A COMMAND STRUCTURE FOR COMPLEX INFORMATION PROCESSING

IPL VI

J. C. Shaw, A. Newell
H. A. Simon*, T. O. Ellis

P-1277

August 20, 1958

Presented at the Western Joint Computer Conference, Los Angeles, May 6, 1958.

*Carnegie Institute of Technology

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

Summary

Recent research into digital computer programs for discovering proofs to theorems in symbolic logic and playing chess has shown the desirability of languages better adapted to the requirements of such non-numeric programming tasks than are present day machine languages. A command structure which allows more indirectness in programming and requires less knowledge of the location and form of the data is described.

A COMMAND STRUCTURE FOR COMPLEX INFORMATION PROCESSING

The general purpose digital computer, by virtue of its large capacity and general-purpose nature, has opened the possibility of research into the nature of complex mechanisms per se. The challenge is obvious: we see humans carrying out information processing of a complexity that is truly baffling. Given the urge to understand either how humans do it, or alternatively, what kinds of mechanisms might accomplish the same tasks, we turn to the computer as a basic research tool. We will understand varieties of complex information processing when we can synthesize them — when we can create mechanisms that perform the same processes.

The last few years have seen a number of attempts at synthesis of complex processes. These have included programs to discover proofs for theorems (5, 7), programs to synthesize music (2), programs to play chess (1, 3), and programs to simulate the reasoning of particular humans (6). The feasibility of synthesizing complex processes hinges on the feasibility of writing programs of the complexity needed to specify these processes for a computer. Hence, a limit is imposed by the limit of complexity that the human programmer can handle. The measure of this complexity is not absolute, for it depends on the programming language he uses. The more powerful the language, the greater will be the complexity of the programs he can write. In our own work, we have sought to increase the upper limit of complexity of the processes we can specify by developing a series of languages, called information processing languages (IPL's), that reduce significantly the demands made upon the programmer in his communication with the computer. Thus, the IPL's represent a series of attempts to construct sufficiently powerful languages to permit the programming of the kinds of complex processes mentioned above.

The IPL's designed so far have been realized interpretively on current computers (4). Alternatively, of course, any such language can be viewed as a set of specifications for a general-purpose computer. An IPL can be implemented far more expeditiously in a computer designed to handle it than by interpretation in a computer designed with a quite different command structure. The mismatch between the IPL's we have designed and current computers is appreciable: 150 machine cycles are needed to do what one feels should take only two or three machine cycles.[1]

The purpose of this paper is to consider an IPL computer - that is, a computer constructed so that its machine language is an information processing language. We will call this language IPL-VI, for it is the sixth in the series of IPL's we have designed. This version has not been realized interpretively, but has resulted from considering hardware requirements in the

[1]. It will become apparent that the difficulty would not be removed by "compiling" instead of "interpreting", to resurrect a set of well-worn distinctions. The operations that are mismatched to current computers must go on during execution of the program. and hence cannot be compiled out.

light of our programming experience with the previous languages.

Some limitations must be placed on the investigation. We will concern ourselves only with the central computer — the command structure, the form of the machine operations, and the general arrangements of the central hardware. We will neglect rather completely input/output and secondary storage systems. This does not mean these are unimportant or that they present only simple problems. The problem of secondary storage is difficult enough for current computing systems; it is exceedingly difficult for IPL systems, since in such systems internal memory is not organized in neat block-like packages for ease of shipment to the secondary store.

Nor is it the case that we would place an order for the IPL computer we are about to describe without further experience with it. The human organism is limited in its power of vision: the designer knows not what he begets. IPL's are sufficiently different from current computer languages that their utility can be evaluated only after much programming. Moreover, since IPL's are designed to specify large complicated programs, the utility of the linguistic devices incorporated in them cannot be ascertained from simple examples.

One more caution is needed to provide a proper context for this paper. Most of the computing world is still concerned with essentially numerical processes, either because the problems themselves are numerical or because non-numerical problems have been appropriately arithmetized. The kinds of problems we have been concerned with are essentially non-numerical, and we have tried to cope with them without resort to arithmetic models. Hence the IPL's have not been designed with a view to carrying out arithmetic with great efficiency.

FUNDAMENTAL GOALS AND DEVICES

The basic aim, then, is to construct a powerful programming language for the class of problems in which we are interested. Given the amount and kind of output desired from the computer, we wish to reduce the size and complexity of the specification — the program — that has to be written in order to secure this output.

The goal is to reduce programming effort. This is not the same as reducing the computing effort required to produce the desired output from the specification. Programming feasibility must take precedence over computing economics; since we do not yet know how to write a program that will enable a computer to teach itself to play chess, it is premature to ask whether it would take such a computer one hour or one hundred hours to make a move. This is not meant as an apology, but as support for the contention that in seeking to write programs for very large and complicated tasks the overriding initial concerns must be to attain enough flexibility, abbreviation, and automation of the underlying computing processes to make programming feasible. And these concerns have to do with the power of the programming language rather than the efficiency of the system that executes the program.

In the next section we will begin a straightforward description of an IPL computer. To put the details in context, we will discuss in the remainder of this section the basic devices that IPL-VI uses to achieve a measure of power and flexibility. These devices include: organization of memory into list structure, provision for breakouts, identity of data with program, two-stage interpretation, invariance of program during execution, provision for responsibility assignments, and centralized signalling of test results.

List Structure

The most fundamental and characteristic feature of the IPL's is that they organize memory into list structures, whose arrangement is independent of the actual physical geometry of the memory cells and which undergo continual change as computation proceeds. In all computing systems the topology of memory — the characteristics of hardware and program that determine what memory cells can be regarded as "next to" a given cell — plays a fundamental role in the organization of the information processing. This is obviously true for serial memories, like tape; it is equally true for random access memories. In random access memories the topological structure derives from the possibility of performing arithmetic operations on the memory addresses that make use of the numerical relations among these addresses. Thus, the cell with address 1435 is "next to" cell 1436 in the specific sense that the second can be reached from the first by adding one to the number in a counter.

In standard computers we make use of the static topology based on memory addresses to facilitate programming and computation. Index registers and relative addressing schemes, for example, make use of program arithmetic and depend for their efficacy upon an orderly matching of the arrangement of information in memory with the topology of the addressing system.

When memory is organized in a list structure, the relation between information storage and topology is reversed. The topology of memory is continually modified to adapt to the changing needs of organization of memory content. No arithmetic operations on memory addresses are permitted; the topology is built on a single, asymmetric, modifiable, ordinal relation between pairs of memory cells which we shall call adjacency. The system contains processes that make use of the adjacency relations in searching memory, and processes that change these relations at will and cheaply in the course of processing.

A list structure can be established in computer memory by associating with each word in memory an address that determines what word is adjacent to it, as far as all the operations of the computer are concerned. We pay the price in memory space of an additional address associated with each word, so that we can change the adjacency relation as quickly as we can change a word in memory. Having paid this price, however, many of the other basic features of IPL's are obtained almost without cost: unlimited hierarchies of subroutines; recursive definition of processes; variable numbers of operands for processes; and unlimited complexity of data structure, capable of being created and modified to any extent at execution time.

Breakouts

Languages require grammar-fixed structural features with the aid of which they can be interpreted. Grammar imposes constraints on what can be said, or said simply, in a language. However, the constraints created by fixed grammatical format can be alleviated at the cost of introducing an additional stage of processing - by devices that allow one to "break out" of the format and to use more general modes of specification than the format permits. Devices for breakouts exchange processing time for flexibility. Several devices achieve this in IPL-VI. Each is associated with some piece of format.

As an illustrative example, IPL-VI has a single-address format. Without breakout devices, this format would permit an information process to operate on only a single operand as input, and would permit the operand of a process to be specified only by giving its address. Both of these limitations are removed: the first by using a special communication list to store operands, the second by allowing the address for an operand to refer either to the operand itself or to any process that will determine the operand.

The latter device - which allows broad freedom in the method of specifying an operand - illustrates another important facet of the flexibility problem. Breakouts are of great importance in reducing the burden of planning that is imposed on the programmer. It is certainly possible, in principle, to anticipate the need for particular operands at particular stages of processing, and to provide the operands in such a way that their addresses are known to the programmer at the appropriate times. This is the usual way in which machine coding is done. However, such plans are not obtained without cost; they must be created by the programmer. Indeed, in writing complex programs, the creation of the plan of computation is the most difficult part of the job – it constitutes the task of "programming" that is sometimes distinguished from the more routine "coding." Thus, devices that exchange computing time for a reduction in the amount of planning required of the programmer provide significant increases in the flexibility and power of the language.

Identity of Data With Programs

In current computers, the data are considered "inert." They are symbols to be operated upon by the program. All "structure" of the data is initially developed in the programmer's head and encoded implicitly into the programs that work with the data. The structure is embodied in the conventions that determine what bits the processes will decode, and so on.

An alternative approach is to make the data "active." All words in the computer will have the instruction format; there will be "data" programs, and the data will be obtained by executing these programs. Some of the advantages of this alternative are obvious: the full range of methods of specification available for programs is also available for data; a list of data, for example, may be specified by a list of processes that determine the data. Since data are only desired "on command" by the processing programs, this approach leads to a computer that, although still serial in its control, contains at any given moment a large number of parallel active programs, frozen in the midst of operation and waiting until called upon to produce the next operation or piece of data. This identity of data with program can be attained only if the processing programs require for their operation no information about the structure of the data programs – only information about how to receive the data from them.

Two-stage Interpretation

To identify the operand of an IPL-VI instruction, a designating operation operates on the address part of the instruction to produce the actual operand. Thus, depending on what designating operation is specified, the address part may itself be the operand, may provide the address of the operand, or may stand in an even less direct relation to the operand. The designating operation may even delegate the actual specification of the operand to another designating operation.

Invariance of Program During Execution

In order to carry out generalized recursions, it is necessary to provide for the storage of indefinite amounts of variable information necessary for the operation of such routines. In IPL-VI all the variable information is stored externally to the associated routine, so that the routine remains unmodified during execution. The name of a routine can appear in the definition of the routine itself without causing difficulty at execution time.

Responsibility Assignments

The automatic handling of such processes as erasing a list, or searching through a list requires some scheme for keeping track of what part of the list has been processed, and what part has not. For example, in erasing a program containing a local subroutine that appears more than once within the program, care must be taken to erase the subroutine once and only once. This is accomplished by a system for assigning responsibility for the parts of the list. In general, the responsibility code in IPL-VI handles these matters without any explicit attention from the programmer, except in those few situations where the issue of responsibility is the central problem.

Centralized Signalling of Test Results

The structure of the language is simplified by having all conditional processes set a switch to symbolize their output instead of producing an immediate conditional transfer of control. Then, a few specialized processes are defined that transfer control on the basis of the switch setting. By symbolizing and retaining the conditional information, the actual transfer can be postponed to the most convenient point in the processing. The flexibility obtained by this device proves especially useful in dealing with the transmission of conditional information from subroutines to the routines that call upon them.

GENERAL ORGANIZATION OF THE MACHINE

The machine we are describing can profitably be viewed as a "control computer." It consists of a single control unit with access to a large random-access memory. This memory should contain 10^5 words or more. If less than 10^4 words are available in the primary memory, there will probably be too frequent occasions for transfer of information between primary and secondary storage to make the system profitable.

The operation of the computer is entirely non-arithmetic, there being no arithmetic unit. Since arithmetic processes are not used as the basis of control, as they are in standard computers, such a unit is inessential, although it would be highly desirable for the computer to have access to one if it is to be given arithmetic tasks. The computer is perfectly capable of proving theorems in logic or playing chess without an arithmetic adjunct.

Memory

The memory consists of cells containing words of fixed length. Each word is divided into two parts, a symbol and a link. The entire memory is organized into a list structure in the following way. The link is an address; if the link of a word a is the address of word b, then b is adjacent to a. That is, the link of a word in a simple list is the address of the next word in the list.

The symbol part of a word may also contain an address, and this may be the address of the first word of another list. As we indicated earlier, the entire topology of the memory is determined by the links and by addresses located in the symbol parts of words. The links permit the creation of simple lists of symbols; the links and symbol parts together, the creation of branching list structures.

The topology of memory is modified by changing addresses in links and symbol parts, thereby changing adjacency relations among words. The modification of link addresses is handled directly by various list processes without the attention of the programmer. Hence, the memory can be viewed as consisting of symbol occurrences connected together by mechanisms or structure whose character need not be specified.

The basic unit of organization is the list - a set of words linked together in a particular order by means of their link parts, in the way we explained. The address of the first word in the sequence is the name of the list. A special terminating symbol, T, whose link is irrelevant, is in the last word on every list. A simple list is illustrated in Figure 1; its name is L_{100}, and it contains two symbols, S_1 and S_2.

We have seen that the symbols in a list may themselves designate the names of other lists.[2] Thus, a list may be a list of lists, and each of its sublists may be a list of lists.

An example of a list structure is shown in Figure 2. The name of the list structure is the name of the main list, L_{200}. L_{200}contains two sublists, L_{300} and L_{500}, plus an item of information, I_4, that is not a name of a list. L_{300}in its turn consists of item I_1 plus another sublist, L_{400}, while L_{500}

[2]. The symbols themselves have a special format, so that they are not names of lists but designate the names in a manner we shall describe.

contains just information, and is not broken out further into sublists. Each of these lists terminates in a word that holds the symbol T.

Available Space List

A list uses a certain number of cells from memory. Which cells it uses is unimportant as long as the right linkages are set up. In executing programs that continually create new lists and destroy old ones, two requirements arise. When creating a list, cells in memory must be found that are not otherwise occupied — that are available for the new list. Conversely, when a list is destroyed — when it is no longer needed in the system — its cells become available for other uses, but something must be done to gain access to these available cells when they are needed.

The device we use to accomplish these two logistic functions is the available space list. All cells that are available are linked together into a single long list. Whenever cells are needed, they are taken from the front of this available space list; whenever cells are made available, they are inserted on the front of the available space list just behind the fixed register that holds the link to the first available space. The operations of "taking" cells from the available space list and "returning" cells to the available space list involve, in each case, only changes of addresses in a pair of links.

Organization of Central Unit

Figure 3 shows the special registers of the machine and the main information transfer paths. There are four addressable registers that accomplish fixed functions. These are shown as part of the main memory, but would be fast access registers.

L_0: Communication List The system allows the introduction of unlimited numbers of processes with variable numbers of inputs and outputs. The communication of inputs and outputs among processes is centralized in a communication list with known name, L_0. All subroutines find their inputs on this list, and all subroutines put their outputs on this same list.

L_1: Available Space List All cells not currently being used are on the available space list: cells can be obtained from it when needed and are returned to it when they are no longer being used.

L_2: List of Current Instruction Addresses (CIA) At any given moment in working sequentially through a program, there will be a whole hierarchy of instructions that are in process or interpretation, but whose interpretation has not been completed. These will include the instruction currently being interpreted, the routine to which this instruction belongs, the superroutine to which this routine belongs, and so on. The CIA list is the list of addresses of this hierarchy of routines. The first symbol on the list gives the address of the instruction currently being interpreted; the second symbol gives the address of the current instruction in the next higher routine, and so on. In this system it proves to be preferable to keep track of the current instruction being interpreted, rather than the next one.

L_3: <u>List of Current CIA Lists</u> The control sequence is complicated in this computer by the existence of numerous programs which become active when called upon, and whose processing may be interspersed among other processes. Hence, a single CIA list does not suffice; there must be such a list for each program that has not been completely executed. Therefore, it is necessary also to have a list that gives the names of the CIA lists that are active. This list is L_3.

Besides these special addressable registers, three non-addressable registers are needed to handle the transfers of information. Two of these, R_1 and R_2, are each a full word in length, and transfer information to and from memory. The register R_1 receives input from memory; R_2 transmits output to memory. The comparator that provides the information for all tests takes as its input for comparison the symbol in R_1 and R_2. This pair of registers also performs a secondary function in regenerating words in memory: the basic Read operation from memory is assumed to be destructive; a non-destructive Read merely shunts the word received from memory in R_1 to R_2 and back, by means of a Write operation, to the same memory cell.

A register, A, which holds a single address, controls references to the memory - that is, specifies the memory address at which a Read or Write operation is to be performed. References to the four addressable Registers, L_0 to L_3, can be made either by A or directly by the control unit itself; other memory cells can be referred to only by A. Finally, the computer has a single bit register which is used to encode and retain test results.

The Environment

We can now indicate how input/output, secondary storage, and high speed arithmetic could be handled with such a machine. The machine manipulates symbols: it can construct complex structures, search them, and tell when two symbol occurrences are identical. These processes are sufficient to play chess, prove theorems, or do most other tasks. The symbols it manipulates are not "coded"; they simply form a set of arbitrary distinguishable entities, like a large alphabet.

This computer can manipulate things outside itself if hardware is provided to make some of its symbols refer to outside objects, and other symbols refer to operations on these objects. It could do high speed arithmetic, for example, if some of its symbols were names of words in memory encoded as numbers - in the usual computer fashion - and others were names of the arithmetic operations. In such a scheme these words would not be in the IPL language; they would have some format of their own, either fixed or floating-point, binary or decimal. They might occupy the same physical memory as that used by the control computer. Thus the IPL language would deal with numbers at one remove - by their names - in much the same manner as the programmer deals with numbers in a current computer. A similar approach can be used for manipulating printers, input devices, and so on.

THE WORD AND ITS INTERPRETATION

All words in IPL have the same format, shown in Figure 4.

The word _a_ is divided into two major parts: the symbol part, _bcde_, and the link, _f_. We have observed that the programmer never deals explicitly with the link, although we shall frequently represent it explicitly to show how manipulations are being accomplished. Since the same symbol can appear in many words, we may need to speak of the _symbol occurrence_ of the symbol in the word _a_.

A symbol occurrence consists of an _operation_, _b_, a _designation operation_, _c_, an _address_, _d_, and a _responsibility code_, _e_. The operation, _b_, takes as operand a single symbol occurrence, which we shall call _s_. The _operand_, _s_, is determined by applying the designation operation, _c_, to the address, _d_. Thus the process determined by a word is carried out in two stages: the first-stage operation (the designation operation) determines an operand that becomes the input to the second-stage operation.

The Responsibility Bit

The single bit, _e_, is an essential piece of auxiliary information. The address, _d_, in a symbol may be the address of another list structure. The responsibility code in a symbol occurrence indicates whether this occurrence is "responsible" for the structure designated by _d_. If the same address, _d_, occurs in more than one word, only one of these will indicate responsibility for _d_.

The main function of the responsibility code is to provide a way of searching a branching list structure so that every part of the structure will, sooner or later, be reached, and so that no part will be reached twice. The need for a definite assignment of responsibility for the various parts of the structure can be seen by considering the process of erasing a list. Suppose that a list has a sublist that appears twice on it, but that does not appear anywhere else in memory. When the list is erased, the sublist must be erased if it is not to be lost forever - and the space it occupies with it. However, after the sublist has been erased when an occurrence of its name is encountered on the list, it is imperative that it not be erased again on the second encounter. Since the words used by the sublist would have been returned to the available space list prior to the second encounter, only chaos could result from erasing it again. The responsibility code would indicate responsibility, in erasing, for one and only one of the two occurrences of the name of the sublist.

Detailed consideration of systems of responsibility is inappropriate in this paper. We believe that an adequate system can be constructed with a single bit, although a system that will handle merging lists also requires a responsibility bit on the link f. The responsibility code is essentially automatic. The programmer does not need to worry about it except in those cases where he is explicitly seeking to modify structure.

Interpretation Cycle

A routine is a list of words - that is, a list of instructions. Its name is the address of the first word used in the list. The interpretation of a program proceeds according to a very simple cycle. An instruction is fetched to the control unit. The designation operation is decoded and executed, placing the location of s̲ in the Address Register, A, of Figure 3. Then operation b̲ is decoded and performed on s̲. The cycle is then repeated using f̲ to fetch the next instruction.

THE OPERATION CODES

The simple interpretation cycle described above provides none of the powerful linguistic features that we outlined at the beginning of the paper: hierarchies of subroutines, data programs, breakouts, and so on. These features are obtained through particular b̲ and c̲ operations that modify the sequence of control. The operation codes will be explained under the following headings: the designation code, sequence-controlling operations, save and delete operations, communication list operations, signal operations, list operations, and other operations.

The Designation Code

The designation operation, c̲, operates on the address, d̲, to designate a symbol occurrence, s̲, that will serve as input, or operand, for the operation b̲. The designation operation places the address of the designated symbol, s̲, in the Address Register.

The designation codes we propose, basing our choice on their usefulness in coding with the IPL's, are shown in Table I. The first four, c̲=0,1,2, or 3, allow four degrees of directness of reference. They are usable when the programmer knows in advance where the symbol, s̲, is located. To illustrate their definition, consider an instruction $\underline{a}_1$, with parts $\underline{b}_1$, $\underline{c}_1$, $\underline{d}_1$, and $\underline{e}_1$, which we can collectively call $\underline{s}_1$. The address part, $\underline{d}_1$, of this instruction may be the address of another instruction $\underline{d}_1=\underline{a}_2$; the address part, $\underline{d}_2$, of $\underline{a}_2$ may be the address of $\underline{a}_3$, and so on.

The code $c_1=1$ means that $\underline{s}$ is the symbol whose address is $\underline{d}_1$, that is, the symbol $\underline{s}_2$. In this case the designating operation puts $\underline{d}_1$, the address of $\underline{s}_2$, in the Address Register. The code $\underline{c}_1=2$ means that $\underline{s}$ is $\underline{s}_3$; hence, the operation puts $\underline{d}_2$, the address of $\underline{s}_3$, in the Address Register. The code $\underline{c}_1=3$ puts $\underline{d}_3$, the address of $\underline{s}_4$, in the Address Register. Finally, $c_1=0$ designates as $\underline{s}$ the actual symbol in $\underline{a}_1$ itself; hence means that $\underline{b}$ is to operate on $\underline{s}_1$. Therefore, this operation places $\underline{a}_1$ in the Address Register.

The remaining two designation operations, $c=4$ and 5, introduce another kind of flexibility, for they allow the programmer to delegate the designation of $\underline{s}$ to other parts of the program. When $c_1=4$, the task of designating $\underline{s}$ is delegated to the symbol of the word $\underline{d}_1=\underline{a}_2$. In this case, $\underline{s}$ is found by applying the designation operation, $\underline{c}_2$ of word $\underline{a}_2$, to the address, $\underline{d}_2$, of word $\underline{a}_2$. An operation of this kind permits the programmer to be unaware of the way in which the data are arranged structurally in memory. Notice that the operation permits an indefinite number of stages of delegation, since if we also have $\underline{c}_2=4$, there will be a further delegation of the designation operation to $\underline{c}_3$ and $\underline{d}_3$ in word $\underline{a}_3$.

The last designation operation, c=5, provides both for delegation and a breakout. With c_1=5, $\underline{d}_1$ is interpreted as a process that determines $\underline{s}$. Any program whatsoever, having its initial instruction at $\underline{d}_1$, can then be written to specify $\underline{s}$. When this program has been executed, an $\underline{s}$ will have been designated, and the interpretation will continue by reverting to the original cycle – that is, by applying $\underline{b}_1$ to the $\underline{s}$ that was just designated. It is necessary to provide a convention for communicating the result of process $\underline{d}_1$ to the interpreter. The convention we have used is that $\underline{d}_1$ shall leave the location of $\underline{s}$ in L_0, the standard communication cell.

Sequence-Controlling Operations

Table II lists the 35 $\underline{b}$ operations. The first twelve of these are the ones that affect the sequence of control. They accomplish five quite different functions: executing a process (b=1,10), executing variable instructions (b=2), transferring control within a routine (b=3,4,5), transferring control among parallel program structures (b=0,6,7,8,9), and, finally, stopping the computer (b=11).

A routine is a list of instructions; its name is the address of the first word in the list. To execute a routine, we designate its name (i.e., its name becomes the $\underline{s}$ of the previous section) and apply it to the operation b=1, Execute $\underline{s}$. The interpreter must keep track of the location of the instruction that is being executed in the current routine and return to that location after completing the execution of the instruction (which, in general, is a subroutine). All lists end in a word containing b=10, which terminates the list and returns control to the higher routine in which the subroutine just completed occurred. (The symbol T is really any symbol with b=10.)

Figure 5 provides a simple illustration of the relations between routines and their subroutines. In the course of executing the routine L_{10} (i.e., the instructions that constitute list L_{10}), an instruction, $(1,0,L_{20})$, is encountered that is interpreted as "execute L_{20}." In the course of executing L_{20}, an instruction is encountered that is interpreted as "execute L_{30}." Assuming that L_{30} contains no subroutines, its instructions will be executed in order until the terminate instruction is reached. Because of the 10 in its b part, this instruction returns control to the instruction that follows L_{30} in L_{20}. When the final word in L_{20} is reached, the operation code 10 in its b part returns control to L_{10}, which then continues with the instruction following L_{20}.[3] This is a standard subroutine linkage, but with all the sequence-control centralized.

The operation code b=2, Interpret s, delegates the interpretation to the word s. The effect of an instruction containing b=2 is exactly the same as if the instruction contained, instead, the symbol, s, that is designated by its c and d parts. One can think of the instruction with b=2 as a variable whose value is s. Thus, a routine can be altered by modifying the symbol occurrence s, without any modification whatsoever in the words

[3] Only the b part, b=10, of the terminal word in a routine is used in the interpretation; the c and d parts are irrelevant.

belonging to the routine itself.

The three operations, b=3, 4, and 5, are standard transfer operations. The first is an unconditional transfer; the two others transfer conditionally on the signal bit. As we mentioned earlier, all binary conditional processes set the signal either "on" or "off." In order to describe operations b=0,6,7,8,9 we need to define the concept of program structure. A program structure is a routine together with all its subroutines and designation processes. Such a structure corresponds to a single, although perhaps complex, process. The computer is capable of holding, at a given time, any number of independent program structures, and can interrupt any one of these processes, from time to time, in order to execute one of the others. All of these structures are coordinate, or parallel, and the operations b=0,6,7,8,9, are used to transfer control (perhaps conditionally) from the one that is currently active to a new one or to the previously active one. In this sense, the computer we are describing may be viewed as a serial control, parallel program machine.

Suppose that we are proceeding with the execution of a particular routine in program structure A. Operation b=6 will transfer control to an independent program structure determined by s; call it B. The machine will then begin to execute B. When it encounters a Stop Interpretation operation (b=0) in B, control will be returned to the program structure, A, that was previously active. But the Stop Interpretation operation, unlike the ordinary termination, b=10, does not mark the end of program structure B. At any later point in the execution of A, control may again be transferred to B, in which case execution of the latter program will be resumed from the point where it was interrupted by the earlier Stop Interpreting command. The operation that accomplishes the second transfer of control from A to B is b=7, Continue Parallel Program s. Thus, b=0 is really an "interrupt" operation, which returns control to the previous structure, but leaves the structure it interrupts in condition to continue at a later point. There can be large numbers of independent program structures all "open for business" at once, with a single control passing from one to the other, determining which has access to the processing facilities, and gradually executing all of them. Operations b=8 and 9 simply allow the interruption to be conditional on the test switch.

Notice that the passage of control from one structure to another is entirely decentralized — it depends upon the occurrence of the appropriate b operations in the program structure that has control.

When control is transferred to a parallel program structure, either of two outcomes is possible. Either a Stop Interpretation instruction is reached in the structure to which control has been transferred, or execution of that structure is completed and a termination reached. In either case, control is returned to the program structure that had it previously, together with information as to whether it was returned by interruption or by termination. Thus, b=0 turns the signal bit on when it returns control; b=10 in the topmost routine of a structure turns the signal off.

The operation, b=11, simply halts. Processing continues from the location where it halted upon receipt of an external signal, "go."

<u>Save and Delete Operations</u>

The two operations, b=12 and 13, are sufficiently fundamental to warrant extended treatment. Consider a word, L_{100}, that contains the symbol I_1:

LOCATION	SYMBOL	LINK
L_{100}	I_1	t

The link of L_{100}, <u>t</u>, indicates that the next word holds the termination operation, b=10. The Save operation (b=12) provides a copy of I_1 in such a way that I_1 can later be recalled, even if in the meantime the symbol in L_{100} has been changed. After the Save operation has been performed on <u>s</u>=L_{100}, we have:

LOCATION	SYMBOL	LINK
L_{100}	I_1	L_{200}
L_{200}	I_1	t

A new cell, which happened to be L_{200}, was obtained during the Save operation from the available space list, L_1, and a copy of I_1 was put in it. We can now change the symbol in L_{100} without losing I_1 irretrievably. Suppose we copy a different symbol, say I_2, into L_{100}. Then we have:

LOCATION	SYMBOL	LINK
L_{100}	I_2	L_{200}
L_{200}	I_1	t

Although we have replaced I_1 in L_{100}, we can recover I_1 by performing the Delete operation, b=13. Before we show how the latter is carried out, it will be instructive to show what happens when the Save operation on L_{100} is iterated. If it is executed again, it will make a copy of I_2. We get:

LOCATION	SYMBOL	LINK
L_{100}	I_2	L_{300}
L_{300}	I_2	L_{200}
L_{200}	I_1	t

Notice that the cell L_{200}, in which the copy of symbol I_1 is retained, was not affected at all by this second Save operation. Only the top cell in the list and the new cell from the available space list are involved in the transaction of saving. The same process is performed no matter how long the list that trails out below L_{100}; thus, we can apply the save operation as many times as we wish with constant processing time.

We are now ready to illustrate the Delete operation, b=13, applied to the symbol I_2 in L_{100}. This operation puts the symbol and link of the second word in the list, L_{300}, into the first cell, L_{100}, and puts L_{300} back on the available space list, with the following result:

LOCATION	SYMBOL	LINK
L_{100}	I_2	L_{200}
L_{200}	I_1	t

We have returned to the exact situation we had before we performed the last Save.

In our description of the Delete operation up to this point, we have considered only the changes it makes in the "push-down" list - in this case, L_{100}. The operation does more than this, however; Delete s also erases all structures for which the symbol s (I_1 and I_2 in our examples) is responsible. When a copy of a symbol is made - e.g., the operation that initially replaced I_1 by I_2 in L_{100} - the copy is not assigned responsibility for the symbol (we set e=0 in the copy). Thus, no additional erasing would be required in the particular Delete operation we illustrated. If, on the other hand, the I_2 that was moved into L_{100} had been responsible for the structure that could be reached through it (if it were the name of a list, for example), then a second Delete operation, putting I_1 back into L_{100}, would also erase that list and put all its cells back on the available space list. Thus Delete is also equivalent to Erase a List Structure.

Communication List Operations

In describing a process as a list of subprocesses, we have bypassed entirely the question of inputs and outputs from the processes. Since each subroutine has an arbitrary and variable number of operands as input, and provides to the routine that uses it an arbitrary number of outputs, some scheme of communication is required among routines. The communication list, L_0, accomplishes this function in IPL.

We require that the inputs and outputs to a routine be symbols. This is no real restriction since a symbol can be the name of any list structure whatever. Each routine will take as its inputs the first symbols in the L_0 list. That is, if a routine has three inputs, then the first three symbols in L_0 are its inputs. Each routine must remove its inputs from L_0 before terminating with b=10, so as to permit the use of the communication list by subsequent routines. Finally, each routine leaves its outputs at the head of list L_0.

The b operations 14 through 19 are used for communication in and out of L_0. Their one common feature is that, whenever they put a symbol in L_0, they save the symbol already there — that is, they push down the symbols already "stacked" in L_0. Likewise, whenever a symbol is moved from L_0 to memory, the symbol below it in L_0 "pops up" to become the top one.[4]

The four operations, b=14, 15, 16, and 17, are the main in-out operations for L_0. Two options are provided, depending on whether the programmer wishes to retain the s in memory (b=14 and 16) or destroy it (b=15 and 17).[5]

Operation b=18 is a special input to aid in the breakout designation operation, c=5. Recall that the latter operation

---

4. To be precise, the responsibility bit travels with a symbol when it is moved. Hence, for example, b=16 and 17, do not, unlike the Delete operation, erase the structure for which $1L_0$ is responsible.
5. The move in operation 15 has the same significance as in 16 and 17; the responsibility bit moves with the symbol, and the symbol previously in the location of s, is recalled.

requires d to place the location of s, the symbol it determines, in L_0. Operation 18 allows the process d to accomplish this.

Operation b=19 provides the means for creating structures. It takes a cell, say L_{200}, from available space, and puts its name – as the symbol $(0,0,L_{200})$ – in the location of the designated symbol, s. The symbol s, previously in this location is pushed down and saved.

Signal Operations

Ten b operations are primarily involved in setting and manipulating the signal bit. Observe that the test of equality (b=20 and 21) is identity of symbols. Since there is nothing in the system that provides a natural ordering of symbols, inequality tests like $s > 1L_0$, are impossible.[6] It is necessary to be able to detect the responsibility bit (b=22), since there are occasions when the explicit structure of lists is important, and not just the information they designate. Finally, although the signal bit is just a single switch, it is necessary to have two symbols, one corresponding to "signal on" and the other to "signal off" (b=26 and 27), so that the information in the signal can be retained for later use (b=28 and 29).

The sense of the signal is not arbitrary. In general "off" is used to mean that a process "failed," "did not find," or the like. Thus, in operations b=6 and 7, the failure to find a Stop Interpretation operation sets the signal to "off." Likewise, the end of a list will be symbolized by setting the signal to "off."

[6]. By $1L_0$ we mean the symbol in L_0.

List Operations

Both the Save and Delete operations are used to manipulate lists, but besides these, several others are needed. The three operations, <u>b</u>=30, 31, 32, allow for search over list structures. They can be paraphrased as: Get the Referent, Turn Down the Sublist, and Get the Next Word of the List. They all have in common that they replace a known symbol with an unknown symbol. This unknown symbol need not exist; that is, the symbol referred to may contain a <u>b</u>=10 operation, which means that the end of the list has been reached. Consequently, the signal is always set "on" if the symbol is found, and "off" if it is not found. One of the virtues of the common signal is apparent at this point, since, if the programmer knows that the symbol exists, he will simply ignore the signal. Instruction formats that provide for additional addresses for conditional transfers would force the programmer to attend to the condition even if it only meant leaving a blank space in the program.

To illustrate how these search operations work, Figure 6 shows a list of lists, L_{300}, and a known cell, L_{100}. Cell L_{100} contains the reference to the list structure. The programmer does not know how the list, L_{300}, is referenced. He wants to find the last symbol on the last list of the structure. His first step is $(30,1,L_{100})$ which replaces the reference by the name of the list, L_{300}. He then searches down to the end of list L_{300} by doing a series of operations: $(32,1,L_{100})$. Each of these replaces one location on the list by the next one.

In fact, a loop is required, since the length of the list is unknown. Hence, after each Find the Next Word operation, he must transfer, on the basis of the signal, back to the same operation if the end of the list hasn't been reached. The net result, when the end of the list is reached, is that the location of the last word on list L_{300} rests in L_{100}. Since in this example he wants to go down to the end of the sublist of the last word on the main list, he next performs $(31,1,L_{100})$. This operation replaces the location of the last word with the name of the last list, L_{700}. Now the search down the sublist is repeated until the end is again reached, at this point the location of the last symbol on the last list is in L_{100}, as desired. The sequence of code follows:

LOCATION	SYMBOL	LINK
	b c d	
	$30,1,L_{100}$	
L_{888}	$32,1,L_{100}$	
	$4,0,L_{888}$	
	$31,1,L_{100}$	
L_{999}	$32,1,L_{100}$	
	$4,0,L_{999}$	

The operations, b=33 and 34, allow for inserting symbols in a list either before or after the symbol designated. The lists in this system are one-way: although there is always a way of finding the symbol that follows a designated symbol, there is no way of finding the symbol that precedes a designated symbol. The Insert Before operation does not violate this rule. In both operations 33 and 34, a cell is obtained from the available space list and inserted after the word holding the designated symbol. (This is identical with the first step of the Save operation.) In the Insert Before operation (b=33) the designated symbol, s, is copied into the new cell, and $1L_0$ is moved into the previous location of s. In Insert After (b=34), the designated symbol is left unchanged, and $1L_0$ is moved into the new cell. In both cases $1L_0$ is moved, that is, it no longer remains at the head of the communication list.

Other Operations

This completes our account of the basic complement of operations for the IPL computer. These form a sufficient set of operations to handle a wide range of non-numerical problems. To do arithmetic efficiently, one would either add another set of b's covering the standard arithmetic operations or deal with these operations externally via a breakout operation on b (not formally defined here) that would move a full symbol into a special register for hardware interpretation relative to external machines: adders, printers, tapes, etc.

The set of operations has not been described for reading and writing the various parts of the word: b, c, d, e, and f (although it may be possible to automatize this last completely). These operations rarely occur, and it seemed best to ignore them as well as the input-output operations in the interest of simple presentation.

INTERPRETATION

In this section we will describe in general terms the machine interpretation required to carry out the operation codes we have prescribed. There is not space to be exhaustive, and we will proceed by discussing selected examples.

Direct Designation Operations

Figure 7 shows the information flows for $\underline{e}=2$, an operation that is typical of the first four designation operations. These flows follow a simple, fixed interpretation sequence. Assume that instruction $(\_,2,L_{100})$ is inside the control unit. The contents of L_{100} are brought into R_1, the input register, then transferred to R_2, the output register, and back to L_{100} again. The $\underline{d}$ part of R_2 now contains the location of $\underline{s}$, and this location is transferred from R_2 to the Address register.

Execute Subroutine (b=1)

When Execute $\underline{s}$ is to be interpreted, the Address register already contains the location of $\underline{s}$, which was brought in during the first stage of the interpretation cycle. L_2, the Current Instruction Address list (CIA), holds the address of the instruction containing the Execute order. A Save operation is performed on L_2, and $\underline{s}$ is transferred into L_2, which ends the operation. The result is to have the interpreter interpret the first instruction on the sublist next, and to proceed down it in the usual fashion. Upon reaching the terminate operation, b=10, the delete operation is performed on $1L_2$, thus bringing back the original instruction address from which the subroutine was executed. Now, when the interpretation cycle is resumed, it will proceed down the original list. Thus, the two operations, save and delete, perform the basic work in keeping track of subroutine linkage.

Parallel Programs

A single program structure - that is, a routine with all its subroutines, and their subroutines and so on - requires a CIA list in order to keep track of the sequence of control. If we wish to have a number of independent program structures, we must have a CIA list for each. L_3 is the fixed register which holds the name of the current CIA list. The name of the CIA list for the program structure which is to be reactivated on completion or interruption of the current program structure is the second item on the L_3 list, and so on. Therefore, the L_3 list is appropriately called the current CIA list list. The Save and Delete operations are used to manipulate L_3 analogously to their use with L_2 described above.

Table III gives a more complete schemat of the interpretation cycle. It has still been necessary to represent only selected b operations.

DATA PROGRAMS

In the section on list operations we described a search of a list. There the data were passive; the processing program dictated just what steps were taken in covering the list. Let us consider a similar situation, shown in Figure 8, where we have a working cell, L_{100}, which contains the name of a list, L_{300}. L_{300} is a data program. There is a program that wants to process the data of L_{300}, which is a sequence of symbols. This program knows L_{100}. To obtain the first symbol of data, it does $(6,1,L_{100})$ – that is, "execute the parallel program whose name is in L_{100}." The result is to create a CIA list, L_{500}, put its name in L_{100}, and fire the program. Some sort of processing will occur, as indicated by the blank words of L_{300}. Presumably this has something to do with determining what the data are, although it might be some bookkeeping on L_{300}'s experience as a data file. Eventually L_{700} is reached, which contains $(0,1,L_{800})$. This operation stops the interpretation, and returns control to the original processing program. The first symbol of data is defined to be $1L_{800}$. The processing program can designate this by $^4L_{100}$, since the sequence of $\underline{c}=4$ prefixes in L_{100} and L_{500} pass along the interpretation until it ultimately becomes $1L_{800}$. Now the processing program can do whatever it likes with the data. It remains completely oblivious of the processing and structure that were involved in determining what was the first symbol of data. Similarly, although it is not shown, the processing program is able to get the second symbol of data at any time simply by doing a "continue parallel program $1L_{100}$" ($\underline{b}=7$).

One virtue of the use of data programs is the solution it offers for "interpolated" lists. In working on a chess program, for example, one has various lists of men: pawns, pieces, pieces that can move more than one square, rooks, queens, and so on. One would like a list of all men. There already exists a list of all pieces and a list of all pawns. We would like to "compose" these lists into a single long list. However, we do not wish to lose the identity of either of the short lists, since they are still used separately. We would like to form a list whose elements are the two lists, but such that, when we search this list of lists, it looks like a single long list. Further – and this is the necessary condition for doing this successfully – we cannot afford to make the program that uses this "list of lists" know the structure. The operation Execute s (b=1) is precisely the operation needed to accomplish this task in a data program. It says "turn aside and go down the sublist s. Since it does not have the operation b=0, it is not "data." It is simply "punctuation" that describes the structure of the data list, and allows the appropriate symbols to be designated. Figure 9 shows a data list of the kind we have just described. We have taken the liberty of writing in the names of the chessmen.

The stretch of code below shows the use of a data program for a "table look up" operation. The table has arbitrary arguments, each of which has a symbol for its value. We have used A_1, A_2, and so on to represent the arguments. To find the value corresponding to argument A_5, for example, we put A_5 in the communication cell with $(14,0,A_5)$. Then we execute the data program with $(6,0,L_{100})$. Control now lies with the table, which tests each argument against the symbol in the communication lists – i.e., A_5 – and sets the signal accordingly. The program stops interpreting (b=8) at the word holding the value only if the arguments are the same. In this case it would stop, designating L_{350}. If no entry was found, of course, control would return to the inquiring program with the signal off.

LOCATION	SYMBOL	LINK
L_{100}	$20,0,A_1$	
	$8,0,L_{300}$	
	$20,0,A_2$	
	$8,0,L_{320}$	
	$20,0,A_5$	
	$8,0,L_{350}$	t

CONCLUSION

The purpose of this paper has been to outline a command structure for complex information processing, following some of the concepts we have been using in a series of interpretive languages, called IPL's. The ultimate test of a command structure is the problems it allows one to solve that would not have been solved if the coding language were not available. At least two different factors operate to keep problems from being solved on computers: the difficulty of specification, and the effort required to do the processing. The primary features of this command structure have been aimed at the specification problem. We have tried to specify the language requirements for complex coding, and then see what hardware organization allowed their mechanization. All the features of delegation, indirect referencing, and breakout imply a good deal of interpretation for each machine instruction. Similarly, the parallel program structure requires additional processing to set up CIA lists, and when a data symbol is designated, there is delegated interpreting through several words, each of which exacts its toll of machine time. If one were solely concerned with machine efficiency, one would require the programmer to so plan and arrange his program that direct and uniform processes would suffice. Considering the size of current computers and their continued rate of growth toward megaword memories and micro-second operations, we believe that the limitation already lies with the programmer with his limited capacity to conceive and plan complicated programs. We certainly know this to be true of our own efforts to program theorem proving programs and chess playing programs, where the IPL languages — or their equivalent in flexibility and power — have been a necessary tool.

Considering the amount of interpretation, and the fact that interpretation uses the same operations as are available to the programmer, e.g., the save and delete operations — one can think of alternative ways to realize an IPL computer. At one extreme are interpretive routines on current computers, — the method we have been using. This is costless of hardware, but expensive in computing time. One could also add special operations to a standard repertoire to facilitate an interpretive version of the language. Probably much more fruitful is the addition of a small amount of very fast storage to speed up the interpreter. Finally, one could wire in the programs for the operations to get more speed yet. It is not clear that there is any arrangement more direct than the wired in program because of the need of the interpreter to use the whole capability of its own operation code.

REFERENCES

1. Bernstein, A., A Chess Playing Program for the IBM 704, Proceedings of the 1958 Western Joint Computer Conference, May 1958.

2. Brooks, F. P., An Experiment in Musical Composition, Institute of Radio Engineers Transactions on Electrical Computers, vol. EC-6, No.3, September 1957.

3. Kister, J., Experiments in Chess, Journal Association for Computing Machinery, 4, 2, April 1957.

4. Newell, A., J. C. Shaw, Programming the Logic Theory Machine, Proceedings of the Western Joint Computer Conference, IRE, February 1957.

5. Newell, A., J. C. Shaw, H. A. Simon, Empirical Explorations of the Logic Theory Machine, Proceedings of the Western Joint Computer Conference, IRE, February 1957.

6. Newell, J. C. Shaw, H. A. Simon, The Elements of a Theory of Human Problem Solving, Psychology Review, 65, March 1958.

7. Newell, A., H. A. Simon, The Logic Theory Machine, Transactions on Information Theory, vol. IT-2, No.3, Sept September 1956.

Table I   Table of c Operations (Designation Operations).

c	Nature of operation for (a)=b c d e
0	(a) is the symbol s.
1	d is the address of the symbol s.
2	d is the address of the address of the symbol s.
3	d is the address of the address of the address of the symbol s.
4	d is the address of the designating instruction that determines s.
5	d is the address (name) of a process that determines s.

Table II Table of b Operations.

b	Nature of Operation
Sequence-control operations	
0	Stop interpreting; return to previous program structure.
1	Execute process named s.
2	Interpret instruction s.
3	Transfer control to location s.
4	Transfer control to location s, if signal is on.
5	Transfer control to location s, if signal is off.
6	Execute parallel program s; turn signal on if stops; off if not.
7	Continue parallel program s; turn signal on if stops; off if not.
8	Stop interpreting, if signal is on.
9	Stop interpreting, if signal is off.
10	Terminate.
11	Halt; proceed on go.
Save and delete operations	
12	Save s.
13	Delete s (and everything for which s is responsible).
Communication list operations	
14	Copy s into communication list, saving $1L_0$.
15	Move s into communication list, saving $1L_0$.
16	Move $1L_0$ into location of s, saving s.
17	Move $1L_0$ into location of s, destroying s.
18	Copy location of s into communication list, saving $1L_0$.

19 Create a new symbol in location of s, saving s.

Signalling operations

20 Turn signal on if s=$1L_0$, off if not.

21 Turn signal on if s=$1L_0$, off if not; delete $1L_0$.

22 Turn signal on if s is responsible, off if not.

23 Turn signal on.

24 Turn signal off.

25 Invert signal.

26 Copy signal into location of s.

27 Copy signal into location of s, saving s.

28 Set signal according to s.

29 Set signal according to s; delete s.

List Operations

30 Replace s by the symbol designated by s, and turn signal on; if symbol doesn't exist (b=10), leave s and turn signal off.

31 Replace s by the symbol in d of s and turn signal on; if symbol doesn't exist, leave s and turn signal off.

32 Replace s by the location of the next symbol after d of s and turn signal on (s replaced by "0,4, (f, part of d of s)"); if next symbol does not exist, leave s and turn signal off.

33 Insert $1L_0$ before s (move symbol from communication list).

34 Insert $1L_0$ after s (move symbol from communication list).

Table III   The Interpretation Cycle

1. Fetch the current instruction according to the current instruction address (CIA) of the currrent CIA list.

2. Decode and execute the *c* operation:

    If *c*=3 replace *d* by *d* part of the word at address *d*, reduce *c* to *c*=2 and continue.

    If *c*=2 replace *d* by *d* part of the word at address *d*, reduce *c* to *c*=1 and continue.

    If *c*=1 put *d* in the Address Register and go to step 3.

    If *c*=0 put CIA in the Address Register and go to step 3.

    If *c*=4 replace *c*, *d* by the *c*, *d* parts of the word at address *d* and go to step 2.

    If *c*=5 mark CIA "incomplete," save it, set a new CIA=*d*, and go to step 1.

3. Decode and execute the *b* operation:

    (Some of the b operations which affect the interpretation cycle follow.)

    If *b*=0 turn the signal on, delete CIA and go to step 4.

    If *b*=1 save CIA, set a new CIA=*d* part of *s* and go to step 1.

    If *b*=2 replace *b*, *c*, *d* by *s* and go to step 2.

    If *b*=3 replace CIA by the *d* part of *s* and go to step 1.

    If *b*=10 delete CIA.

    If no CIA "pops up" turn signal off, delete CIA and go to step 4.

    If "popped up" CIA is marked "incomplete" fetch the current instruction again, move $1L_0$ into Address Register and go to step 3.

    Otherwise go to step 4.

4. Replace CIA by the f part of the current instruction and go to step 1.

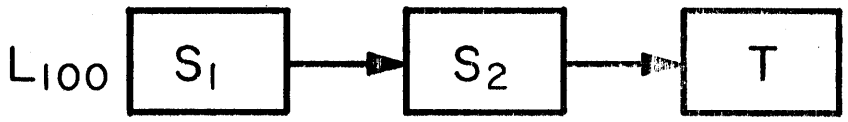


Fig. 1 — A simple list

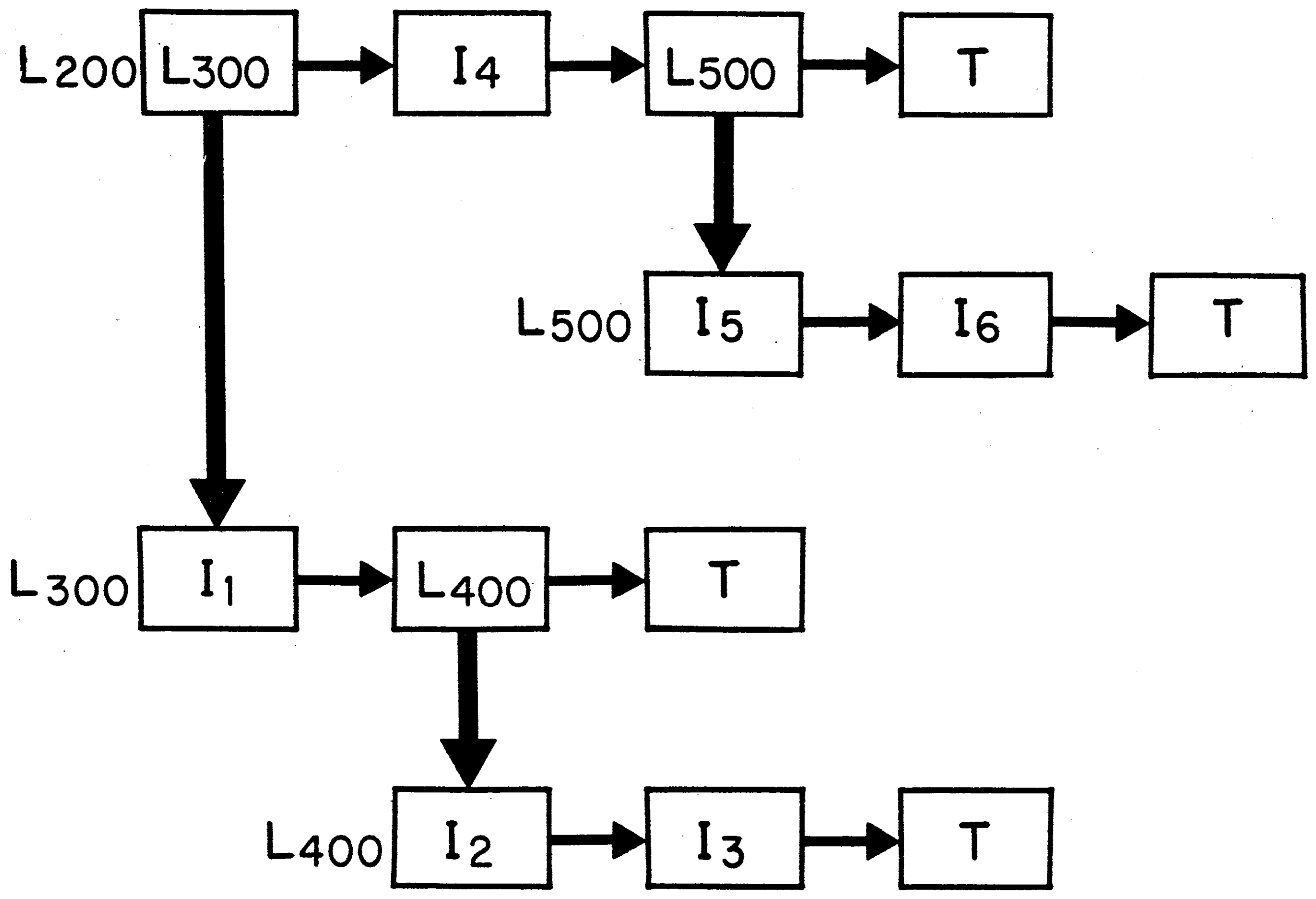


Fig. 2 — A list structure

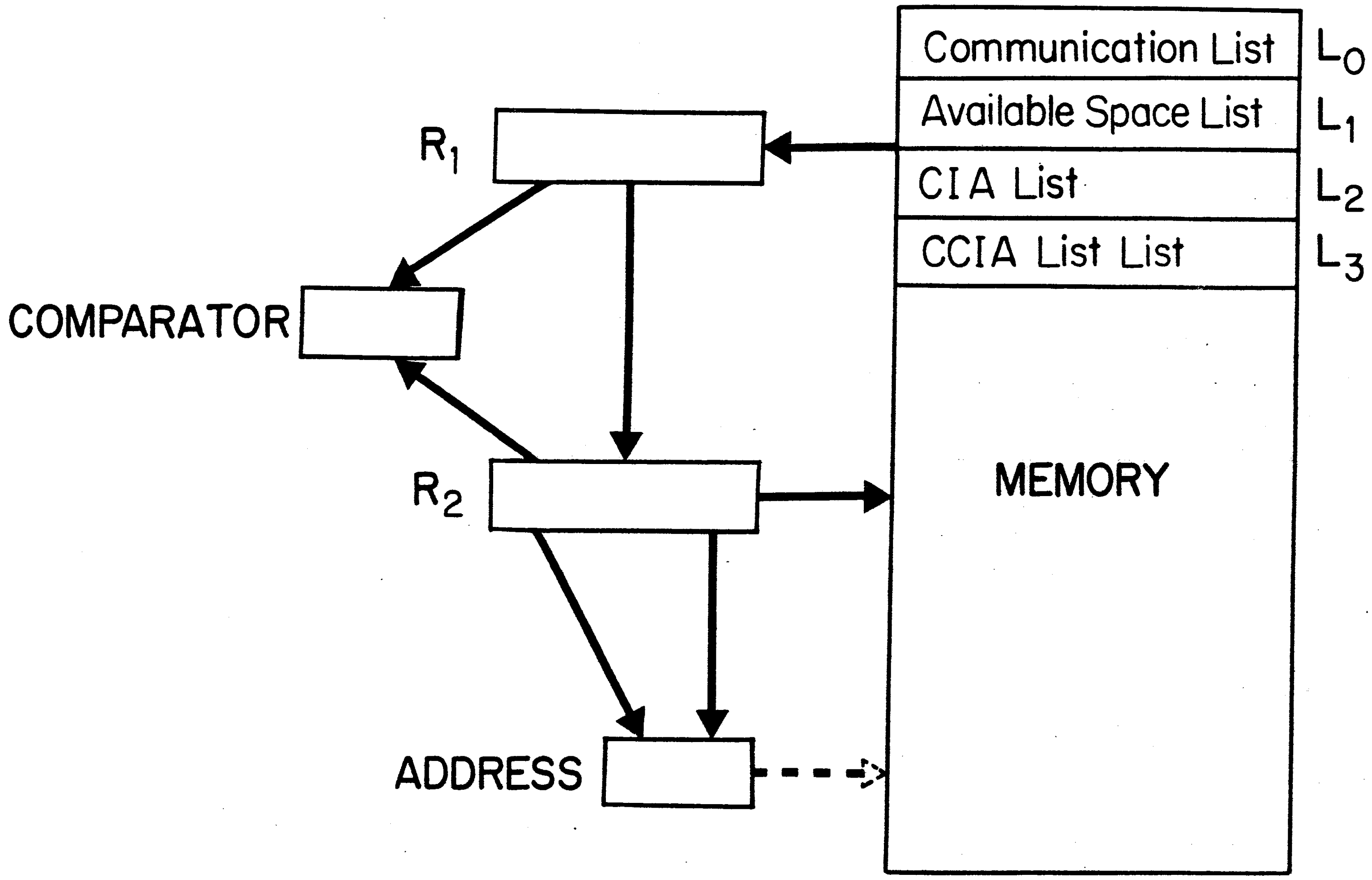
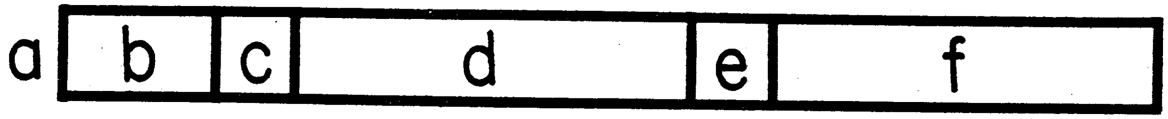


Fig. 3 — Machine information transfer paths



a: Location of word
b: Operation code
c: Designation code
d: Address field
e: Responsibility code
f: Link to next word

Fig. 4 — IPL word format

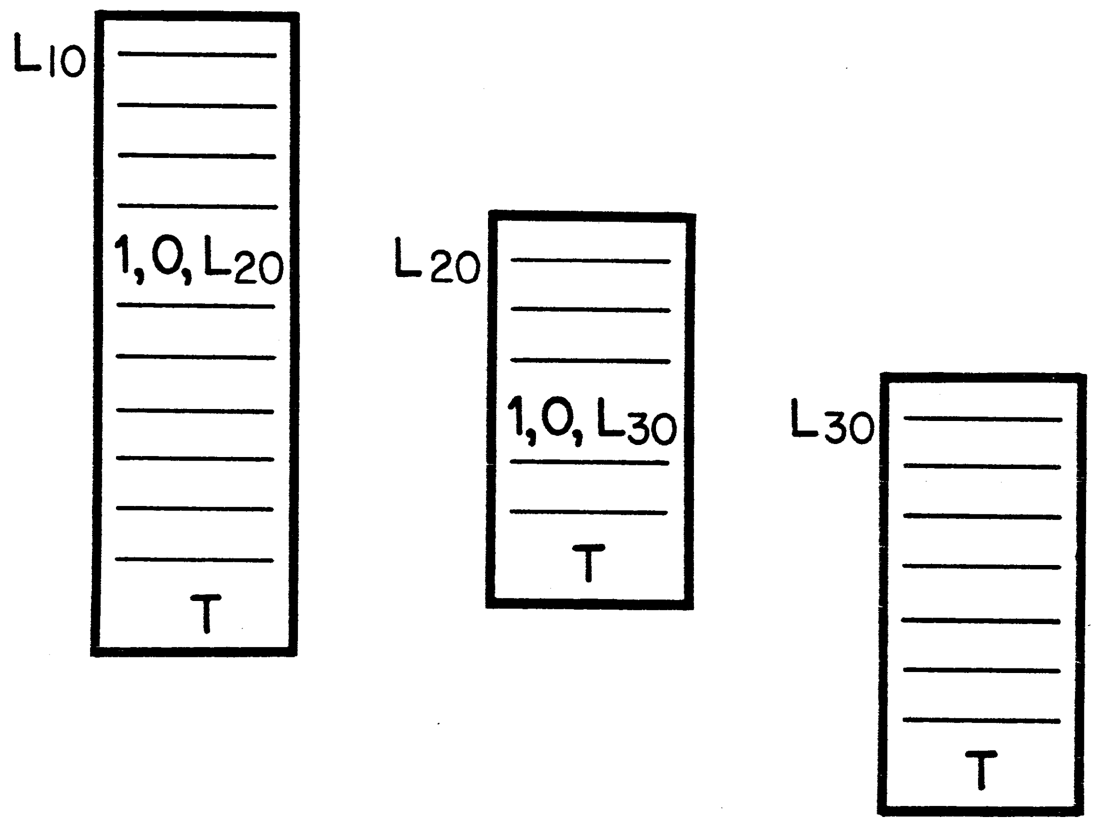
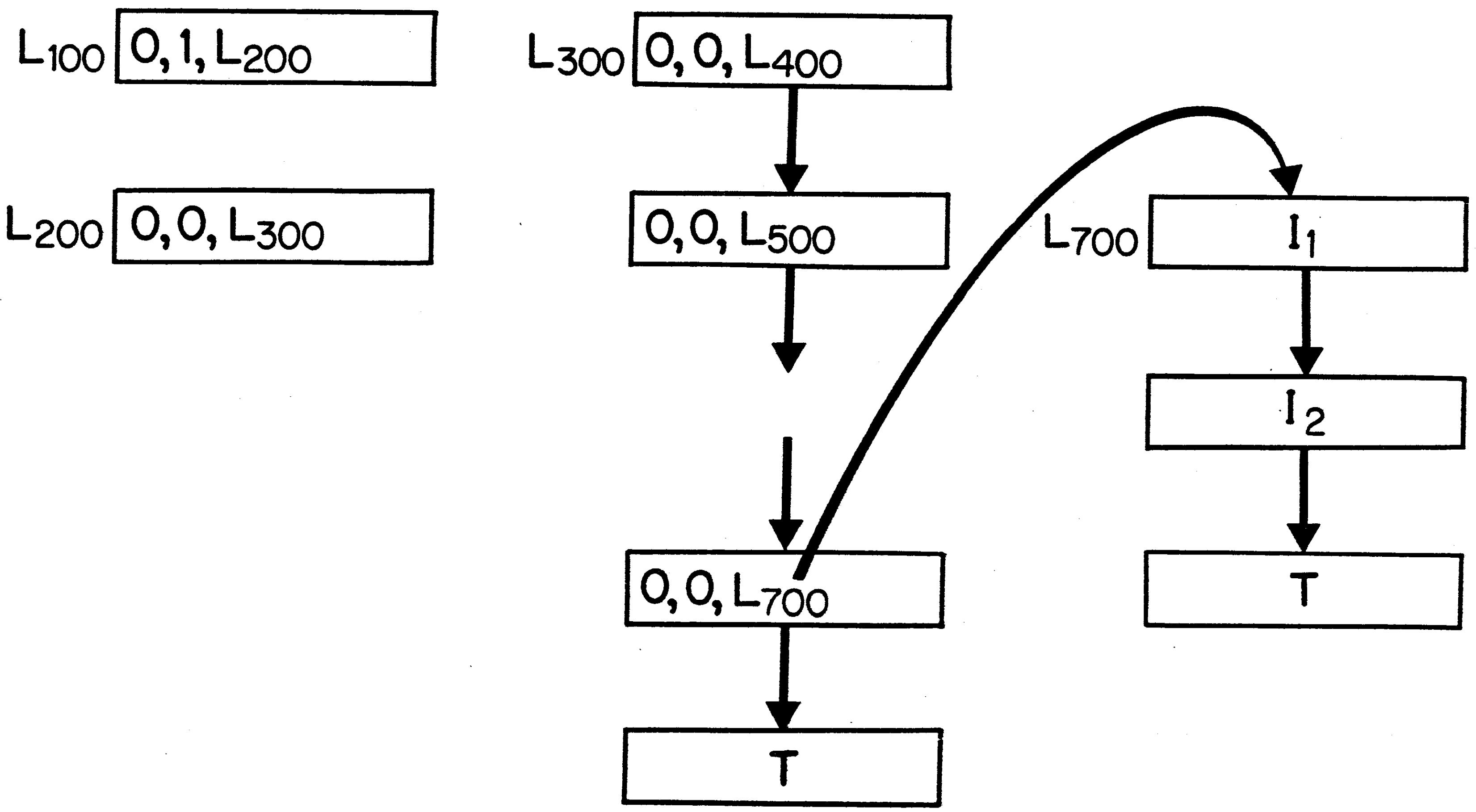


Fig. 5—A simple subroutine hierarchy



L_{100} 0, 1, L_{200}
L_{200} 0, 0, L_{300}
L_{300} 0, 0, L_{400}
0, 0, L_{500}
0, 0, L_{700}
T
L_{700} I_1
I_2
T

Fig. 6—Example of finding last item of last sublist

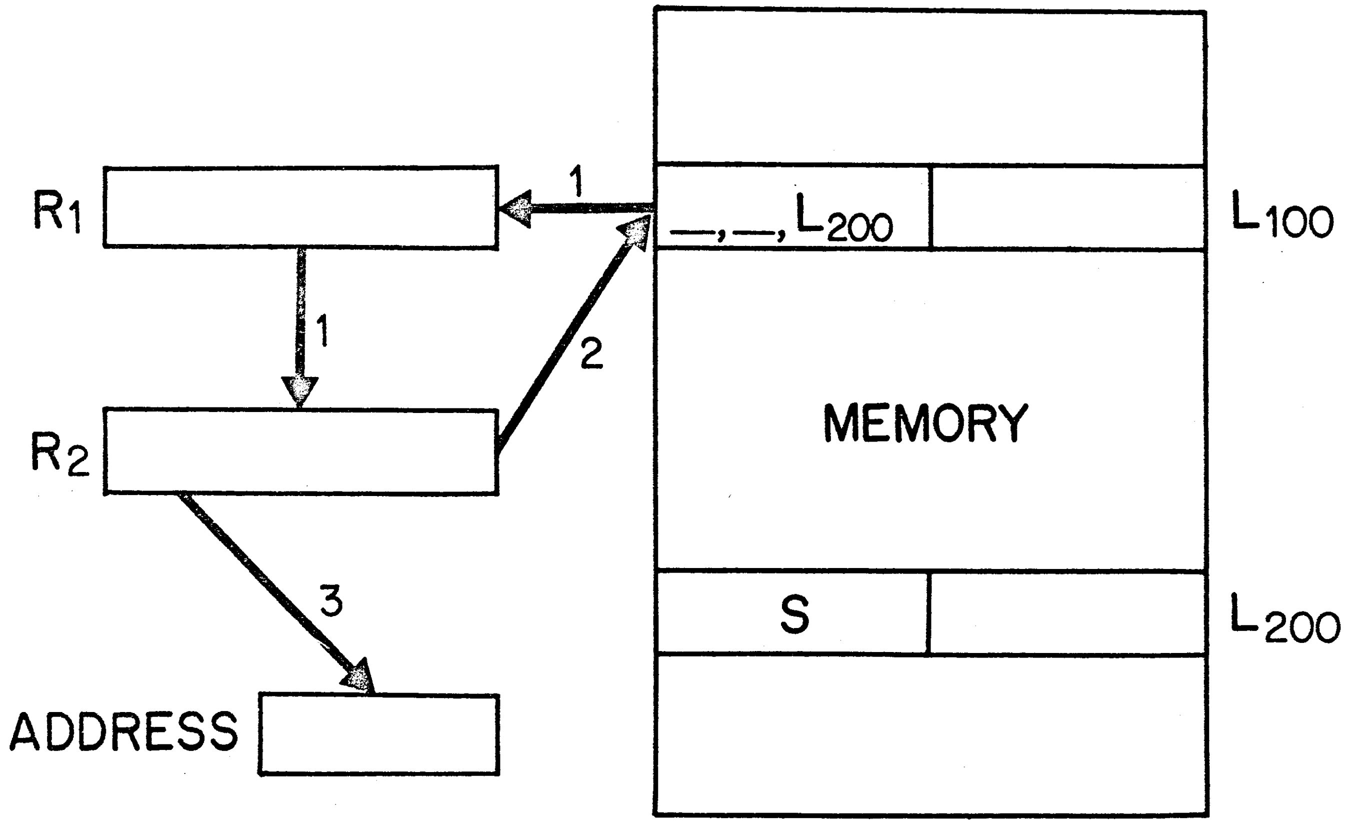


Fig. 7 — Information transfers in C-2 operation

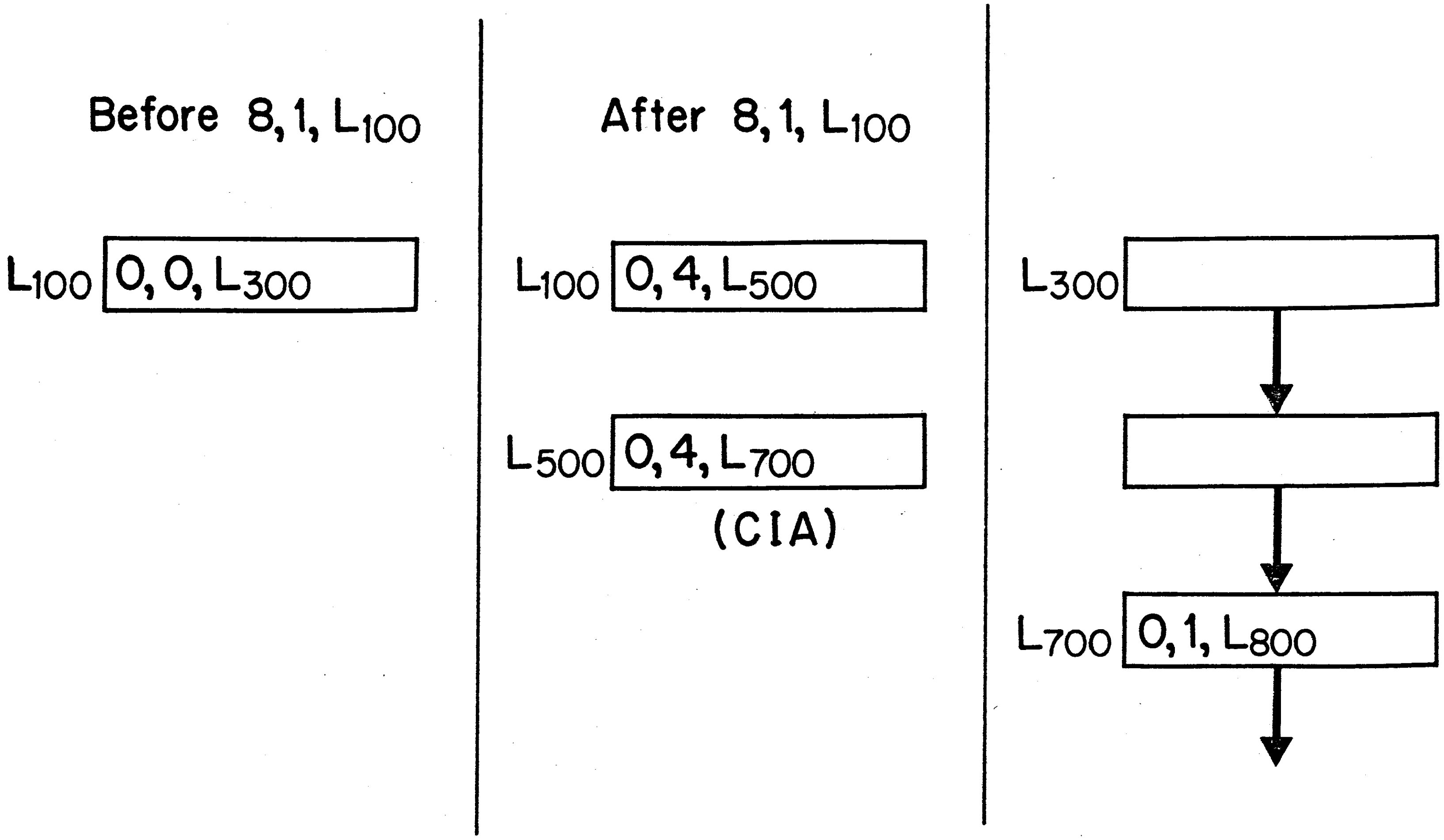


Fig. 8 — Example of a data program

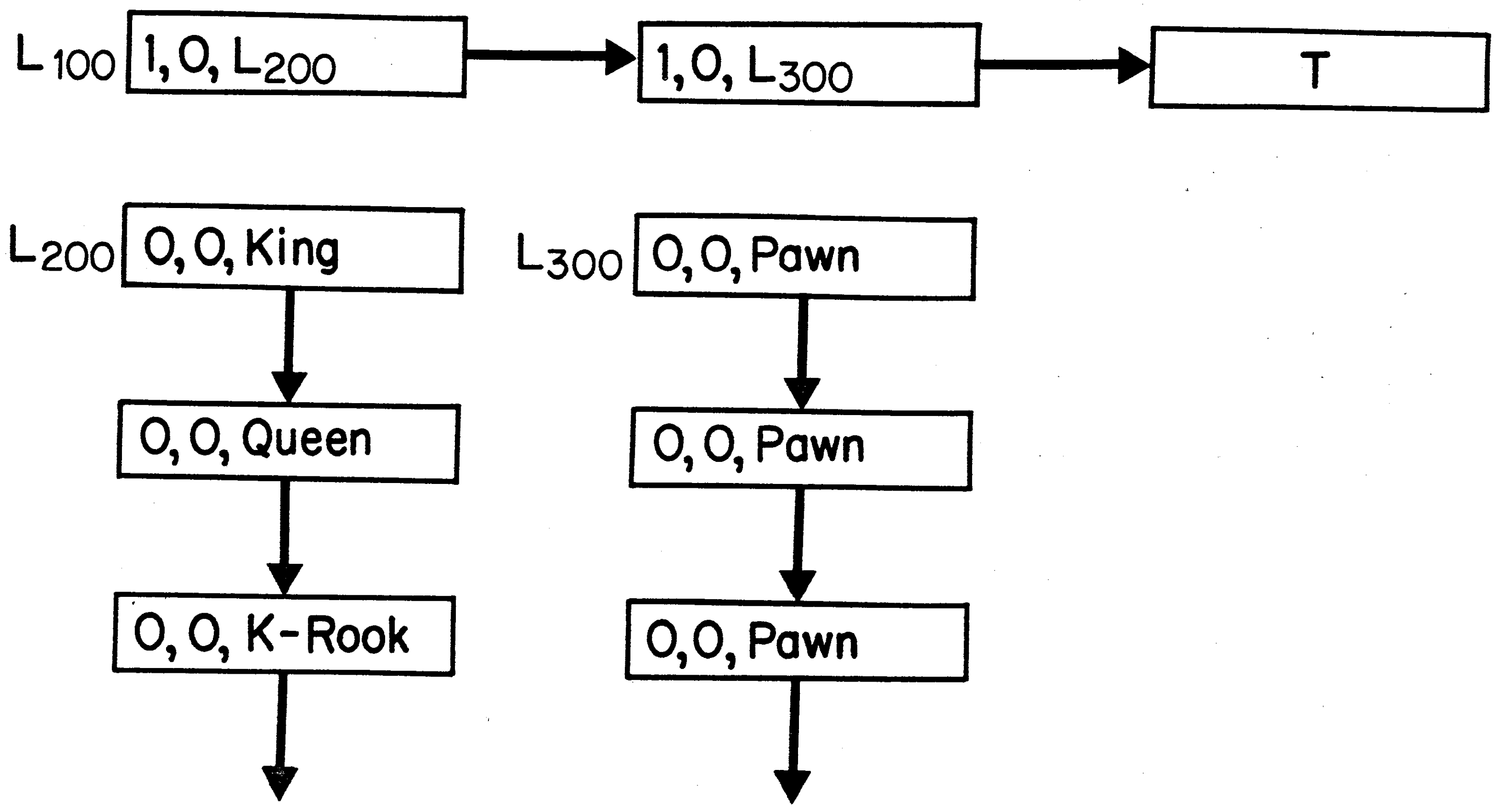


Fig. 9 — Application of a data program to chess