PROGRAMMING THE LOGIC THEORY MACHINE

BY

A. NEWELL AND J. C. SHAW

Reprinted from the PROCEEDINGS OF THE WESTERN JOINT COMPUTER CONFERENCE
Los Angeles, California, February 1957

PRINTED IN THE U.S.A.

Programming the Logic Theory Machine*

A. NEWELL† AND J. C. SHAW†

INTRODUCTION

A COMPANION paper[1] has discussed a system, called the Logic Theory Machine (LT), that discovers proofs for theorems in symbolic logic in much the same way as a human does. It manipulates symbols, it tries different methods, and it modifies some of its processes in the light of experience.

The primary tool currently available for studying such systems is to program them for a digital computer and to examine their behavior empirically under varying conditions. The companion paper is a report of such a study of LT. In this paper we shall discuss the programming problems involved and describe the solutions to these problems that we tried in programming LT.

The aims of this paper are several. First, it serves to amplify and make more precise its companion paper. Second, progress in research on complex information processing demands a heavy investment in technique. It is not sufficient simply to specify a rough flow diagram for each new system and to program it in machine code on a one-shot basis. We hope this paper not only shows the techniques and concepts we found useful, but also emphasizes the role played by flexible and powerful languages in making progress in this area.

Finally, LT is representative of a large class of problems which are just beginning to be considered amenable to machine solution; problems that require what we have called heuristic programs. A description of the problems encountered in LT may give some first hints about the requirements for writing heuristic programs.

NATURE OF THE PROGRAMMING PROBLEM

To avoid too much dependence on the companion paper, we will repeat a few general statements about LT in the context of programming. LT is a program to try to find proofs for theorems in symbolic logic. In this type of problem, a superabundance of information and alternatives is provided, but with no known clean-cut way of proceeding to a solution. These situations require "problem-solving" activity, in the sense that one has no

* This paper is part of a research project being conducted jointly by the authors and H. A. Simon of Carnegie Institute of Technology. All of us have shared in the development of most of the ideas in the language.

† The RAND Corp., Santa Monica, Calif.

[1] Newell, Shaw, and Simon, this issue, p. 218.

path to the solution at the start, except to apply vague rules of thumb, like "consider the relevant features." Playing chess, finding proofs for mathematical theorems, or discovering a pattern in some data are examples of problems of this kind. Occasionally, as in chess, one can specify simple ways to solve the problem "in principle" —given virtually unlimited computational power—but, in fact, limitations of computing speed and memory make such exhaustive procedures inadmissible.

LT, as an example of a heuristic program, may be expected to yield some clues about constructing this type of program. Actually, LT is still very simple compared to the complexity in learning, self-programming, and memory structure that seem necessary for more general problem solving. Thus, we think that LT underestimates the flexibility and programming power required in complex problem-solving situations.

Perhaps the most striking feature of LT when compared with current computer programs is its truly nonnumerical character. Not only does LT work with other symbols besides numbers, but many of its computations either generate new symbolic entities (*i.e.*, logic expressions) that are used in subsequent stages of solution, or change the structure of memory. In contrast, in most current computer programs, the set of entities that are going to be considered (the variables and constants) is determined in advance, and the task of the program is to compute the values of some of these variables in terms of the others. Such forward planning is not possible with LT. Although there are fixed entities in LT—which remain constant over the problem and provide a framework within which the computation takes place—these are complex affairs, rather than symbols. An example of such an entity is a list of subproblems. The elements on this list are variable: each problem is a logic expression which is generated by LT itself and may carry with it various amounts of descriptive information. The number, kind, and order of these logic expressions are completely variable.

The program of LT is also very large. There are large numbers of different features under consideration and large numbers of special cases. All of these features and cases require special routines to deal with them, and, by a kind of compounding rule, the existence of numerous subroutines requires yet other subroutines to integrate them. This is further compounded in LT, because no one way of proceeding ensures solution of a given logic problem, and hence, many alternative subroutines exist. Their existence again implies routines to choose among them. Some reduction in the total size of the program is achieved through multiple use of routines, but this increases the complexity of the subroutine structure considerably. The hierarchies of routines become rather large: 13 or 14 levels are common in LT.

Another characteristic of LT is its use of information about the workings of the program—how much memory is being used for particular purposes, and how much effort is allocated to various subprocesses—to govern the further course of the program. LT uses such information in its "stop rules," by which it passes from one problem to another, and in its choice between recomputing and storing information. It is cheaper in terms of total amount of computation to compute information and then store it; LT does this as long as memory space is available. When memory becomes scarce, LT shifts to recomputing information each time it is needed.

LT also contains routines for recording the results of its operation, so that we can study its behavior. It is built to permit easy and rapid change of program, in order to let us study radical program variations. These additional features do not add anything qualitatively to the features mentioned above, but they do add to the total size and complexity of the program.

Requirements for the Programming Language

We can transform these statements about the general nature of the program of LT into a set of requirements for a programming language. By a programming language we mean a set of symbols and conventions that allows a programmer to specify to the computer what processes he wants carried out.

Flexibility of Memory Assignment:

1) There should be no restriction on the number of different lists of items of information to be stored. This number should not have to be decided in advance; that is, it should be possible to create new lists at will during the course of computation.

2) There should be no restriction on the nature of the items in a list. These might range from a single symbol or number to an arbitrary list. Thus, it should be possible to make lists, lists of lists, lists of lists of lists, etc.

3) It should be possible to add, delete, insert, and rearrange items of information in a list at any time and in any way. Thus, for example, one should be able to add to the front of a list as well as to the end.

4) It should be possible for the same item to appear on any number of lists simultaneously.

Flexibility in the Specification of Processes:

1) It should be possible to give a name to any subroutine, and to use this name in building other subroutines. That is to say, there should be no limitation on the size and complexity of hierarchies of definitions.

2) There should be no restriction on the number of references in the instructions, or on what is referenced. That is, it should be possible to refer in an instruction to data, to lists of data, to processes, or what not.

3) It should be possible to define processes implicitly; *e.g.*, by recursion. More generally, the programmer should be able to specify any process in whatever way occurs naturally to him in the context of the problem. If the programmer has to "translate" the specification into a fixed and rigid format, he is doing a preliminary processing of the specifications that could be avoided.

4) It should be unnecessary to have a single integrated plan or set of conventions for the form of information; that is, for symbols, tags, orderings, in lists, etc.

On the other hand, it should be possible to introduce conventions locally within parts of the problem whenever this will increase processing efficiency.

These requirements are neither precise nor exhaustive. Except in a world where all things are costless, they should not be taken as general programming requirements for all types of problems. They characterize the kinds of flexibility we think are needed for the sorts of complex processes we have been discussing.

Solutions of the Program Language Requirements for LT

The requirements stated above for LT were met by constructing a complete language, or pseudo code, which has the power of expression implied by the requirements, but which the computer can interpret. A first version of the language was developed independently of any particular computer and was used only to specify precisely a logic theory machine.[2] A second version is an actual pseudo code prepared for use on the RAND JOHNNIAC,[3] and it is this version that we will describe here. We have had about fifty hours of machine computation using the language and hence, we can evaluate fairly well how it performs.

The present language has a number of shortcomings. It is very costly both in memory space and in time, for it seemed to us that these costs could be brought down by later improvement, after we had learned how to obtain the flexibility we required. Further, the language does not meet the flexibility requirements completely. We will comment on some of these deficiencies in the final section of this paper.

The language is purely a research tool, developed for use by a few experienced people who know it very well. Thus a number of minor rough spots remain. Further, we used available utility routines, fitting the format and symbols of the language to a symbolic loading program which already exists for JOHNNIAC. This loader accepts a series of subroutines coded in absolute, relative, or symbolic addresses (symbolic within each routine separately) and assigns memory space for them.

Description of the Language

The description of the language, which we shall call IPL, falls naturally into two parts. First, we shall describe the structure of the memory and the kinds of information that can be stored in it. Then we shall describe the language itself and how it refers to information, processes, and so on.

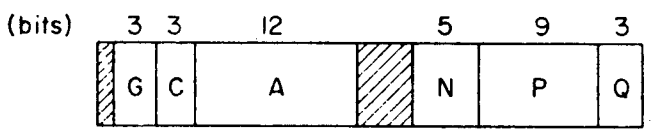
The Memory Structure

LT is a program for doing problems in symbolic logic. Basically, then, IPL must be able to refer to symbolic logic expressions and their properties. It must also be able to refer to descriptions of the expressions which are properties only in an extended sense. For example, an expression may have a name, or it may have been derived in a given fashion, or by using a certain theorem, and IPL must be able to express these facts. LT needs to consider lists of expressions and lists of processes used to solve logic problems, and there must be ways to express these facts.

Elements: The basic unit of information in IPL is an *element*. An element consists of a set of symbols, which are the values of a set of variables or attributes. There are different kinds of elements to handle the different kinds of information referred to above. The two most important elements are the logic element, which allows the specification of a symbolic logic expression, and the description, which is a general purpose element, used to describe most other things, and which carries with it its own identification.

Each element fits into a single JOHNNIAC word of 40 bits. The symbols are assigned to fixed bit positions in the word, so that the element is handled as a unit when it comes to moving information around, etc. Each variable and symbol has a name which is used in IPL to refer to it. The name of a symbol is the address of a word that contains the appropriate set of bits. Since JOHNNIAC has instructions corresponding to the logical "and" and complementation, the name of a variable is the address of a word that holds the mask necessary to extract the bit positions corresponding to the variable.

Logic elements are the units from which logic expressions are constructed. Fig. 1 shows what variables and



G Number of negation signs
C Connective (or variable)
A Location of logic expression
N Name
P Position number
Q Level in expression

Fig. 1—Logic element.

symbols comprise a logic element. Expressions in symbolic logic are much like algebraic expressions: each element consists of an operation (called a "connective" in logic) or a variable, together with the negation signs (if any) that apply to it. We use a parenthesis-free notation, in which the position of each element in a logic expression is designated by a number—this number, therefore, being one of the symbols in the element. For example, the logic expression $p \rightarrow (-q \vee p)$ would be represented by five elements as shown in Fig. 2. Each logic element consists of six variables (each taking on a variety of values) all of which fit into a single word: the number of negation signs, the connective, the loca-

[2] A. Newell and H. A. Simon, "The logic theory machine," IRE Trans., vol. IT-2, pp. 61–79; September, 1956.

[3] The JOHNNIAC is an automatic digital computer of the Princeton type. It has a word length of 40 bits with two instructions in each word. Its fast storage consists of 4096 words of magnetic cores and its secondary storage consists of 9216 words on magnetic drums. Its speed is about 15,000 operations per second.

of the total number of instructions interpreted, whereas the unit cost of interpretation of a higher instruction is only two and a half times as great as for a primitive (about 50 ms to 20 ms). Thus interpretation of all the higher routines accounts for less than 30 per cent of the total cost of interpretation.

Additional Deficiencies of IPL

Experience in writing programs in IPL has revealed a number of additional deficiencies. Perhaps the one that strikes the programmer most is the artificiality of the distinction between the element and the list. By packing a set of symbols into a single JOHNNIAC word we gain in memory space over schemes that use one full word for each variable. The net result, however, is that certain properties, those packed into an element, are treated in one way, and others, those expressed by the lists or by the description elements, are treated in another. Elements are brought into working storage for processing; since lists have various sizes and shapes, they cannot be handled in this fashion. Information that must be kept as a list is handled by indirect reference, through an element in working storage that refers to it. Information that can be fitted into an element is handled directly in working storage. For example, an element and a one-element list must be processed very differently in IPL.

A second deficiency is the restriction to certain forms of referencing. IPL has great flexibility in the specification of operations, that is, an operation can be specified by giving an expression in the language for that operation. We have allowed no such flexibility in the specification of the other references. There are only three ways of giving the information to be used in a routine: by giving the address of the element, the name of the working storage that holds the element, and the name of a reference place that refers to the element. These methods allow certain indirect references, but they still lack flexibility. A rather simple example, but one that is typically annoying, occurs when we want to refer to a name of a routine, that is, to a symbol like L082, which is the address of a directory element. This symbol is used in many places throughout the program, but there is no simple way of getting to it. There is no reason why there should be less power of expression for information references than for operations. It should be possible to give a reference by giving an expression for determining that reference, just as is now done in IPL for operations.

There are other unsolved problems. For instance, we have no satisfactory way of erasing in the association memory. The problem is not how to delete items and make their space available again, which we think is done fairly well in IPL. The problem is how to know what can be erased, since there is no direct way of knowing what else in the system may be referring to the items about to be erased. References are directional, so that if location word A refers to item B, there is no way of knowing this, when only the address of B is known. Uniform two-way referencing seems to be an expensive solution, although it may be the only one. In simpler programs this erasing problem is handled by having the programmer know at all times exactly what refers to what. But if we move to programs in which all lists are set up during operation by the program itself, such solutions are not adequate, and the problem soon becomes acute.

Conclusion

IPL is an experimental language that was built to find ways of achieving extreme flexibility. It was developed in connection with a particular substantive problem—proving theorems in symbolic logic—which requires great flexibility in the memory structure, and powerful ways of expressing information processes.

The language achieved its purpose: we have a running program for LT which has allowed us to explore its behavior empirically with a number of variations. On the other hand, the language is relatively crude, viewed as a general language for specifying programs like LT. It is very costly; it shows the "provincialism" of too close a connection with symbolic logic; and it still has a number of rigidities.

We believe that the basic elements of the language are sound, and can be used as the ingredients of languages having considerably greater powers of expression and speed. We are currently engaged in the construction of a new language patterned on IPL, which we hope will serve us as a general tool for the construction and investigation of complex information processes.

Discussion

L. P. Meissner (Nol, Corona): Do you have a list of those lists which do not list themselves?

Mr. Shaw: Without going further into paradoxes except to say that there is not a direct answer to this question, but the debugging list does list itself.

P. Sayre (Northrop): Would you reiterate or expand your remarks on the next version especially with regard to automatic programming?

Mr. Shaw: No, except to suggest that programming itself is a field of complex information. Processing such is the field we are studying.

J. Matlock (Douglas): Can you give an example of a subroutine using itself?

Mr. Shaw: I think the best example of this is the matching routine which is asked to match one expression to another. The first part of this routine merely looks at the main connectives. If it is successful in matching the given expression to the second one, it then takes a look at the lower left element of each expression and there again it is faced with exactly the same problem as it was faced with initially. Again the matching routine is asked to match this expression. So, at this point the routine recourses and calls upon itself to match the expression it is faced with to the second expression. Eventually, of course, it comes to the termination on these trees and proceeds to back off. So, it says, "I am done" to itself, reiteratively, and then backs up to a certain point at which it proceeds down the right branch.