any storage memory, except I and P. 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), which 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 can 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 very 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. These locations are designated by letters, A, B, C, . . . In Figure 2, Instruction #1 transfers control 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.

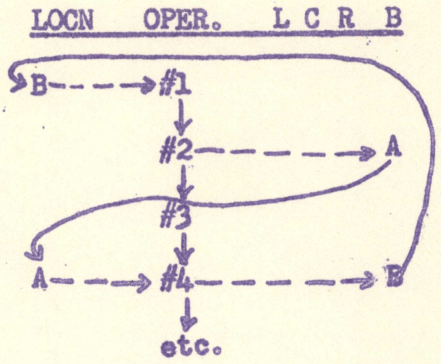


Figure 2

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 the Section III, and its structure 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 this 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 F instructions are very similar: FL (Find the left subelement) and FM (Find the main element).

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

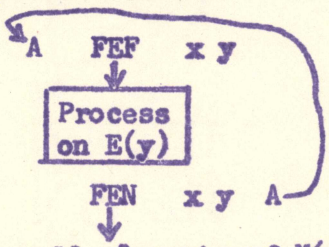
OPER.	L	C	R	B
FEF	x	y		A

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; it is always combined with an instruction, FEN for continuing and terminating the search:

OPER.	L	C	R	B
FEN	x	y		A

Some sort of imp. error on this page — perhaps diagrammatic

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]

---

[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 the Appendix, but in Part II of the Appendix it is shown how they can be defined in terms of simpler elementary operations.

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 on any but a temporary basis.

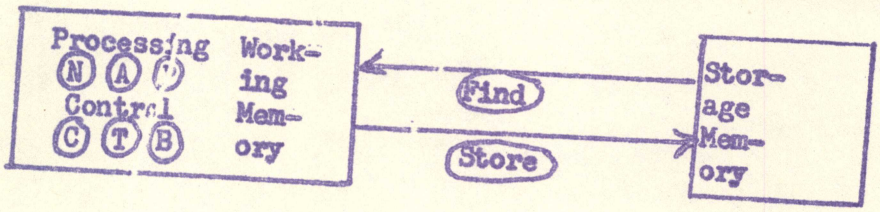


Figure 3

Put instructions transfer information and symbols around the working memory. (Notice that, unlike these, Find and Store instructions dealt only with whole elements.) 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	PK	PU
	PC→		PUB

Numerical instructions carry out various kinds of arithmetic and logical 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, each keeping track of the names or lists already in use. 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 and 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	B
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. These, then are the three Branch operations:

B BHB BHN

It will clarify matters, and provide some introduction to the complete program given in Section III, to 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 the 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 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 having 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.