P-1906

ON PROGRAMMING A HIGHLY PARALLEL MACHINE TO BE AN INTELLIGENT TECHNICIAN

Allen Newell*
The RAND Corporation
Santa Monica, California

Summary

This paper speculates on how to program a machine that is suitable for microelectronic components to be an intelligent technician. The point of departure is from a class of machines described by J. H. Holland in a concurrent paper entitled, "On Iterative Circuit Computers Constructed of Microelectronic Components and Systems." These machines consist of a regular lattice of active modules, each possessing both processing and memory functions. The goal is a machine with the problem solving capabilities of a smart human technical assistant, and the volume processing capabilities normally associated with digital computers. This goal is chosen because it coincides with many current developments. After discussing the eventual capabilities desired and the most striking features of Holland's machines, the speculation proceeds by considering the basic organization for information processing. This is followed by briefer treatments of the organization for problem solving, supervision, interpretation and production.

Introduction

This is a speculative paper. I have been invited to provide a link in a chain of reasoning stretching from the impending advances in microelectronic components to their social consequences. I take as given a class of machines postulated by J. H. Holland in the preceding paper[1], which I will henceforth call Holland machines. A Holland machine is very different from any current computer, reflecting both potential virtues and vices of microelectronic techniques. On balance, of course, the great increase in speed of action and number of components guarantees a great increase in processing potential. I am to show how this potential can be realized.

* All my thinking on intelligent processes and computer languages has occurred in the context of joint research with J. C. Shaw and H. A. Simon. For the speculative application of these ideas to Holland's machines, I alone must take responsibility, of course. I am indebted to J. H. Holland. He not only provided the basic stuff out of which to fantasize, but he helped me to clarify the underlying concepts and suggested implications of the modes of organization I was trying to achieve.

A Holland machine is so strange on first contact that the problem seems to be to regain the programming facility attained on current machines in a decade of effort. But magnitude increases in basic capacity should yield equivalent increases in delivered power. The prospect of working hard to reestablish what already exists is not inviting. Hence, my task includes providing some vision commensurate with the new capacities.

The givens and the desired are yet further apart. A Holland machine is an abstract automaton, and I am free-that is, forced--to specify the exact version that will suit my needs. The boundary between the logical designer and the programmer has shifted so that by any normal job description I play both roles. The difference lies in the components available to me. I have not AND's and OR's, but "semicomputers," out of which to construct a machine.

The Vision

An Intelligent Technician

Of the myriad things that will occur with the continued development of information processing machines, I will select just one for consideration--a machine that is an intelligent processor of information. There is nothing very dramatic about such a machine, except perhaps the results that are achievable with it. It is difficult to differentiate this machine from an extremely compliant, fairly bright, human technical assistant, backed up by an impressive computing establishment. The user will converse with such a machine about his problem with the freedom of ordinary technical discourse. The language may not be English, but it might as well be for the freedom and flexibility it will afford. The machine will return answers with a rather breath-taking rapidity, and it is difficult to predict how rapidly a human-plus-machine can advance into a problem. Like all machines (and humans) it will have finite capacities. Ordinary intelligence could pose useful problems that require processing that the machine can do in a reasonable time. Ordinary muddleheadedness could so confuse the machine that it would not know what is desired. Ordinary knavery could fool it into doing the wrong thing. In short, the total product will depend on the joint intelligence and processing capacity of the user-plus-machine. A smart, articulate, well prepared man will produce more with the machine than a dull, ambiguous, unprepared man.

All our lives are spent learning to live with other intelligences, and relating to an intelligent machine will seem more familiar than strange. Since we want machines to help us solve problems, the more intelligent we are able to make it, the more unobtrusive it should be in providing this help. Contemplating the more extreme forms of this vision, there is little to describe about the machine except that it gives a great deal of help quickly and with very little pain. Not even this picture of productive harmony is safe, since we will soon accept all the benefits that flow from the new capability and will become aware of the constraints and annoyances that mark the new boundary of limitations. Joining a reasonable level of intelligence with the volume processing characteristics we already associate with the large computers will produce large and dramatic consequences. But these require detailed consideration for each subject matter, a task exemplified by the last paper in the chain by C. W. Churchman.[2]

Why This Vision?

I select this particular vision because I believe it lies along the main path of computer development quite independent of microelectronic advances.

The history of computers is marked by a sequence of innovations, each eliminating some factor limiting the rate of information processing. Most of these limitations were the time it took humans to make decisions or take action. The addition of memory and control to a fast arithmetic device bypassed the human decision time in doing simple numerical procedures. This innovation produced the computer essentially in its present form. The excessive time that it takes for humans to code has given rise to the assemblers, algebraic compilers, list languages, and the like. The wastage from a half-second human tending a microsecond machine is forcing the development of supervisory routines and interrupt systems that will eventually eliminate the operator.

The intelligent technician is simply another step in this development. The time for human invention and accurate specification of memory organization and elementary procedures in most programs is already too long. It will become intolerable with Holland machines made from microelectronic components. We need a "problem-oriented machine" instead of a "machine-oriented machine"—and such a machine, truly conceived, is equivalent to an intelligent assistant.

I do not consider the vision radical. Indeed, the programming and computing world is already on its way to achieving it.

Fundamentals

The Requirements of Intelligent Action

When applied to humans, intelligent action means action that achieves desired ends. There is no reason to modify this usage with respect to machines. An intelligent technician must be able to achieve desired ends. As normally used the term is ambiguous, since the resources given to the intelligence are not stated. This ambiguity is essential, since high intelligence in the real world means the ability to surmount the "givens" if they get in the way of problem solution. It means the ability to achieve desired ends even when defined with surprisingly little information. Intelligence is our word to indicate the fact of successful performance against theoretically inadequate conditions. Intelligence is the resolution of ambiguity by means of an adequate theory.

The notions expressed above trace capricious paths through the concept of intelligence. They can be summarized in four requirements for the behavior of the intelligent technician.

Indefinite Resources. The intelligent technician must be able to solve problems. More importantly, he must be able to work on any problem that can be stated to him. No limit can exist to the resources available for working on a given problem. These need not be all the methods available nor very good ones, but the machine should not quit because it has no more things to try. An intelligent human working on some hard differential equations will try to solve them analytically; then will turn to power series; then to a book on trigonometric series so he can apply these; then to a treatise on differential equations for some new clue (finding this via a reference in his standard text on the subject); then to numerical procedures. On and on it goes. If there is any fixed limit—any sharp restriction to a special class of methods—we will recognize soon enough that the intelligent technician cannot be left on his own, that it is really not intelligent enough.

Indefinite Awareness. A distressing feature of current computers is their instability in the face of trivial errors, a feature often cited in making comparisons between brains and computers. To be intelligent is not to be trapped easily—not to require external help for little things. The machine itself must be aware of its own behavior and of the context in which its work is done. No sharp boundary can exist for which particular features the machine is aware , even though it cannot be aware of everything. This would soon reveal itself as a "quirk" or "blind spot" that limits the intelligence.

Indefinite Language. The machine's capabilities are mirrored accurately in the external language used between it and the human user. For example, if the language is limited to procedures, then it is not possible to ask the machine to solve problems, such as "Find x such that sin x - x = .5." The model of language required might be labeled the "Hide and Seek" model. Any information the transmitter can hide in the expressions being communicated, the receiver can find, providing the rules of hiding are given in the language. Any restriction to a fixed vocabulary, a fixed grammar, or a fixed technical area will be viewed immediately as a limitation in intelligence.

Indefinite Accumulation. Anything that is reasonably intelligent learns from what it does. Indeed, problem solving involves learning in many ways—e.g., learning about trigonometric series in the example above. Part of the stream of facts, procedures, clues, theories, and so on that pass through an intelligent machine will be accumulated for later use. Again, no definite boundaries can exist on what is selected or how it is filed and indexed. Limitations of this sort will reveal themselves as inappropriate repetitions, and will immediately be seen as a deficiency in intelligence.

Perhaps I am grooming the machine to be a genius, rather than a mere technician. I do not think so. Educated men possess these characteristics to a remarkable degree. Limits will exist, or the machine would be smarter than most men. But these limits will be indefinite and shifting, and will express themselves in a global measure of the machine's intelligence.

It may seem that all the emphasis is on intelligence and none on the volume processing capacities that are also needed. On the contrary, the machine is for high volume work. The intelligence forms the connective tissue that allows the machine to rapidly organize itself to be highly repetitive and efficient.

General Layout of the Machine

Although I cannot provide a complete picture, I will give some considerations about an organization for such a machine. These fall under the headings of information processing, problem solving, supervision, and interpretation and production. These headings are machine oriented and are not coordinate with the requirements of intelligent action. The general layout is shown in Figure 1.

An External Language is used between the machine and those things with which it communicates. This language is completely independent of the internal structure of the machine. Only incidentally will it refer to things and properties inside the machine. The expressions in the external language are taken in bodily and made available in the Input-output area.

There is a single Processing area, structurally homogeneous, but divided into two functional areas: the Interpreter and the Factory. The Interpreter produces the processes indicated by the External Language expressions in the Input-output area. These action processes are constructed in the Factory, and constitute the activities that the machine does in response to the external world.

The processes in the Factory are in the Internal Language, as are the interpretation processes in the Interpreter. In Holland machines it is difficult to distinguish programs from processes—expressions that say what to do from structures that do it. A language expression written in a set of modules may be sufficient to convert these modules into a process that behaves according to the expression. Hence, "internal language," "program" and "process" are used interchangeably.

Besides the Input-output area, there are three stores for information. Each is structurally distinct because of different reading and writing requirements. The Program Store contains the large number of program forms required by the Interpreter and the Factory. The Associative Store contains entities with the properties normally associated with symbols. The Warehouse is a tertiary store that backs up all the other areas. It is a reminder that, even in a machine with 10^{11} components, access time must be traded for space in order to remember enough information.

Holland Machines

A Holland machine consists of a regular lattice of identical modules, as shown in Figure 2. Each module is directly connected with its immediate neighbors in the lattice. These direct physical connections can form paths between distant modules. A module's capacity for paths is limited. Once formed, the paths give immediate access independent of length. Each module has both memory and processing functions, and may be active simultaneously with and independently of other modules. The functions of a module are represented by settings of bit patterns, so that a module may take on any possible character by writing a new bit pattern into it. Operations change the state of a module as a function of the states of other modules connected to it by paths. These include operations for reading and writing information into modules and for building up paths. The machine is basically synchronous.

Within the bounds indicated almost any kind of a system is possible. There is freedom to specify the number of immediate neighbors, the kinds of information held by each module, the operations performed by each module, and the sizes of the paths. For a given amount of basic componentry the more complex the module the more components it will take per module and the fewer modules that will be available. The original papers should be consulted for more detail.

The following features summarize most of the advantages and problems of Holland machines:

Indefinite Parallelism. The most striking feature of Holland machines is their active nature. Many processes can run in parallel, each of arbitrary composition. Our experience is almost entirely with completely serial machines. Machines with a small number of independent processors are just coming into being. Although a telephone system is like a big parallel computer in many ways, it accomplishes unsophisticated functions compared to those needed for general computation. There is little work in the literature on programming systems of this sort. Besides Holland's papers, work by Unger[3] and Selfridge[4] is relevant, although they both focus on pattern recognition.

Industrial organizations are examples of highly parallel systems, and face many problems similar to this one. We can expect to be involved in the same crucial issues of centralization-decentralization, coordination, and division of labor. We can also expect the growing rationalization of management, typified by operations research, to be a prime source of clues for programming these systems.

Local Action. The modules of Holland machines work by local contact along paths. Information is designated by pointing to it, rather than by symbols that refer to it. A major gain recently made in programming was finally to create an entity that behaves in many ways like a linguistic symbol (the address with a list of associated information). No simple correspondent to this exists in Holland machines.

Iterative Structure. The modules in a Holland machine are identical, although different Holland machines can be used for the gross areas of Figure 1. This iterative character, besides being suitable microelectronic production technique, provides a "space" already rich in possibilities in which structures can be created at will by information transfer operations rather than actual construction operations (in the carpenter's sense of the word). However, this implies that most modules will have only a small fraction of their componentry utilized.

Fixed Network of Connections. The multidimensional connections between modules seem at first glance to be a blessing. The virtue fades as soon as arbitrary structures must be processed. The information is invariably incommensurate with the fixed network. "Dead ends," in which all the paths to a module are occupied and no way exists to gain access to the module, are continually a problem. Simple data organizations must be used, even though they prohibit clever ways of encoding particular information.

Information Processing

The first level of organization of the machine is the information processing level. It contains means for building up structures of information and for forming processes to operate on these structures. It involves specifying the Holland machine for the Processing area (the Interpreter and the Factory) and indicating how the problems of memory and program organization will be solved. In the following I draw heavily on the programming experience with the problem solving programs described in the next section and on the list languages[5] constructed to aid in programming them. Properties of the Processing area will be accumulated; then a Holland machine for them will be sketched.

Units and a Principle of Homogeneity

In current machines and coding a large discontinuity exists between the machine level and all structures built up from this level. A subroutine is not like an instruction; a double precision number is not like a number contained in one word. The temptation to create Holland machines similarly is strong--to create neat, powerful, elementary operations for modules and then to build up everything from these in structures that look very different from modules. There would be a preferred size of channel (the one between modules), a preferred size of information (the word in the module), and a preferred set of operations (the operation code of the module).

Contrariwise, the Processing area satisfies a Principle of Homogeneity, according to which a module and a higher unit cannot be distinguished by any of the conventions for dealing with them. The basic structure of the Processing area is the following:

1. The Processing area consists of a stack of planar fields, each field containing spatial units. These are rectangular in shape and vary in size. Their boundaries do not cross each other, but units may exist inside other units.

2. The units are connected by paths. These run in horizontal and vertical segments and may be of any width. Two paths may cross each other at right angles, but otherwise two paths may not occupy the same space.

3. Each unit has registers that hold information in bit patterns. There can be units with any number of registers, and registers with any number of bits.

4. Each unit can accomplish an information process involving reception from some paths, transmission along others, and changes in internal registers.

These properties allow units to be built up from networks of other units in order to accomplish more complex processes. Similarly, units may be analysable into subunits connected by paths. If analysis proceeds far enough, units are reached that are not further analysable. These may be either modules or aggregates of modules that, for reasons of speed and space, have been arranged internally in ways that do not correspond to all the conventions of units.

It is now unimportant where the modular level is, and what operations and registers modules contain. I am now free to specify flow diagrams of units with arbitrary functions connected by paths of arbitrary information content. No assumptions need be made about the module operations, except completeness. For the machine, the problem of organizing itself has been much simplified.

Spatial Operations

The machine needs ways to assemble and manipulate units and paths. These should also be homogeneous and should not require extensive knowledge about the distribution of units in the field. The following additional properties seem appropriate:

5. Units can be moved in any of the six directions (vertical movement between layers is needed). The entire unit moves at once with a velocity of one module per basic cycle (variable or faster speeds are complicated).

6. Movement takes place freely into space not occupied by paths or other units. A moving unit coming into contact with a stationary unit or path sets the new unit in motion in the same direction. Two colliding units stop. A moving unit stops upon contact with the inside boundary of a unit.

7. Paths remain connected to moving units, growing and contracting as required.

8. Units can expand and contract. In expansion the same conventions hold as for a moving unit. It sets other units in motion and stops when it contacts the inside of a boundary. In contraction, the inward moving boundary sets the inside units in motion, and all motion comes to a halt when the subunits are jammed together.

9. Units can be copied. This always occurs in the vertical plane, as shown in Figure 3 (a direct horizontal copy is difficult). The figure shows a horizontal copy obtained by copying-up, moving-over, and moving-down.

These capabilities provide convenient ways to manipulate structures. Movement is controlled by always working inside a higher unit as a sort of corral. Rearrangements occur automatically, since all units tend to move out of the way.

Information Structures

The more complex the programs, the more irregular and dynamic are the structures of information built up of units and paths. The machine must be prepared for network-like affairs, as shown in Figure 4. The left side shows a simple tree; the right side indicates that matters are not always so simple. To put such structures (and their more elaborate cousins) into the rectangular grid of the Processing area, a standard outline form is used (Figure 5). Each unit is allocated a horizontal band. Subunits are indented to the right and put immediately below their superunits. The connecting paths use the space made available to the left. As the tree grows all the units are put in downward motion. Each unit can also grow independently to the right in its band. The right hand limit is given by the boundary of the superunit that contains the entire structure.

Splitting Operations. The simple outline is insufficient for general networks, as an attempt to add the paths for the right side of Figure 4 shows. Already paths can cross paths, but they must also be able to cross units. Splitting the unit is one solution:

10. A unit can be split into two parts, either vertically or horizontally, and these two parts may be moved indefinitely far apart. The space between the parts contains only paths. Elongations of the internal paths of the unit run across the split, so that the unit remains unchanged as far as information flows are concerned. Other paths can run along the split, and so traverse a unit.

The outline conventions and the splitting operation allow the machine to put any kind of information structure into the Processing area. This stylized form is often less compact than other forms, but it avoids solving many problems in memory organization.

Definitional Control

No sharp distinction exists between informational units and processing units. Most data is contained in units that actively transmit the data over paths, rather than waiting passively to be read. Both the Interpreter and the Factory are systems of many interconnected, simultaneously active units, and techniques must be provided for control and coordination.

Experience at the programming level is exclusively with sequential control. Sequencing information is given independent of the content of the processes, usually by the order of instructions. A form of sequential control could be adapted to the present machine. Instead, a more radical, fully asynchronous procedure is used. This is called definitional control, since it rests on the fact that procedures work from the defined towards the undefined. Although similar to existing asynchronous techniques, I am not familiar with any discussion of it for general procedures. However, some work by Steward[6] is relevant.

Any procedure can be put into the form of a hierarchy of subprocedures: The lowest procedures work on the input data; the next procedures work on the outputs from these; the next higher ones work on the outputs of these; and so on. Figure 7 shows the computation of Ke^{-2K} in this form, as it would occur in the Factory. Iterative, recursive, and conditional processes can be analysed analogously.

Assume that each process is sensitive to whether its inputs are available and to whether its output can be accepted. As long as K remains undefined no computation takes place. As soon as K becomes defined, say by some other process feeding a number into it, then the first multiplier can operate, since both of its inputs are defined. Once its computation is finished and it transmits the product, the exponential process can operate; once this is finished, the final multiplication operates. Finally the result is printed. Each process operates as soon as it can--that is, when the situation is defined for it. Until that time, it simply waits.

Suppose a process at the bottom was feeding numbers into K at a faster rate than the printer could operate. Until the printer accepts a result, the top multiplier does not accept the inputs from K and the exponential; the exponential does not accept the product from the lower multiplier; and it, in turn, does not accept the next value of K. The computation automatically becomes paced by the printer.

This form of control partially rationalizes the construction of procedures. No independent sequencing problem must be solved. A process that is put together correctly according to the flow of information is ready to go, and will sequence itself automatically. Since the machine constructs its own programs, this seems a desirable property.

To obtain definitional control, the following are needed:

11. Every register is either defined or undefined; this can be detected on any path leading into or out of the register. (This is simple enough unless there are multiple transmitters and receivers, in which case things get complicated.) No transmit operation will be executed into a defined register (it's already occupied), and no read operation will be executed from an undefined register.

12. Each unit is either active or inactive. If a unit becomes active all its subunits become simultaneously active also. If it is inactive it will not perform any processing (spatial operations excepted), and all of its registers will be undefined. If it is active, it will perform its processes as soon as all paths involved are admissible (inputs defined and output receivers undefined).

Modules and Mass Operations

I must at least indicate how the numerous asserted properties might be achieved in a Holland machine. The ideas seem independently interesting because they involve mass operations, which affect all modules in a region simultaneously and identically. Little experience exists with such operations[3]; but they seem worthwhile exploring.

Each module consists of two parts: a regular part and a substrate. The regular part involves operation codes and storage registers from which processing and memory functions will be synthesized. The substrate takes care of defining units with their spatial and activity properties. The regular part satisfies the homogeneity principle. The substrate achieves the homogeneity principle for aggregates of modules and has no counterpart at the unit level. The regular part need not be defined because of the homogeneity principle, but the substrate needs discussion.

The substrate consists of a state for each module, some operations for changing state, and some paths interconnecting modules. Every module is either:

1. occupied or unoccupied;
2. active or inactive;
3. repetitive or nonrepetitive;
4. stationary or in motion in one of six directions (four horizontal and two vertical);
5. Open or closed, independently, in each of the four horizontal directions to the transmission of influence.

All the modules in a plane are interconnected so that they must all change state simultaneously. If one module goes from inactive to active, all modules go to the active state. This network of influence, which extends from any module to all other modules, can be broken by modules that are closed. If some modules are surrounded by a boundary of modules, all closed with respect to transmission outward, then changes inside affect everything inside but affect nothing outside the module. A unit is a rectangular set of modules with a boundary that blocks influence from the inside out but transmits influence from the outside in. This last condition makes possible hierarchies of units. Anything that occurs in the larger unit affects the subunits (like moves, executes, stops, etc.), but not vice versa (a subunit can move about in the unit without making the unit move).

A module in motion copies its content into the next module in the given direction. This leaves the motion state unchanged, and motion continues until something turns one of the modules to stationary. As soon as this happens, all modules within the unit instantly become stationary and the unit stops moving.

A unit is made active by changing any of its modules from inactive to active. All the modules are simultaneously made active, and the unit immediately begins to "push" to execute its process, since those parts which can operate go into immediate operation. Two states of operation are needed, repetitive and non-repetitive. A repetitive module remains active even though it has just executed its operation; it immediately tries again. A non-repetitive module sets itself inactive as soon as the operation is accomplished. This change immediately affects the entire smallest unit that contains the module, but it does not affect the modules outside the unit. Incidentally, this shows that influence must depend on state changes and not states, since otherwise the existence of active modules outside would immediately reactivate a module that had just turned itself off.

Although this development is incomplete, the outlines are clear, including the power of the mass operations.

Symbols and the Associative Store

Both human language behavior and the programming of complex systems point to the need for a stock of symbols for general information processing. Symbols are entities with the following properties:*

1. It is always possible to obtain easily a new symbol, not otherwise being used (within the limits of the total stock).

2. It is possible to produce indefinitely many occurrences of a symbol (symbol tokens) and these can be moved around and placed in structures.

3. Given the occurrence of two symbols, it is possible to determine whether they are the occurrences of the same or different symbols.

4. A symbol is a locus of associations. This means the following operations can always be performed:

a. An associative link can be formed from a symbol S to a symbol S', and the link labeled with the symbol S". Only one link with a given name, S", exists for a given symbol, S.

* I reiterate that this formulation, and several others throughout the paper, come from joint efforts with J. C. Shaw and H. A. Simon.

b. The symbol associated to the symbol S by the link labeled S" can be found. This is the inverse operation to the one above, and forms the sole significance of the "associative link." No association with label S" needs to exist, but if it does it is unique.

c. The link labeled S" for the symbol S can be destroyed.

This set of properties ties the concept of a symbol to the concept of an associative memory. Likewise, these properties give one definition of an associative memory, which is equivalent to being able to form and distinguish arbitrary single-valued functions of one discrete variable. It implies that an adequate concept of symbol and reference can be formed from the concept of a locus of associations.* Without arguing any of these points, this formulation is taken as the requirement for giving a machine symbols.

Although the Processing area contains an elaborate system for processing information, it provides nothing that resembles a symbol. One unit designates another, not by having a symbol that names it, but by directly pointing to it. Although the bit patterns can be copied, transferred and compared, they do not allow associations. They are used entirely as objects and never refer to anything.

A separate Holland machine, the Associative Store, provides symbols. The same difficulties in trying to map a free structure into a rectangular grid arise here. To each symbol there corresponds a structure that encodes an indefinite number of associative links, so that the total store is a completely arbitrary, dynamically changing network.

Figure 8 shows a scheme for the Associative Store. The Holland machine for the Store consists of planes of modules. Each symbol has two adjacent lines of modules in the Store for its associations and an unique bit pattern for its token. Each association takes two modules, one in the lower line and one right above it in the upper line. The lower module is concerned with the symbol that labels the link, and the upper module is concerned with the symbol to which the link is made.

Each module has a single register that holds a token. The horizontal data paths, which transmit tokens, are open, so that all the modules in a line see the same token simultaneously. The horizontal command paths are

* This is an old notion with psychologists. However, this associative system is considerably more powerful than any proposed for organisms by psychologists. The essential difference is that here the links themselves can be labeled with symbols and hence can take associations.

directional to the right, so that a command issued by a module is seen simultaneously by all modules to the right but by none to the left. The vertical command paths connect each module to its immediate neighbor above.

Each module is either in the "defined" or "undefined" state. Each can record the token on the data path in the register; transmit the token in the register onto the data path; compare the token in the register with the token on the data path; and receive and transmit commands. Three specific patterns of these actions are required for the associative operations. (The 'link token' means the token of the symbol that labels the link; the 'symbol token' means the token of the symbol to which linkage is made.)

1. Form an associative link: The symbol token is put on the upper data path and the link token is put on the lower data path. Each link module in the defined state compares the link token with the one in its register, and if they are equal commands the symbol module above it to record the symbol token in its register. Each link module in the undefined state records the link token in its register, commands the symbol module above it to record the symbol token in its register, and sets itself to be defined. Further, all modules that took action command all modules to the right that have this link token registered to set themselves undefined. This order overrides any action to set a module defined. The net result is that one and only one module-pair records the association. Many modules may have the information in their registers, but only the leftmost one will be in the defined state. This assures that there cannot be two links with the same label.

2. Find an associated symbol. The link token is put on the lower data path. Each link module in the defined state compares this with the token in its register. If they are the same, it commands the symbol module above it to transmit the token in its register onto its data path as the symbol token. By the nature of the system either no module transmits or exactly one module transmits.

3. Destroy a link. The link token is put on the lower data path. Each link module in the defined state compares this with the token in its register. If they are the same, it sets itself to the undefined state.

The scheme is still incomplete in essential respects. Most important is the connection between the token, which is a bit pattern, and the two lines of the associative store that correspond to its symbol. The mechanism described simply delivers and remembers bit patterns in response to bit patterns transmitted on special data paths. Some way must exist for units in the processing area to take a token and construct a path for themselves to the data paths in the associative store. Once they connect they must be able to command the associative operations and receive bit patterns back along the path in return.

This problem is strikingly similar to that of a telephone system. Connections must be made from a set of subscribers, the units, that are widely scattered throughout the processing area. The connections are only temporary; once formed they persist for variable periods. Finally, only a few subscribers want access to the store at any one time. The token bit pattern, then, is a "telephone number" and the units "dial" the associative store.

Problem Solving

The organization of the machine at the information processing level is now clear enough to allow a brief consideration of higher levels of organization. Most important is an ability to problem solve.

Combinatorial Problems

Recent work on heuristic programs provides considerable information on the processes involved in problem solving. Programs have been written for tasks which are sufficiently complex and difficult to require intelligent action by humans. These tasks include theorem proving in elementary domains, chess and checker playing, musical composition, and some management science problems[7]. All these programs formulate the problem in a common way and attempt to solve the problem by a common approach.

The problems all involve a set of objects and a set of operators for producing new objects from old objects. The goal is to find a sequence of operators that produces an object with certain desirable properties, given some initial objects. For example, in chess the objects are positions; the operators are legal moves; and the goal is to produce a sequence of moves that will result in a winning position. In theorem proving, the objects are theorems; the operators are rules of inference; and the goal is to produce a sequence of inferences that will result in the theorem to be proved.

These problems are combinatorial in nature. They involve the selection of a sequence with certain properties out of the set of all possible sequences. The possible sequences can be represented as an expanding tree: given an initial object, application of the operators in all possible ways yields a set of new objects. Applying the operators in all possible ways to each of these objects yields another (much larger) set of objects two steps removed from the starting point. This can be repeated to generate the tree to any depth by taking all of the objects obtained at depth D and applying the operators in all ways to them to get the objects at depth D+1. If approximately B applications are possible for each object, there are B branches at each node, and of the order of B^D objects are generated at depth D. Thus the number of paths in the tree goes up exponentially with depth.

If the solution lies at depth D, then of the order of B^D paths must be examined to discover the solution. It is easy to show that immense numbers of paths occur for any real problem. For example, chess has a B of about 30 and a D of about 80, yielding about 10^{120} paths. Since the entire tree cannot be explored, it is necessary to be sophisticated in searching--to look only where the answer is. In hard problems no single item of information tells exactly where the desired object is, and many different bits of information, each of low calibre, must be used to restrict the search to a reasonable number of branches. These odd bits of information and the ways of organizing them are called heuristics. An example from geometry is "Don't try to prove two angles equal unless they appear about equal in a well drawn diagram." This heuristic reduces the average number of branches at each node by rejecting some that would otherwise have to be explored. The effect of heuristics is to reduce the parameters, B and D, in the search formula, B^D, leaving the form of the problem unchanged--search in an exponentially expanding space.

Problem solving by heuristic search is not the universal solvent that dissolves all particular methods. It is the last ditch defense when no special methods are known. The machine has many special methods, from analytic differentiation to Newton's method, and uses them wherever appropriate. It also needs some model for all the other problems that arise. Heuristic search is the best available.

Parallel Search

The problem is to organize the machine for problem solving. In current serial machines a single program, the problem solver, conducts the search. It carries with it an accumulation of information about the maze and its activities in it, and combines this with its more general heuristics to decide which branches to explore. If on the average it takes C time units per node, then the search time is of the order of CB^D.

Being parallel, the machine is not limited to a single problem solving process. With P problem solvers, the time of solution is of the order of $(C/P)B^D$. This speeds up the process by a factor of P, but does not affect the exponential, which governs the growth of the search tree. Coordination problems also exist, since no one problem solver accumulates all the information. Each must contain additional processes for transmitting its information and analysing the information received from the other problem solvers.

Parallelism can be pushed further. Each problem solver can have the task of not only creating the B branch points, but of making B copies of itself and of setting one copy to work independently on each branch. Now every branch point is active, and even if it takes an additional K time units to produce each new problem solver, the total search time to depth D is of the order of (C+BK)D. If this equation were only true, all the world's problems could be solved in a day! It claims that the search time, instead of being exponential with depth, is linear with depth. Thus, at a microsecond per chess position, it would take only a few milliseconds to compute the perfect game. The absurdity of this is manifest, but it is instructive to examine why it cannot hold.

Parallel computation trades space for time. The space required to hold the data of a problem tree also increases exponentially with time (although serial strategies exist that keep the space proportional to D or BD). In the fully parallel case space must be found to put the exponentially increasing number of problem solvers. Since space becomes available at the rate at which units can move apart, the ultimate determiner of the growth of the tree is the velocity of movement. Since this is always limited (to one module per cycle for this machine), the time to search the tree remains exponentially related to depth.* This is simply the Malthusian problem for programs, in which a population of programs reproduces itself geometrically, while its sustenance, space, grows only linearly.

Parallel search is still an effective strategy. By suitably spacing the initial points, extremely rapid exploration can be had of the first part of the tree. The devices described earlier for information processing make the mechanization of these procedures straightforward. Independent problem trees can exist simultaneously in the processing area, each encased in an expanding boundary. The entire set of growing units moves around in the yet larger bounded area that contains them all, utilizing the total space in a reasonable, if not optimal, fashion.

Supervision

Earlier a principle of awareness was stated by which the machine should be able to detect the consequences of its actions and have some ability to correct, modify, or prevent them. This covers reliable computation as well as more subtle things, such as inconsistencies in the user's data.

* Even if space expands in N dimensions, the volume cannot grow fast enough to keep up with an exponential demand. However, added dimensions help. If N dimensions are available for expansion, the search time becomes like $B^{D/N}$.

An approach to this problem is to provide reliable containment of consequences and supervision by independent processes that are sufficiently intelligent to handle the errors that occur.* The unit boundaries, impervious to outward flowing influence but transparent to inward flowing influence, are naturally suited to contain consequences. The supervisory processes are units that sit outside the units they supervise, checking and correcting certain classes of errors. Not all errors can be checked, since only a finite amount of effort can be spent on supervision, and there are many errors the machine cannot possibly rectify.

Two aspects of the problem can profitably be discussed further at this level. First, who shall guard the guardians? The simplest answer is to have another supervisor, as shown in Figure 9. Each lower system is contained in a walled cell, and the supervisor outside has paths into it for monitoring it. The problem is to avoid the implied infinite regress. One solution is to adhere to the rule that any unit that acts outside itself must be supervised, but any unit that only takes in information need not be supervised. In Figure 9, assume the action program is for a user and will result in an external response. It must be supervised, so Supervisor #1 must exist. As long as Supervisor #1 takes no action on the action program, it need not be supervised. The moment Supervisor #1 goes into action, Supervisor #2 must come into existence, and so on through an upward recursion of supervisory routines. As units complete their tasks the supervisory units disappear again. If supervisors are constructed to act rarely, say only when errors occur, then the height of the recursion is governed by the successive powers of the error probability and is kept under good control.

The second aspect is the nature of general computation. Current programs normally consist of a single procedure, all of which must be gone through to produce the final result. Contrariwise, most procedures carried out by people are done in the context of a tree of alternatives, where at each step, although one action is taken, others are possible to achieve the same end. Thus, if the computed value of the cosine is wrong, it can be recomputed; if still wrong, it can be looked up in a table; if still wrong, the look-up procedure can be checked; and so on. The strategy is not to correct errors directly, but to bypass them and produce the result a different way. All computations are to be carried out as a problem solving search to discover the solution.The nodes in the search tree are stages in the computation; the branches are the possible computational actions. It makes little difference whether a path to the answer is rejected because it is "a wrong computational step" or because it is "a right computational step with an error."

* An alternative approach, which has been much pursued, is to achieve the reliability at the level of the smallest component by such means as error-correcting codes and multiplexing. This approach tries to solve the problem without recourse to the larger context in which the computation occurs.

Interpretation and Production

There are two conflicting requirements for this machine. For volume processing of data, its programs should be as efficient as possible. But, since these programs come to it from outside, they should be as easily communicated as possible. The external language and the internal language are completely divorced from each other in order to deal with the conflict. The external language is communication oriented; the internal language is production oriented. The intelligence of the machine mediates between them.

The general operation of the machine is as follows: The Input-output area contains expressions in the external language, generated by both outside users and internal processes. The expressions are analysed by a process in the Interpretive area, which builds up a process in the Factory that will take the action appropriate to them. The Program Store provides the parts from which these action processes are constructed, ranging from small parts, like multiplication, to entire programs, like matrix inversion. These parts are forms; they are identical to the final process except for certain variable units and paths, which must be specified to make an actual process. They are moved out of the Program Store along unoccupied planes and copies of them deposited in the Factory. Then the various unspecified units and paths are identified and brought together. For this purpose, the variable parts of the forms carry descriptions of their function. For example, the form for the process in Figure 7 might be xe^{ax} and would look identical to Figure 7 except for an x and an a in place of the K and -2, and an expression xe^{ax}, in place of the print.

Once a complete process is assembled, and the interpreter decides the interpretation is correct, it is executed. The interpreter remains intact during execution to handle difficulties, to remove the action process at the end, and to decide if anything should be salvaged for later use.

The behavior of the machine is a collection of such efforts, each composed of an action process governed by expressions in the external language and mediated by an interpretive process. Some are built up in response to simple external requests and are concluded almost immediately and some involve long production runs. Others constitute strategies of internal operation, such as the allocation of effort between production and improvement. The rate at which the machine processes information is determined by whatever limit is reached first: production time, Factory space, input-output equipment, interpretive time, and so on.

Productive Efficiency

Little study has been given to the theory of efficient computation.[8] However, certain general features of the machine permit efficient programming.

One source of inefficiency is excessive interpretation. This generally occurs in repetitive processing, where interpretation of the same procedure occurs over and over again. It is avoided by using the first interpretation to set up an efficient procedure, so that subsequent interpretation isn't needed. The division of labor between the Factory and the Interpreter permits this efficiency. A more subtle version of interpretive inefficiency is the use of symbols and other indirect forms of designation. The use of direct paths in the Factory permits the elimination of most of these costs.

The machine operates through the use of standard building blocks, which it puts together in various standardized ways to construct yet bigger building blocks. Often a process so built up can be made more efficient, both spacewise and timewise, by reconstructing it out of more microscopic processing units. One of the continual activities of the machine is to invent more efficient units for processes that are important to it. Independent of immediate demands, it takes program forms from the Program Store, attempts to improve them, and returns them to the Store. The discrete nature of the machine allows the problem of improving a program to be phrased in terms similar to the problem of playing chess or proving theorems.[9]

Interpretation as Problem Solving

The only acceptable dictum for designing the external language is that it cover the full range of expressive devices. It is easy to state some of these. We want the ability to substitute expressions for terms; to add new terms and abbreviations; to express new language conventions; to make ambiguous and vague statements; to state problems as well as procedures; to mix syntax and semantics; and so on. Each of these is needed, not only for communication, but because the machine uses this language for all its internal control and problem solving. Consider the possibility for ambiguity, which may seem an odd property to desire. If the machine cannot state an ambiguous notion, it can never get started on a series of successive approximations to achieve an exact notion. And if the man must be unambiguous in communicating to the machine, he can never get it to help him formulate a difficult computation.

A discussion of the mechanisms required to interpret these features is beyond the bounds of this paper. However, one general mechanism is relevant to all of them--the treatment of interpretation as problem solving. The interpreter in the machine for a given expression is a branching tree of processes. Each node of the tree represents a set of hypotheses about the meaning of the expression (more precisely, about the implication of the expression for current action). Each branch represents an additional hypothesis about the meaning, so that different branches at a node represent alternative interpretations of the expression. The processes at the nodes carry out the analysis of the expression. Each is specialized to the hypotheses at the node, and operationally represents these hypotheses. These processes detect information that would confirm or reject the hypotheses, and if the latter, terminate interpretation along their branch. They find new information that leads to additional hypotheses, and create processes to continue the interpretation within these more restrictive bounds. They also take direct action in building up the action process in the Factory.

Viewed this way, interpretation is the process of discovering by a series of interpretive acts a final action process that is consistent with all the information extracted from the expression. The search goes on in parallel with all alternative interpretations being carried forward. If the expression is clear, then one line emerges cleanly and immediately. If the expression is full of little errors, then many short branches occur at each node, all but one of which prove false. If the expression is ambiguous, more than one line of interpretation continues through to a complete action program, and information from more distant sources must be sought.

To give one simple example of this process, consider the interpretation of the request "Compte Ke^{-2K} for K = 1 and 2." Assuming that nothing is known about the expression, interpretation starts with a standard process. "Compte" is recognized as not corresponding to a word, and the interpretive hypothesis is taken that the user meant "Compute." On this basis a new interpretive process is set up that assumes some formula for numerical computation will follow. When Ke^{-2K} is found this is confirmed. This action is dependent on the first. If the initial word had been "Differentiate," then a form for numerical computation would be wrong. Also, the hypotheses needn't have been confirmed. If the original expression had been "Compte is to mean the same as Compute," then a numerical computation was not desired and an alternative hypothesis is needed, say that "Compte" is being used as a name for itself. Getting the form constitutes a second interpretive act and another interpretive process is created which matches the form to the expression. This results in additional interpretive acts as x is identified with K and a is identified with -2. The final interpretive act occurs when the print process is put in. Since no indication is given for what the machine is to do with the computation, this is a hypothesis that the user wants the information given to him immediately. Its confirmation must wait until after the action process has been executed, and the user reacts to the result.

Although this mechanism does not solve the problems of interpreting the devices mentioned earlier, it seems to be a necessary feature of an intelligent interpreter.

Last Thoughts

Speculation stops here. The problems of making an intelligent technician from a Holland machine form an expanding tree which could be explored both broader and deeper.

Each of the topics touched on in the paper was left incomplete: the nature of the modules that can achieve the properties stipulated; the nature of the telephone exchange; the mechanisms for interpreting the external language; and so on. Major aspects were not even mentioned: the problem of going from an object to its symbol; the problem of organizing the machine's knowledge; the problem of context; and so on. But at this stage of development, with the microelectronic techniques still a little ways distant, it is appropriate to end with a sense of incompleteness and need for further exploration.

References

1. Holland, J. H., "On Iterative Circuit Computers Constructed of Microelectronic Components and Systems," These Proceedings. See also, "A Universal Computer Capable of Executing an Arbitrary Number of Sub-programs Simultaneously," Proceedings of the 1960 Eastern Joint Computer Conference, December, 1959.

2. Churchman, C. W., "On a Potential Customer for an Intelligent Technician," These Proceedings.

3. Unger, S. H., "A Computer Oriented Toward Spatial Problems," Proceedings of the 1958 Western Joint Computer Conference, May, 1958.

4. Selfridge, O. G., "Pandemonium," Proceedings of the Symposium on the Mechanization of Thought Processes, Teddington, England, 1959.

5. McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine," Quarterly Progress Report No. 53, Research Laboratory of Electronics, Massachusetts Institute of Technology, April, 1959; or J. C. Shaw, A. Newell, H. A. Simon, and T. O. Ellis, "A Command Structure for Complex Information Processing," Proceedings of the 1958 Western Joint Computer Conference, May, 1958.

6. Steward, D. V., "On an Algebraic Foundation for Constructing Optimum Algorithms," General Electric Technical Report, GEAP 3159, August, 1959.

7. For example, see H. Gelernter, "Realization of a Geometry Proving Theorem Proving Machine," Proceedings of the International Conference on Information Processing, UNESCO, June, 1959 (in press); A. Newell, J. C. Shaw and H. A. Simon, "Chess Playing Programs and the Problem of Complexity," IBM Journal of Research and Development, 2, 4, October, 1958; or A. Samuel, "A Checker Playing Program," IBM Journal of Research and Development, 3, 3, June, 1959.

8. However, see Jeenel, J., "Programs as a Tool for Research in Systems Organization," IBM Journal of Research and Development, 2, 2, April, 1958.

9. Kilburn, T., R. L. Grimsdale and F. H. Sumner, "Experiments in Machine Learning and Thinking," Proceedings of the International Conference on Information Processing, UNESCO, June, 1959 (in press).

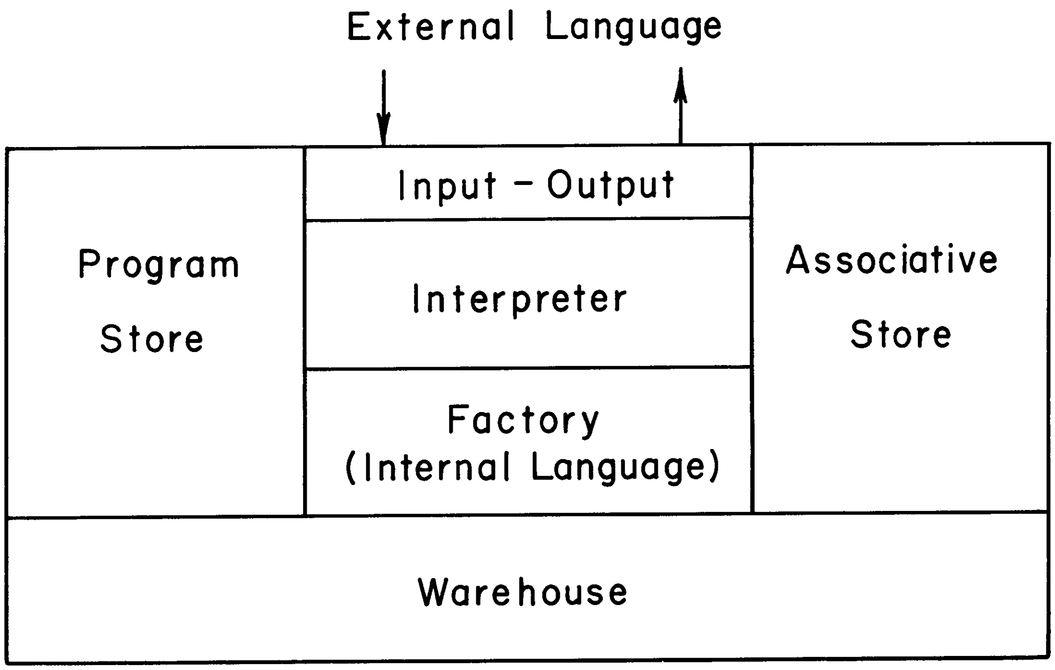

Figure 1 Layout of the Machine

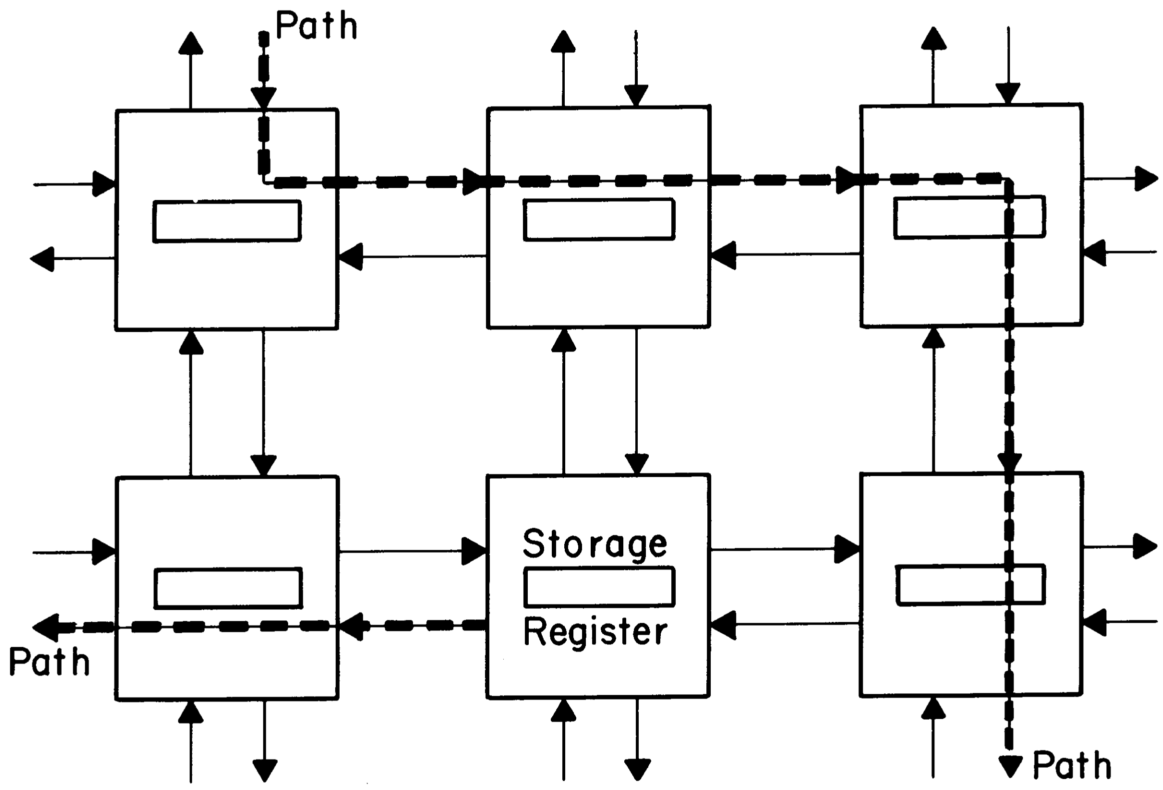

Figure 2 Structure of a Holland Machine

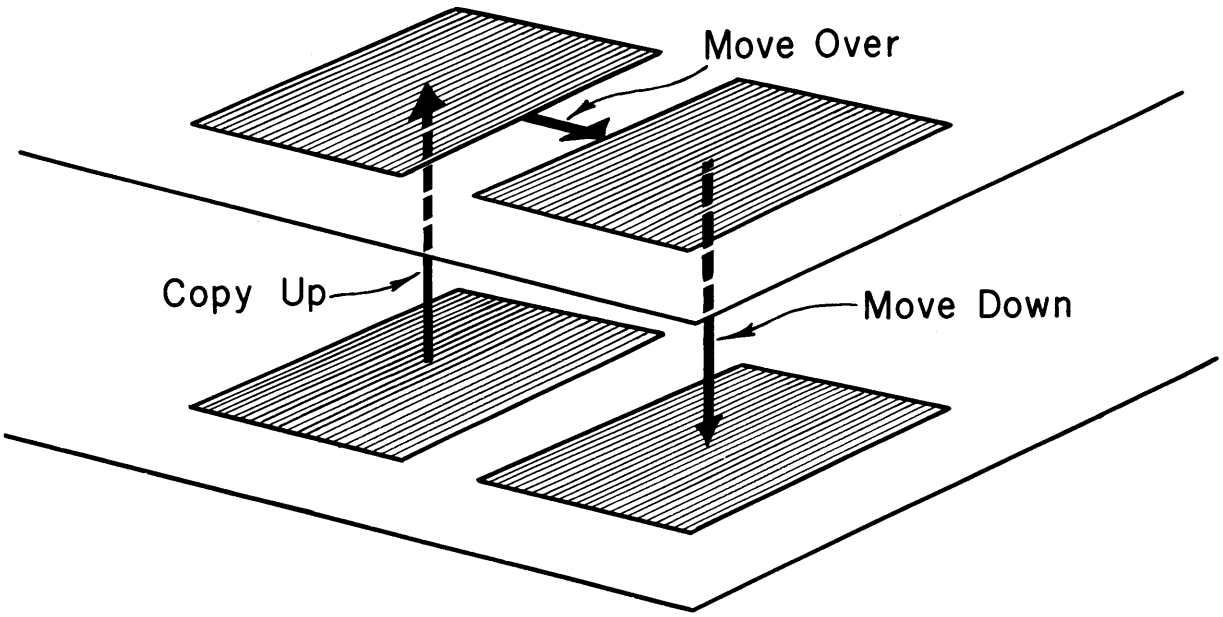

Figure 3 Spatial Operations

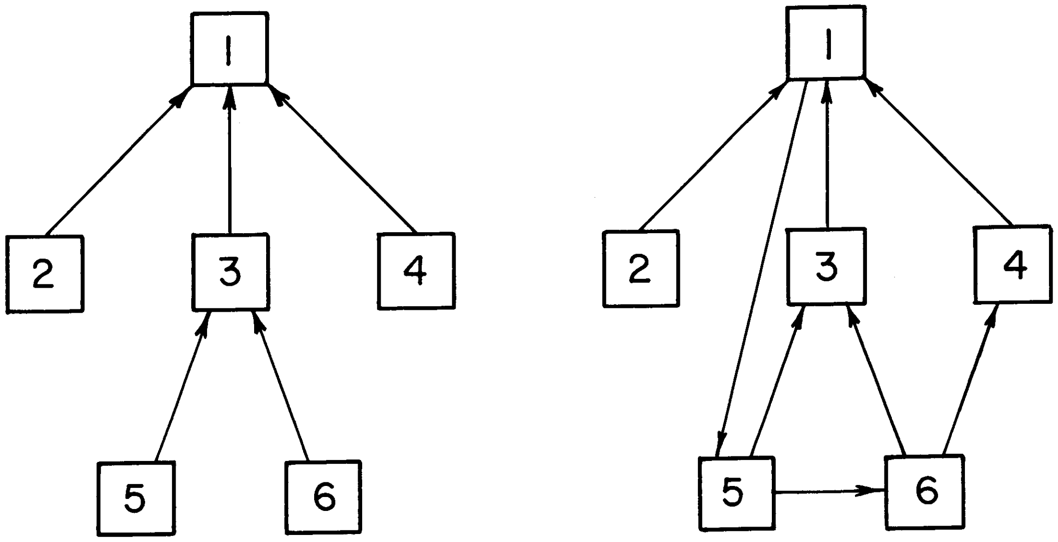

Figure 4 Information Structures

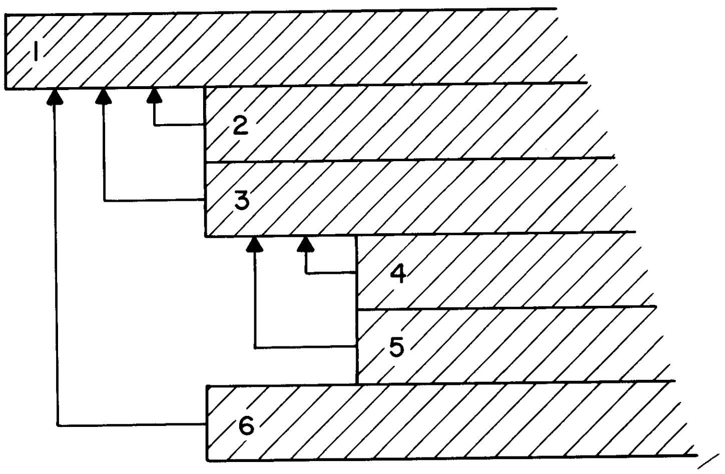

Figure 5 Outline Form for Information Structures

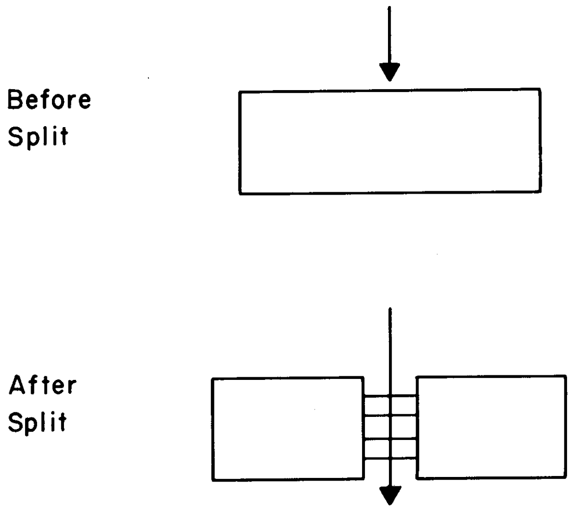

Figure 6 Splitting Operation

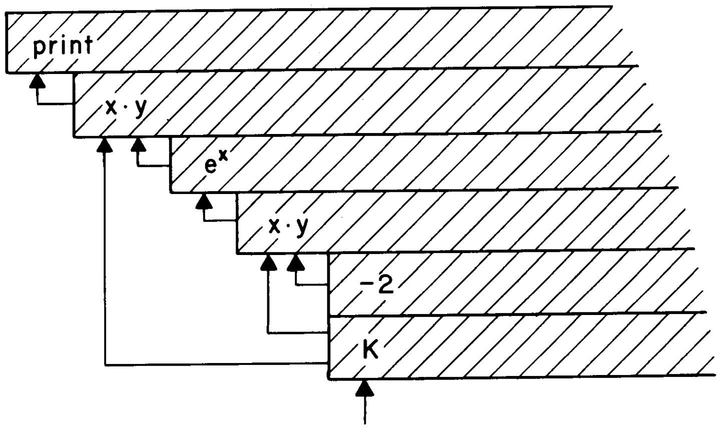

Figure 7 Definitional Control

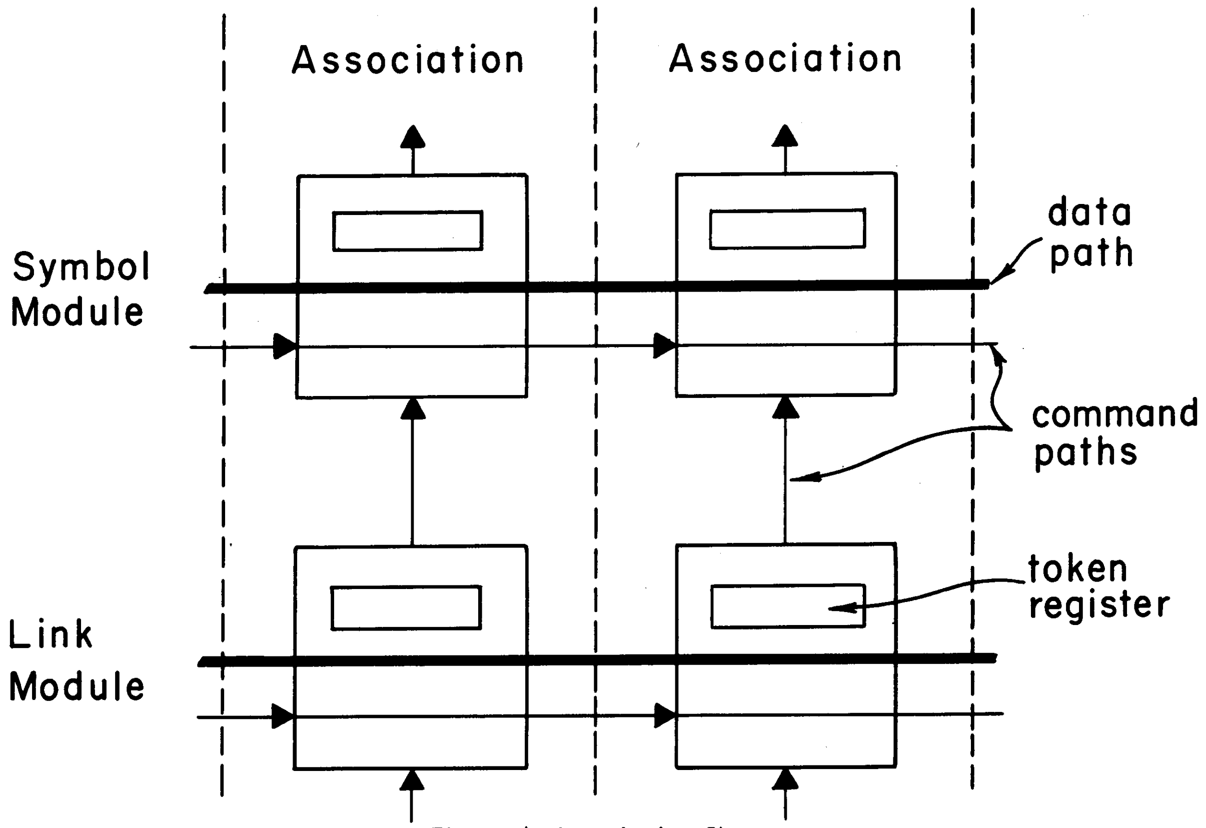

Figure 8 Associative Store

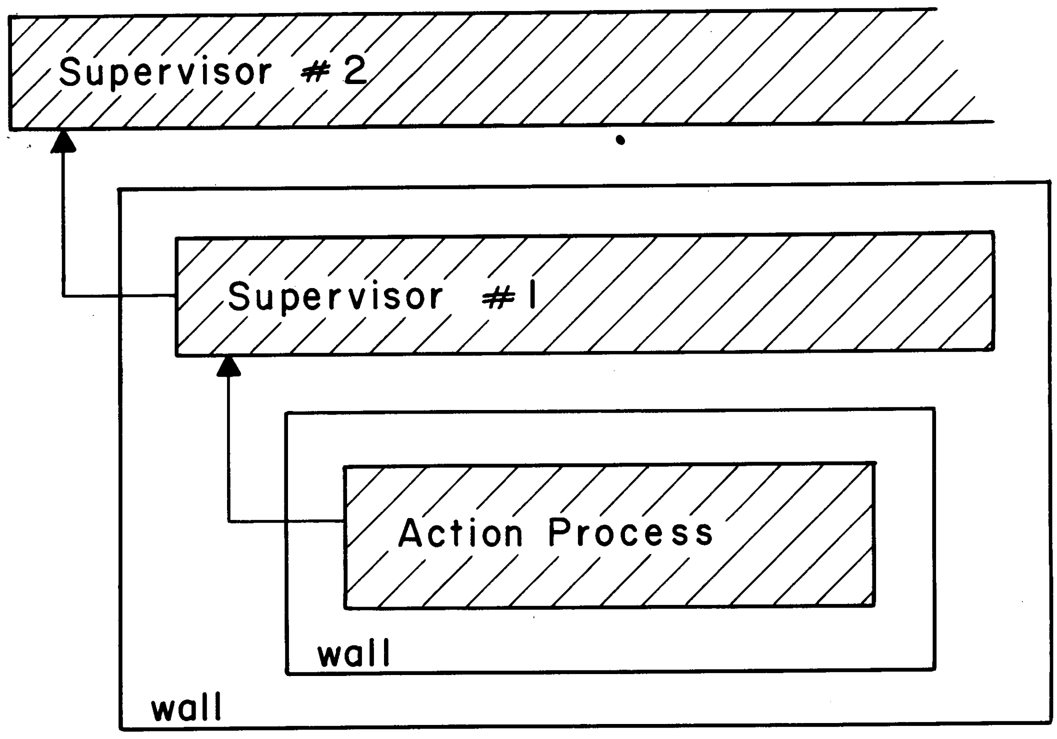

Figure 9 Supervision