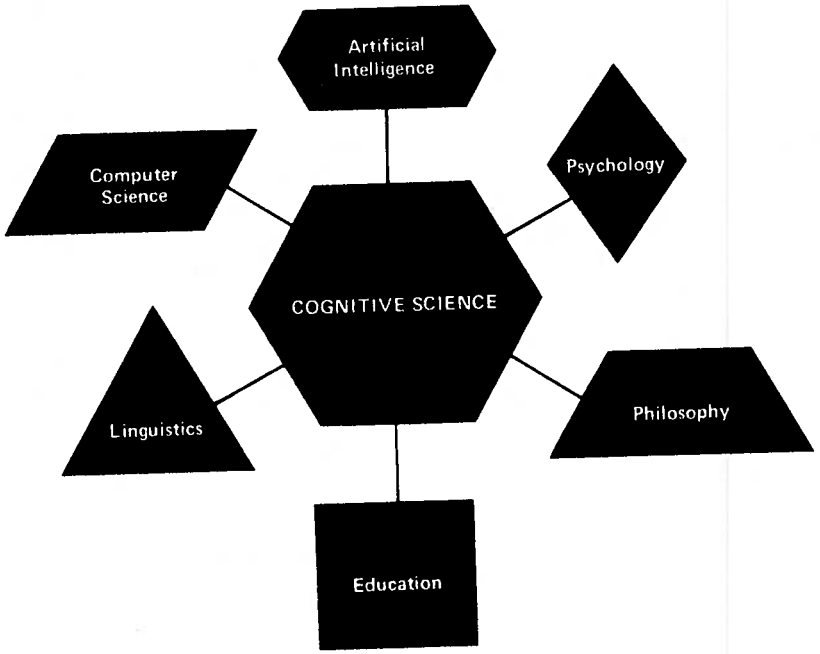
LANGUAGE, THOUGHT, AND CULTURE
Advances in the Study of Cognition

# REPRESENTATION AND UNDERSTANDING

## Studies in Cognitive Science




*Edited by*

*Daniel G. Bobrow / Allan Collins*

*Academic Press, Inc.* A Subsidiary of Harcourt Brace Jovanovich, Publishers

REPRESENTATION AND UNDERSTANDING
Bobrow / Collins

Academic Press

## FRAME REPRESENTATIONS AND THE DECLARATIVE/PROCEDURAL CONTROVERSY

**Terry Winograd**
*Computer Science Department*
*Stanford University*
*Stanford, California*



### I. INTRODUCTION

Any discussion today of "the representation problem" is likely to entail a debate between proponents of *declarative* and *procedural* representations of knowledge. Sides are taken (often on the basis of affinity to particular research institutions) and examples are produced to show why each view is "right". Recently a number of people have proposed theories which purport to solve many representational problems through the use of something called "frames". Minsky (1975) is the most widely known example, but very

similar ideas are found in Moore & Newell's (1973) MERLIN system, the networks of Norman & Rumelhart (1975), and in the schemata of D. Bobrow & Norman (Chapter 5). Parts of the notion are present in many current representations for natural language [see Winograd (1974) for a summary].

This chapter is composed of two distinct parts. In the first half I want to examine the essential features of the opposing viewpoints, and to provide some criteria for evaluating ideas for representation. The second half contains a very rough sketch of a particular version of a frame representation, and suggests the ways in which it can deal with the issues raised.

## II. THE SIMPLE ISSUES

First let us look at the superficial lineup of the argument. It is an artificial intelligence incarnation of the old philosophical distinction between "knowing that" and "knowing how". The proceduralists assert that our knowledge is primarily a "knowing how". The human information processor is a stored program device, with its knowledge of the world *embedded* in the programs. What a person (or robot) knows about the English language, the game of chess, or the physical properties of his world is coextensive with his set of programs for operating with it. This view is most often associated with MIT, and is emphasized by Minsky and Papert (1972), Hewitt (1973), and Winograd (1972).

The declarativists, on the other hand, do not believe that knowledge of a subject is intimately bound with the procedures for its use. They see intelligence as resting on two bases: a quite general set of procedures for manipulating facts of all sorts, and a set of specific facts describing particular knowledge domains. In thinking, the general procedures are applied to the domain-specific data to make deductions. Often this process has been based on the model of axiomatic mathematics. The facts are *axioms* and the thought process involves *proof procedures* for drawing conclusions from them. One of the earliest and clearest advocates of this approach was McCarthy (see



McCarthy & Hayes, 1969), and it has been extensively explored at Stanford, and Edinburgh.

From a strictly formal view there is no distinction between the positions. Anyone who has programmed in languages like LISP has been forced into believing that "programs are data". We can think of the interpreter (or the hardware device, for that matter) as the only program in the system, and everything else as data on which it works. Everything, then, is declarative.

From the other end, we can view everything as a program. Hewitt, Bishop, & Steiger (1973) actually propose this view. A fact is a simple program which accepts inputs equivalent to questions like "Are you true?" and commands like "Assume you are true!". It returns outputs like "true" and "false", while having lasting effects which will determine the way it responds in the future (equivalent to setting internal variables.) Everything is a procedure. Clearly there is no sharp debate on whether a piece of knowledge *is* a program or a statement. We must go below these labels to see what we stand to gain in *looking at* it as one or the other. We must examine the mechanisms which have been developed for dealing with these representations, and the kind of advantages they offer for epistemology. In this entire discussion, we could divide the question into two aspects: "What kind of representation do people use?" and "What kind of representation is best for machine intelligence?" I will not make this distinction, first of all since the issues raised are quite similar, and second, since I believe that at our current stage of knowledge these questions are most profitably attacked as if they were the same.

### A. The Benefits of Declaratives

*Flexibility - Economy.* The primary argument against procedural representation is that it requires that a piece of knowledge be specified by saying how it is used. Often there is more than a single possible use, and it seems unsatisfactory to believe that each use must be specified in advance. The obvious example is a simple universal fact

like "All Chicago lawyers are clever." This could be used to answer a question like "Is Dan clever?" by checking to see if he is a Chicago lawyer. It might be used to decide that Richard is not from Chicago if we know he is a stupid lawyer, or that he is not a lawyer if he is a dim-witted Chicagoan. Each of these might be done in response to a question which is asked, or as a response to a new piece of information which has just been added. In a strictly procedural representation, the fact would have to be represented differently for each of these deductions. Each would demand a specific form, like "If you find out that someone is a lawyer, check to see if he is from Chicago, and if so assert that he is clever." Traditional logic, however, provides a simple declarative representation in the predicate calculus:

$$\forall(x)\ [\text{Chicagoan}\ (x)\ \&\ \text{Lawyer}\ (x) \Rightarrow \text{Clever}\ (x)]$$

The different uses for this fact result from its access by a general deductive mechanism. In adding this formula to the system, we do not have to anticipate how it will be used, and the program is therefore more flexible in the sorts of deduction it will be able to make. Associated with this flexibility is an obvious economy in making multiple uses from a single statement.

*Understandability - Learnability.* The simplicity of the declarative statement above is more important than just an economy of storage. It also has important implications for the case of understanding and modifying the body of knowledge in a system. If the knowledge base is a set of independent facts, it can be changed by adding new ones, and the implications of each statement lie directly in its logical content. For programs, on the other hand, the implications lie largely in questions of how a routine is to be used, under what conditions, with what arguments, etc. Minor changes can have far-reaching effects on other parts of the program. In addition, it is not easy to split programs into independent subprograms. Thus a single "piece" of knowledge may be embedded as a line in a larger integrated program. Changing or adding is much more difficult. This issue of modification applies both to programmers trying to build a system, and to any sort of self-modification or learning.



*Accessibility - communicability.* Much of what we know is most easily statable as a set of declaratives. Natural language is primarily declarative, and the usual way to give information to another person is to break it into statements. This has implications both for adding knowledge to programs, and communicating their content to other people. Quite aside from how the eventual program runs, there may be important advantages in stating things declaratively, from the standpoint of building it and working with it.

## B. The Benefits of Procedures

*Procedural - Modeling.* It is an obvious fact that many things we know are best seen as procedures, and it is difficult to describe them in a purely declarative way. If we want a robot to manipulate a simple world (such as a table top of toy blocks), we do it most naturally by describing its manipulations as programs. The knowledge about building stacks is in the form of a program to do it. Since we specify in detail just what part will be called when, we are free to build in assumptions about how different facts interrelate. For example, we know that calling a program to lift a block will not cause any changes in the relative positions of other blocks (making the assumption that we will only call the lift program for unencumbered blocks). In a declarative formalism, this fact must be stated in the form of a *frame axiom* which states something equivalent to "If you lift a block X, and block Y is on block Z before you start, and if X is not Y and X is not Z and X is unencumbered, then Y is on Z when you are done." This fact must be used each time we ask about Y and Z in order to check that the relation still holds. Note that this knowledge is taken care of "automatically" in the procedural representation because we have control over when particular knowledge will be used, and deal explicitly with the interactions between the different operations.

In using procedures we trade some degree of flexibility

for a tremendous gain in the ease of representing what we know about processes. This applies not only to obvious physical processes like moving blocks around, but equally well to deductive processes like playing games or proving geometry theorems.

*Second Order Knowledge.* One critical component of our knowledge is knowing about what we know and what we can know. We have explicit facts about how to use other facts in reasoning, and this must be expressed in our representation. These are not sophisticated logical tools, but straightforward heuristics. For example, if we want to generate a plan for getting to the airport, we know "If you do not see any obvious reason why the road should be impassable, assume you can drive." The catch is obviously the word "obvious". It is quite distinct from logical notions of truth and provability, referring to the complexity and difficulty of a particular reasoning process. Another such fact might be "The relation *NEAR* is transitive as long as you don't try to use it too many times in the same deduction."

It is theoretically possible to express second-order knowledge in a declarative form, but it is extremely difficult to do so outside the context of a particular reasoning process. In a procedural representation we can talk directly about things like the depth or duration of various computations, about the particular ways in which facts will be accesssed, etc.

*The Need for Heuristic Knowledge.* The strongest support for procedural representation comes from the fact that it works. Complex AI programs in all domains have a large amount of their knowledge built into their procedures. Those programs which attempt to keep domain-specific knowledge in a nonprocedural data base do so at the expense of limiting themselves to even more simplified worlds, and the simplest of goals. The obvious reason for this is that much of what we know about an area is *heuristic*. It is neither "simple facts" nor general knowledge about reasoning, but is of the form "If you are trying to deduce this particular sort of thing under this particular set of conditions, then you should try the following



strategies." In theory, this knowledge could be kept separate and integrated with the declarative knowledge of a domain. In practice, it is not at all clear how a real system could do this. Most systems which have been based on a declarative formalism have only the general heuristics built into the interpreter, and do not make it easy to add domain-specific strategies. By putting knowledge in a primarily procedural form, we gain the ability to integrate the heuristic knowledge easily. The deduction process is primarily under control of the specific heuristic knowledge.

## III. SOME UNDERLYING ISSUES

At this point it is tempting to look for a synthesis -- to say "You need both. Some things are better represented procedurally, others as declarative facts, and all we need to do is work on how these can be integrated." This reaction misses what I believe is the fundamental ground for the dispute. It is not simply a technical issue of formalisms, but is an expression of an underlying difference in attitude towards the problems of complexity. Declarativists and proceduralists differ in their approach to the duality between modularity and interaction, and their formalisms are a reflection of this viewpoint.

In his essay, The Architecture of Complexity, Simon (1969, p. 100) describes what he calls *nearly decomposable systems*, in which "...the short-run behavior of each of the component subsystems is approximately independent of the short-run behavior of the other components.... In the long run, the behavior of any one of the components depends in only an aggregate way on the behavior of the other components." One of the most powerful ideas of modern science is that many complex systems can be viewed as nearly decomposable systems, and that the components can be studied separately without constant attention to the interactions. If this were not true, the complexity of real-world systems would be far too great for meaningful understanding, and it is is possible (as Simon argues) that it would be too great for them to have resulted from a process of evolution.

In viewing systems this way, we must keep an eye on

both sides of the duality -- we must worry about finding the right decomposition, in order to reduce the apparent complexity, but we must also remember that "the interactions among subsystems are weak *but not negligible*". In representational terms, this forces us to have representations which facilitate the "weak interactions".

If we look at our debate between opposing epistemologies, we see two metaphors at opposite poles of the modularity/interaction spectrum. Modern symbolic mathematics makes strong use of modularity at both a global and a local level. Globally, one of the most powerful ideas of logic is the clear distinction between *axioms* and *rules of inference*. A mathematical object can be completely characterized by giving a set of axioms specific to it, without reference to procedures for using those axioms. Dually, a proof method can be described and understood completely in the absence of any specific set of axioms on which it is to operate. Locally, axioms represent the ultimate in decomposition of knowledge. Each axiom is taken as true, without regard to how it will interact with the others in the system. In fact, great care is taken to ensure the logical independence of the axioms. Thus a new axiom can be added with the guarantee that as long as it does not make the system inconsistent, anything which could be proved before is still valid. In some sense all changes are additive -- we can only "know different" by "knowing more".

Programming, on the other hand, is a metaphor in which interaction is primary. The programmer is in direct control of just what will be used when, and the internal functioning of any piece (subroutine) may have side effects which cause strong interactions with the functioning of other pieces. Globally there is no separation into "facts" and "process" -- they are interwoven in the sequence of operations. Locally, interactions are strong. It is often futile to try to understand the meaning of a particular subroutine without taking into account just when it will be called, in what environment, and how its results will be used. Knowledge in a program is not changed by adding new subroutines, but by a *debugging* process in which existing structures are modified, and the resulting changes in interaction must be explicitly accounted for.



If we look back to the advantages offered by the use of the two types of representation, we see that they are primarily advantages offered by different views toward modularity. The flexibility and economy of declarative knowledge come from the ability to decompose knowledge into "what" and "how". The learnability and understandability come from the strong independence of the individual axioms or facts. On the other hand, procedures give an immediate way of formulating the interactions between the static knowledge and the reasoning process, and allow a much richer and more powerful interaction between the "chunks" into which knowledge is divided. In trying to achieve a synthesis, we must ask not "how can we combine programs and facts?", but "How can our formalism take advantage of decomposability without sacrificing the possibilities for interaction?"

## IV. STEPS TOWARD A MIDDLE

If the declarative and procedural formalisms represents endpoints on a spectrum of modularity/interaction, we should be able to see in each of them trends away from the extreme. Indeed, much current work in computing and AI can be seen in this light.

### A. Modular Programming

One of the most prominent trends in computer programming today is toward some kind of *structured programming* as exemplified, say, by the work of Dahl, Dijkstra, & Hoare (1972). It represents a response to the tremendous complexities which arise when a programmer makes full use of his power to exploit interactions. Advocates maintain that understandability and modifiability of systems can only be maintained if they are forced to be "nearly decomposable". This is enforced by severely limiting the kinds of interaction which can be programmed, both in the flow of control and in the manipulation of data structures. This represents a move toward the kind of *local modularity* between the individual pieces of knowledge, and is carried to a logical extreme by Hewitt's actors.

Recent AI programming languages represent an attempt to achieve some *global modularity* within the programming context. [For a good overview, see Bobrow & Raphael (1974).] Through the use of pattern-directed call, and search strategies such as backup, they attempt to decouple the flow of control from the programmer. Procedures are specified whose meaning is in some sense free from the particular order in which they will be called, and the system has some sort of general mechanism for marshalling them in any particular case. So far, the general procedural knowledge is extremely primitive, and the potential modularity is rarely used. It is exploited much more fully in the *production systems* of Newell & Simon (1972). We can view production systems as a programming language in which all interaction is forced through a very narrow channel. Individual subroutines (productions) interact in only two ways, a static ordering and a limited communication area called the *short-term memory*. They can react to data left in the short-term memory and modify it as their means of communication. Their temporal interaction is completely determined by the data in this STM and a uniform ordering regime for deciding which productions will be activated in cases where more than one might apply. The orderings which have been most explored are a simple *linear* ordering of the productions, and a restricted system in which it is considered an error if there is not a unique production which matches the contents of short term memory at any time. Thus instead of the full ability to specify just what will happen when, the programmer can only determine the ordering, and the rest of the interaction is out of his hands. Of course, it is possible to use the STM to pass arbitrarily complex messages which embody any degree of interaction we want. The spirit of the venture, however, is very much opposed to this, and the formalism is interesting to the degree that complex processes can be described without resort to such tricks, maintaining the clear modularity between the pieces of knowledge and the global process which uses them.



### B. Compiling Facts

Starting with a traditional logical-declarative system, Sandewall (1973) attempts to build in some of the specific control interactions which make programs effective. He compiles declarative information into *operators* in which specific decisions are embedded about what knowledge should be called into play when. These are then included as part of a system which in many ways is similar to GPS, the intellectual forerunner of the production systems. Another approach from this direction is Sussman's (1973) attempt to combine the effectiveness of procedures with the ease of modification which comes from modularity. His system contains both declarative and procedural knowledge, but combines them by being an *active programmer*. There is a body of declarative data about the specific subject domain, but it is not used directly in this form. As each piece is added, whether as a statement from a teacher, or from an experience in the model-world, it is perused by a programmer-debugger. General knowledge about procedures is brought into play to decide just how the new knowledge should be integrated into the domain-specific programs, and how the resulting interactions might be anticipated and tested. Thus what the system knows may be decomposed into "procedure" and "domain fact" modules, but these are internally combined into a procedural representation.

## V. A FIRST ATTEMPT AT SYNTHESIS

Recently much excitement has been generated by the idea of a representational format called "frames" which could integrate many of the new directions described in the previous section. So far, this work is in a beginning state of development, and none of the available papers work out the actual implementation of such a scheme and its application to a significant set of problems. Therefore, most of what can be said is at the level of general system criteria, and ideas for organization.

In this section I give a simple example of a system which represents knowledge in a frame-like notation. I warn the reader strongly that this does not purport to be

an explanation of what frames "really are", or to represent anyone else's understanding of what they should be. Many different issues are still unsettled (many more perhaps unrecognized), and it will be a long time before any agreements on notation and operation can be reached. The notation also does not represent a worked-out design, but is intended to be suggestive of the necessary formalism. This example is specifically chosen to *avoid* many of the most interesting problems, in order to gain at least a small foothold for attacking them sensibly. Thus it oversimplifies and misrepresents my own views of frames as well. This version grew up in the same environment as Minsky's (1975) frames, but with a slightly different emphasis since they were initially applied with a view toward natural language rather than vision. Their development over the last year has been strongly influenced by discussions with Andee Rubin at MIT, and Daniel Bobrow at Xerox PARC. Further work is continuing in conjunction with Bobrow, leading toward the specification of the frame formalism and the development of reasoning programs which use it.

With this caveat clearly in mind, let us look at the problem of understanding the connection between days, dates, and numbers. The context for this problem can be best understood by imagining some sort of program (or person) acting as an office assistant in matters such as scheduling. The assistant must be able to do things like writing schedules, accepting information about events to be scheduled, and accepting facts about dates in a natural input. This does not necessarily mean natural language, but in a form whose structure corresponds roughly to that which a person might use to another person.

## A. The Generalization Hierarchy

We first note that the system would have a set of internal concepts appropriate to the subject matter. These might include things such as those in Fig. 1. These concepts are arranged in a *generalization hierarchy*, a structure of *isa* links connecting concepts to those of which they are *specializations*. This hierarchy contains both



specific objects (like *July 4, 1974*) and general objects (like *day, holiday,* and *Thursday*). I will avoid discussing the problems inherent in the combination of these in a single hierarchy, as that deserves at least another entire paper.

Associated with each node in this hierarchy is a *frame* tying together the knowledge we have of that concept. Often there will also be a particular English word or phrase associated with it, but that is not a necessary condition. Reference within the system can be made by using the internal concept names as shown in the hierarchy.

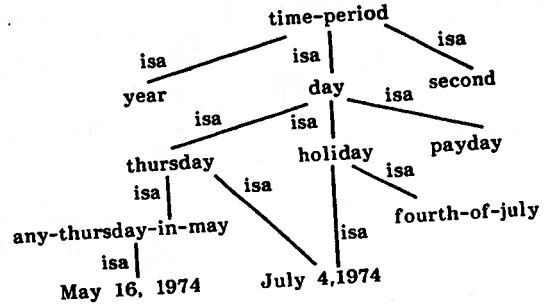



**Fig. 1. A generalization hierarchy.**

It is clear that this cannot be a simple tree, since often two different generalizations apply to the same specific concept. *July 4, 1974* is both a *fourth-of-july* and a *Thursday*. The use of this hierarchy is primarily through *inheritance of properties* Any property true of a concept in the hierarchy is implicitly true of anything linked below it, unless explicitly contradicted at the lower level. This is an old idea, to be seen, for example, in the work of Raphael and Quillian (in Minsky, 1968). At times it has been used to cover too much; however, as one of many deductive mechanisms, it is particularly efficient and intuitively reasonable. Again, many issues can be raised about the problem of "multiple isa links". If two generalizations apply, how do we choose the appropriate one to look for the properties we want, and how do we resolve

disagreements between them? For the purposes of this elementary discussion, we will ignore these (which of course does not make them go away for good).

## B. Description and Classification

In the previous section we used the word "generalization" to describe the relationship between frames linked in the hierarchy. Another way to think of it would be as a system of *classification*. Each frame in fact represents a class of objects, and an *isa* link connects a class to some superclass which properly contains it. From an operational viewpoint it seems more useful to think of this as a hierarchy of *descriptions*. Of course there is a duality between the idea of descriptions and the idea of the classes of objects to which they apply. In making use of these frames, we will be specifying and modifying descriptions, not dealing with the sets of objects (mental or physical). Thus we will use the notion that a general description can be *further specified* to any one of a number of more specific ones. At the bottom we have descriptions which apply to unique objects in the system's model of the world. In attaching particular properties or facts to a node of this hierarchy, we are saying "Anything to which the description at this node applies also has these additional properties."

This operation of applying classifications is one of the basic modes of reasoning. We decide on the basis of partial evidence that some particular object with which we are concerned belongs to a known class (i.e., there is a particular frame for it in our hierarchy). On the basis of that decision, we can apply a whole set of additional knowledge associated with that frame. We can use the frame to guide our search for the specific facts associated with the object, or to make assumptions about things that must be true of it without checking specifically. [Minsky (1975) has an extended discussion of this vital aspect of frames as does Kuipers, Chapter 6.].

This type of reasoning can be extended to handle things more like analogy. In attempting to apply a full description to an object which would not normally be in the corresponding class, the frame focuses our attention on those particular properties or facts which are applicable, and those which are contradictory.



## C. Important Elements

What then are the elements attached to a description? In a simple predicate scheme, our classifications correspond to the set of predicates. All axioms (or clauses) containing a predicate are attached implicitly by the fact that they contain the symbol. Any statement which contains the predicate *day(x)* says something about the concept *day*. In a semantic net system, the attachments are explicit, and essentially constitute all the links from the concept to anything else. The frame idea adds an important additional element to this -- the idea of *importance* or *centrality*. For each frame, there is a set of other frames marked as *important elements* having a specific relation to it. These are what Minsky (1975) calls "slots", Newell calls "components of the beta-structure", etc. The details of how these are determined and treated is probably the key area of frame theory, and the one on which there is the least agreement. Again with the warning that this oversimplification does not represent anyone's full views (including mine), I will pare away most of the interesting problems and go on a set of intuitive but unexplored assumptions for this particular case.

Associated with the frame for *day*, the important elements (or *IMPS*) might be, for example, *year, month, day-numbers, day-of-week, sequence-number* and *ASCII-form*. Immediately we note that the idea of important elements must be sensitive to what we are doing. The idea of *day* would have very different elements if we were thinking of it as a typical sequence of events for a person (getting up, going to work,....), as an astronomical phenomenon (period of rotation, orientation of axis,...), as a set of natural events (sunrise,...,sunset, night), etc. We need separate frames for things like *day viewed as a calendar object, day viewed as an event sequence*, etc. Clearly more mechanism is needed to explain the connections between these (how do they relate in more ways than the fact they use the same

word), and the ability to choose the appropriate one for a given context. The idea of attaching IMPs to frames is not intended to avoid the more usual types of deduction and association, but to augment it by providing some kind of formalism for talking about the importance of certain ideas and relations with regard to a central concept (frame) within a context.

**TABLE I.**
**Descriptions of the IMPs for "day"**

| IMP | DESCRIPTION |
|---|---|
| Year | Integer |
| Month | Month-name |
| Day-number | (Integer range (interval min 1 max 31)) |
| Day-of-week | Weekday-name |
| Sequence-number | Integer |
| ASCII-form | (Integer<br>length 6<br>structure (concatenated-repetition<br>element (integer length 2)<br>number 3))) |

When we further specify a frame (that is, move down the generalization hierarchy) we also further specify the IMPs that go with it. This makes sense only if we think of each IMP as being another frame (which in turn fits into the hierarchy, and so on recursively). Thus the IMPs associated with *day* might be initially filled by the descriptions of Table I. These of course imply a much more extended set of frames to cover things like "integer", "string", "6", the relation "length", etc. This knowledge will be common to many of the knowledge domains represented in a computer system, and one of the basic prejudices of this approach is that in looking at the



knowledge of any part of the system we must do it in terms of its dependency on other parts. Our world knowledge is all tied together, and in any particular area we make use of our more general concepts. We have adopted a simple notation in which a frame is given by putting the name of its generalization (the thing to which it has an *isa* link) followed by a list of IMP names and their contents. If the appropriate frame is a simple named one, whose IMPS are not further specified, that name is used directly. This should definitely not be interpreted in the usual LISP sense of atoms and functions, and is also not isomorphic to a kind of straightforward atom-property list notation. Since the problems of all this do not arise at the level of detail used in the discussion below, they will be ignored for now [see Moore & Newell (1973) for a discussion of some of them].

At first glance these descriptions look like traditional computer language data types--integer, string, etc. Indeed, data types are one simple example of a shallow generalization hierarchy, but do not extend nearly far enough. We must be able to provide descriptions at any level of detail rather than choosing from a small finite set of categories. Newer computer language formalisms are beginning to move in this direction. [See Dahl, Dijkstra, & Hoare, 1972; Cheatham & Wegbreit, 1973.] Also, it should be pointed out that the choice of descriptions is not strictly determined. The fact that the standard ASCII representation of a date is a six-digit integer is simple. But the fact that this integer is a sequence of three two-digit integers is less straightforward. Clearly any six-digit integer can be viewed in this way, but the fact that this description is used explicitly is a statement that this way of looking at it is in fact relevant and important for the purposes of this frame. One of the key points of the notation is that it allows (encourages) us to include facts which although in themselves "epistemological" (to use McCarthy's term) give us heuristic clues by their very presence in the representation. We want to represent not just the bare essence of what we know, but also the scheme of how we think about it. Here we are treading on that middle ground between declarative and procedural knowledge, by selecting our declarative statements with specific concern for how they will be used.

## D. Relations between IMPs

We can now express a frame for a particular day as in Table II or a more specific but still general day as in Table III.

TABLE II.
**Further Specification of IMPs for "July 4, 1974"**

| IMP | DESCRIPTION |
|---|---|
| Year | 1974 |
| Month | July |
| Day-number | 4 |
| Day-of-week | |
| Sequence-number | |
| ASCII-form | 740704 |

TABLE III.
**Further Specification IMPs for "Any Thursday in May"**

| IMP | DESCRIPTION |
|---|---|
| Year | |
| Month | May |
| Day-number | |
| Day-of-week | Thursday |
| Sequence-number | |
| ASCII-form | |



In each case, we have filled in those IMPs which correspond to the "natural" way of describing that day. The *filler* for each IMP is a further specification of the same IMP in the *day* frame. For example, *4* is a further specification of *(integer range (interval min 1 max 31))* which in turn is a further specification of the general frame for *integer* in the part of the generalization hierarchy dealing with numbers. It is obvious that a more extended formalism is needed to express what we know about dates. The set of IMPs is not independent. Quite the opposite -- it is the interrelation between these important elements that provides most of the useful knowledge connected with the frame. Thus we need a notation which allows specific reference to IMPs associated with a frame. Newell seems to want to avoid this in his beta-structures, but it is not clear that this can be done at all, and certainly not without paying an untenable price in combinatorial explosion. We have chosen to represent an element of a frame as a path of IMP names implicitly beginning with the frame in which the path appears. Once again there are many issues buried here about variable binding, scoping, use of pointers versus names, etc., and the notation is far from settled. A list beginning with a "*!*" is used to indicate such a reference path. Thus in Table IV we include additional knowledge about our general concept of *day*, such as the fact that the year is actually the digits "19" concatenated with the first two digits of the ASCII form, or that we can expect each month name to have associated with it an IMP named *length*, and that the range of the *day-number* is determined by it. In adding this information, we have used material from still more of the fundamental set of frames used by a problem solver, such as *1-1 correspondence* and *position-in-list*.

## E. Procedural Attachment

So far we have been building a declarative data structure. It contains somewhat more information than a simple set of facts, as it has grouped them in terms of importance, and made some of them implicit in the hierarchy. We can deduce that "The ASCII code for July 4

TABLE IV.
Relations between IMPs

| IMP | DESCRIPTION |
|---|---|
| Year | (Integer<br>structure (concatenation<br>first "19"<br>second<br>(! ASCII structure first))) |
| Month | (Month-name<br>(position-in-list<br>list "January, February,...,December"<br>element (! month)<br>number (! ASCII structure second))) |
| Day-number | (Integer<br>range (interval<br>min 1<br>max (! month length))) |
| Day-of-week | (Weekday-name<br>(position-in-list<br>list "Sunday, Monday,...,Saturday"<br>element (! day-of-week)<br>number (integer<br>range (interval min 1 max 7)))<br>(1-1 correspondence<br>set1<br>(! day-of-week position-in-list number)<br>set2 (quotient-mod-7<br>dividend (! sequence-number)))) |
| Sequence-number | Integer |
| ASCII-form | (Integer<br>length 6<br>structure (concatenated repetition<br>element (integer length 2)<br>number 3))) |

is a six-digit integer." using only the hierarchical link, while deciding "Its final four digits are 0704." would



involve more complex computations, using the pointers and IMPs.

At this point we might try to describe a general deductive mechanism which made use of this elaborated structure, taking advantage of the additional information to help guide its computations. It seems possible to do so. The special use of the generalization hierarchy does not really provide any different power than that which could be expressed by a series of simple axioms like

$\forall(x)$ **Holiday(x)** $\Rightarrow$ **Day(x)**.

There have been various schemes to use groupings of axioms to guide the theorem prover in selecting which to use (although it might be very difficult to prove things formally about its behavior). To the degree such a scheme succeeded, we would have been successful at embedding heuristic knowledge into the declarative structure. I believe, however, that to be useful, a representation must make much more specific contact with procedures.

In one sense, the idea of frame structure provides a framework on which to hang procedures for carrying out specific computations. Table V indicates some of the procedures which might be attached to our frame for *day*. The notations *TO-FILL* and *WHEN-FILLED* correspond loosely to the antcedent/consequent distinction of Planner-like formalisms, and indicate the two fundamental reasons for *triggering* a computation. The mechanism for doing this should, however be more general than the syntactic pattern match which controls the calling of procedures in these languages. Triggering should be based on a more general notion of mapping [see Moore & Newell (1973) and Minsky (1975) for different versions of this.] We will describe later one aspect of how this interacts with the abstraction hierarchy.

The procedures attached to a frame or IMP are not necessarily equivalent to the factual information in the same place. One obvious example is the procedure *use-calendar* associated with *day-of-week*. Our system would have a special algorithm which can take as inputs the day-number, month, and year, and produce as output a week-day. Its steps might be paraphrased as "Find a calendar. Find the page corresponding to *month-name*. Find the number corresponding to *day-number* on the page. Look at

TABLE V.
**Possible Procedures to Attach to the "Day" Frame**

| IMP | PROCEDURE |
|---|---|
| Year | |
| Month | |
| Day-number | WHEN-FILLED<br>(CHECK-RELATION day-number, month) |
| Day-of-week | TO-FILL<br>(APPLY calendar-lookup<br>TO year, month, day-number)<br>(APPLY anchor-date-method<br>TO year, month, day-number) |
| Sequence-number | |
| ASCII-form | WHEN-FILLED<br>(FILL year, month, day-number) |

the top of that column." This is the algorithm most of us use most of the time for this computation. It depends on a batch of additional knowledge about calendars and how they are printed, pages, columns, etc., but in using this procedure we do not worry about why calendars work -- only the direct steps needed. At a deep level it is based on the facts listed in Table IV, but that is far removed from the procedure itself. This is an important element of our frame notation -- there is a large degree of redundancy between the procedural and declarative knowledge. Many procedures are specific ways to deduce things which are implicit in the facts. Many facts are essentially statements about what the corresponding procedures do. Neither one is the "fundamental" representation--in any individual case one or the other may be learned first, one or the other may be used in more circumstances.

There may be a number of different procedures attached to any frame or IMP. In this example, we have another procedure for calculating dates, based on the use of an "anchor date". If asked "What day is June 28 this year?" I



may perform a computation whose trace would be: "I know that May 12 is a Sunday, so the 19th is a Sunday, the 26th is a Sunday, the 33rd is a Sunday, but May has only 31 days, so June 2 is a Sunday, so the 23rd is, so the 28th is a Friday." This procedure involves a specific algorithm for beginning with some known date, stepping forward by 7 (or multiples of 7), accounting for the number of days in a month, and finally counting backward or forward from a known day of the week. (This last calculation often involves the use of peripheral digital devices (fingers) as well.) This procedure is much more closely tied to facts like the connection of the weekdays to the sequence of day numbers, but is not simply an application of them. The organization of these facts into a specific algorithm represents the addition of relevant knowledge, and its inclusion is not purely redundant. In operating it will also make fairly direct use of facts which are in the simple declarative form, such as that the *length* of May is 31. This new procedure is more generally applicable than the calendar one -- it works when we don't have a calendar at hand, and even works if we say "Imagine that May had 32 days this year. Then what day would June 28 be on?"

We can view the procedures as covering a whole spectrum, from very specific ones (attached to frames near the bottom of the hierarchy) to highly general ones nearer the top. If we did not have a specific procedure for finding the name of the month from the ASCII representation, we could look at the facts and try applying whatever procedures were connected to the frame for *1-1 correspondence.* In fact we should have a number of different frames which are further specifications of *1-1 correspondence* for different cases. There may be only an abstract mathematical correspondence in which we know only of its existence, but not how to compute the actual correspondence between particular elements; a functional correspondence in which given a domain element we have a direct way of computing a range element, but not vice versa; a testable-pair correspondence in which we can test whether two elements correspond, but have no good way to search for one, etc. This enrichment of the set of terms used in the facts is crucial to integrating them properly with the procedures. The set of notations which

corresponds loosely to the quantifiers and connectives of logic would be much larger and richer, involving a correspondingly expanded set of inference rules. Again, this does not increase the abstract logical power, but allows us to include much more procedurally related information ("What can you do with this fact") into the declarative notation.

The procedural information attached to frames may be very specific, or may be simply a guide of what to do, like the statment (FILL month-name, year, day-number) in Table V. This does not indicate the specific procedures to be used for each of these, but is rather a piece of *control-structure* information. It tells the system that on filling the ASCII representation, it might be a good strategy to do these other computations. The system then must look for procedures attached to those paricular IMPs (or generalizations of them) to do the work. One important area to be explored is the way in which this sort of explicit control structure can be included without paying a high price in loss of modularity.

### F. Problems of Learning and Modularity

One important aspect of procedural attachment is that it should be dynamic -- the basic ways the system learns are by adding new facts (either from being told or by inducing a generalization), and by creating new procedures based on applying general procedures to specific facts. This is the type of learning described by Sussman (1973). We can think of his work as beginning with a very general set of procedures (near the top of our hierarchy) and a set of specific facts. The outcome is a set of more specific procedures associated with the particular tasks to be done at all levels of the hierarchy. The frame framework does not attack the details of this kind of debugging, but provides a way of representing what is going on and integrating the use of the results. Similarly we can think of the task of *automatic programming* as "pushing procedures down the hierarchy". Given very general procedural knowledge and declarative facts about further specified cases, the task is to write the procedures suited to those cases.



Returning to the issue of modularity, we find that it has been attacked directly by adding a whole new layer of structure. Instead of taking a modular view (each fact is independent) or a highly integrated view (everything knows enough about the others to know how to use them) we have created a set of mechanisms for allowing the writer (whether human or program) to impose a kind of modularity through the decision of what will be included in a frame, and how the procedures will be attached. Modularity then becomes not a fixed decision of system design, but a factor to be manipulated along with all the other issues of representation.

Most of what the system knows is included in *both a modular and an integrated form.* The procedures for learning and debugging continually use general knowledge of programming to take the individual facts and combine them into the specific integrated procedures which do most of the system's deductions. Faced with a problem for which no specific methods are available, or the ones available do not seem to work, the system uses the specific facts with more general methods. There is no sharp division between specific and general methods, since there is an entire hierarchy of methods attached at all levels of the generalization hierarchy for the concepts in the problem domain. The most critical problem for the representation is to make it possible for this shifting between levels of knowledge to occur smoothly, without demanding that the programmer anticipate the particular interactions.

## VI. CONCLUSION

This formalism has clearly achieved one goal it set out to do: to blur many of the distinctions such as declarative/procedural, and heuristic/epistemological in the discussion of representations. It is yet to be seen whether this blurring will in the end clarify our vision, or whether it will only lead to badly directed groping. Many more issues are raised by it than are immediately solved, and for each solution there are many more problems. Hopefully this book will be one step on the way to sorting out what

is useful and providing a new generation of representational tools for artificial intelligence.

## REFERENCES

Bobrow, D. G., & Raphael, B. New programming languages for artificial intelligence research. *Computing Surveys*, 1974, *6(3)*, 153-174.

Dahl, O. J., Dijkstra, E., & Hoare, C. A. R. *Structured programming*. New York: Academic Press, 1972.

Hewitt, C., Bishop, P., & Steiger, R. A universal modular ACTOR formalism for artificial intelligence. *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 1973, 235-245.

McCarthy, J., & Hayes, P. Some philosophical problems from the standpoint of artificial Intelligence. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 4*. Edinburgh, 1969, 463-502.

Marvin Minsky (Ed.), *Semantic Information Processing* Cambridge: MIT Press, 1966.

Minsky, M. A framework for representing knowledge. In Winston, P. (Ed.), *The psychology of computer vision*. New York: McGraw-Hill, 1975.

Minsky, M., & Papert. S. Progress Report. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1972.

Moore, J, & Newell, A. How can MERLIN understand?. In Gregg (Ed.), *Knowledge and cognition*. Baltimore, Md.: Lawrence Erlbaum Associates, 1973.

Newell, A., & Simon, H. A. *Human Problem Solving*. Prentice Hall, 1972.

Sandewall, E. Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. *Proceedings of the Third International Conference on Artificial Intelligence*, 1973, 230-234.

Simon, H. The architecture of complexity. In *The Sciences of the Artificial*. MIT Press, 1969.

Sussman, G., *A computational model of skill acquisition* (MIT-AI TR 297), 1973.

Winograd, T. *Understanding Natural Language*, New York: Academic Press, 1972.

Winograd, T., *Five lectures on artificial intelligence* (Stanford AI-Memo-246). September 1974.