P-954

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.

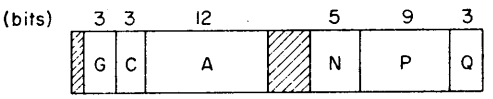
The Memory Structure

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 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 $p \rightarrow (-q \text{ v } p)$ 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-



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.

[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.

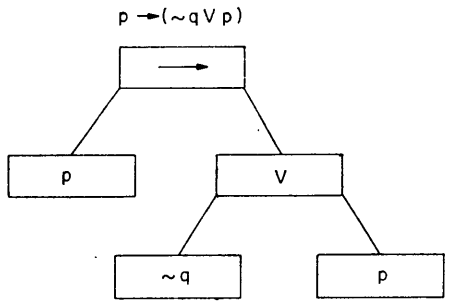


Fig. 2—Logic expression.

tion of the list which holds the entire logic expression of which this is an element, the name of the variable, the position number, and the number of levels down from the main connective.

Description elements consist of two symbols, as shown in Fig. 3. There are many different types of descriptions,

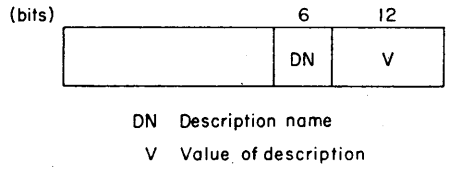


Fig. 3—Description element.

such as the name of a logic expression, the method used to derive a logic expression, or the number of different variables appearing in a given logic expression, and each type has a name. The left-hand symbol in the element gives the name of the type of description. The right-hand symbol gives the value of this description for some logic expression with which this description is associated. Thus, for example, in considering a certain logic expression the description element 012-L082 might be found. The 012 indicates that this description element gives the method used in deriving the expression, and the L082 is the name of the actual method used, in this case, the method of detachment.

Lists: Lists are the general units of information in the memory. A list consists of an ordered set of items of information. Any item on a list may be either a list or an element, and these are fundamentally different types of units, as we shall see later (the difference arises mostly from the fact that an element is contained in a single JOHNNIAC word). Since a list is itself an ordered set of items which may themselves be lists, we obtain most of the flexibility we desire in the memory structure. There is no limit to the complexity of the structures that can be built up, provided that one knows how to use them, except the total memory space available. Also, there is no restriction to the number of lists on which an item can appear. For example, if we have a list of items, we can construct one or more indexes (lists) on each of which an arbitrary subset of the items of the original list appears.

With each item located in a given list we may associate descriptive information without disturbing the general structure of the lists. That is, each item can have a list of description elements associated with it. As many descriptions may be put on the list as desired, and, since they are self-identifying (by means of the description names they contain) they may be put on in any order. Descriptions are associated with the item *on a given list;* hence, if an item is on several lists, it can have several distinct description lists.

Forming Lists: This memory structure has most of the flexibility that we specified earlier as desirable. There are information processes that can create new lists at any time; or that can add items to a list at any time, either in front, in back, or in some relation to other locatable items in the list. Likewise, items can be deleted from lists at any time, or moved from one list to another, or simply "adjoined" to a new list without being deleted from the old one.

All this flexibility in the memory is achieved by the single expedient of divorcing the ordering relations among items of information from the ordering relations built into the address structure of the computer memory. Let us sketch how this is done in JOHNNIAC for IPL.

To form a list, we use a set of *location words*, each containing two addresses. One address locates an item on the list, the other address locates the next location word. Fig. 4 shows how this is done for a list of three elements.

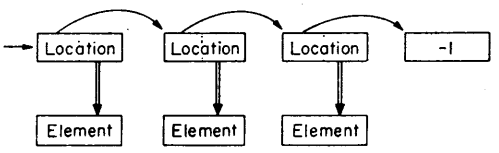


Fig. 4—List of elements. The left half of the location word contains the address of the next location word (single arrow); the right half contains the address of the element (double arrow).

A location word holding a negative number serves to terminate the list. Since the JOHNNIAC word holds two instructions and hence, contains two addresses, it is very convenient for this scheme. The left address is the address of the next location word, the right address is the address of the item on the list. In order to permit the general list structure indicated earlier, each location word contains a code telling whether the item it refers to is an element (001), in which case it contains information, or a list (000), in which case it is the beginning of another list; *i.e.*, of a series of location words. Fig. 5 shows a general list containing both elements and lists.

Each item on a list is uniquely determined by one of the location words. To associate a description list with this item, we insert a location word for the description list right behind the location word of the item with which it is to be associated. We use a code 002 to distinguish the location word of the description list from the location word of the next item. Since a description requires only half a word to hold its two attributes, we put the location of the next description in the list in the other half of the same JOHNNIAC word (Fig. 6).

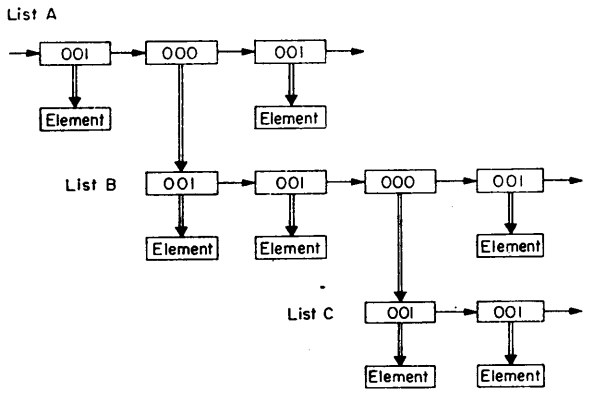


Fig. 5—General list. List B is the second item on list A; list C is the third item on list B.

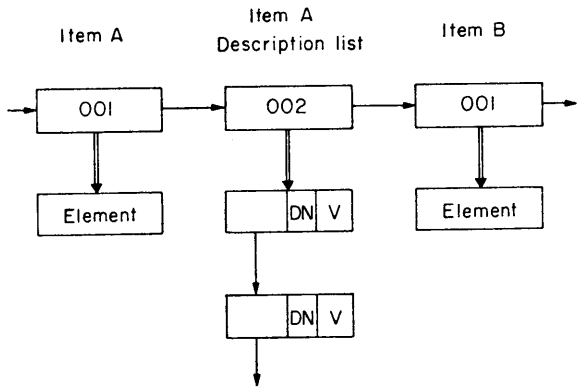


Fig. 6—Description list. The description list for item A is inserted immediately behind item A, and distinguished from the next item, B, by a 002 location word.

The address of the next item or location word in a list need bear no particular relation to the address of a given location word; they need not be adjacent, for instance. Hence, an item is deleted from a list simply by deleting its location word. Suppose, as in Fig. 7, we have three

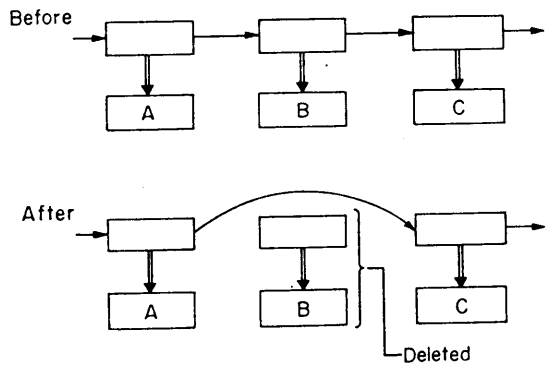


Fig. 7—Deletion of an item from a list. Item B deleted by changing address in location word of A to refer to location word of C.

items, A, B, and C, on a list. To delete B, we simply change the address in the location word of A to refer to the location word of C. Because of this same freedom of position of the words in a list, location words on different lists may hold the address of the same item of information. Hence, a single item of information may be on as many lists as we please.

Perhaps the major problem in creating a flexible memory is the housekeeping necessary to make available unused words after they have become scattered all through the memory because of repeated use and reuse. When a word is deleted from a list, we must be able to "recapture" this word, in order that it may be used subsequently for other purposes. The association memory (the name we use for this type of memory) starts with all "available space" on a single long list, called the *available-space list.* Whenever space is required for building up a new list, this is obtained by using the words from the front of the available-space list, and whenever information is erased and the words that held it become available for use elsewhere, these words are added to the front of the available space list. In Fig. 7, the deletion process would be completed by tying all of the deleted words into a list and attaching this at the front of the available-space list. Thus, the fact that unused space is scattered all through the memory creates no difficulty in finding new space, for there is a single known word (the head of the available-space list) that always contains the address of the next available word. Hence, the use of the memory is not complicated by any natural ordering like the natural sequence of machine addresses.

Since all lists obtain their new space from the same list, the only restriction on amounts or degrees of complexity of lists is the total size of memory. Thus, it is clear why there are no separate limits to the number of lists, their maximum size, how "stacked" up they can be, and so on. In this sense the language is easy to learn and use.

Language Structure

The basic form of the language is the same as in all current programming languages. The terms of the language are *instructions.* Each instruction specifies a complete information process; that is, it can be followed by any other instruction. Thus, the syntax of the language is basically identical with that of machine codes or flow diagrams: sequences of instructions are carried out in succession, with conditional transfers of control to permit alternative subsequences to be carried out as a function of the process. (IPL is slightly more general than this, as will be seen subsequently.) Also, as is usual in this general type of language, each instruction specifies separately: 1) an operation and 2) the information upon which it operates.

In IPL, a *program—e.g.,* LT—is a system of *subroutines.* Each subroutine is a sequence of instructions. Each IPL instruction is defined by a subroutine (more precisely, each particular occurrence of an instruction has its operation part carried out by some subroutine), usually called the *defining subroutine* of the instruction. Subroutines may be written either in JOHNNIAC machine language or in IPL (whenever "routine" is used in this paper it always means IPL routine, unless stated otherwise). Correspondingly, there are two kinds of IPL instructions; *primitives,* whose defining subroutine

is written in machine language, and *higher instructions* whose defining subroutine is written in IPL.

The system of subroutines is organized in a roughly hierarchical fashion. There is a "master routine," each instruction of which is defined by another routine; the instructions in these subroutines, in turn, are defined by yet other subroutines, and so on. Eventually, primitive instructions are reached, and their defining subroutines, which are in machine language, are executed.

Instructions: Fig. 8 shows a typical instruction for-

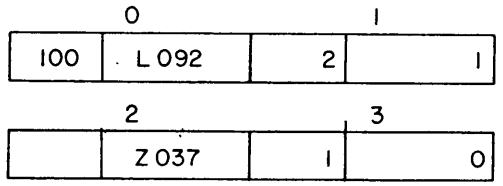


Fig. 8—Typical IPL instruction. The small numbers over the half words give the number of the reference place.

mat. An instruction is a vertical sequence of JOHNNIAC words; a routine is a vertical sequence of instructions. Each half-word in an instruction is a *reference place*, the first being numbered 0, as shown in the figure. There may be any number of reference places in an instruction, and the number need not even be constant from one use of the instruction to another, provided that the subroutine which carries out the operation understands how to use the references. Each reference states (by code): 1) the *type of reference* (the small space on the left side of each reference) and 2) the *reference*. All references are to elements; hence, to refer to a list in an instruction, it is necessary to refer to an element that refers to the list.

There are three types of references (coded 0, 1, 2). Type 0 gives the location of an element in memory by specifying either the absolute or relative address of the word containing the element. Type-0 references are used for the fixed names of things, like constants (Z037) or subroutines (L092). Within each routine, instructions are located by symbolic addresses (such as * 32) which are also of type 0. In this scheme there is no general way to make reference in one routine to an arbitrary instruction in any other routine, although there are some important special ways of referring from one routine to another which are considered below.

Each subroutine has its own working storage, consisting of an indefinite number of elements. These are referred to by type-1 references: 1-0, 1-1, 1-2, · · · . When a subroutine is completed, these working-storage elements are automatically erased and made available for reuse.

As stated above, each subroutine carries out the operation for the instruction it defines, called the higher instruction of that subroutine. Since this higher instruction has variable references that differ with each occurrence of the instruction, some way must exist of referring to these variable values within the defining subroutine. This is done by the type-2 references. The symbols 2-0, 2-1, 2-2, · · · , in a subroutine refer to the reference places 0, 1, 2, · · · , of the higher instruction defined by the subroutine. Thus, the type-2 references are indirect, referring to an element by referring to a reference place, that, in turn, refers to the element. The situation is shown in Fig. 9, where a given routine, L081, uses an

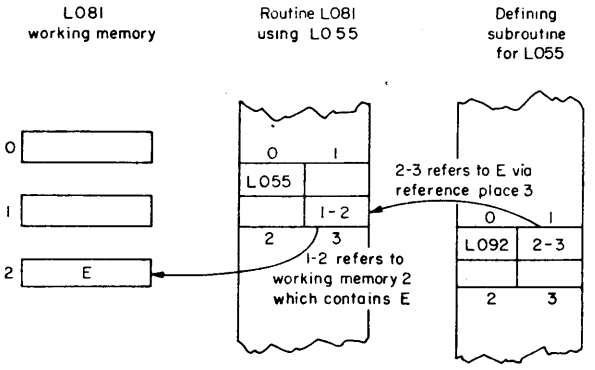


Fig. 9—IPL Type 2 reference.

instruction, L055, which is a higher instruction. Thus, L055 is defined by a subroutine, part of which is shown further to the right. In the working memory of L081, shown at the far left, the element E is located in cell 2. The instruction L055 refers to E by using symbol 1-2 in reference place 3. The instruction L092, which is in the defining subroutine of L055, refers to E by 2-3.

The first reference place (number 0) in an instruction determines the operation; or, more precisely, refers to the subroutine that will carry out the operation. All other reference places may refer to anything needed by the subroutine. Thus, an instruction is simply a format for a general process that is a function of an arbitrary number of variables.

Execution of Instructions: Access is gained to the subroutine that defines an operation by reference to an element which contains the location of that subroutine. These elements are normally collected in a directory (the Lxxx region), but may be put on lists and processed like other elements.

From the point of view of coding for JOHNNIAC, the language is entirely interpretive. When the interpreter picks up an IPL instruction, it obtains the address of the directory element from reference place 0. Besides giving the location of the defining subroutine, the directory element tells whether the instruction is a primitive or a higher instruction.[4]

In case an instruction is a primitive, the defining subroutine is in machine language, and the interpreter transfers control to it. This subroutine then either moves the referenced elements into fixed positions or adapts its instructions to the addresses of the referenced elements, and carries out the operation. Upon finishing it returns control to the interpreter.

The directory element also gives other information which is described in the section, "Other Details."

In the case of a higher-level instruction, the defining subroutine is also written in IPL and requires further interpretation. To interpret the subroutine, the interpreter sets up several lists (obtaining space for these from the available space list). The first list contains the referenced elements in the instruction; it is the "2" list from the point of view of the subroutine. The second list is the "1" list, which will hold the working memory elements, as they are set up in the subroutine. Finally, before beginning to interpret the subroutine, the interpreter must add to the next-instruction list the location of the instruction following the one it is currently interpreting.

Within the subroutine the interpreter picks up the first instruction and repeats the process described above. Thus, no matter how many levels there are in the hierarchy, the interpreter continues to set up the lists described above for each successive subroutine until it reaches a primitive instruction. After the primitive is executed, the interpreter proceeds to the next instruction in the lowest subroutine. When this subroutine is completed, the interpreter backs up to the next lowest subroutine, and so on. In operation, the memory structure for interpretation looks like a gigantic yoyo: lists of references are set up successively one "below" another as the interpreter goes down in search of a primitive, and then these lists are erased again in reverse order as the routines they correspond to are finished.

Primitive Processes

So far we have described only the outline of a language—the structure of memory and the format of the instructions. The power of the language to express complex processes depends on the set of primitive processes out of which all the others must be built.

The set of primitives in IPL is built to reflect the principle that the programmer should need to know as little as possible about the storage of information in memory. One of the clear lessons from programming experience is that small differences in what the programmer must know about the information in memory have important consequences for ease of programming. Much of the power of automatic computation derives from the fact that in order to program it is sufficient to know only the location of a number, and not the number itself. Further, large gains in programming efficiency have come from allowing the programmer to know this location only as a symbol or a relative address, rather than as an absolute address.

In IPL an attempt was made to carry this principle one step further. The concept of *working memory*, already encountered earlier, is used to divide the memory into two parts, so that all the intricate processing is done in working memory. The remaining memory, which we shall call the *list memory*, is used for permanent storage of information. This division of memory separates the primitive operations into two groups. One group of operations finds information in the list memory, makes it available in the working memory, and stores it back in the list memory again. The other group of operations processes information in working memory. There are also primitive operations for input and output, which will be discussed in the next section.

Working-Memory Operations: The primitives for processing information in working memory are roughly similar to typical machine instructions for a two-address computer. An example will make this clear. Fig. 10

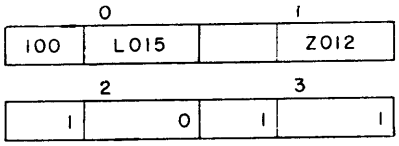


Fig. 10—IPL addition instruction.

shows a typical occurrence of L015, the addition instruction. The instruction adds a value stored in working memory 1–0 to a value in working memory 1–1. Since a working memory holds an entire element, which is a collection of attribute values, it is necessary to indicate which attribute is being added; the Z012 in reference place 1 designates this. Z012 is the name of an attribute: in this case the number of negation signs of a logic element. Hence, this instruction reads, "add the number of negations in the element 1–0 to the number of negations in element 1–1, and place the result in 1–1." This type of instruction requires the programmer to know what information is in the working memory elements and defines some elementary process involving two of them.

The set of primitives for processing information in working memory includes addition and subtraction instructions; test instructions for equality and inequality with a conditional transfer of control to some other part of the subroutine; and instructions for copying information from one working memory to another. All of these instructions use a reference, like the Z012 in the example, to designate which attributes in the element are being considered.

Find and Store Operations: The find and store instructions, which pass information between list memory and working memory, are quite different in nature from the instructions discussed above. To avoid having the programmer know anything in detail about the location of information in the list memory, all the find and store instructions take the form of searches through a list with tests to identify the information desired.

An example will make this clear. Referring back to Fig. 8, L092 is a primitive find instruction that obtains information about a logic expression. A logic expression is stored as a list of elements (see Fig. 2) in the list memory. The order of symbols in a logic expression is specified by position numbers and is unrelated to the ordering of the elements in the list. Given the position number of a logic element it is easy to compute the position number of the element that is in any given relative position to it, say, its left subelement. L092, then, is an instruction that finds an element in a logic expression

which bears a specified relative position, (*e.g.*, Z037) to some element (*e.g.*, in 2-1) already known, and that puts it in a working memory (*e.g.*, 1-0) where it can be processed further. Thus, the programmer only has to know that the element he wants bears a given relation to some known element, and he need know nothing about the actual location of this element in the list or about the rest of the logic expression. Each logic element carries as one attribute the location of the list of the logic expression containing it, so this does not have to be found separately. Typically, when an element is called for by an instruction, it is not known whether the desired element even exists; hence, L092 provides a conditional transfer of control if the desired element is not found. This particular instruction is written as a primitive because the programming problem it solves—to find a logic element bearing a given relation to a known logic element—occurs repeatedly in LT.

The instructions for finding descriptions provide a second example of how the instructions concerned with the list memory use search and test processes. As stated earlier, a list of description elements can be associated with any item in a list. An instruction to find a description requires the programmer to know the item to which the description applies. The programmer must also know the name of the description he wants. The operation then searches the list for the item, and when it finds it, searches the description list associated with that item for the description with the indicated name. Again there is no guarantee that the item is on the list, that the description is on the description list, or even that a description list exists; and the failure to find the desired description is signaled with a conditional transfer of control.

Like the find instructions, none of the store instructions depend on the precise location of an item in a list. A typical store instruction is L023, which moves descriptions from working memory to the description list of a known item on a known list. L023 searches the list until it identifies the item, then searches down the description list until it identifies the description name of the description it is storing. If it finds it, it stores the new value; if it does not find it, it stores the description as a new item on the description list. L023 must also be prepared to set up a description list in case it does not find one at all. One of the important features of the descriptions is that no space needs to be reserved for them until they are actually created.

Other Processing Instructions: Besides find and store instructions for the various types of lists, there are instructions for erasing lists, for creating lists, and for moving items from one list to another directly. There is no erasing problem in the working memory, since working memory elements are erased automatically when a subroutine has been carried out. In erasing items from lists, the instructions require only that the programmer know what item is to be erased and on what list it occurs, but not its location on the list. Likewise, the programmer does not have to know anything at all about the structure of a list to erase it, but only where it starts. The erase operations are constructed to explore all possible extensions of a list and erase them all.

Other Details

No attempt has been made with this language to build a repertoire of service routines or to make input and output exceptionally convenient. For output, the JOHNNIAC has either punched cards or a high-speed numeric printer, but we use the printer almost exclusively. There is a "print list" primitive, which prints any list however complicated and extensive. This single primitive essentially suffices for our output needs, since, if we have several lists we wish to print, we simply put them on a new superordinate list in the right order, and apply the "print list" instruction to this superordinate list. The instruction then prints out the several lists in the indicated order. We can suppress all the location words, so that only the items of information print.

JOHNNIAC has punched-card input. We use a card format for giving an arbitrary list to the computer, so that a single "read list" primitive suffices for data input. The program input is handled by the symbolic loading routine mentioned earlier.

The use of the interpretive mode for the language allows the computer easy access to its own process. As a matter of course we trace the IPL instructions that are being performed. The trace can be selective, each directory element indicating whether the trace of that instruction is to be printed or not. What is printed is the name of the subroutine (*i.e.*, the relative address of the directory element) indented according to its level in the hierarchy of routines. Since we wish to study the course of the processing as well as end results, the trace is a prime source of data.

Also as a matter of course, we keep tallies of the number of times each instruction is performed, both for our use as data and for the program's use in operating. The directory element also tells the address of the tally. For example, LT allocates its effort by using such tallies to see how much effort it has devoted to a given problem.

The devices mentioned above provide us with some debugging facilities. Since all the information connected with the hierarchy of routines is on lists (see the section on the language structure), we can print a single debugging list which contains these plus a number of other lists as items. The printing of this list (with all location words being printed) gives us most of the information we need. We also use the tracing with a selective suppression of details to aid in debugging. This procedure traces all instructions within the subroutines of interest, and none of the instructions in those of no interest.

The JOHNNIAC's 4096 words of high-speed, random-access core storage is not adequate for a program and data lists of this size. LT in operation has about 1600 words of interpretive code, about 1600 words of machine code, and about 400 words of directories, constants, etc.; hence, a total storage of about 3600 words for the program alone. We have been forced to

utilize secondary storage, which for JOHNNIAC, is a drum of 9216 words. Storage hierarchies are notorious for presenting difficult problems of accessibility, and the type of program we are working with, with its avoidance of consecutive blocks of words, simply compounds the the difficulties. So far, we have used the drum only for the program, and not for data; we are keeping almost all the higher routines on it.

When the interpreter goes to the directory element of a given instruction, it discovers whether the defining subroutine is in cores, or on the drum. If the subroutine is on the drum, it is fetched into the next available stretch in a large consecutive block in core storage. As the interpreter works down the hierarchy, more and more subroutines are brought in from the drum and gradually fill up this large block. Each subroutine remains intact until it is finished, but no attempt is made to plan or schedule trips to the drum. As soon as a subroutine is completed it is "discarded" and the next routine from the drum is placed in the same stretch of the core storage block.

Evaluation of the Language

The previous section has given a picture of the solutions we tried in programming LT. We will now consider more critically what this language accomplishes, and what its shortcomings are.

Association Memory

We have made a great issue of the flexibility of memory—the ability to create lists at will and to add and delete items from existing lists. This has certainly simplified a number of housekeeping tasks. For instance, the entire structure involved in the hierarchy of subroutines with their indefinite numbers of working memories was easily handled by means of the association memory. Similarly, in a primitive like "erase list," which must search out all items in a list of arbitrary structure, there is a need to remember an indefinite number of junctions in exploring the list. The flexible memory allows the primitive to build up a list of these points of choice, adding each new one to the front of the list.

We have made extensive use of the flexibility throughout LT, the one major program we have written in IPL. Our most complicated structure to date is a list of lists of lists connected with a routine that modifies the list of theorems used by LT as a function of experience. This same structure also has theorems (a list of logic elements) as items on multiple lists.

The association memory also has severe costs. The most obvious cost is the extra memory space needed for location words. Location words occupy about one half of the list memory, since it takes one location word to refer to each "item" word in a simple list. The proportion of location words is not much greater than one half, since the space devoted to simple lists greatly exceeds the space devoted to the more complicated structures that take additional location words. This cost factor is rather difficult to estimate, however, since alternative schemes for achieving the same total program are not known. Any component comparison is somewhat misleading, since the virtues of the association memory arise from the avoidance of planning, of reserving blocks of storage, and so on.

Another cost, which may be the more serious one, is the loss of ability to compute addresses. In a computation which can be well laid out in advance, it is often possible to assign addresses to data in such a way that the addresses can be computed in a simple fashion. For example, instead of searching a table for a function value corresponding to a given argument, the address of the function value can be made a simple function of the argument, say the argument plus a constant, and the value obtained almost without effort. This is not possible with the association memory, where the only function the address can perform is to designate the location of another word in a list.

The Language Structure

Some of the flexibilities of the language structure have provided greatly increased power in the language whereas others have not. We have not made much use of the variable number of reference places if one measures use in terms of variability of that number. Most of our instructions have about four references: the operation and three pieces of information. Both examples described in this paper are of this size. Whenever a routine exceeds about six references—one of the executive routines has 15—the references are not used as "variables" but to transmit data. In the case of the executive routine, for example, the 15 references provide a convenient place to hold all the parameter values for a run of LT. On the other hand, we have used the variable number of references considerably as a flexible communication device up and down the hierarchy of routines. Thus, in making changes in the program it is often convenient to transform what was a constant into a variable. This can be done simply by adding a new reference place to the higher instruction and replacing the constant by a type-2 reference, say 2-6, if the original instruction previously had only references 0 through 5.

We have used extensively the hierarchical properties of the language—the ability to define new subroutines in terms of old ones. The number of levels in the main part of LT is about 10, ignoring some of the recursions, which sometimes add another four or five levels. It would be interesting to compare the size of the LT program written in IPL and the program written in machine code. This is very difficult to do, since when writing in machine language one makes use of subroutines, and even of subroutines of subroutines. Hence there is no standard machine language program for comparison. However, the following figures give a rough approximation. IPL consists of about 45 primitive instructions, which take an average of about 70 JOHNNIAC instructions each. Instructions are packed two to the JOHNNIAC word, so the number of words used is roughly 35 per primitive. In addition the ma-

chine-language subroutines all include some initial code either to position the words used by the subroutine, or to adapt its instructions to the addresses of the words. This can be an appreciable fraction of some of the simpler primitives like L015, the addition instruction. Further, these statistics do not reflect the fact that the primitives themselves use a number of closed subroutines.

The LT program described in this and the companion paper contains about 45 different higher instructions, defined by 45 higher routines. A typical higher routine contains about 16 primitives and two higher instructions. If we expand the entire hierarchy for LT, ignoring recursions, we find that LT can be written as about 8000 primitives. Since the average primitive instruction takes about two JOHNNIAC words to write, it is clear that some hierarchization of subroutines is needed to compress a program like LT into manageable size.

The fact that the operation part of an instruction is a reference place like all the others, and can be treated as such, gives additional power to IPL. An operation is normally referred to by its "name," which is the relative address of the directory element that leads to the defining subroutine; *e.g.*, L015, L092, etc. However, an operation can also be referred to by a type-1 reference, such as 1-3, if the correct element is in the working storage. For instance, LT uses a set of routines, called methods, which are, roughly speaking, alternatives to one another, and are used in about the same way. There is a list of methods, which is simply a list whose items are the directory elements of the methods. The executive routine executes a method by searching the list until it finds the desired one, bringing it into a working memory (*e.g.*, 1-3) and then performing an instruction with 1-3 in the 0 reference place. If this method does not work, the executive routine finds the next method and repeats the process. Thus the executive routine is able to perform a simple iteration over the set of methods. We use this device also to compute sets of descriptions of logic expressions.

We can also use a type-2 reference for an operation. This essentially makes the operation a variable and dependent on information in the higher routine. This device is used in several places in LT, but only to allow fixed specification at a higher level. We have no examples where the operation is determined by a computation in the higher routine, although this is possible.

An entirely different kind of power arises from the flexibility of the hierarchy—the ability to do recursions. An instruction may be used in its own defining subroutine, or in any of the subroutines connected with its definition, in any way whatsoever provided that the routine does not modify itself and that the entire process terminates. The restriction on self-modification is clearly needed if the same routine is to be available at more than one level. All the information necessary to carry out the routine must be stored in the working memory, which is set up separately for each occurrence of the routine, and not within the routine. In LT there are no higher routines that modify themselves. The impetus for self-modification of routines usually arises from the use of iterative loops. In LT all iterations are accomplished by means of lists. A succession of elements is brought in from a list to fixed working-memory references, and the iteration terminates when the end of the list is reached.

There are two kinds of recursions in LT. The matching routine, which compares one logic expression with another, is an example of the first kind. The routine starts with the main connective of the expression and proceeds recursively down the tree of the expression element by element (see Fig. 2). The recursion is bound to stop, since the number of elements in any expression is finite. This recursion could also be expressed as an iteration through the list of the expression, although perhaps not so neatly.

A more fundamental recursion occurs at the highest levels of the program. Here LT has an executive routine which governs its whole problem-solving behavior. Within this routine, that is, at some lower level, are methods that generate subproblems. Also within this routine are subroutines that select the subproblem to be worked on next. A subproblem does not differ from the original problem with respect to the methods and techniques used to solve it. Hence the appropriate programming technique is to apply the entire executive routine to the subproblem; that is, to perform a recursion with the entire program. Such a recursive system will terminate if a solution is found, but since no guarantee exists that the problem will be solved there is no guarantee the machine will stop. In LT we add such a guarantee simply by having LT stop after a certain total amount of effort, a rather trivial but effective device.

The language also has its drawbacks. It is expensive; the over-all average time for a primitive is about 30 milliseconds. JOHNNIAC performs an add order in about 80 microseconds. Thus if we consider L015, the addition instruction, and compare it with a direct replication of its operation in machine language, we find we lose a factor of about 60. This is one of the more extreme cases. If we consider an instruction like L092, which is typical of the list operations, the loss factor drops to about 5. However, as in the case of the association memory, a component comparison is somewhat misleading, since all the virtues of the interpretive scheme arise from its automatic handling of the entire problem. For example, the hierarchy provides a way of keeping track of some 50 words of data in process, and it would seem that this information must be maintained if the problem is handled in any other way. The appropriate comparison is with an alternative way of coding a total problem such as LT, and no comparable alternative currently exists.

The large hierarchy with its multiple levels may seem a very expensive feature. However, its cost appears to be less than the cost of interpreting the primitives, primarily because of the infrequency of higher routines in comparison with the number of primitives. All the higher instructions account for only about 10 per cent

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.