# CURRENT DEVELOPMENTS IN COMPLEX INFORMATION PROCESSING

by

Allen Newell and Herbert A. Simon

P-850

May 1, 1956

The RAND Corporation
1700 MAIN ST. • SANTA MONICA • CALIFORNIA

(38)

# CURRENT DEVELOPMENTS IN COMPLEX INFORMATION PROCESSING

Allen Newell and Herbert A. Simon*

## I. INTRODUCTION

The term "complex information processing" has been chosen to refer to those sorts of behaviors -- learning, problem solving, and pattern recognition -- which seem to be incapable of precise description in any simple terms, or perhaps, in any terms at all. The advent of the high-speed electronic digital computer has focused attention on the possibilities of specifying and creating information processes of almost unlimited size and complication. But one of the sober facts about current computers is that, for all their power, they must be instructed in minute detail on everything they do. To many, this has seemed to be harsh reality and an irremovable limitation of automatic computing. It seems worthwhile to examine the necessity of the limitation of computers to easily specified tasks. First, we will give a more complete picture of what is meant by complex information processing. Then we will explore in some detail the sorts of processes computers will be required to perform if they are to do complex processing. Next we will consider the problem of instructing the computer which is the issue most closely related to the question raised

*Paper read in a series of Lectures on Computers and Automation in Washington, D.C. May 2, 1956, sponsored jointly by the local chapters of ACM, ASA, ORSA, SAM, SIAM, and TIMS. The authors are grateful to The RAND Corporation and to the Ford Foundation which have supported their respective participations in this study. Mr. Newell is a member of the staff of The RAND Corporation, Mr. Simon a member of the faculty of the Graduate School of Industrial Administration, Carnegie Institute of Technology.

above. Finally, we shall discuss learning and its relation to complex processes.

Current Research. One of the justifications for considering this topic is that it is rapidly leaving the domain of speculation and becoming an area of hard work. Hence, we shall illustrate the various points by sketching some of the current developments in the field. The major example that we will use is some of our own work on programming a computer to find proofs for theorems in mathematical logic. We call this the Logic Theory Machine. This research is currently being done by the authors and Cliff Shaw of The RAND Corporation. The work on the Logic Theory Machine is very recent -- only a few months old -- and has not previously been published or announced.

Application. The examples we will be talking about are chess, abstract mathematical theorizing, and visual pattern recognition. These may seem a bit frivolous and far removed from "real" applications, but, in fact, they are not. For those of you who do not find these tasks intrinsically interesting, we suggest that you view them as the places where we are gaining our fundamental knowledge for the applications you do find interesting. Chess, in particular, is no longer just a game, but is fast becoming a classic task for the fruitful scientific study of complex information processing.

## II. DIFFICULTY AND COMPLEXITY

Chess. Let us consider the playing of chess as a task to be performed, either by a human or a machine. At each position, when the player has the move, the rules of the game provide him with a set of legal alternatives, about thirty in number on the average. His problem is to choose one. His opponent then makes a move, and the player is again presented with a set of about thirty legal alternatives under somewhat changed circumstances. This continues until either a won or lost position is reached, which takes on the average some forty moves.

If we think of the player as exploring each possible sequence of alternatives, we find that there are some $10^{120}$ continuations to be explored. Alternatively, there are some $10^{40}$ positions to become acquainted with. Chess is a difficult task indeed. However, we must distinguish carefully between the difficulty of a task and the complexity of the processes that are used to solve that task.

Simple Processes. The player could perform his task by choosing moves at random. What is meant by saying the task is difficult is that he would surely lose if he used such a simple process. However, the fact that a task is difficult does not necessarily mean it can't be handled by simple processes. For example, complete exploration of all continuations is a simple process, although it is also a very extensive one. There is no difficulty in specifying precisely how it is to be done. A fairly simple machine with sufficient speed and memory could solve even the very difficult task of chess by this simple process. We shall find that we pay the price of complexity of process in order to carry out difficult tasks with limited computing power -- i.e. limited in speed and memory.

Most of the serious exploration of chess and other games prior to 1955 was concerned with simple processes and how well they might actually operate. These efforts stem from a paper by Claude Shannon in 1949,* which is the first serious discussion of which we are aware. The simple processing schemes considered by Shannon involve exploring all sequences of legal moves to a depth of _n_ moves; where _n_ is determined by the computing power available. A numerical evaluation function is applied to each final position. This function usually consists of some sort of weighted sum of features of the position considered to be important on the basis of human experience. Given the evaluation, it is possible to work backwards to decide what move should be made.

The only machine we know of that has actually been coded for chess is a Russian digital computer, ** and we know very little yet about how it plays. On the other hand, at least two machines have been coded for checkers*** and there has been some hand simulation of chess machines. The upshot of

---

* Shannon, C.E., "Programming a computer for playing chess", _Phil_. _Mag_., 41:256-75 (March 1950)

** The computer is BESM at the Institute for Precision Mechanics and Calculating Technology in Moscow. The machine played some demonstration moves for a visiting group of American engineers. See _New York Times_, December 11, 1955.

*** For one of the checker players, see Strachey, C.S., "Logical or nonmathematical programmes", _Proc. Assoc. Computing Machinery_, September 1952, pp. 46-9. The other checker player has been programmed by A. L. Samuel at IBM, although we know of no published report of its program.

these efforts, as far as we can now evaluate them, has been to show that simple processes do not nearly suffice to play good chess; but that, on the other hand, they do not produce completely absurd results. The amount of information available is still very meager.

Complex Processing. We may now ask, with respect to our example, what complex processing might look like. Let us consider just a few of the features of a human player's processes. He does not look at all alternatives. Those that he does look at, he examines in varying degrees. He makes evaluations to determine where good moves might be discovered and where danger should be expected. Sometimes he draws inferences from the position, sometimes from the history of his opponent's moves; and sometimes he makes no inferences at all, but just proceeds with an attack already under way. He perceives the game in phases -- opening, middle, and end -- and changes his method of analysis accordingly. He works with global terms like "attack", "defense", and "control", and interprets these in each particular position even though he has never seen exactly that position before.

This is a good enough sample. In fact, we probably cannot name all the different processes that go on. Our inability to do so seems to be a major characteristic of complex processes -- they consist of a great many subprocesses, each quite different from the others. It also seems characteristic that the complexity lies not in the component processes, but in the ways these are organized, so that the choice among them at each moment is highly conditional and flexible.

Even though our language must still remain vague, we can at least be a little more systematic about what constitutes a complex information process.

1. A complex process consists of very large numbers of subprocesses, which are extremely diverse in their nature and operation. No one of them is central or, usually, even necessary.
2. The elementary component processes need not be complex; they may be simple and easily understood. The complexity arises wholly from the pattern in which these processes operate.
3. The component processes are applied in a highly conditional fashion. In fact, large numbers of the processes have the function of determining the conditions under which other processes will operate.

Current Chess Studies. No one really knows how complex a process human chess playing is, nor does anyone know what the effect is of various patterns of elementary rules and discriminations in a chess-playing machine. A number of people are currently programming computers for chess in order to explore these questions. Among the current explorers are John McCarthy at Dartmouth, Hal Judd at IBM and ourselves.*

---

* For some of our preliminary thinking, see A. Newell, "The chess machine: an example of dealing with a complex task by adaptation", Western Joint Computer Conference, March 1955. Published by I.R.E. pp. 101-8.

To my knowledge, no one is actually getting results yet, but these efforts should culminate in an extensive exploration of chess playing as a complex information process. It is worth emphasizing that the results of such exploration will be brand new empirical information about complex processes of a kind we have never possessed before.

## III. TYPES OF COMPLEX PROCESSES

Logic Theory Machine. Chess has provided an example for sketching the general nature of complex information processes. The characterization was mainly negative, but actually much can be said about the various component processes and how they are organized. In this part of the paper, we will consider a specific example, the Logic Theory Machine, which will be called LT for short. In describing it, we will exhibit and illustrate a number of processes and modes of organization that typify complex processes.

There are two preliminary remarks. First, the component processes are thoroughly familiar ones. There are, as far as we know, no hidden, undiscovered processes that contain the "key" to complex processing.

Secondly, we will be describing a particular machine.*
This machine exists as a code in an interpretive language that will be described later. This language is not yet coded for a digital computer, but the machine was especially devised to

*We understand that some similar work is being done by Trenchard More at M.I.T., but we are not familiar with any details of his machine.

be simple enough for hand simulation. This has allowed us to explore a little and for instance, to verify that the machine will prove theorems. However, this simplicity means that certain of the processes only show up in embryonic form.

Symbolic Logic. Before we describe the machine, we must describe its task. We will do this very sketchily, since it is only necessary to get the flavor of what the machine must accomplish. LT proves theorems in symbolic logic -- more specifically, in the elementary propositional calculus without operators.

Symbolic logic is a formal system of mathematics, just like Euclidean geometry or algebra. There is a certain set of elements, p, q, r, ..., with which the theory deals. Normally, these elements are interpreted as propositions. For instance, p might stand for, "the moon is made of green cheese." These elements can be combined into expressions by means of connectives, which are similar to the plus and minus of ordinary algebra. Only three connectives concern us:

1. $\sim p$, which means "not p", or "It is not the case that the moon is made of green cheese."
2. $p \vee q$, which means "p or q or both", or "Either the moon is made of green cheese, or the house is painted blue, or both."
3. $p \supset q$, which means "p implies q", or "If the moon is made of green cheese, then the house is painted blue, but if the moon is not made of green cheese, the house may or may not be

painted blue." This is not a primitive connective since it is defined to be "$\sim p \vee q$".

Thus, we may form expressions like:

A: $p \supset (q \vee p)$.

This says, "If p is true, then the sentence (q or p) is also true, but if p is false, nothing can be asserted about (q or p)."

Theorems in symbolic logic are expressions that are universally true. Thus, expression A is a theorem because it is true no matter what content we give p and q. Now in the manner of formal mathematics, five axioms are given which are assumed to be universally true. Then two rules of inference are given whereby new theorems can be deduced from others known to be true. These two rules of inference are (1) substitution and (2) detachment.

In substitution any expression can be substituted for a variable in an expression, provided that the substitution is made throughout the latter expression. Thus in A we could substitute $r \vee s$ for p, getting a new expression, B:

A: $p \supset (q \vee p)$

B: $(r \vee s) \supset [q \vee (r \vee s)]$.

By the rule of substitution, B could be written as a true theorem.

Detachment is the rule for making logical inferences. If we know that C is true, and we know also that $C \supset D$ is true, then we can conclude the D is true. Thus, if we knew that p was true, we could write:

$$C: \quad p$$

$$A = C \supset D: \quad p \supset (q \vee p)$$

$$\therefore D: \quad q \vee p.$$

We will conclude this brief statement on symbolic logic by listing the axioms, although they will not concern us further. Equation 1.1 is the definition of "implies" in terms of "not" and "or", and it is a legitimate operation to replace a connective by its definition selectively in any expression.

1.1 $p \supset q = \sim p \vee q$

1.2 $(p \vee p) \supset p$

1.3 $p \supset (q \vee p)$

1.4 $(p \vee q) \supset (q \vee p)$

1.5 $[p \vee (q \vee r)] \supset [q \vee (p \vee r)]$

1.6 $(p \supset q) \supset [(r \vee p) \supset (r \vee q)]$.

The task of LT is to accept a new logic expression as a conjectured theorem, to try to find a proof, and, if it is successful, to print the proof out. A <u>proof</u> is a sequence of expressions that starts with any axioms or known theorems, and by successive applications of the rules of inference and definitions, terminates in the expression to be proved. Hence our machine is not simply to find true theorems, nor is it to do any kind of routine calculation. It is to discover proofs to given theorems introduced from the outside.

Those of you who find symbolic logic a little strange may substitute for the description above an equivalent one in terms of Euclidean geometry. If this were a Geometry Theory Machine instead of a Logic Theory Machine, its task would be to prove the Pythagorean theorem, or to prove that two circles intersect in at most two points.

LT: Methods. We can now begin to describe LT. First of all, there is no one way to prove theorems. As should be expected of a complex process, LT has a number of methods for proving theorems or for taking significant steps towards a proof. Although LT has six methods, we shall only write down three - Detachment, Substitution, and Chaining - since two of the others are variations of Chaining, and the sixth will be taken up later. The figure, the start of a flow diagram, shows these methods:

DT          SB          CH.

The similarity between these names and the rules of inference is not accidental. The methods are as broad -- and in this sense, contentless -- as are the rules. They do serve, as we shall see, to organize subprocesses.

Substitution is the fundamental method, since it produces actual proofs, whereas the others only produce new problems. Substitution states: if you wish to prove A, find a known theorem B which is similar to A and make it identical with A by suitable substitutions for variables and replacements of connectives. Thus this method is built around the rule of substitution as a valid rule of inference.

Detachment is built around the rule of detachment. It states: if you wish to prove A, find a known theorem of form

$B \supset A$, and then the problem can be reduced to proving *B*. This is simply an inversion of the rule of detachment.

*Chaining* is built around the transitivity of the connective "implies". It states: if you wish to prove a theorem of the form $A \supset C$, find a known theorem of the form $A \supset B$, and then the problem can be reduced to proving $B \supset C$. The concept of chaining -- if $A \supset B$ and $B \supset C$ then $A \supset C$ -- is not a primitive rule of inference in the particular system of symbolic logic with which we are working (Whitehead and Russell's *Principia Mathematica*), although it is easily shown to be valid. In the present context it becomes a method for finding proofs, which, once found, can be written down using only the two legitimate rules.

A characteristic feature of these methods is that they offer no guarantee that they will work in any particular instance. Substitution may not get a proof and Detachment and Chaining may not produce new subproblems. If we also observe, once again, that some of these methods do not provide proofs directly, and hence must be used jointly, we are led to consider how methods get organized into a single operating machine.

*LT: Master Routine*. There is another component of LT called *M*, the *Master Routine*, whose function it is to decide which methods shall operate on which problems and when. We add this to the flow diagram:

$$M \left\{ \quad DT \qquad SB \qquad CH \right.$$

The fundamental problem for the Master Routine is to allocate computing effort to methods and problems. Limitation of effort is an essential concept in complex systems because if there are no such limitations, there will always exist a simple process that will perform the task. Suppose there were no effort limitation for LT. Then it could always find proofs by a simple process very similar to the ones discussed earlier for chess. It would start with the axioms and apply the rules of inference in all possible ways. It would then take the resulting set of theorems (which, incidentally, would already be rather large) and apply the rules again in all possible ways. It would repeat this process until the desired theorem was produced. (This technique guarantees a proof if it exists. There are some problems related to an infinite variety of substitutions, but these may be easily circumvented.) The only reason this process for producing proofs seems fantastic is that, in fact, there always are limitations of effort.

To do its job the Master Routine has certain techniques available. For instance, to decide whether to continue a given method or problem it has <u>stop rules</u>. The important feature about these stop rules is that they must be, in a sense, "irrelevant" to the problems they are applied to. That is, they must not involve, explicitly or implicitly, finding the solution to the problem. They will be similar to certain common aspects of human behavior: levels of aspiration and perseverance. The stop rules will consist of <u>norms</u>, i.e., how long proofs are expected to take, and <u>cues</u>, i.e., the complexity of the expressions that are encountered in proving a theorem.

Another technique available to the Master Routine is to build up a *hierarchy of problems* and subproblems. Since some of the methods produce new problems, the Master Routine can direct effort to solving these. These efforts may lead to still further subproblems, and so on. An essential feature of the hierarchizing is that the entire problem-solving resources of the machine must be available for use on any problem, no matter how derivative it is from the original. If this cannot be done, the amount of space required to hold the large number of separate processes required to deal with problems at each level would prove prohibitive.

In LT, the Master Routine is rather rudimentary, due almost entirely to the limitations imposed in order to make hand simulation feasible. Thus LT utilizes the methods in a fixed order, and uses only substitution to try to solve the subproblems that are created by the other methods. We symbolize this in the flow diagram by drawing arrows from SB to DT and CH.

$$\begin{tikzcd} M \quad \{ & DT & SB \arrow[l] \arrow[r] & CH \end{tikzcd}$$

*LT: Search and Description*. This gives us the grand design for LT - the methods and their organization by the Master Routine. We return to the individual methods and the processes they organize. We will concentrate on Substitution as an illustration, since all methods make use of the same subprocesses.

Substitution requires (1) *finding* a theorem similar to the one to be proved, (2) *comparing* it with the desired theorem, and (3) trying to *match* the two by suitable substitutions and replacements. Consider, first, the task of finding similar theorems. In LT this requires three things: a memory to hold known theorems, a process for searching this memory, and a process for describing logic expressions (which defines similarity). In operation, a description is made of the expression to be proved; then the theorem memory is searched for all the theorems that have identically the same description. These are selected out, and made available to the matching processes.

The description and search process acts like a filter that passes theorems which have a higher chance of working in the following stages of the total method, than would theorems selected capriciously. The major question about such filters is economic -- whether they save more effort than they use. On the debit side, a certain amount of computing power is required to make each description and to make each comparison of descriptions, whether the theorems are selected for further processing or not. On the credit side, the filter selects out a small fraction of all the theorems for further expensive processing, and those selected have a much higher probability of success than the others. Whether one such filter is worthwhile, or perhaps a whole cascade of them, depends on the relative inexpensiveness of the search and description and the enrichment gained.

The descriptions used in LT consist of simple measurements on the isolated logic expressions that indicate only very gross features. To some extent these are patterned after the features we think humans respond to when they glance at an expression in symbolic logic. LT counts *variables*; it counts the number of *variable places*; and it counts the maximum number of nested parentheses, or *levels*, in an expression. This latter number is interpreted by us to be a rough measure of complexity of the expression, but this is pure heuristic. In expression *A*, for example, which looks like,

$$A: \quad p \supset (q \vee p),$$

there are two variables, three places where these variables appear, and three levels (if a variable is counted as a level). These measures taken on expressions and subexpressions constitute the descriptions.

Both Detachment and Chaining use the same processes, though in different ways. We symbolize this in the flow diagram by using FT for "Find theorem", and *D* for "description".

$$\begin{tikzcd}
M & \{ & DT & SB \arrow[l] \arrow[r] & CH & & \\
 & & FT \arrow[u] & FT \arrow[u] & FT \arrow[u] & \} & \boxed{\begin{array}{l}\text{THEOREM}\\\text{MEMORY}\end{array}} \arrow[l] \\
 & & & D \arrow[ul] \arrow[u] \arrow[ur] & & &
\end{tikzcd}$$

LT: Diagnosis and Matching. The remaining steps of Substitution consist in taking one of the similar theorems and trying to make it identical with the desired theorem. The first step of this process is to compare the two expressions and note differences. If a difference is found, this generates a subproblem, for unless a series of legitimate logical operations can be performed on the theorem to eliminate the difference, Substitution will fail. Since LT is called upon to compare logic expressions often, there are certain differences that arise over and over again, and for which relatively simple solutions exist. The appropriate technique for handling these is to be able to recognize them directly, and simply to apply the corrective operations. For the other differences that arise there is no simple method of eradication, and the Substitution Method fails when such differences are encountered in the matching process.

This method of direct diagnosis and application of a known solution procedure is the sixth proof method mentioned earlier. One can expect it to be used in complex processes whenever it is applicable frequently enough to justify holding it continuously ready, and stopping to make the diagnosis over and over again.

Substitution for variables and replacement of connectives form the basis of most of the known solutions used in LT. For example, if a variable $q$ in a theorem has $\sim p$ as a correspondent in the desired theorem, then a substitution of $\sim p$ for $q$ provides the solution to this small matching problem.

Detachment and Chaining also use the same matching and diagnosing operations, although again in different ways. Using MC for "matching process", S for the logical operation of "substitution", and R for "replacement", we get the final flow diagram for LT.

$$\begin{tikzcd}
 & & & S \arrow[dr] & & R \arrow[dl] & & & \\
 & & & & MC \arrow[dll] \arrow[d] \arrow[drr] & & & & \\
M & \{ & DT & & SB \arrow[ll] \arrow[rr] & & CH & & \\
 & & FT \arrow[u] & & FT \arrow[u] & & FT \arrow[u] & \} & \boxed{\text{THEOREM MEMORY}} \arrow[l] \\
 & & & & D \arrow[ull] \arrow[u] \arrow[urr] & & & &
\end{tikzcd}$$

<u>LT: Conclusion</u>. This completes the description of LT. We have omitted a few other processes that are required in a minor way. For instance, the machine does do a certain kind of learning in the retention of proved theorems for future use. Perhaps we should not treat the existence of these other processes so casually, since one of the main points about complex processes is that they are made up of many little pieces.

The Logic Theory Machine just described is essentially the first version we have completely specified. The differences are for purposes of exposition. We know, through hand simulation, that the one we have specified exactly will prove theorems. It will prove the 60-odd theorems of Chapter 2 of <u>Principia Mathematica</u>. For example, it will prove:

$$p \vee q \supset [(p \supset q) \supset q].$$

With minor modifications it could probably handle all of elementary symbolic logic. However, it is unimportant exactly what LT will do. The relevant point is that LT is a complex process made up of very simple and familiar components, and complete enough so that it does something.

## IV. INSTRUCTING THE COMPUTER

Languages. We have given an extensive view of what a complex system is, and have specified a wide array of processes that are involved. At the beginning of the paper we focused on the problem of instructing a computer and remarked on the amount of detail and planning required. We now need to build the bridge from the wide-open systems we have been talking about to the computer.

In putting LT on a computer, we work through an intermediate language. This language is similar to a computer code in that it consists of sequences of instructions that the computer can execute. However, to execute these instructions, the computer uses a special program that translates the intermediate language instructions into machine code. Once we have correctly written LT in the intermediate language, we can be assured that the computer can perform the program.

Such languages are becoming more and more common in automatic computing. They usually are called pseudo-codes or interpretive languages. They have been developed in response to the amount of detailed work involved in coding. They differ from the language used for LT in their preoccupation with standard mathematics. Whereas LT requires primarily logical and procedural operations, the core of most current pseudo-codes is the ability to write equations in standard mathematical notation; i.e., $a + bx^2 - \sin x$.*

All pseudo-codes, including the one for LT, are attempts to free the user from the additional planning and envisioning of information flows that is required if a problem is put in machine code. This freedom is usually considered a mere convenience, important because it reduces the opportunities for error and utilizes programming manpower efficiently.

A price is paid for pseudo-codes, since it takes both time and memory space to translate them into machine code. This is an offsetting factor to the efficiency and convenience criteria mentioned above, and provides a natural limit to how involved and far-removed from machine code the pseudo-codes can get and remain practical.

The reason for emphasizing these considerations is that the point of view taken here is a little different from the usual one. There are limits on the information processing capacity of humans and,with any given language, there will be limits to the complexity of the systems a human can meaningfully consider. Thus the role of the intermediate language for us is to allow the specification of complex processes that would be impossible to describe in machine code -- not logically impossible, of course, just impossible to think through. Even though the language is necessary, we should not forget that we

*See, for example, Proceedings of a Symposium on Automatic Programming for Digital Computers, May 1954. Office of Technical Services, U. S. Department of Commerce are paying a price in computing power to get it.

To fix ideas, let us characterize briefly some of the features we would like in a language. Most of those that appear here correspond to restrictions that have arisen in machine codes.

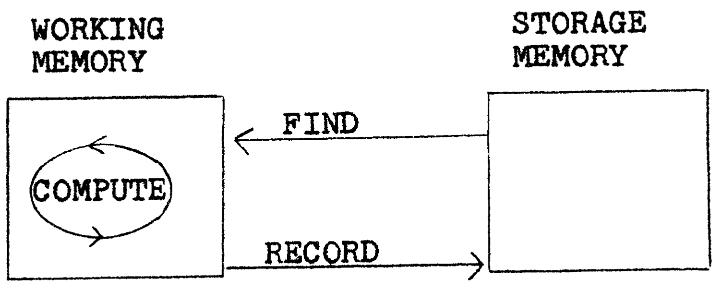
1. We should like to be free from specifying where information is to be stored; and to be able to get information in terms of our need for it without having to remember where it is.
2. We should like plenty of working space, so that we needn't be concerned about writing over good information, or have to do too much information shuffling.
3. We should like to be able to define new concepts whenever we find them useful, so as to express our ideas in compact form.
4. We should like to have operations which are at the natural level for thinking about complex processes. This holds for:
    a. Computational operations such as addition and subtraction;
    b. Control operations which make decisions as to what processes shall be carried out; and
    c. Procedural operations which set up computations, see that the right information is in the right place, and keep track of the status of computation as it progresses (also called housekeeping or red-tape operations).

LL: Basic Concepts. The language uses instructions that are very similar in format to machine instructions. There are routines which consist of sequences of instructions, just as in normal machine coding. Likewise, the usual concept of conditional transfers to instructions is used to make decisions. Each instruction has an operation part, and two reference places, which are similar to the address parts of machine instructions:

OPERATION LEFT REF RIGHT REF

The language allows new definitions. A new instruction may be created to stand for a whole routine. These defined instructions may again be used in the definitions of still other instructions. Thus, an extensive vocabulary can be built up by a hierarchy of definitions.

LL: Memories. In order to effect freedom of memory reference, the memory is divided into two parts, Working Memory and Storage Memory. This is pictured in the figure below.

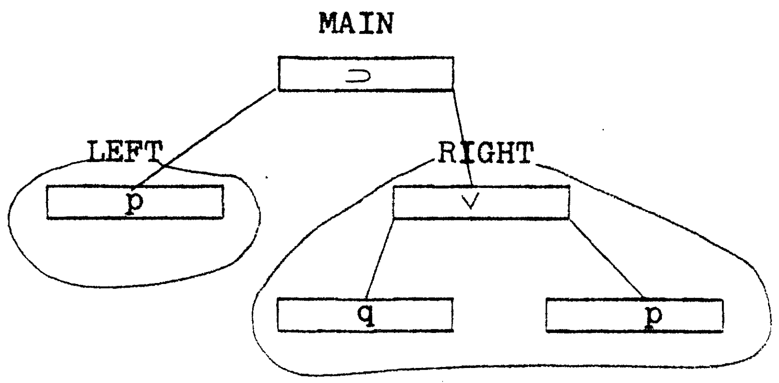


There will be Find instructions to bring information about logic expressions from the Storage Memory to the Working Memory, and Record instructions to put information back. All of the computation, however, will be done only on information stored in the Working Memory.

The Left and Right references may be used to refer to working memories, but they may not refer to storage memories. There is no way to refer directly to any memory cell or information in Storage Memory. The underlying notion of how we find information in Storage Memory is that the desired information is always related to something else that is already in Working Memory. What we want can always be found by means of its relation to information that is already known. Consider an example. Logic expressions are stored away as a set of elements of information, one to each symbol. Suppose, for example, the expression is

A: $p \supset (q \vee p)$.

This would be stored away in a section of Storage as follows (the rectangles represent the elements of information):



Suppose we had the Main element in working memory x, and wanted to find the element that represented its right-hand subexpression; that is, $\boxed{\quad\vee\quad}$ . We would use an instruction like:

Find the right subelement of the element in working memory x and put it in working memory y .

This would be written formally as:

```
OPERATION  LEFT  RIGHT
     FR      x     y.
```

Thus we can find this new element because of its relationship to the element already in a working memory (i.e. the element in x ).

The total specification for a machine like LT consists of a large number of separate routines, corresponding to the various subprocesses. The working memories referred to within each single routine are automatically kept entirely separate from all others. Thus there is no need to worry about destroying valuable information relevant to other processes. There is an unlimited number of working memories available within any routine; hence, there is no necessity for involved information-shuffling to get things into place where they can be worked on. On the other hand, once a routine has been completely computed, there is no way of referring to the information in its working memories from other routines. All the results of the computation that are wanted permanently must be put back in Storage Memory.

LL: Instructions. In order to discuss the level of detail at which instructions are defined, we will consider an example of each of the three types of instructions mentioned earlier. The example of a computational instruction is one concerned with the description of logic expressions. We wish to determine the number of variable places in an expression. Suppose we had

$$A: \ p \supset (q \vee p).$$

Then P, the number of places where variables appear, is three. There is a way for numbers like P to be associated with a working memory. Thus, the computational operation needed is:

Add 1 to the P associated with working memory x .

Formally we may write:

| OPERATION | LEFT | RIGHT |
|---|---|---|
| NAP | 1 | x. |

Here N stands for computational instruction (Numerical), A for add, and both the amount to be added (1) and the working memory (x) are given in the reference places. The computational instructions are defined at about the same level of detail as in machine code, which is about the level at which the user must think about his problem.

An example of a control instruction is one that discriminates whether an element of a logic expression is a variable or not. As was mentioned earlier, all conditionals in this language are made by a transfer of control to some other instruction in the routine. However, instead of referring to an instruction by memory address, as is done in machine code, arbitrary names are given to those instruction locations to which it is necessary to refer (normally referred to as symbolic coding). Thus we have:

If the element in working memory x is not a variable, transfer control to instruction J.

Stated formally:

```
        OPERATION   LEFT   RIGHT
              TV      x      J.
```

Here T is for test and V is for variable. Such a decision is again at a level of detail that is meaningful to the user. This is in contradistinction to machine codes that admit as conditions for transfer only whether a particular number is positive, negative, or zero.

As an example of procedural instructions, we will give a pair that work together to run through a repetitive operation a number of times and then stop the process. This is a very common occurrence in all computing and is usually called looping, cycling, or iterating. In LT, iteration is required when the same processing must be done on all the elements of a logic expression. One instruction, FEF, is used to start the process off, and the other, FEN, is used to recycle it and stop it when through. The instructions assume the elements are in some linear order in the Storage Memory, which is always the case. For the first instruction we have:

Find the first element of the logic expression indicated in working memory x and put it in working memory y.

Formally:

```
        OPERATION   LEFT   RIGHT
              FEF     x      y.
```

For the other instruction, we have:

> Find the element that is next to the element in working memory x and put it in x (obliterating the old element). Transfer control back to instruction J, unless there are no more elements, in which case continue control in sequence.

Formally:

```
OPERATION   LEFT   RIGHT
      FEN     x      J.
```

Here the front F stands for "Find", the E for "Element", the rear N for "Next" and rear F for "First". Again, the procedure is formulated at the level at which the user naturally thinks about it, whereas with machine code all the details have to be programmed.

LL: A Sample Routine. An example of the use of the language might make the previous instructions a little clearer and tie things together. Consider the computation of P, the number of variable places. This computation occurs often enough to make an instruction desirable. We define the new instruction as:

> Count the number of variable places in the expression indicated in working memory x and put the number in the associated P.

Formally, we define:

```
OPERATION   LEFT   RIGHT
      NP      x.
```

To find P we need to examine every element in the logic expression, decide if it is a variable and, if so, count one for it. We wish to do the counting in P, since that is where the final answer must be. The program follows:[*]

| LOCN | OPERATION | LEFT | RIGHT |
|---|---|---|---|
| | NP | x | |
| | FEF | x | y |
| J | TV | y | K |
| | NAP | 1 | x |
| K | FEN | y | J. |

The FEF and FEN set up the search through the logic expression indicated in working memory x. TV asks each element in turn whether it is a variable. If it is, NAP counts 1 in the P of working memory x. Then FEN gets another element and sends control back to TV which starts the cycle over again. If an element is not a variable, TV just skips control around NAP to FEN, thus avoiding the count. Finally, when all elements have been counted, FEN sends control on, and NP has been accomplished.

LL: Conclusion. This completes the description of the language used for LT. LT has been completely coded in its terms. However, nothing has been said about how to code such languages for digital computers. As was stated earlier, the language is fundamentally similar to a number of pseudo-codes which have been coded for computers and are in use. We have not coded the exact language described here, although we are

[*]This is a simplified program for expositional purposes. Several important details have been glossed over, such as how the defining routine "knows" what x is, since NP may be used in a number of different routines.

currently working on one of this general variety. Thus, for instance, it is not possible yet to make any statements about the price in computing power that must be paid to use languages like this for specifying complex systems.

Our purpose in exhibiting the language in so much detail has been to show that the bridge exists between the complex processes talked of earlier in the paper and the computers. The essential contribution of this language is in freeing of the user from the detailed planning of the information flows involved in executing the program.

## V. LEARNING

It remains to consider learning as it relates to the complex processes we have been discussing. We can do this rather briefly, since our main point is that learning is not at all different from what we have talked of so far. Put differently, we already have all the component processes; it only remains to combine these to obtain "learning". Further, systems that learn will not look very different from systems like LT.

Learning consists only in this: that some processes within the total system have as their output the structure or operating conditions of other processes. For then the same system presented with the same external environment will behave differently a second time than it did the first. This is about all that learning comes to, except that one may wish to limit the term to "improvements" -- an ambiguous restriction at best.

Sufficiently complex processes can hardly avoid learning and adapting.

Let us make this point a little more concrete by describing some current work aimed at putting learning programs on computers. Here again there have been some explorations of simple processes.* In these, the performance system (the processes that accomplish whatever the system is supposed to do) has a number of numerical parameters. Feedbacks are programmed that modify these parameters as a function of the discrepancy between performance and a criterion. Thus the system gradually learns its way to perfect performance.

In the last two years, however, an attempt has been made by Oliver Selfridge and Gerald Dinneen of the Lincoln Laboratories of MIT to explore learning as a much more complex process. Since a fairly good description of their work exists in the literature,** we will not go into much detail. However, a brief sketch will show both the nature of their investigations and the similarity between their machine -- which is a learning machine -- and LT -- which is primarily a pure performance machine.

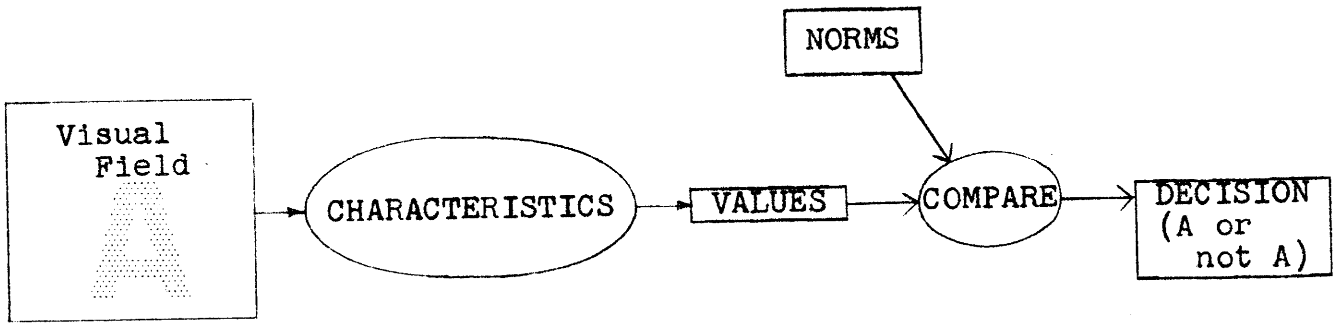
Because the task of their machine is the recognition of visual patterns, we will call it VPR. VPR is presented, from

---

*A. G. Oettinger, "Simple learning by a digital computer", Proceedings of the Association for Computing Machinery, Toronto, Ont. September 1952

**O. G. Selfridge, "Pattern recognition and modern computers", and G. P. Dinneen, "Programming pattern recognition" both in Proceedings of the 1955 Western Joint Computer Conference, Los Angeles, Calif. I.R.E., March 1955 outside, with a two-dimensional field -- a 90 x 90 square of black or white dots -- on which patterns may be created with the black dots. Selfridge and Dinneen have worked with A's, O's, triangles, and squares, etc. VPR has a performance system made up of a number of processes that can recognize a pattern -- that is, can answer whether a given field is A or not.

At a sufficiently gross level, the performance system is quite simple, as shown in the figure:



A number of characteristics of the visual field are computed. A "characteristic" is a computer routine that we will explain in more detail later. These computations yield numbers -- the values that characterize this particular visual field. These are compared with norms that are associated with the specific pattern under question, say an A. If there is sufficient agreement between the values characterizing this field and the norms characterizing the pattern an affirmative decision is reached by VPR.

There are four things that turn VPR into a learning machine and into a complex process in the sense in which we have been using that term. First, there is a language for characteristics. Although it is very rudimentary, this is a true language. It has basic terms -- three symbols, I, II, and III -- which stand for <u>elementary operations</u>. These transform a 90 x 90 visual field into a modified 90 x 90 visual field. The basic terms may be combined into expressions that symbolize characteristics. The expression for a characteristic is a sequence of transformations, say: I II II III I III III. Translated into a computation, this means: take the original visual field, transform it with I, then II, then II again, etc.; when this has been done, count the number of black spots left and that number is the value of the characteristic for that visual field. (The counting operation is not as irrelevant as it sounds since the transformations are such as will generally reduce the total number of black spots in any field. For more details of the nature of the operation the reader is referred to the papers cited.)

The second thing VPR can do follows closely upon the first. It can create new characteristics for itself. Its learning problem may now be stated. It has available a space of characteristics formed from all possible sequences of three symbols (say $3^{10} \cong 6x10^4$ possible sequences of length 10). Its performance system consists of some very small number of these characteristics (say 20) which should discriminate A's from non-A's. Its problem is to search the space in order to discover a particular set of twenty that provides good discrimination. Notice that the access to this space is through generation; no very large set of possibilities could ever be held extensively in memory.

The third thing VPR has is a learning loop for eliminating a poor characteristic once it has been generated and VPR has decided to give it a try.* This learning loop goes outside the machine. VPR can learn about patterns only extensively, by being presented with a variety of examples (particular visual fields which it is told are A's) and inducing the pattern. VPR achieves this separation of good from poor characteristics by accumulating the experience from a number of visual fields on each character it tries. This includes information from outside on whether each pattern was an _A_ or not. If its summary of experience reveals a characteristic of good discriminating power -- giving always a certain value when an _A_ occurs, always a different value when other patterns occur -- then the characteristic is retained. Otherwise, the characteristic is eventually discarded in favor of another. The value of a characteristic that typically occurs when an _A_ is presented is made the norm for that characteristic.

The fourth and final thing that VPR has is a second learning loop for the generation of new characteristics. Since VPR, like all complex processes, operates under a severe effort

---

* This is the classic learning loop of stimulus-response learning theory in psychology -- the selection of a response from a set of responses on the basis of externally applied reinforcement.

constraint, the whole function of the first loop is to remove worthless processes that absorb good computing effort. This makes way for new characteristics to be tried. But first, such characteristics must be found; that is, selected from the large space of all possible characteristics. The process in VPR that generates new characteristics does so as a function of the accumulated experience with the characteristics already generated. The philosophy of the process -- if we may use the term -- is to choose new characteristics similar to those old ones that were successful, yet to allow an occasional "wild" choice. The program achieves this by building up new sequences of elementary operations based on the same conditional probabilities as occur empirically in the set of good characteristics -- already found and tested. Again, the reader is referred to the original papers for details.

This completes the description of VPR. The program has been coded for the Memory Test Computer at MIT and some exploration done. It is not claimed -- either by Selfridge and Dinneen or by us -- that this machine is a good pattern recognizer or that its learning is at all adequate either qualitatively or in terms of rate of convergence to an adequate set of characteristics. VPR has shown a little learning, although it has a tendency to fixate in one kind of characteristic and improve it to the exclusion of others.

We think that just how good VPR is is unimportant. As with LT, we have here a complex process made up of many quite different subprocesses, all of which are quite elementary.

These are organized together in a highly conditional and interactive way. VPR does do something. More processes could be added at anyone's whim to sophisticate the process: a learning loop in the overall criterion, more elaborate discriminations of the experiences with characteristics, etc. The value of both LT and VPR -- meaning precisely the two machines so far programmed -- is to show that we are in the right general area for fruitful exploration.

## VI. CONCLUSION AND SUMMARY

This paper has attempted to convey a picture of an area of current scientific activity that we have called complex information processing. We have not aimed at a survey of all work that is pertinent.* Rather, our objective has been to characterize the field, and to provide a specific example of a complex information processing system.

In the first section we showed that the attempt to perform difficult tasks (e.g. playing chess) with a simple information processing system leads to exorbitant demands for computing power. We asserted that such tasks can be performed with much smaller requirements of computer speed, time, and memory by use of a complex process -- a process made up of a very large

*The most serious omissions are the work on automata, which includes the work on abstract nerve nets, and the work on mechanical translation of languages. See C.E. Shannon and J. McCarthy (eds.), Automata Studies, Annals of Mathematics Studies, #34, Princeton, 1956; and W.N.Locke and A.D.Booth (eds.), Machine Translation of Languages, Wiley, 1955. Attention should also be called to the work of W.R.Ashby. See Design for a Brain, Wiley, 1952.

number of subprocesses, the elementary component processes being simple but combined in complex patterns, and the whole process constituting a strategy (each step being highly conditional on what went before) rather than a simple linear sequence.

Next, a machine, LT, was described that employs a complex process to discover proofs for theorems in elementary symbolic logic. The machine consists of a hierarchy of routines: a master routine, some proof methods, and subroutines that are employed to carry out the proofs. These routines constitute a heuristic that permits the machine to discover appropriate steps in a proof-chain with far less random search than if it proceeded to construct and examine all chains permitted by the rules of logic.

To put the logic theorist, LT, on a computer, it is highly advantageous -- and perhaps necessary -- to work through an intermediate language (LL), of the kind now generally referred to as an interpretive language. The program for LT is written in the language LL, and then LL is coded for a computer. Important characteristics for a language to be used in specifying complex processes include: (1) relative freedom from reference to the absolute addresses where information is located; (2) adequate working space so that information does not have to be rearranged repeatedly in the memory; (3) freedom to define new concepts whenever they appear to be useful; (4) operations defined at a level of "grossness" that is natural for thinking about complex processes. These characteristics are illustrated by the structure of LL.

The logic theorist is primarily a performance, rather than learning, machine. In a final section we indicated by reference to a machine, VPR, developed by Selfridge and Dinneen, how learning can be described as a complex information process that is not fundamentally different in kind from the process of LT.