IN DEFENCE OF LOGIC

P.J. Hayes
Essex University
Colchester, U.K.

Introduction

Modern formal logic is the most successful precise language ever developed to express human thought and inference. Measured across any reasonably broad spectrum, including philosophy, linguistics, computer science, mathematics and artificial intelligence, no other formalism has been anything like so successful. And yet recent writers in the AI field have been almost unanimous in their condemnation of logic as a representational language, and other formalisms are in a state of rapid development.

I will argue that most of this criticism misses the point, and that the real contribution of logic is not its usual rather sparse syntax, but the semantic theory which it provides. AI is as much in need now of good semantic theories with which to compare formalisms as it always has been. I will also re-examine the procedural/declarative controversy and show how regarding representational languages as programming languages has, ironically, made procedural ideas as vulnerable to the old proceduralists' criticisms as the classical theorem-proving paradigm was. I will argue that the contrast between assertional and procedural languages is false: we have rather two kinds of subject-matter than two kinds of language.

This paper is deliberately polemical in tone. Much has been written from the proceduralist point of view. It's time the other arguments were put.

Logic is not a programming system

It will, and has been, said that to defend logic is to adopt a reactionary position. Logic has been tried (in the late sixties) and found wanting; now it has been superceded by better systems, in particular, procedural languages such as uPLANNER [17], CONNIVER fl8] and more recently KRL [2].

But logic is not a system in this sense. It's not a style of programming. It entails no commitment to the use of any particular process organisation or technique of coding. To think that it does is to make a category error.

Logic is a collection of ideas on how to express a certain kind of knowledge about a certain kind of world. The moetatheory of logic is a collection of mathematical tools for analysing representational languages of this class. What these tools analyse is not the behaviour of an interpreter, or the structure of processes in some running system, but rather, the extensional meaning of expressions of a language, when these are taken to be making claims about some external world.

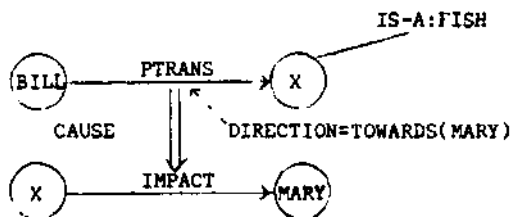
These two distinct topics - the meaning of a language and the behaviour of an interpreter for it - are related in various ways. They meet in particular, in the notion of inference. Logical meaning justifies inferences. A running system performs inferences: some of its processes are the making of inferences.

But two different systems may be based on the same notion of inference and the same representational language. The inference structure of the language used by a system does not depend on the process structure. In particular, a system may have a logical inference structure - may be making deductively valid inferences - without being a classical uniform theorem-prover which just "grinds lists of clauses together".

What logic is: the extensional analysis of meaning

One of the first tasks which faces a theory of representation is to give some account of what a representation or representational language means. Without such an account, comparisons betweer representations or languages can only be very superficial. Logical model theory provides such an analysis.

Suppose it is claimed that:



means that Bill hit Mary with a fish (to take a representative example), or that:

((DO(^AGENT)^BADTHING))CAUSE(^AGENT)DISPLAY
(tfNEGATIVEEMOTION)))

means that people often seem upset when bad things happen (to take another). How could one judge whether they really do mean those things? What would count as a specification of their meanings? Several answers can be suggested.

The first might be called "pretend-it's-English". Here, one takes the primitive symbols to stand for their ordinary English meaning, and gives a way of translating the grammar of the representation into English surface syntax (this is often left implicit but fairly obvious). The first example above then is to be read as something like "Bill moved some object - which was a fish - in the direction of Mary, thus causing the object to make an impact upon Mary". One now has to judge whether this English sentence has the same meaning as the original English sentence ("Bill hit Mary with a fish"). The English rendering of tl.e second example is even more obvious.

This way of analysing meaning has the virtue of simplicity, and it also requires very little technical expertise. It is widely used in modern

linguistics, where it often goes hand in hand with the assumption there is some finite collection of basic words in terms of which the meanings of all sentences can be explained.

But there are many problems with this simple idea. For a start, it's perilously vague. It's always hard to judge whether two English sentences have the same meaning. It depends what you mean by "meaning" - with a very tight sense of'meaning", the sentences "John hit Mary" and "Mary was hit by John" are different in meaning. Second, it's an essentially linguistic view of meaning. While this doesn't bother many workers in the natural language area, it should bother anyone who believes that at least some knowledge representations need to be independent of any particular sensory bias. (We can all look at a scene and describe what we see. How is information transferred from the visual recognition process to the linguistic representation?) Much of what a vision program needs to represent may not be readily expressible in English (e.g., 2-dimensionalpatterns of light and shade). Third, it provides no useful guidelines for how a system might use the representation. Given that the network of the first example is supposed to mean the same as its anglicisation, does anything follow concerning what inferences can or should be made from the network?

This last point is really a symptom of the most basic problem, which is that on this account we could just as well use the English sentences themselves as their own representations. The symbols in the formalism might as well be English words. (Wilks [20] states this explicitly.) Until some independent account of the formalism is provided, no actual analysis of meaning is forthcoming.

The model-theoretic approach to meaning interprets an expression of a formalism as making a claim about the way the world is. Suppose we give some criteria by which we can judge whether a suggested possible world satisfies the expression, or whether on the contrary it is a counterexample to the claim made by the expression. Then these criteria can be used as an account of meaning. An expression means what it claims about a possible world. Two expressions which are satisfied by the same possible worlds are identical in meaning. A natural notion of inference follows also. If every counterexample to E1_ is also a counterexample to E2, then we can infer *E*± from E_2: for then all the possible worlds which are consistent with the claim we make when E_2 is asserted also satisfy Ei.

Notice that on this account an expression can usually not be said to definitely correspond to anything in tire actual world. Its meaning is fixed only with respect to a possible world. In order to pin down its meaning (we should say 'referent') more precisely in the actual world, we must add more assertions so as to cut down the set of possible example worlds. Take for example the expression "MARY", which is intended to denote a particular lady in the real world. In order to achieve this identification, we would have to asser assert enough axioms containing the expression "MARY" to ensure that in any possible world satisfying them, the denotation of "MARY" corresponded to the particular lady in question in the actual world. These axioms will contain other names and relations symbols, and we cannot in general say conclusively that any of these is defined in terms of some particular subset of the others. The entire web of logically connected assertions is presumably tied down to the actual world by some of them having an interpretation as observations, in the case of an actual robot with these beliefs in its head. On this account, perception is a form of inference: inference which involves observational assertions. (This is not to say that we can deductively derive beliefs from observations, which is of course not time in general. The required relationship is consistency: beliefs must be kept consistent with observations.)

This model-theoretic account of meaning corresponds exactly to Bobrow and Winograd's |_2] view that "a description ... cannot be broken down into a single set of primitives, but must be expressed through multiple views" and "... there would be no simple sense in which the system contains a 'definition' of the object, or a complete description in terms of its structure". Their subsequent remarks suggest, however, a confusion between the logical notion of meaning and the pretend-it's-English notion using "primitives".

The problem with this approach to meaning is, of course, to specify what we mean by a possible world in such a way that we can state the meaning criteria - the truth-conditions as they are usually and somewhat misleadingly called. First-order logic makes only very elementary assumptions. A logically-possible world is a set of individuals (each name denotes some individual) and a set of relations between them (each relation symbol denotes some relation). The rules for deciding which worlds are examples for an expression and which are counterexamples, are well known, yielding the usual notion of deductive inference.

Model theory, unlike pretend-it's-English, gives an account of extensional meaning relative to an exact notion of possible world. One might object that this notion is mistaken. Perhaps the real world isn't like that, does not consist of individuals with relations between them. Certainly this notion of world seems too simple. Are liquids individuals, for example? Either answer (yes or no) gives rise to certain problems. There is much scope for ingenuity in giving precise descriptions of more interesting classes of possible worlds. It would be interesting to see a class of worlds in which there was a fixed notion of causality, for example [8] , Notice how such an enterprise would differ from the 'analysis' of CAUSE provided by pretend-it's-English. The latter yields no account of what a causally possible world would be like, nor does it explain what constitutes causally valid inference.

An important property of the model-theoretic account is that it enables one to judge a proposed representation by imagining the circumstances which would render it true. Of course this is only a heuristic remark, but I find that it is an important feature. One way to test a proposed representation is to run it, if possible on a computer, but perhaps only in a pencil and paper sense, i.e., write down some formal consequences of it using whatever inference structure comes with the

representational language. But this does not always generate insight into errors or inadequacies of the representation, because a characteristic symptom of such a situation is that nonsense becomes derivable, or alternatively that nothing useful is derivable at all, neither of which is very much help. Another way to test it however is to attempt to understand it as a description of a world, and to imagine what the world would have to be like to make it false. I find the latter the most useful.

For example, suppose one is trying to formal ise knowledge about liquids, and one writes something like

```
IN(LIQUID, CONTAINER) &MOVES(CONTAINER)
=>IN(LIQUID.CONTAINER)
```

Is this a reasonable assertion? In order to answer that question, one would at least have to say whether it were usually true. What would the world have to be like to render it false; what would be a counterexample? Well, what does it mean? It's not clear, since we have no model theory. Presumably IN is a relation, but is MOVES then a relation? The intention behind this semiformal axiom can be crudely expressed thus:

```
IN(LIQUID,CONTAINER,STATE)
=>IN(LIQUID,CONTAINER,MOVE(STATE))
```

where MOVE is a function from states to states. Now the ontology is clear, anyone who has picked up an overfull cup of coffee can easily imagine a counterexample. Without a model theory - albeit perhaps an informal one - we would not be able to so connect expressions of the formalism to possible configurations of a world that it would even be possible to imagine such counterexamples. A formalism without a model theory can hardly be said to constitute a representational language at all.

None of these basic semantic ideas say anything about the syntax of the expressions used to encode facts. The same meanings can be expressed in a wide variety of syntactic forms. There is thus an a priori possibility that some already existing language may be best interpreted as another syntax for predicate (or even propositional) calculus. "Semantic networks" are a good example, as several recent writers have observed (see for example Woods [2] and Schubert [15]). If someone argues for the superiority of semantic networks over logic, he must be referring to some other property of the former than their meaning (for example, their usefulness for retrieving relevant facts from a database - an aspect of a possible process structure - or their attractive appearance on a printed page). A more recent example is KRL. Virtually the whole of KRL-0 can be regarded as merely a new syntax for first-order predicate logic.

Now it must be admitted that sometimes semantic networks (for example) are used in ways which do not reflect their obvious logical meaning. For example, there is often a sort of implicit uniqueness condition which prevents two nodes from denoting the same entity in any interpretation. Without such a condition, for example, the 'pedestal' network of fig. 1 would be merely an instance of the 'arch' network, got by identifying B2 and B3 (much as P(x, x) is an instance of P(x, y) in the usual syntax).

Similarly, frames are a syntax which have been used to convey a variety of meanings. They can be understood as a strange syntax for logic in at least 2 distinct ways (either frames are objects and slots 2-place relations, or frames are n-place relations), they are used in GUS [11] to represent conversational sequencing, in EVIL \H] to represent perceptual hypotheses. One syntax, four different meanings.

Representation and control

Almost every idea on representation in AI has eventually appeared in the guise of a programming language. This is in part the legacy of the procedural/assertional debate, which was won fairly conclusively by the proceduralists. It is worth, however, going back over the old history of this dispute rather carefully, as the ground of the argument has shifted subtely but significantly over the years.

Classical theorem-proving operated in the general problem-solver paradigm. This takes the form of a competitive game between he who designs the theorem-prover and he who provides the axioms on which they are tested. The aim of the game is to write theorem-provers which can solve really hard problems, and which are general. To cunningly adapt the axioms so that the theorem-prover is able to prove the theorem is cheating and is frowned upon, like cheating at cards. Moreover, the theorem-prover, being general purpose, has no bias to any particular domain. The result is that classical theorem-provers know very little about what to do, and are incapable of being told it.

This was the position the proceduralists attacked, and their argument was, I think, conclusive. It has to be possible to tell a system what to $2/ what inferences to make and when to make them (and not to make them), as well as what is true. In a word, a system has to be programmable.

Contrast the problem-solver methodology with the programming language designer's methodology. The latter does his best to make the workings of the language interpreter available, or at least visible, to the user, even to the extent in some cases of writing a manual (the ultimate anathema for problem-solvers: a handbook for cheats). The difference is ultimately one of where the responsibility for a system's behaviour lies: the problem solving system designer retains it, the programming language designer gives it to the user, to the person who composes the knowledge representations. What more natural, then, than to regard a representation language as a programming language?

It is important to emphasise this contrast of methodologies because it is the only significant difference between the proceduralist position, as it was argued in the early seventies, and the traditional theorem-proving view. In particular, the procedural languages, offered in this period as replacements for logic, have very similar inference structures to predicate calculus. The proceduralist's own remarks about how to represent facts reiterate the basic semantic intuitions of formal logic (see Winograd [22] for example). The inference structure of uPLANNER is a subset of predicate

calculus, augmented with THNOT. Even the newer languages, such as KRL, based on different and apparently rival intuitions (see Minsky's broadside in [I2]) display some remarkably logical features.

Nor is there any important difference in underlying mechanisms of implementation. The basic, and quite old, mechanism of an and/or tree with variable-sharing across and nodes, implemented using invocation records with separate access and control links and local environment bindings, underlies theorem-proving programs, AI programming interpreters, production systems and ACTORS \9*j* .

But this methodological difference runs very deep, and does have technical consequences. A procedural language to represent knowledge has two distinct tasks to perform. It must encode facts and inferences about external domains (and hence have some kind of inference structure which we might try to analyse using logical tools); and it must also express strategies of behaviour for its interpreter to obey, some of which will presumably be strategies of inference. It will have both an inference structure and a process structure, both usable by the programmer. This is a tall order, and nobody has managed to build a satisfactory such language yet. There have been essentially three ideas on how to do it.

The first idea is to specify control by the way in which one states the facts. Supposing that there are a few predefined strategies which the interpreter can use to process an assertion: then one provides the user with just enough syntactic variants for stating facts to enable him to implicitly tell the interpreter which strategy to use. This is the uPLANNER idea (THCONSE and THASSERT), also underlies Kowalski's more recent proposal to treat predicate logic as a programming language [II], and has been used by some "natural deduction" theorem-provers which find it much easier to prove AoB than AVB. But this idea is far too inflexible: one rapidly finds that one wants to specify behaviours which cannot be encoded as some simple combination of the predefined strategies.

The natural reaction to this situation is to build systems which provide the necessary machinery but make few commitments as to how it should be used. To build systems, that is, in which specialist interpreters can be implemented. This is the second idea. CONNIVER has just this relationship to pPLANNER, for example. CONNIVER was a toolkit for implementing PLANNER-like systems and, more usefully, for experimenting with coroutine control structures. The KRL authors similarly insist that a representation language "must provide a flexible set of underlying tools, rather than embody specific commitments about either processing strategies or the representation of specific areas of knowledge", [2], page 4.

But what then happens to the inference structure of the representational language? We have now moved to a lower conceptual level, the level of the interpreter rather than the level at which substantive claims about some domain are made. What we now have is purely a programming language, and not a descriptive language. The objects which, in a descriptive language, would be meaningful assertions or descriptions or names - meaningful if the exact sense that their relationship to a possible external world was defined by the meaning of the language - these objects appear merely as data structures in an interpreter-implementing language. And of course it is part of the philosophy of programming language design that the interpretation of what a data structure mean8 must be left to the programmer.

Going down a level thus renders vacuous the original claims of the proceduralists with regard to representation. To argue that CONNIVER is better than predicate calculus is to compare incomparables. CONNIVER is about processes and their behaviour: logic is about assertions and their meaning. CONNIVER is one of the programming languages one might use to implement a system with a logical inference structure (or indeed any other structure).

There is still a procedural problem, in any case. The interpreter-defining language has to be based on some control regime. CONNIVER and INTERLISP use coroutineing, for example. But whatever control regime is used, at this level it is deterministic code - a sequence of instructions - which actually runs.

A widespread dissatisfaction with purely procedural languages stems from the feeling that procedural code is too rigid a language to express interesting behaviours (see [2], page 36, for example). One can use "pattern-directed invocation" (i.e., resolution*), or "procedural attachment", or whatever, to make a more sensitive choice of which procedure to run: but when that choice has been made, deterministic code is found in its body. It all gets down to LISP in the end. Using the CONNIVER (GEDANKEN, PAL ...) idea of frozen process states allows a certain amount of freedom: but still we have the feeling that control is like a baton being passed from hand to hand. If one process doesn't know who to hand it to, everything comes unstuck. All runnable code, while running, has total responsibility for keeping the whole system alive.

While this does make some ingenious programming possible, especially when combined with a database of assertions used to 'simulate' a world (see Fahlman [17] for a beautiful example), it still lacks the flexibility and opportunism which we need.

We need to have several coexisting processes, each acting for itself without needing to be explicitly called from some other process. The obvious idea then is some form of multiprocessing, where the interpreter maintains a queue of processes and runs them all from time to time, according to some strategy. This is the third idea. Calling a process is putting it on the queue. This makes apparently hard code, like: beginF(); G(); H() end; into something much softer, since exactly what will happen depends on what other processes there are around. When F is called, that doesn't mean that it's actually called, only that it's put on the agenda.** Maybe some other process

* Resolution is an inference rule, not a "strategy" or a "method".

** agenda = queue. A lot of impressive renaming goes on in this business. For example, good old environments appear in KRL under the titles 'procedure directory' and 'signal path'.

will run first and flush F before it has a chance to run, for example.

This very old (c.f., Elcock and Foster [6]) idea is currently popular. But we have now come full circle, to a classical problem-solving situation. How can the interpreter decide what order to run the processes in? It doesn't know anything about any particular domain, so it can't decide. So we have to be able to tell it. But how?

This is exactly the situation with which we began, the situation the proceduralists attacked. In removing the decision to actually run from the code and placing in in the interpreter, advocates of multiprocessing systems have re-created the uniform black-box problem-solver.

The next step then is to design a language in which the programmer can control the agenda. The simplest such idea is to use numbers: the agenda has levels numbered from zero, and process calls specify their level. This is used by KRL-0 and the Graph Traverser [Y]. A somewhat more sophisticated idea is to allow descriptors for subqueues and allow processes to access these descriptors, as in POPEYE *[I6]*. But none of these ideas seem very convincing. And we have now moved down another level, to the interpreter of the interpreter-writing language of the representational language.

The only way out of this descending spiral is upwards. We need to be able to describe processing strategies in a language at least as rich as that in which we describe the external domains: and for good engineering, it should be the same language. The aspects of procedural languages - THNOT of uPLANNER, passing context frames as parameters in CONNIVER, defaults in KRL - which resist simple syntactic mappings into logic, are all places where the languages refer to their own interpreter's behaviour. THNOT means not provable (from current resources): passing a context frame is proving something about another proof; a default value is one which is taken unless there is a proof that its value is different. It is this reflexive nature of these languages which gives them their 'non-logical[1] features.

But this is a question of what knowledge is represented, not of what language it is represented in. These reflexive assertions, referring to the system's own internal states, can also be expressed in logic, with the same gains in ontological clarity as are realised in other areas. This distinction between logic and procedures is then seen as a distinction between kinds of domain rather than kinds of language: the proceduralist position leads one to envisage a system which can describe its own inferential processes and thus make inferences about its own behaviour.

In order to design the interpreter for such a system, one needs a framework in which these behaviours can be adequately described. Logic provides - in the notion of proof - a richer such framework than any of the usual procedural ideas.

What logic isn't

It's worth spending a little time laying to rest some misunderstandings I've met about logic.

(1) Logic isn't a programming system.

(2) Logic isn't a particular syntax.

(3) Logic does not assume that the world is made up of concrete physical individuals without "abstract" individuals such as properties, events, nations or feelings. This view is nominalism, and leads to a quite different sort of semantic intuition, in which, for example, red denotes not a property of physical individuals, but the (rather disconnected) individual consisting of all pieces of red stuff in the world.

Other similar confusions are also made. For example, logic is no worse (and no better) than Conceptual Dependency at representing warm, human facts about people hitting each other,

(4) Logic doesn't give "the ultimate in decomposition of knowledge". Winograd, in his widely cited discussion [23] of the assertional/procedural controversy, draws a distinction between logic's atomistic view of knowledge, in which a representation is seen as a set of separate disconnected facts, and the proceduralist's holistic view in which interactions between procedures have prominence. But this is exactly the opposite of the truth. The interactions sanctioned by logic between assertions are far richer and more complicated than the interactions between procedures in a procedural language (any procedural language). Thus, explicit recursive procedure calls (LISP) are more restricted than explicit coroutine calls (SIMULA), these more restricted than pattern-directed coroutineing (CONNIVER), these more restricted than resolution (which allows both caller and callee to have variables bound during the matching process) and finally resolution itself is a special case of general logic inference rules of instantiation and cut. In each case, one pattern of interactions is a special case of, and can be imitated by, the next. In each case, the more general interaction pattern allows more interactions and hence yields a more complex search space, and a more difficult search problem. It is precisely the restrictions on interactions in procedural languages which make them so useful.

Again, Winograd claims that procedures, unlike assertions, mean very little in isolation but acquire meaning from their interactions with other procedures; and again has got it exactly the wrong way round. A procedure may well mean a lot in isolation. RANDOM(), for example, or PRINT(X): any procedure whose body contains code but no calls of other procedures. Whereas the function and predicate symbols in a logical axiomatisation, like the tokens at nodes of a semantic net, literally mean nothing unless their meaning is specified by axioms. The model-theoretic account of meaning makes this absolutely precise; as one conjoins assertions, so the set of interpretations possible for the symbols occurring in them is restricted, and the set of possible inferences from them is enlarged. Their meaning is progressively tightened, as more facts involving them become inferrable.

(5) The tendency to replace representational languages by purely procedural languages goes hand in hand with a tendency to judge representational issues in computational terms. Thus Minsky 12) in attacking what he sees as the malevolent influence of logic, dimisses predicate calculus by observing that the machinery of P.C. inference - instantiation and tree-growing, basically - is available as a

simple byproduct of the more sophisticated symbol-manipulation operations needed for analogical reasoning. But this, while perhaps true, misses the point: it is the meaning of those operations, interpreted as inferences, of which logic provides an analysis.

Again, Winograd [23] identifies the procedural /assertional distinction with the program/data-structure distinction, a completely false analogy. The latter distinction is to do with two different relationships a piece of data can have to an interpreter (including, ultimately, the hardware CPU): the former with the meanings of those structures. An assertion can be treated as a datastructure or interpreted as a program, just as a procedure can. The distinctions are orthogonal.

Last word

I have argued the case for taking logic's notion of meaning seriously. I do not, however, wish to argue that this is the only important issue in considering representational languages. Process control is important, of course: questions of ease of retrieval, of focussing of attention, of relevance, are also of great significance. Neither is syntactic convenience completely unimportant. These issues are however all receiving considerable attention already. Semantics - questions of meaning - tend to be discussed less.

Acknowledgements

I have had helpful conversations and correspondence with Bruce Anderson, Richard Bornat, Eugene Charniak, Jack Lang, Bob Wielinga and Yorick Wilks. Alan Bundy, Aaron Sloman and Terry Winograd made useful comments on the first draft of the paper. This work was supported in part by the Science Research Council.

References

1. Bobrow, et al (1976), GUS, a frame-driven dialog system, Xerox Palo Alto Research Center.

2. Bobrow and Winograd (1976), An overview of KRL, A Knowledge Representation Language, Xerox Palo Research Center.

3. Brady and Wielinga (1977), "Reading the writing on the wall", Proc. Workshop on Computer Vision, Amherst, Mass.

4. Brooks and Rowbury (1976), An EVIL primer, Memo CSM-14, Essex University.

5. Doran and Michie (1966), "Experiments with the Graph Traverser program", Proc. Royal Society (A) 294, pp. 235-59.

6. Elcock and Foster (1969), "ABYSY 1: an incremental compiler for assertionsi an introduction", Machine Intelligence 4., Edinburgh University Press.

Fahlman (1974), "A planning system for Robot Construction Tasks", A.I.J. 5, pp. 1-49.

Hayes (1971), "A logic of actions", Machine Intelligence 6, Edinburgh University Press.

Hewitt, Bishop and Steiger (1973), "A universal modular ACTOR formalism for artificial intelligence", Proc. 3rd I.J.C.A.I., Stanford.

10. Huet (1972), Constrained Resolution, report 1117, Case Western University.

11. Kowalski (1975), "Predicate calculus as a programming language", Proc. I.F.I.P. 75.

12. Minsky (1974), "A framework for representing knowledge", Memo, M.I.T. A.I. Lab. (unexpurgated version).

13. Pietrykowski and Jensen (1973), "Mechanising w-order type theory through unification", Report CS-73-16, University of Waterloo.

14. Schank (ed.) (1975), Conceptual Information Processing, North-Holland.

15. Schubert (1975), Extending the Expressive Power of Semantic Networks, University of Alberta.

16. Sloman (1976), (personal communication).

17. Sussman, et al (1970), Micro-planner reference manual, M.I.T. A.I. Memo 203.

18. Sussman and McDermott (1972), "From PLANNER to CONNIVER-- A genetic approach", A.F.I.P.S. Fall Joint Computer Conference.

19. Anderson, et al (1972), After Leibnitz, A.I. Memo, Stanford University.

20. Wilks (1975), "Primitives and Words", Proceedings of the Conference on Theoretical Issues in Natural Language Processing, Association for Computational Linguistics

21. Wilks (1973), "An artificial intelligence approach to machine translation", Computer Models of Thought and Language, W.H. Freeman.

22. Winograd (1972), Understanding Natural Language, Academic Press.

23. Winograd (1975), "Frames and the declarative-procedural controversy", Representation and Understanding, Academic Press.

24. Woods (1975), "What's in a link", Representation and Understanding, Academic Press.

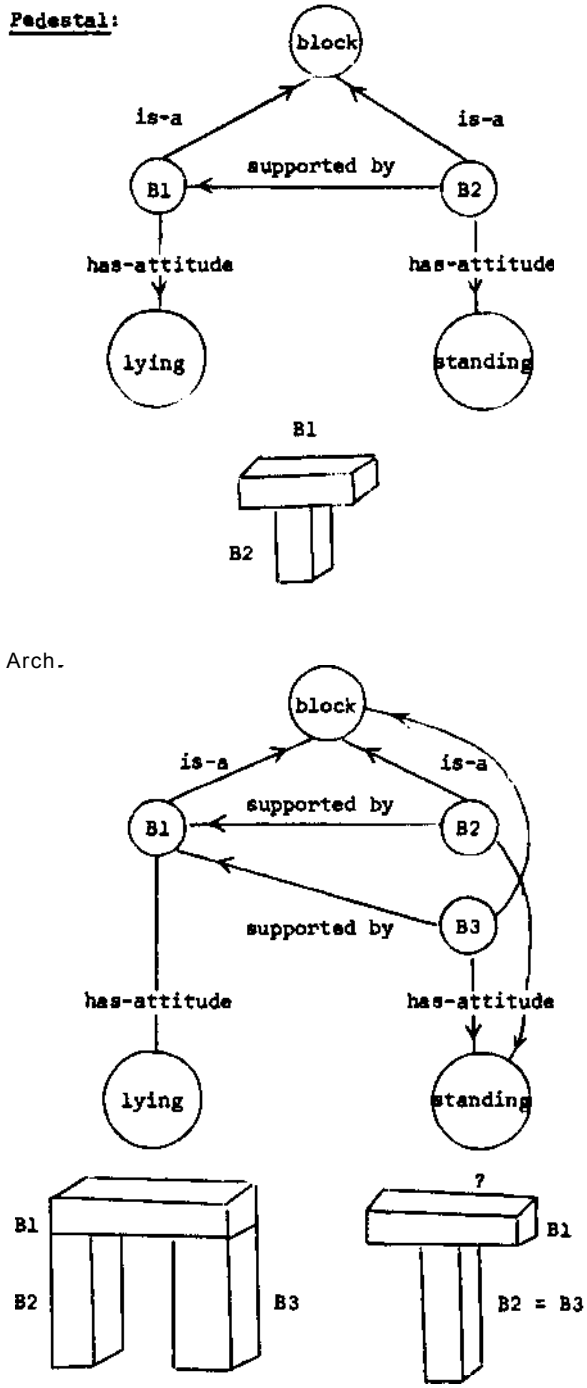


Figure 1

A pedestal is an arch