WORKING PAPER 58

FUNCTIONS AND FRAMES
IN THE LEARNING OF STRUCTURES

by

MICHAEL J. FREILING

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

December 1973

Abstract

This paper discusses methods for enhancing the learning abilities of the Winston program, first by representing functional properties of the objects considered, and secondly by embedding individual models in a hierarchically organized system to provide for economy of recognition. An example is presented illustrating the use of these methods.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advance Research Projects Agency of the Department of Defense and monitored by the office of Naval Research under Contract Number N00014-70-A-0362-0005.

Working Papers are informal papers intended for internal use.

Table of Contents

1. Introduction p.3

2. Function p.7

2.1 Motion p.11

2.1.1 General Motion Primitives p.13

2.2 Holes p.17

2.2.1 Hole Types p.18

2.3 A Formalism For Function p.21

2.4 Use of functional representation p.24

3. A Hierarchically Structured Knowledge System p.29

3.1 The Individual Test-Frame p.34

3.2 Utilizing the Test-Frame Structure p.38

3.3 Constructing and Debugging p.42

3.3.1 Local Debugging Rules p.43

3.3.2 General Debugging Techniques p.49

3.4 An Example p.50

4. Grouping p.58

5. Conclusions p.63

References p.64

1. Introduction

The structural description learning program of Pat Winston <6> achieved great progress in answering questions concerning possible mechanisms for acquiring the ability to categorize and compare objects. His program was capable of building general descriptions of classes of objects, understanding differences between descriptions to the extent that it could solve simple analogy problems, and in some cases dividing a scene into component structures. The limitations of his work however, raise a number of other interesting questions.

First of all, his program described objects only in terms of their structure. Such one-sided description has inherent limitations in that

(a) It does not provide for recognition of objects on the basis of less concrete criteria, say the uses to which they may be put.

(b) It does not permit generalization to deal with classes which may be different structurally but similar in other respects. For example, the class of possible supports for a television set includes tables and shelves, which share little in the way of common structural properties.

(c) In many practical problems, it is of great advantage to have many ways of describing a particular object. Limitation to one specific mode of description severely restricts the class of problems which may be tackled.

These limitations suggest that we search for other ways to describe objects aside from pure structural form, and attempt to understand the relationships between different ways of describing objects. The richness of such interrelationships should provide us with useful ways to

accelerate our ability to understand the objects, as well as greatly expand our ability to use such descriptions effectively.

Secondly, while Winston's program was quite capable of understanding the differences between particular items or classes of items in a given domain, the descriptions were not organized in a systematic fashion capable of representing any structure inherent in the domain. For instance, there is no provision for economically representing subclasses of given structures. Understanding the structure of a domain should enable us to

(a) classify items much more easily,

(b) propose more refined theories about the structure of the domain,

(c) provide valuable ideas on the nature of learning since much of learning and problem solving is concerned with exploring and comprehending the inherent structure of general classes of "problem spaces" <4>.

This paper represents a collection of ideas and proposals aimed at providing first steps in overcoming both these limitations. It should be stressed that most if not all of the ideas are tentative and in a state of flux. They are not meant to be as yet a complete or consistent theory of general types of representation or learning of structures of various domains.

The particular domain I have chosen to discuss is a slight extension of Winston's domain--that of structures built from blocks. The ideas of functional description which will be elaborated are of course, very specific to the blocks world domain, though it is reasonable to expect that in some modified form they will have applicability to the problem of

recognizing real physical objects. The ideas of structuring a recognition system, however, are far less domain-dependent, and should find application in almost any recognition scheme.

With respect to the problem of representing aspects of objects other than form, it seems appropriate to make an attempt at describing objects on the basis of their function. There are several reasons for this:

(a) It may be possible to construct simple representations of the functions of objects in the blocks world in terms of simple concepts of motion and areas of unoccupied space.

(b) Many structures in the blocks world have real-world counterparts which are classified in actuality on the basis of function. An arch is principally something we can pass through. A table is principally a structure we can put things on.

(c) There are many relationships between the form and function of objects. Since the possible functions of a class of objects are generally much simpler to enumerate (assuming we have the proper tools!) than the possible structures, functional description enjoys the advantages of more concise forms of representation with a corresponding increase in our ability to manipulate overall descriptions of objects.

As far as the problem of organizing descriptions is concerned I am proposing a system constructed on the lines of general vision frame systems discussed by Minsky <3>. Such systems have several advantages. They have a well defined inherent structure, they provide for economic representation of information, they are relatively easy to patch locally, and they are simple to construct in a step-by-step fashion. Furthermore, frame-like systems posess a generality which makes them applicable to many domains.

It should be mentioned here that I have felt it desirable as a first step to divorce this work from problems of vision or sense.

Consequently, It is assumed that all structures presented are described in a "God's eye" view, that is, such descriptions are complete in the sense that nothing is obscure or hidden from the viewer, and that the descriptions contain no information which is applicable only to a specific viewing position. In addition, the descriptions are assumed to contain complete information about the attachment of blocks to each other. Such data could be gathered by a machine which attempts to move different parts of a scene it is viewing. It is hoped that these simplifications will provide a basis for fuller understanding of the relationships explored, which understanding may later, perhaps, be applied to aid in the solution of vision and sense problems.

For the reader who desires merely an overview of the work described, it is suggested that the example of section 3.4 be perused and the first sections of chapters 2 and 3 be read.

2. Function

When we classify objects in everyday life, we generally do so on the basis of their function. When one sits down at a desk, for instance, one is not usually concerned with the shape of its legs, or the number of drawers, but more directly with the fact that it has a flat surface to write upon. When looking for a hammer one doesn't care if the handle is round or square or octagonal, or if one side of the head is a claw or a ball. What we want to know is, is one side of the head flat enough that we may use it to drive a nail, and is the handle long enough to provide sufficient leverage? If one were to worry about classifying the hammer down to the last irrelevant detail, one would waste inordinate amounts of time doing such calculations. Rather, it makes more sense to concentrate on those specific properties which are especially pertinent to the desired function. Classification of an object by its functional features provides us with a computationally useful tool for quickly finding the objects we desire.

For this same reason, it seems quite desirable that any computer program to deal with classification of objects on a more than trivial level should be capable of providing representation of a class of objects by their function, or by the specific properties directly relevant to their function. This is not to say that detailed structural descriptions are not desirable, but that functional representations will generally expedite computation, even in cases where we may later desire to examine

the detailed description.

Consider for example the problem of finding all instances of an arch in figures 2.1 and 2.2. When working on 2.1, Winston's program first groups the three basic arches (i.e. those whose support consists only of a single block) and then groups each bottom arch with a support (E with ABC and F with GHI) to form a fourth "generalized arch," since any arbitrary structure may be a support for the arch. Following this logic, there are 15 possible matches for arches in figure 2.2, which one might expect the Winston program to produce. Although such fine distinction may be necessary in a particular puzzle domain ("Find an arch in this scene such that the number of blocks in the arch is one less than the number of blocks in the scene which are not in the arch."), and even desirable to be able to make, one would certainly balk at the prospect of carrying around descriptions of 15 different arches every time one encountered this configuration in some larger scene. In any application where one plans to utilize the arch for some purpose, one would expect to be almost exclusively interested in the arch containing all blocks in the scene. Especially so in a domain of constructed objects, where one might expect all the blocks in 2.2 to be attached to each other, thus rendering it impossible to physically isolate any of the other arches. (Consider the prospect of having to find all arches in a brick wall!)

As a step towards resolving these problems, I would like to propose a scheme for representing block structures on the basis of properties closely related to their function. This scheme will rely principally on the ideas of motion and holes in achieving its goal. The next section

Fig. 2.1

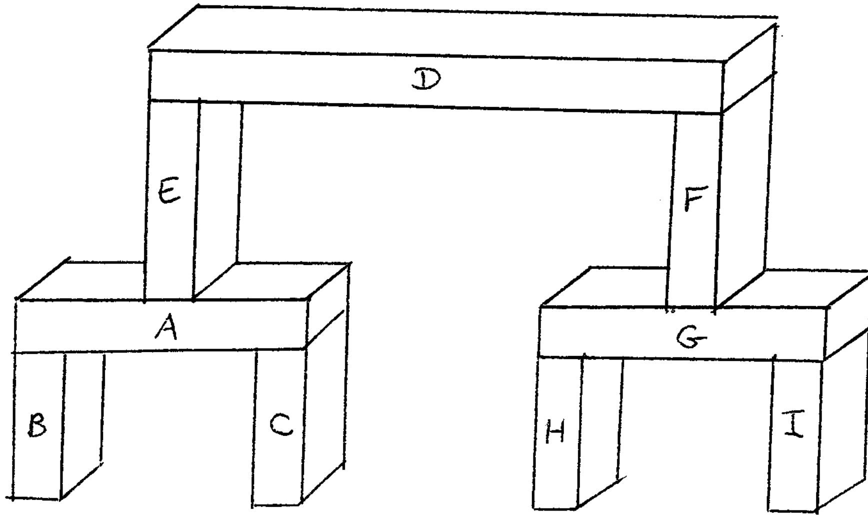
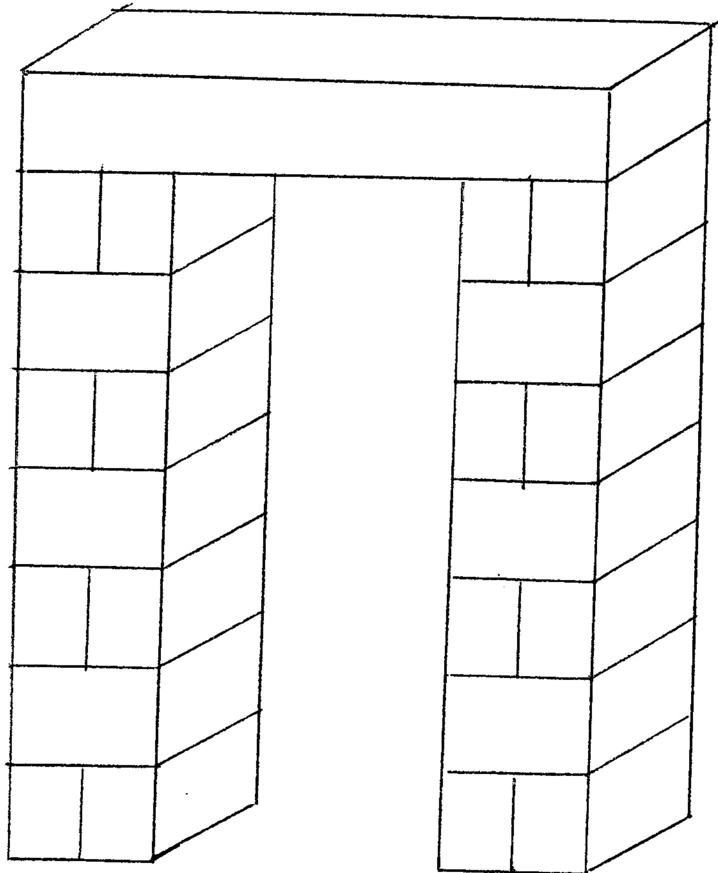


Fig. 2.2



will discuss my ideas on motion and after that I will say a little about holes. In the third section a formalism for representing function will be discussed, and finally, the possible use of such descriptions in a learning program.

Before that however, let us speculate about a possible solution to the arch problem mentioned above. A loose functional definition of an arch might be "something one can walk through." Although this is not sufficient to define our arch (one can walk through Building 10, for instance), if we couple it with some model of an arch, we can immediately discover the essential property of the arch. That is, the two supports do not touch. We might use this information to model an arch functionally as a hole surrounded on three sides, representing the hole as a block of air. Applying this to 2.2, we would find one hole and thus, only one arch need be considered when using functional criteria. This last qualification is important, because it is conceiveable that in some cases we would prefer to invoke a more form-oriented arch finder. However, the speed and simplicity with which the function-oriented representation may be used recommends it as a strong heuristic. Applying our function-oriented representation to figure 2.1, we encounter a slightly different problem. The two holes corresponding to arches A-B-C and G-H-I may be found easily. The central hole, however, might be considered slightly ambiguous. If we are choosing to represent holes as blocks of air, we see that the central hole may be represented either as one T-shaped block or as one brick of air "lying" atop another brick of air. Consideration of the former would yield the generalized arch while

consideration of the latter will yield arch D-E-F surrounding hole alpha (figure 2.3). Beta will not have a corresponding arch because it is not "covered" by a solid object. There may be some question about whether we should consider D-E-F to be functionally an arch or not. As will be discussed later, this will ultimately depend on other functional criteria such as whether there is attachment between E and A and/or between F and G.

2.1 Motion

In the domain of objects constructed out of blocks, most of the functions one considers seem to be contingent upon the idea of motion or the restriction of motion. The hole is the principal part of the arch because it enables the arch to achieve its function, i.e. one's ability to move through it. The concept of support, or potential support, which may be considered the function of objects like a pedestal or table, may be simply defined as the restriction of motion in a downward direction. A wall may be considered as a structure which prevents one from proceeding in a given direction unless a detour of some sort is taken. It seems clear that if we are going to deal with such functions in a program, we should have a set of primitive concepts with respect to motion available for use. This section will consider a set (which is by no means to be thought of as complete) of such primitives as a basis for the study of functional representation.

The motion we will initially be primarily concerned with will be motion in a straight line, or a sequence of straight lines. We will probably want to consider the motion of three types of objects: (a) that of a bird, which may move in any direction; (b) that of a ball, whose motion must have no purely upward components; and (c) that of a creature with legs, for whom the vertical components (e.g. climbing a staircase) must be suitably small.

It should be noted here, that this definition of "suitably small" is

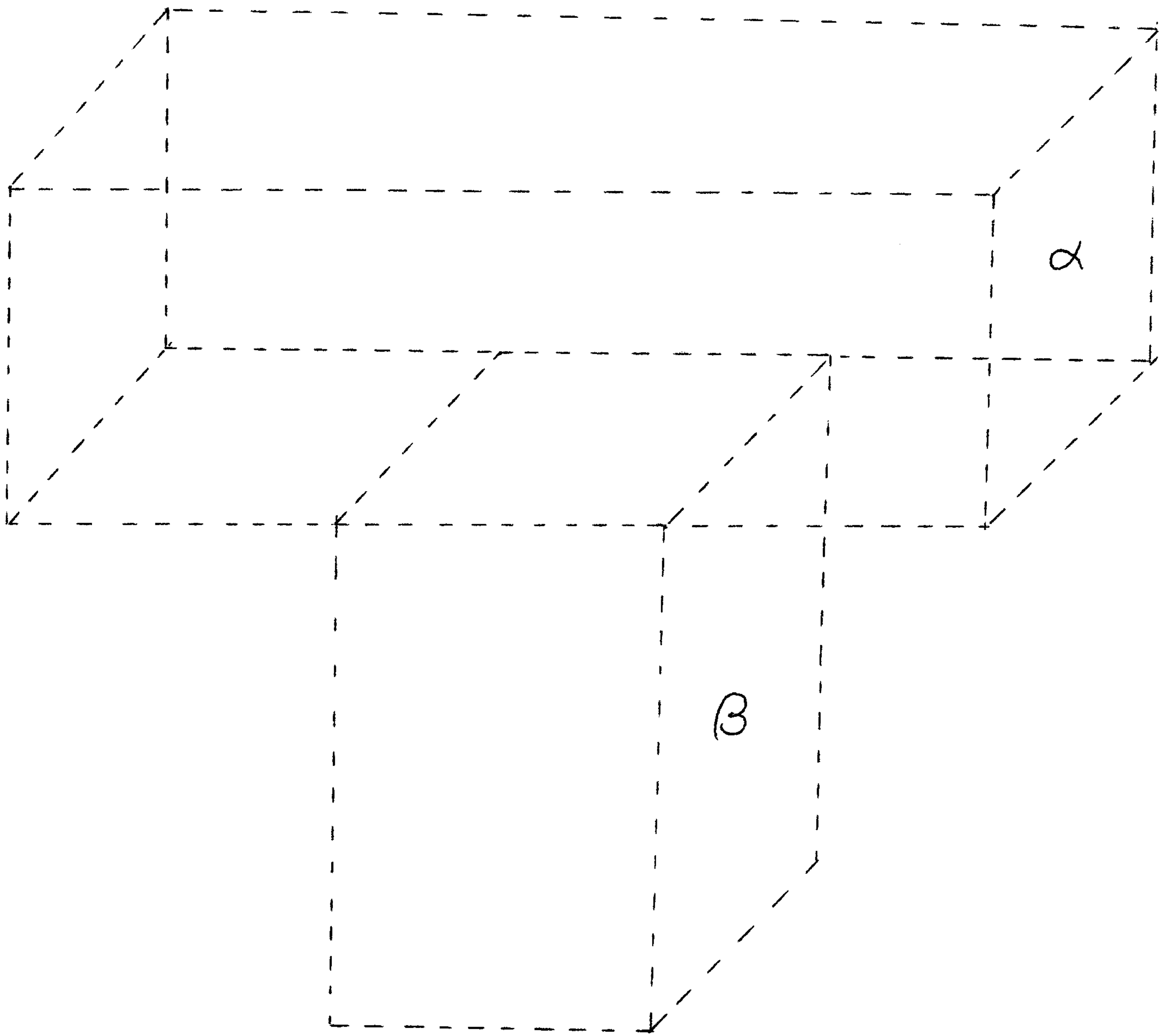


Fig. 2.3

of necessity rather vague. There will be many other similarly relative concepts mentioned later, in conjunction with ideas about containment, windows, doors, etc. At the present there does not seem to be a general, systematic method for handling these concepts adequately. The fuzzy logic work of Lakoff <2> and Goguen <1> is not very satisfactory. What is more desirable for our limited application is a set of very simple rules for determining a threshhold condition with respect to a given context. Fuzzy Logic work generally tends to be quite complicated and ignore the importance of context. For the time being, I intend to use simple ad hoc and perhaps slightly arbitrary conditions, e.g. "Sufficiently small vertical components with respect to a moving object will be less than one fourth the height of the object." Since the moving object is hypothetical, (and thus imaginary) its dimensions will be determined from the structure under consideration, i.e., it must be small enough to get through the doors.

One can see that these semi-quantitative concepts like "sufficiently small", as well as the proposed representation of holes as blocks of air (see section 2.2), will require descriptions at some level to be more quantitative in terms of dimension than the purely qualitative relations handled by the Winston program. In a hierarchically organized knowledge system (see chapter 3) these should naturally be placed at the lowest (i.e. most specific) frame level and invisible at any higher level, where the concepts of relative size and position may be introduced. The information will then be available for access, but need not clutter up higher level comparisons. Furthermore, such specific information need

only be present in a specific scene being viewed. It can be completely eliminated from most models the system will choose to keep around for a while.

It should be realized that the descriptions of motion and holes in the next sections are not intended to be the basis of a general theory of motion or space. Rather, they should be viewed as gross simplifications of complicated concepts which are intended to permit easy description of more abstract relations. Their chief feature is a large amount of expressive power at a low level of complexity.

2.1.1 General Motion Primitives

The general primitives I wish to consider deal specifically with the relationship of a (potentially) moving object with respect to its environment. Basically, they are:

OBSTRUCTION -- A rolling object will be considered obstructed in a specific direction (perpendicular to the vertical) if it meets an obstacle on traveling in that direction and must make a suitably long detour (say greater than or equal to the distance already traveled) before it can continue in that direction. Thus obstruction is a function of the length of the obstructing object and its position relative to the moving object.

In figure 2.4(a), the moving object is obstructed to the "east" because in order to move east it will be forced to move north from X1 to X2 or south from X1 to X3, a distance greater than it traveled from X0 to X1 in the easterly direction. In figure (b) the object is not obstructed because the detour is relatively short. In figures (c) and (d) we would consider the object obstructed if the perpendicular component of the total path traveled becomes greater than the component in the desired direction. In the case of backward motion (e) in either of the two directions, this may be

simply subtracted from the forward component, since one can generally trace an alternative path which does not contain the components backtracked over. As with all the motion primitives, it will be useful to add the qualifier RELATIVELY when an object contains sufficiently sparse (say no more than one fifth the total length) holes which permit the desired motion, but would represent an obstruction if these holes were blocked. (This is to be seen as a tentative answer to the "window" problem where, topologically speaking, a building with an open window does not enclose anything.)

COVERING -- An object will be considered covered if in traveling upwards, the resultant of motions perpendicular to the upward directions ever becomes greater than the upward component of the motion. As with obstruction, backtracking is subtractive, and there is an analogous notion of relatively covered.

SUPPORT -- It is of interest that the existence of gravity dictates that the concept of support be not quite analogous to that of covering. First of all we assume that an object cannot be at rest unless it is supported by another object or group of objects which are at rest. The actual definition of support, however, is likely to give us some trouble. We could define support in such a way that an object is supported by all the objects in contact with its bottom. But this ignores the question of what would happen if we wanted to remove some of the objects beneath the supported one. A simple defintion of support is that an object is supported by any set of points such that one cannot pass a vertical plane through the center of gravity of the object which places all members of the set on one side. Such a definition at first glance may seem computationally messy, but these are several alleviating factors. For one we will be dealing with blocks, and in general the supports will be surfaces rather than points. Any points which do appear will of necessity arise from pyramid type objects which prevent their being placed arbitrarily close. In any case, the rule used to determine if an object is supported will not affect the use of the support predicate in higher level computation.

Another concept which is potentially useful is that of SAFE SUPPORT. An object is safely supported if it cannot be rolled to a position where it will drop vertically, i.e. there are no points under it directly in contact with its surface. Since a relatively safe support would not really be very safe, there doesn't seem to be much use for it.

SURROUNDED -- We say an object is surrounded if (a) it is safely supported and (b) it is obstructed in all horizontal directions. In other words, if it were a ball, its motion would be restricted to a fixed horizontal surface. The concept relatively surrounded refers to the relative obstruction in all directions.

(a)

X_1

X_0

X_2

(b)

X_0

(c)

(d)

(e)

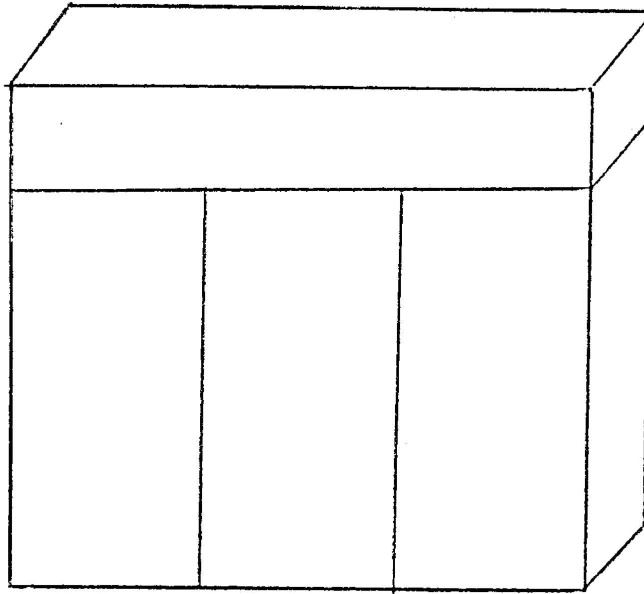
alternate path

subtractive paths

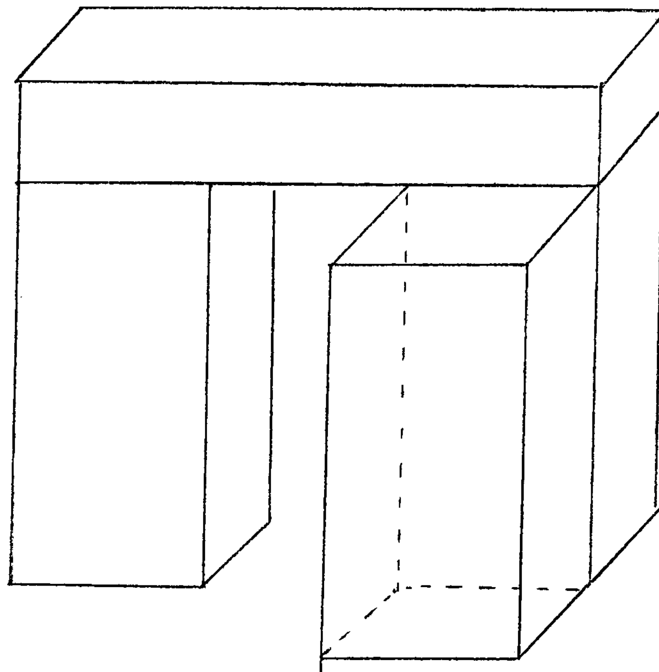
Fig. 2.4

Fig. 2.5

(a)



(b)



(c)

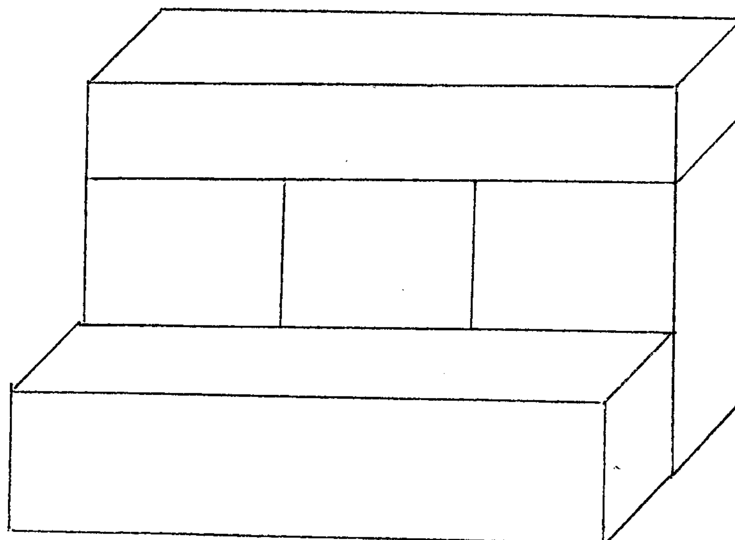


Fig. 2.6

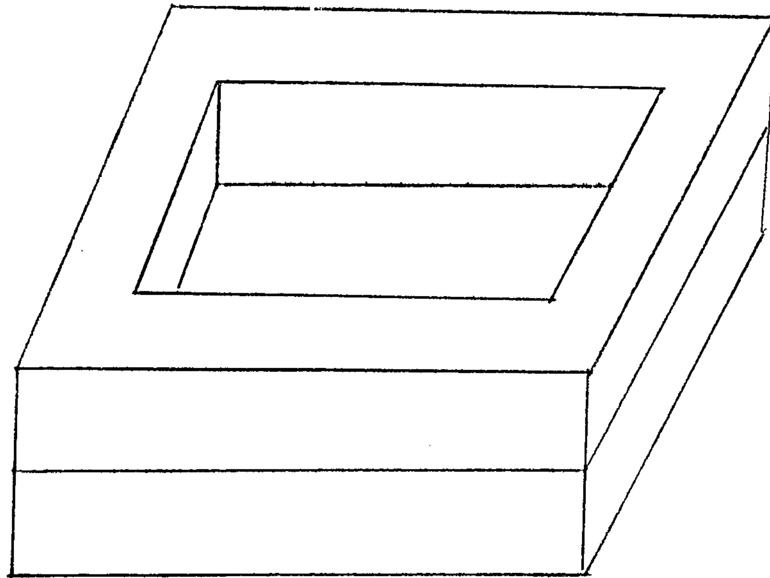
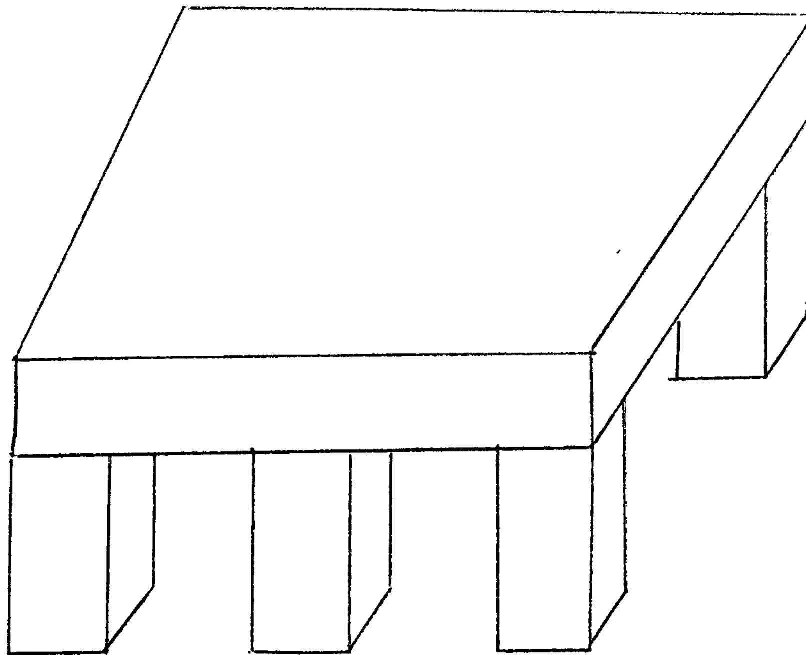
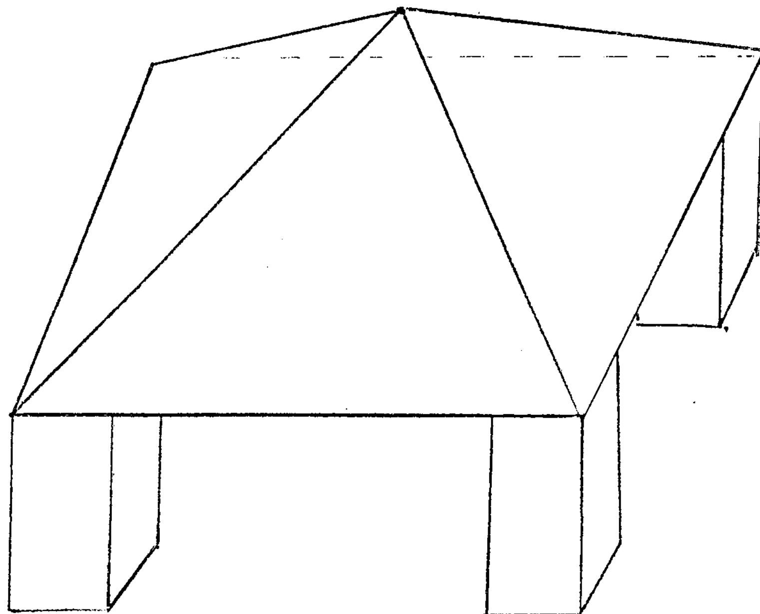


Fig. 2.7

(a)



(b)



CONTAINED -- an object is contained if its movement (in any direction) is restricted to a fixed subspace of the world space. We may make boundary conditions explicit by considering the "world" we observe as a fixed 3 dimensional rectangle with clear walls. All things will be considered contained in the world (unlike Columbus, we do not have to face the prospect of falling off the edge). An object will be considered relatively contained in the usual manner, as long as it is safely supported.

ATTACHMENT -- Objects which are not attached to each other may move independently. There are two types of attachment I feel should be considered -- face attachment and edge attachment. If a face (or suitable subregion thereof) is attached to to the face of another object, the two objects are essentially one object in that they must move together. If an edge of an object is attached to some other object, the former object is said to be edge attached and is free to pivot about that edge with respect to the other object. This will enable us to deal with items such as doors and gates in a structure. We will ignore tolerance problems in door jambs. For example in figure 2.5 (a) if the marked edge is attached as indicated, we will assume the block A may move freely to the position indicated in 2.5(b), provided, of course, it is not obstructed as in (c).

Using these concepts we can provide simple definitions for many common block structures. A box (figure 2.6) is any structure capable of surrounding an object. A canopy (figure 2.7) is any structure capable of covering an object. A door (figure 2.5) is a block which is edge attached to an arch such that in one possible position, the arch and door form a wall. A wall is any group of objects which obstruct motion in some direction for a moving object sufficiently close. Addition of the word "group" is important because otherwise, any object could serve as a wall for a near enough object.

It seems evident that in a system using both functional and structural definitions, we must be careful not to confuse them. Hierarchies formed on the basis of function may differ greatly in their organization from those formed on the basis of structure. Functionally,

a table may serve as a pedestal and vice versa, although they differ structurally. Despite the fact that functional criteria may prove a valuable aid in choosing candidates for a class, we will not in general wish to define classes solely in terms of function.

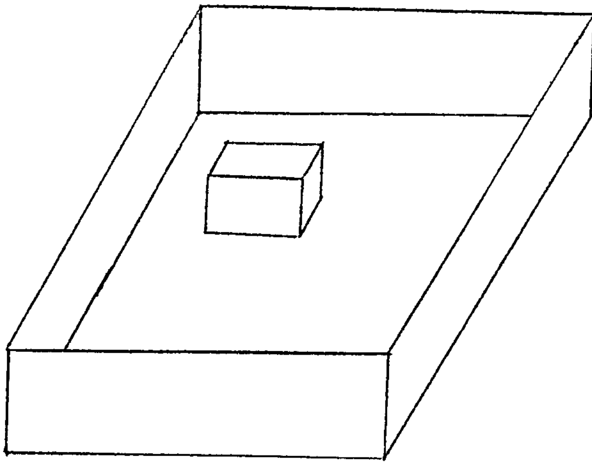
2.2 Holes

Since an object may only move through space which is unoccupied, it is a logical step to desire that freedom and restriction of movement be represented in terms of unoccupied space or holes. In general, holes are not an easy thing to represent. Bob Moore's statement "Holes are the complements of simple objects, and the complements of simple objects are not in general simple" seems to shed some light on the fact. The contour of free space in a given room at a given time may be exceedingly complex. However, for the purposes considered here, it should not be necessary to worry about such complicated questions. Basically, it would be desirable if our representation of holes did not differ too greatly from our representation of the other items in our world. This suggests that we consider holes as composed of blocks of free (or as we shall see later, potentially free) space, generally rectangular in shape. Such a representation also seems advantageous for other reasons. Dividing the free space up into blocks will also give us clues as to which parts of a structure should be grouped together. But perhaps most important, the primitives relating to motion which were discussed in the previous section lend themselves readily to analogy with holes.

It seems advantageous to define holes with respect to a given structure or group of structures. Thus a hole may in part consist of solid objects not attached to the given structure. The reason for this is that any unattached object may be moved independently of the given structure. Suppose we have a box with a block in it (figure 2.8), and

Fig. 2.8

(a)



(b)

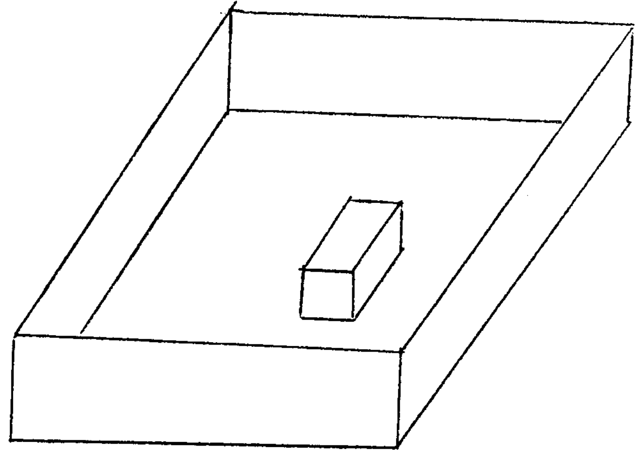


Fig. 2.9

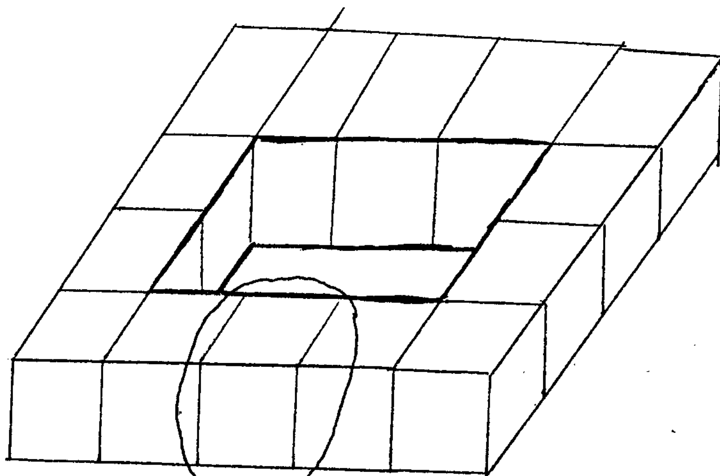
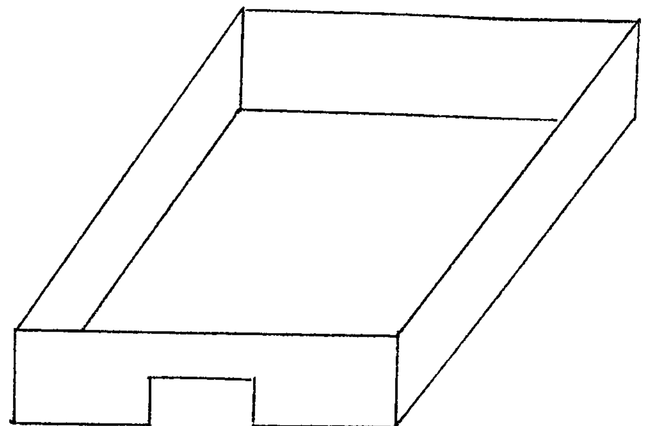


Fig. 2.10



we move the block to a different position in the box. We do not want to be forced to consider these situations as representing two different holes, so we choose to include the unattached block within the hole which describes the interior of the box. Besides making our task simpler, however, this scheme should give us a certain power in altering descriptions. For instance, suppose we have a ring of blocks surrounding an object (figure 2.9), all of which are attached except one. Then with respect to the rest of the structure that block represents a hole (albeit a hole which has been temporarily filled) and the presence of that block is not essential. If we are interested in forming an entrance to the area surrounded by the blocks, we know that all we need do is push the block out and we have our hole. (In a sense we will get this information for free if the unattached block is already considered as a hole relative to the attached ring.)

2.2.1 Hole Types

Rectangular holes may be classified according to the number of sides on which they are bounded. Conveniently the number of bounding sides coincides with the general purpose of such a hole.

Passages (ramps) -- Passages are holes which are bounded by three edges, one parallel to the ground and two which are vertical and parallel, above the horizontal edge. The key function of a passage is that it limits non-flying objects to motion along only one line. Though hard to visualize as holes, they are useful in understanding the functions of roads and bridges.

Ports-- A port is a space bounded by four edges, all perpendicular to a given plane. They are similar to passages, in that they restrict motion to a line. A port may be long, e.g. a tunnel; or short, e.g. an arch and its supporting surface. Generally, their purpose is to provide for motion from one region to another. Unlike passages ports restrict the motion of any object.

Niches--A niche is a space bounded by five edges, the lowest of which must be parallel to the ground. Niches generally provide places for objects to rest or be contained. Boxes (figure 2.6) and wall indentations are both examples of niches. Any niche supports an object, while a niche with only one edge parallel to the ground will safely support any object inside.

Rooms--Rooms (for want of a better word) are considered to be areas of space bounded on all sides--i.e. completely enclosed. They represent the idea of containment.

It should be mentioned that the edges bounding holes do not have to be solid. They may contain reasonably small (say not more than one sixth total surface area) holes, with the general provision that the floor be solid. For example (figure 2.10), a box with a port in one side is still considered a niche, or a tunnel with a window would still be considered a tunnel. Thus most of the motion concepts to which holes correspond are the "relative" counterparts of those concepts.

It is interesting to note that some of the motion concepts discussed in the previous section have direct representation in terms of holes. The notion of constraint to one direction of motion and safe support may be achieved by either a port or a passage. The notion of surrounded may be directly represented by a niche with only one edge parallel to the ground. The notion of containment translates directly into a room. More specific examples of the transition from function to representation will be provided in section 2.4.

It must be mentioned that for the sake of simplicity, some perfectly

natural conditions have been neglected. Consider for example a room with a sunken area in the middle. One would like to consider this still to be a room, but it might conflict with the notion of containment since containment implies safe support, and a large enough niche in the center might cause us to consider an object in this particular room not safely supported. The bug here is probably with our notion of "safe-support." We probably want to permit "sufficiently small" drops. This would allow us the liberty of considering structures like staircases to provide safe support. For the time being, such fine points will be left open.

2.3 A Formalism For Function

Any system which plans to provide some representation for function must also provide a formalism for such representation. The preliminary formalism I will describe is exceedingly simple (no doubt reflecting the simplicity of the domain) but some aspects suggest generalization to more complicated areas. Syntactically, ?X will represent a pattern match which binds X to any item occurring in that positon, much like the pattern matching rules of Planner or Conniver. The symbol "$" will represent "self", i.e. "$" is considered a reference to the object whose function we are describing. For example, (SUPPORTS $ X) indicates that the object we are describing supports an object named X. Our formalism basically consists of a predicate, POSSIBLE, the logical connectives AND, OR, NOT, and CHOICE, which corresponds to exclusive or, e.g.

(CHOICE (HAVE ?X CAKE) (EAT ?X CAKE))

and some functions and predicates (IN ?X ?Y), (IS ?X ?Y) <true if X is a member of class Y>, (SUITABLE-OBJECT ?X ?MODE), and PASS, SURROUNDED-BY, SUPPORTED-BY, CONTAINED-IN, OBSTRUCTED-BY, and COVERED-BY.

(IN ?X ?Y) returns T if X is located in some hole which is part of the description of structure Y. (SUITABLE-OBJECT ?X ?MODE) generates a structure representing a movable object whose size is reasonable with respect to structure X. MODE is optional. Where specified, it refers to FLY, WALK, or ROLL depending on which motion abilities we desire the generated objects to posess. The others are all predicates of the form (PREDICATE ?X ?Y ?MODIFICATIONS) where the possibilities for

MODIFICATIONS vary with the predicate. For COVERED-BY, SURROUNDED-BY and CONTAINED-IN, MODIFICATIONS may be RELATIVELY or NIL. For SUPPORTED-BY it may be SAFELY or NIL. For OBSTRUCTED-BY, MODIFICATIONS is a list of the form (DIRECTION, MOD2) where DIRECTION represents a direction and MOD2 represents RELATIVELY or NIL. For PASS, MODIFICATIONS may be ON or THRU or BETWEEN. ON may apply only to passages, THRU applies to holes in general and BETWEEN to a list of two objects.

POSSIBLE is a general predicate which operates on the motion primitives. It asserts that there is currently no condition which prevents the relation on which it operates from taking place. If it can make the predicate true it returns T, otherwise NIL. CHOICE is a predicate operating on a list (L1 L2 ....LN) of predicates, and can best be understood in terms of a predicate CAN-MAKE (similar to Planner's THGOAL) which succeeds if it proves its argument can be realized through limited manipulation of the structures involved. (CHOICE L1 L2 . . . LN) is equivalent to:

```
(AND (CAN-MAKE L1)
     (CAN-MAKE L2)
           .
           .
           .
     (CAN-MAKE LN)
     (NOT (CAN-MAKE (AND L1 (OR L2 ....LN))))
                             .
                             .
                             .
     (NOT (CAN-MAKE (AND LN (OR L1 ....LN-1)))))   .
```

Examples:

ARCH

```
(POSSIBLE (PASS (SUITABLE-OBJECT $ WALK) PORT1 THRU)
```

DOOR

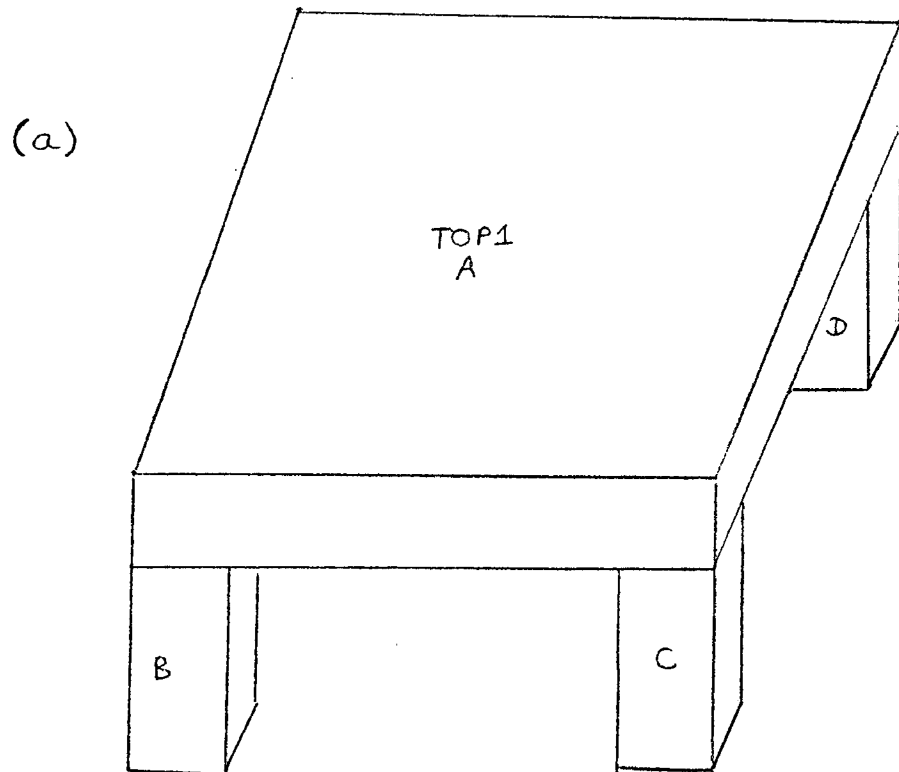
```
(CHOICE
   (POSSIBLE (PASS (SUITABLE-OBJECT $ WALK) (B1 B2) BETWEEN))
   (NOT POSSIBLE (PASS (SUITABLE-OBJECT $ WALK) (B1 B2) BETWEEN)
```

ROAD

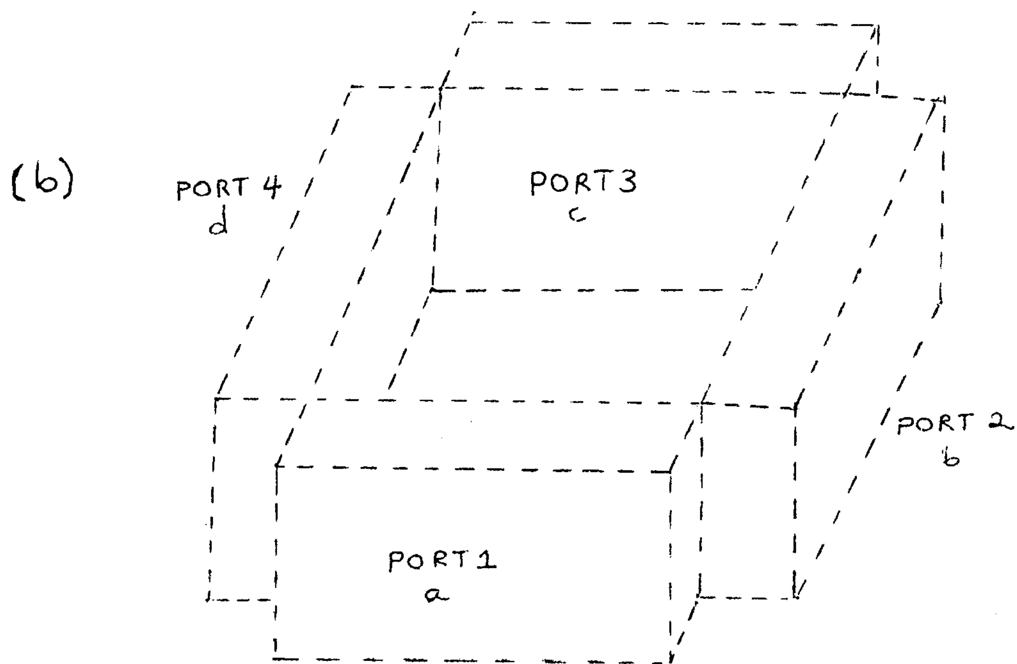
```
(AND
   (POSSIBLE (PASS (SUITABLE-OBJECT $ WALK) $ ON))
   (POSSIBLE (SUPPORTED-BY (SUITABLE-OBJECT $ WALK) $ SAFELY)
```

Fig. 2.11

(a)



(b)



2.4 Use of functional representation

Assuming we have a program which takes a structure and interprets it in such a way as to discover all the pertinent holes, we may use the holes and other information to construct a list of possible functions for the structure. Let us look for example at a table (figure 2.11(a)). Our hole finder will find the four ports shown in (b) and produce a description like (c). Furthermore the large square area of the top suggests that it will support something. Consequently the list of possible functions will be:

```
(POSSIBLE (PASS (SUITABLE-OBJECT $) PORT1 THRU))
    "        "              "        PORT2   "
    "        "              "        PORT3   "
    "        "              "        PORT4   "
(POSSIBLE (SUPPORTED-BY (SUITABLE-OBJECT $ ROLL) TOP1 ))
(POSSIBLE (COVERED-BY (SUITABLE-OBJECT $ FLY) TOP1))
```

If we are now searching this structure for a table, and the functional representation of table is

```
(POSSIBLE (SUPPORTED-BY (SUITABLE-OBJECT $) T4)
```

where T4 points to the table top in an internal description of the table, we immediately have an anchor with which to begin our comparison with the table description. Winston's program would have to search the entire description before deciding to link the two table tops. Furthermore, if we were looking for a house in figure 2.11(a), and assuming the house had a functional representation

```
(POSSIBLE (CONTAINED-IN (SUITABLE-OBJECT $) R1 RELATIVELY))
```

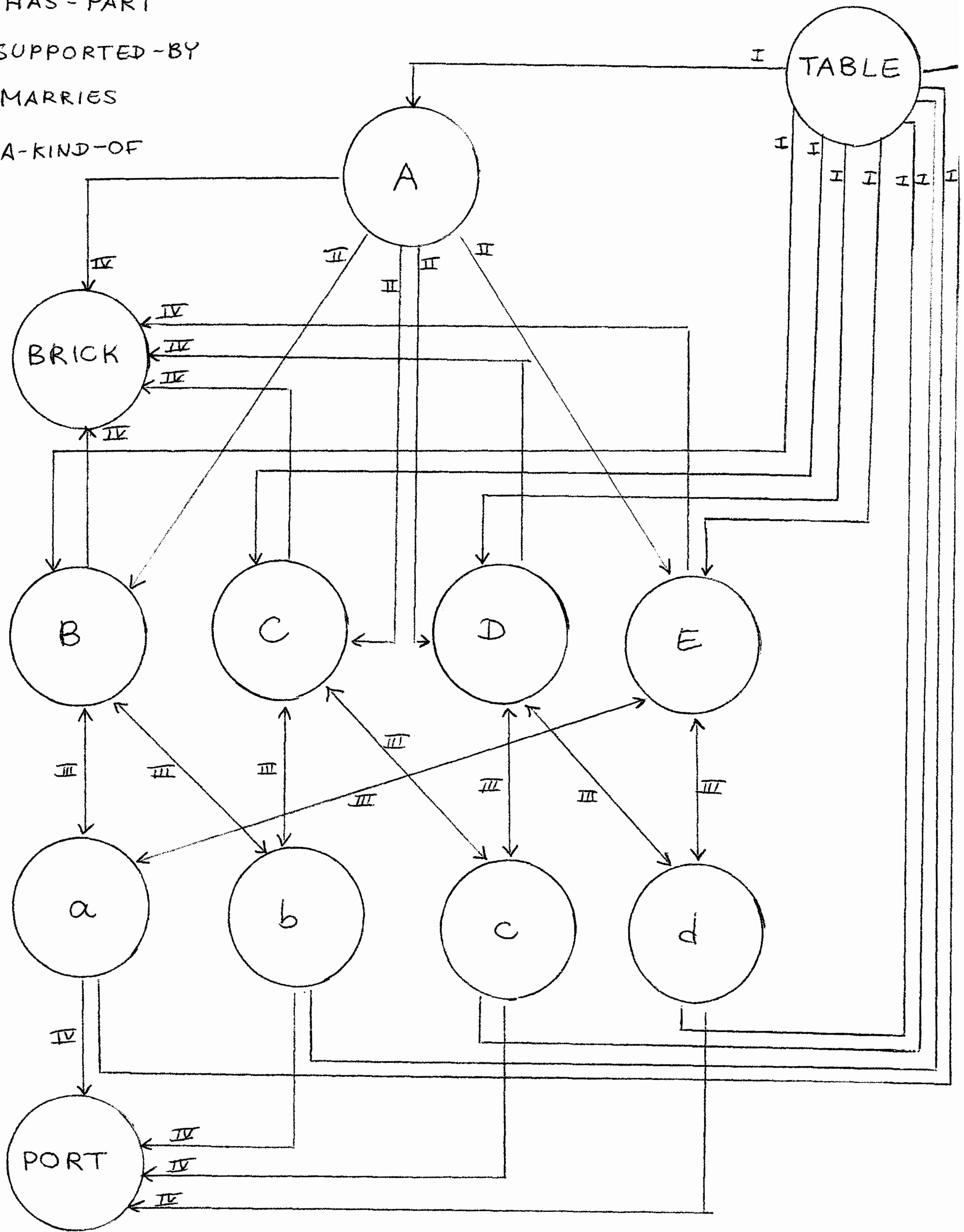


Fig. 2.11 (c)

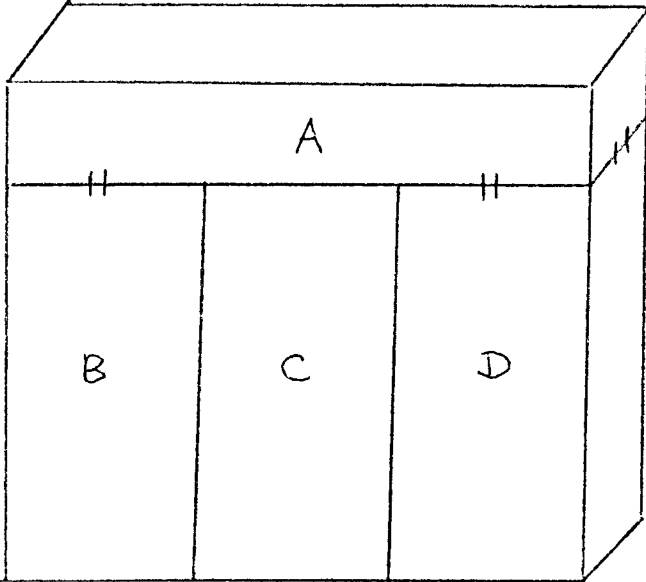
we would see that this does not occur in the description and would not even have to bother trying to match the two descriptions. It is interesting to note that if we are actually looking for a pedestal (which will have essentially the same functional description as a table) we will succeed on the functional description but then end up with a bad match. But we haven't lost entirely, because the machine has discovered an important thing. While not strictly speaking a pedestal, the object of figure 2.11(a) may be used for a pedestal if one is needed and none are around. If at some later point the machine wishes to build something, this information may prove invaluable. Consequently, we see that certain advantages will accrue from keeping our knowledge of functional properties separate (in some sense) from our knowledge of structural form. This will be discussed more completely in the next chapter on the general hierarchical system which will be the backbone of the program. Although I do not propose presently to deal with the problems of planning and constructing structures, such would be a logical and perhaps insightful extension of the program I have in mind.

Let us return now to the triple-arch example of figure 2.1. Our function list will look something like

```
(POSSIBLE (PASS (SUITABLE-OBJECT $) PORT1 THRU))
    "        "         "            PORT2  "
    "        "         "            PORT3  "
```

Where PORT1, PORT2, and PORT3 point to the respective ports in this scene. When looking for arches, we again get immediate links. Here, the time saved is much greater because we do not have to go searching for crosspieces among every object in the scene. It is perhaps important to

Fig. 2.12



╫ → attached blocks

note that this simplified process will fail to find the simple top arch. This is because the top arch strictly speaking, does not contain a port. As discussed in section 2.0, this may or may not be desirable. One practical way to fix this bug, if we want that arch, would be to consider the top structure separately by moving it (in the machine's imagination) to a separate place and then discovering the port. Another is giving the machine the knowledge that an inverted passage may serve as a port if the object rests on a suitably large surface. Both these methods presuppose proper attachments, and a grouping of items in a scene by attachment (see chapter 4). Both have heuristic merits, which should be considered in a final program.

Another interesting problem is that of finding an arch in a scene like figure 2.12, where the hole is blocked. Again, assuming an initial grouping by attachment, and that block C is not attached to the others, C will be represented as a hole with respect to substructure A-B-D. The arch will be easily found. Winston's program would have to explore possibilities A-B-C and B-C-D as well as A-B-D in determining the arch. Suppose, now that C were edge-attached to B or D. Then by considering the extremes of its motion (figure 2.5(b)), our function-finding program should know enough to construct:

```
(CHOICE
  (POSSIBLE (PASS (SUITABLE-OBJECT $) (B D) BETWEEN))
  (NOT (POSSIBLE (PASS (SUITABLE-OBJECT $) (B D) BETWEEN))))
```

matching the description of door.

I do not think the problem of translating from function descriptions into the structure necessary to fulfill those descriptions will arise

until a constructor is built. So I will just mention the problem briefly with reference to one example. Consider the function description

(POSSIBLE (PASS (SUITABLE-OBJECT $ WALK) ?HOLE THRU)) .

Since passing through refers to space (i.e. holes) in general we will be able to generate several structures to fill this need. Any port or passage will do, as well as any structure consisting of a box or room with two ports or passages. The first two would likely be generated on a first call since that is their sole function and they are simple to construct. The second pair should not be ignored, however, and may be tried if the machine fails to find a port or passage and is desperate for something to fill this function.

It is appropriate to wonder if the domain discussed here is not too simple to afford effective extension to other areas. This is a difficult question. Certainly at some level of structural complexity much more sophisticated theories of physical laws will be needed to adequately describe function. And it is possible that even the generalized formalism would be of no use at all for describing the functions of classes of non-physical objects. It may be that both the key functional features and the short functional descriptions are much too simple to admit useful extension. Nevertheless, it is quite striking that the functional concepts discussed find an easy and direct representation in terms of specific structural properties and that certain structural properties may immediately be singled out to provide clues as to functional use. Perhaps this is an artifact of our simple blocks world. But perhaps not. Considering chess as one example, specific types of

moves often translate readily into specific objectives. Simultaneous attack of two pieces may be easily achieved by a fork. Immobilization of a desired piece suggests immediate examination of the possibilities of a pin. And when one's queen is in trouble, a saving tactic is often attacking the opponent's queen. Of course a simple one-to-one correspondence between structural properties and functional use does not hold in general, or even in these simple examples. But in any case where a good method of functional representation narrowly defines the choices, such representation is definitely useful.

3. A Hierarchically Structured Knowledge System

In chaper 7 of his thesis, Winston discusses the identification problem, a generalized application of the simple matching techniques developed earlier. There are many aspects of this identification problem which may be illuminated by the following questions:

Given a scene X, is X a structure of type FOO?
Does X contain any substructures of type FOO?
Do we have a structure which matches X?
What substructures are present in X?
What in general can we say about X?

These questions show a progression from less to more general. The questions are closely related, because the answering of one question may involve the asking of several others. For example, if a structure FOO necessarily contains some substructures of type BAR, then in answering the question "Is X a FOO?" we will want to answer "Does X have any substructures of type BAR?" But in answering this question we will certainly want to ask of a substructure Y of X "Is Y a structure of type BAR?" Clearly these different questions we will want to ask will require quite similar processes, and many of the procedures used will be shared by routines answering all of these questions.

Winston's program answers the question "Do we have a structure which matches X?" by matching all structures against X. If the number of structures in our data base is large, this sort of procedure may be extremely wasteful, since many structures will not be at all like X, and those which will be like X will require much redundant computation.

Winston proposed two ideas to help improve matters.

(1) abstracting certain essential properties of a scene into a skeleton which could be matched against X to quickly decide if it was worth continuing to match X against this structure in detail.

(2) When certain structures are sufficiently similar, pointers may be inserted in them so that if the matching program runs into difficulty matching x against FOO, but the difficulty strongly suggests X is really a BAR, the machine can quickly switch to the BAR description with a reasonable expectation of success.

Both of these ideas are quite valuable for shortening the amount of computation required in deciding such problems. However, they still have some drawbacks in the form proposed. Their construction and application is somewhat haphazard, and lacking in definite structure. Solution (2), the "similarity network" <6; p.232> does enable certain crucial differences to change the model being compared, but it does so on a highly local, one-difference level, without using the overall evidence gained already (perhaps including failures in previously tried models) to present an appropriate sequence of likely choices. Such a system will work well when there are not too many structures around and few of them are related. But if we are dealing with a world of structures where the relationships and similarities are many, in other words, a world with a definite structure of its own, we would like our machine to be more systematic in the application of these processes so that its internal models resemble the structure of the world to at least a reasonably close degree.

It is also interesting to note that the question "What substructures are present in X?" is answered in the Winston system by checking a list of all substructures known. It would be quite nice, of course, if a

program looking at the structure could quickly generate lists of likely and unlikely substructures, i.e. tell us which substructures we should be looking for. It may be asked if this is really a meaningful question, or simply one of "puzzle value" as figure 7-31 <6; p.238> in Winston's thesis might indicate. My personal feeling is that it is in fact quite meaningful. When presented with raw data in the form of a large and complex structure which is not artificially contrived, we have an excellent chance of determining its principal features if we can simplify the description by a judicious labeling of substructures. This, however, depends on our ability to cheaply decide which substructures we should look for. We should be able to make use of clues provided by specific remarkable features of the object as well as those provided by context. (What type of structure is expected here? What substructures are known to be common building blocks in our current world?)

In this light, it is instructive to consider the properties we would find desirable of a general description mechanism. Perhaps most important is the consideration of speed. At the brute face level, finding appropriate descriptions of structures requires exponentially exploding search if all possible substructures are to be considered. The amount of search actually needed in solving problems provides a convenient metric against which to measure performance. Each level of complexity in our descriptive power can only be attained practically if such descriptions may be recognized in a reasonable amount of time. In order to recognize a complicated structure reasonably fast, we must be able to recognize its simpler components very fast. This requirement

strongly suggests that we have the power to concentrate on description one level at a time, so that we do not become confused by irrevelant detail. Winston <6; p.202> recognized this problem and proposed a method for solving it. It seems desirable to generalize his solution.

A second desirable property of a general description processing system is that the system be easy for the machine itself to construct and debug. If we merely handed these structures to the machine on a platter, the best claim we could make would be that we ourselves had a reasonably good understanding of the structure of the domain we were working on. But it would be far more interesting if the machine itself could construct a working system, with some help from us of course. Then we could say in some sense that the machine itself was capable of understanding the structure of our particular domain. This can best be achieved by a great degree of simplicity, as well as the ability for errors to be patched locally without having to worry about possible side effects on the entire system. The ability to divide the system into small blocks of well defined structure would greatly facilitate this.

There are probably many possible systems one could build which meet these criteria to a reasonable degree. In this chapter, one alternative will be discussed which owes its conception to the frame systems proposed by Minsky <3> as a general structure for dealing with many problems in artificial intelligence. Such a system has many of the desirable properties mentioned above. A hierarchy of frames provides something like an automatic control structure. Through judicious organization and use of pointers one should be able to eliminate virtually all unnecessary

search, and thus provide tremendous gain in speed. The ability to isolate single frames provides conveniently for local repair and construction. And since a frame system is in general hierarchial, differences between structures may be represented at different levels, enabling the machine to temporarily focus attention on specific questions and effectively share descriptions of several similar objects.

A short overview of the proposed orgainzational system will be given here, and dealt with in more detail in subsequent sections. The description in this chapter should not be considered complete, since there are several problems which have not yet been resolved satisfactorily. These will be mentioned where they occur.

Basically, the system will consist of a set of hierarchies (each a tree or perhaps a lattice structure) of small modules called test-frames, which will represent complete and incomplete descriptions of various classes (or examples of classes) of objects. The hierarchies will in general each represent a general class of objects at their top level, with subclasses represented at lower levels. Models of specific members of each class may be tacked on at the bottom level of classification. These will be of use in processing complicated descriptions and in debugging our system when errors are noticed. Certain test-frames will be considered "entry points" to the system. Any frame which has a specific name associated with it will be an entry point. This will permit descriptions in one part of the system to reference components by name, which may be examined directly without having to pay attention to test-frames above the named class. Other entry points will be defined by

a "skeleton" <6; p.233> consisting primarily of functional information described in the previous chapter. The top test-frame in each hierarchy will be an entry point. Names and function descriptions will be indexed by a program which provides pointers to the various entry points, and may be considered the super-top-level test-frame. The purpose is to provide a small list of entry points arranged in some order of likelihood which will eliminate search of most of the test-frame structure. All frames will provide information which uses the result of prior frame description operations to reject a description or accept it, and if accepted, to point to subsequent frames in the hierarchy where it might belong. Each entry point will contain more complete information to compensate for the fact that it may be entered without the prior processing having been done. A description will first be processed by the super-top-level frame which will return a list of plausible entry points. These will be explored in a given order. Some may define substructures which will be noted and others will (hopefully) lead to a complete classification of the description. Other substructures may be searched for only when specifically requested by a frame, since it is desirable to eliminate a costly search for every possible substructure.

3.1 The Individual Test-Frame

The original network matching program described by Winston <6; chapter 4> was reasonably symmetrical with respect to the two networks

being matched. This has certain advantages in applications such as the analogy problem, since it permitted models to be formed from descriptions and then compared with other models. But, in some cases, this completely symmetric match proved to be a drawback. In finding substructures of a scene, for instance, the matching program produced such a proliferation of c-notes that it was extremely difficult to isolate the proper information. These c-notes ended up being discarded <6; p. 239>. Continually generating c-notes only to discard them later is a wasteful process. The essence of our system is that one does not want to carry out an entire match at once, but only small portions at a time, deciding what to do next on the basis of the previous results. Consequently, it is desirable to have a matching routine which is by nature asymmetric, considering the network description required by a test-frame (the frame model or FM) as a pattern which must be matched in the structural description (SD) of the item we are attempting to classify. Features present in the SD but not in the FM will generally be ignored unless specifically noted, while those in the FM which are absent from the SD will be of critical importance. Furthermore, it is desirable to simplify those portions of the description processed by each frame so that they may be readily accessed by lower frames. Such simplification may take two forms:

(a) labeling certain portions of the SD with a tag which may be referenced by lower frames, and

(b) replacing certain sections of the SD by simpler descriptions to speed up work by lower frames. For instance, it would in many cases be desirable to replace the subdescription of a row of attached

bricks by the simple tag (WALL). When this is done, a record of the transformation should be kept someplace, so that it may be undone if that should later be necessary.

It is reasonable, considering this, to view the operation of the matching program as a progressive transformation of the SD into a more and more general representation. In cases where the area of the SD being examined is very small, say one or two nodes, it may prove possible to bypass any interpretive mechanism and represent the requirements explicitly by a few lines of code. I anticipate that this will generally be the best representation for the difference operations on descriptions which are coming from immediately preceding test-frames. The case of entry points will be different, however. Here the matching will be much more extensive and we will probably desire to invoke the matching program. In a language such as CONNIVER, the FM may represented by a set of assertions in a context associated with the frame, and all we need to do is add a line of code to call the matcher.

At this stage the usual distinction between data and program is becoming quite blurred -- in one sense each test-frame represents a description of a certain class of objects, and in another sense it represents a portion of a procedure which moves us around the system. For example, if we were interested in whether the structure under consideration contains an arch we could represent this as

```
(AND (SETQ L2 (CONTAINS SS15 ARCH))
     (GO-FRAME G0106))
```

where CONTAINS is a high level function which searches for substructure ARCH in SS15 and returns it as a value. Or we could say

```
(AND (MATCH FM SD) (NOT (NULL L2)) (GO-FRAME G0106))
```

with a network FM:

(HAS-PART SS15 ?L2)
(KIND-OF L2 ARCH)

where the "?" indicates that L2 should be bound to the appropriate substructure.

Also associated with the entry point frames will be a name or a skeleton or both. The total content of the skeleton is flexible, but it will definitely contain any functional information abstracted from the structure under consideration. Other characteristics of great interest will be added, presupposing that the machine has some means of deciding which characteristics are of great interest for given frames, and also knows how to find these quickly in a complex structure.

3.2 Utilizing the Test-Frame Structure

The key to success of the test-frame system will be a reliable set of skeletons indexed in the super-top-level test-frame. Given a skeleton which has been created for a structure under consideration, matching this skeleton against skeletons of entry points should yield generally three sets of entry points:

(a) entry points with very simple skeletons which are small subsets of the SD skeleton.

(b) entry points with more complicated skeletons which are subsets or very nearly subsets of the SD skeleton.

(c) complicated entry points which almost completely match the SD skeleton.

Entry points in class (a) will represent simple structures which are very likely to be the basic components of a large structure (walls, arches, etc.). Entry points in class (c) will represent likely candidates for descriptions which match the entire structure. Entry points in class (b) may be either large components of the structure or candidates to match the structure which are not as likely as those in (c). First, entry points of class (a) will be tested until exhausted. When appropriate subgroups of the SD are discovered, they will be marked accordingly, in order to form a more generalized SD which will cause less confusion to later test-frames. An entry point may be indicated more than once, say by different portions of the SD which are instances of the same general type of substructure. The entry point will be tested once for each of these. Certain frames may be marked as terminal for a given class, even

if they point to frames below them. The meaning of a terminal frame is that if a substructure being tested reaches a terminal frame, it is a "success", i.e. it may be considered a valid member of the class of objects represented by that particular frame. Terminal frames thus in some sense provide failpoints in a frame system. If a substructure which we are trying to classify is rejected at some particular frame, it simply backs up to the nearest terminal frame where it succeeded. When we have decided that a substructure should be replaced by a more general description, (usually just a name) in the larger SD, we hang the substructure on the appropriate terminal frame as an "example" with a unique label, and then replace it in the original SD by the name associated with the terminal node and a pointer to the label. Thus if at some time we wish to revoke our assignment of this substructure or explore its properties more thoroughly we will not have lost any information, while in the mean time we avoid cluttering up the SD.

After the SD has been simplified by replacing all the substructures represented by class (a), we can begin deciding about the structure itself by comparing the SD with frames whose entry points are in class (c), in order of the completeness of the match. If we process a terminal node in which the entire SD is accounted for, then we have a description of the entire structure. If we reach a terminal which accounts for some portion of the SD, we have isolated another substructure, and may further simplify the SD as described above. We may proceed to the next entry point if we fail to reach a terminal point, or decide to abandon a path which is taking too much time. If we exhaust all entry points in (c)

without finding a match we may try those in class (b), which represent less likely candidates and intermediate subgroups. If we are able to simplify the SD still further, we may return to frames in places we previously abandoned because of computational difficulty.

This procedure for using the test-frame structure is not final. Principally it represents a sort of compromise between the classical "bottom up" (starting with the most basic substructures and combining them to form more complicated substructures until the entire structure is categorized) and "top down" (looking only for specific substructures when their presence is indicated by a particular attempted categorization). The "bottom up" scheme would require many attempted matches and be quite costly in terms of time. The "top down" scheme will be dealing with quite complicated SD's whose simplification will only be incidental. The system proposed hopes to significantly simplify the SD by finding many simple (i.e. cheaply identified) substructures and then operating in a top down fashion where further subgrouping occurs only when specifically requested by a frame or is "stumbled upon" by the classification process. Such a strategy may not prove optimal for this application and will be open to change should a better one present itself.

It should be noted that there are no hard and fast guidelines for separating the entry points into classes (a), (b), and (c). Such separation will depend on a small program, perhaps with special purpose knowledge, which knows enough about the domain to make the proper choices. There would be a great advantage gained if this program itself could be made to learn from experience. Currently, I do not feel I

understand the problem fully enough to have suggestions as to how this might be accomplished, but it would necessitate collecting some data as to the reliability of the skeleton system and proposing amendments to it. As the program grows in complexity, it may prove useful to embed the skeletons in a mini-frame system, which may be debugged in a fashion similar to the large one.

3.3 Constructing and Debugging

The test-frame system we have been examining is oriented toward ease of construction by a machine. This form of construction must necessarily be an incremental process, changes occuring each time an error is made in classifying structures. It may be helpful to think of the system as a "theory" of the structure of the problem domain, initially quite meager, which is progressively fleshed out and debugged as new examples are encountered. The test-frame structure is designed so that most debugging will take the form of fairly local patching. Since too many local patches in a system of this sort could easily compromise its structural generality, we would like our debugging system to be as conservative as possible about amending the existing system. Usually we will be faced with choosing from among two or more alternatives which divide roughly into three categories:

(A) adding a new test-frame or entry point
(B) producing a cross link between hierarchies
(C) changing existing conditions for rejection or frame selection.

(C) will be preferable where the differences are small, that is if an SD ends up fairly close to where it should be, local changes should not be too disasterous. However, in cases where a serious mistake has occurred it will in general be better to create a new entry point somewhere than provide a link between hierarchies. The problem here is

that if too many entry points with the same or similar skeletons are generated, we may end up having to do a lot more processing than we really want to. In cases like this, we will need a program which collects a set of entry points with similar description and compares them, perhaps changing the skeletons of some to permit more efficient selection for a given SD. In addition we would like the entry points to be fairly evenly spaced throughout a hierarchy. If the entry points are too dense they defeat their purpose by forcing more computation in the initial entry point selection. And if they are too sparse, too much work will have to be done in the frame system, and we run the risk of better matches in the wrong places for a given SD. Also, at some time we may want to totally or partially reorganize a hierarchy, perhaps deleting entry points which don't do much good. This will be discussed later.

3.3.1 Local Debugging Rules

There are basically two different types of definite errors which may occur. One occurs where a particular SD is accepted as a member of a class when it should not be. Another is when all entries in a particular hierarchy are rejected, and the SD actually fits somewhare in that hierarchy. Since the machine is going to need some kind of information concerning the proper position or positions (we will assume a given structure may actually fit in more than one position in the frame system) we will allow the machine to request the classifications when it has

finished processing an SD.

This form of questioning generalizes upon Winston's program which accepts the information about whether a structure does or does not belong to a specific class. Other forms of questions will be discussed later which will augment the basic teacher-student relationship and permit the machine to learn more quickly.

There are four fundamental debugging processes in the frame system. They are :

(1) providing for rejection of an SD when such is needed,

(2) rearranging a particular test-frame so that a particular description or class of descriptions gets pointed in a different direction,

(3) adding a new terminal node for a particular classification,

(4) processing a set of test-frames to provide for a structural description being accepted which was formerly rejected.

(1) When we wish to provide a rejection of an SD somewhere in our system, we generally want to place it as close to the entry point as possible so that the amount of useless processing is small. By examining portions of the SD which were not looked at and found acceptable by the system we may focus on the probable causes for rejection, and we may further narrow our possibilities by comparison with already learned examples of the particular concept. Then we proceed up the hierarchy, one test-frame at a time, making sure that no examples attached to each test-frame have the putatively undesirable properties. As soon as we hit a frame which does accept such examples, we may place that particular rejection information in the frame immediately below it.

(2) If we want to rearrange a test-frame so as to redirect a particular class of SD's we must make sure that no previously learned SD's get redirected also. Since the direction indicators as presently conceived will each be a test or two followed by a frame pointer, we need only worry about examples tacked on to frames descendent from the test-frame whose pointer we wish to amend. We may take the examples from these frames and test them to make sure each goes in the proper direction after our patching. If not, we will be forced to further amend the test-frame which is giving us trouble. We should generally not have to look at any lower frames.

(3) When we want to add a new terminal node under a particular class name, we will assume we have fixed things so that the SD in question reaches the top test-frame processing members of this class. We are then faced with two possibilities: (a) this frame rejects our current SD or (b) the SD is accepted, but rejected before any terminal test-frame of this class (or any of its subclasses) is reached. (If it is accepted down to the terminal of any subclassification we have an error since we assume the teacher is being honest and giving us the most accurate classification possible for the particular SD.) In case (b) we will simply add an extra terminal frame with description of the SD immediately below the classification frame and provide (using (2)) an acceptable pointer to it. In case (a) however, there may be good reason to preserve the other members as a separate subclassification, so we will replace the top frame for this class by a test-frame with two pointers -- one to our new terminal and one to the previous top frame, which will now be a

subframe of the class. In some cases it may be desirable to provide a more suitable entry point for our new terminal, either by changing an entry point above it or creating a new entry point. We can do this by comparing a proposed skeleton of the new item with properties of nearby frames and the immediately preceeding entry point. If we can add the right items to the entry point skeleton (even if its position must be shifted slightly) this is preferable. Otherwise we will have to create a new entry point, placing it so that it covers as much area as possible. If for some reason we do not wish to create a new entry point, we may wish to add linking pointers to our new terminal at other entry points whose skeleton match is better than the terminal's immediately preceeding entry point.

(4) In cases where we would like to place an SD in a given class, but it is rejected before we reach there, we must first amend the rejection test so that the SD is accepted at this stage. This will be somewhat risky, unless we have kept around examples of rejected structures. Furthermore, problems of this type may arise from instances where we misinterpreted the offending property in a structure we desired to reject, and thus in reality we may be better off eliminating the rejection test altogether. Eliminating the test will not really put us in serious difficulty (though it means we have wasted an example or two due to bad sequencing) because if we were ever to encounter again the structure which originally caused us to insert the rejection test, we would be forced to produce a better test by virtue of the fact that we would now have an example which demands acceptance.

Fig. 3.1

F_1 entry point

F_2 Test i

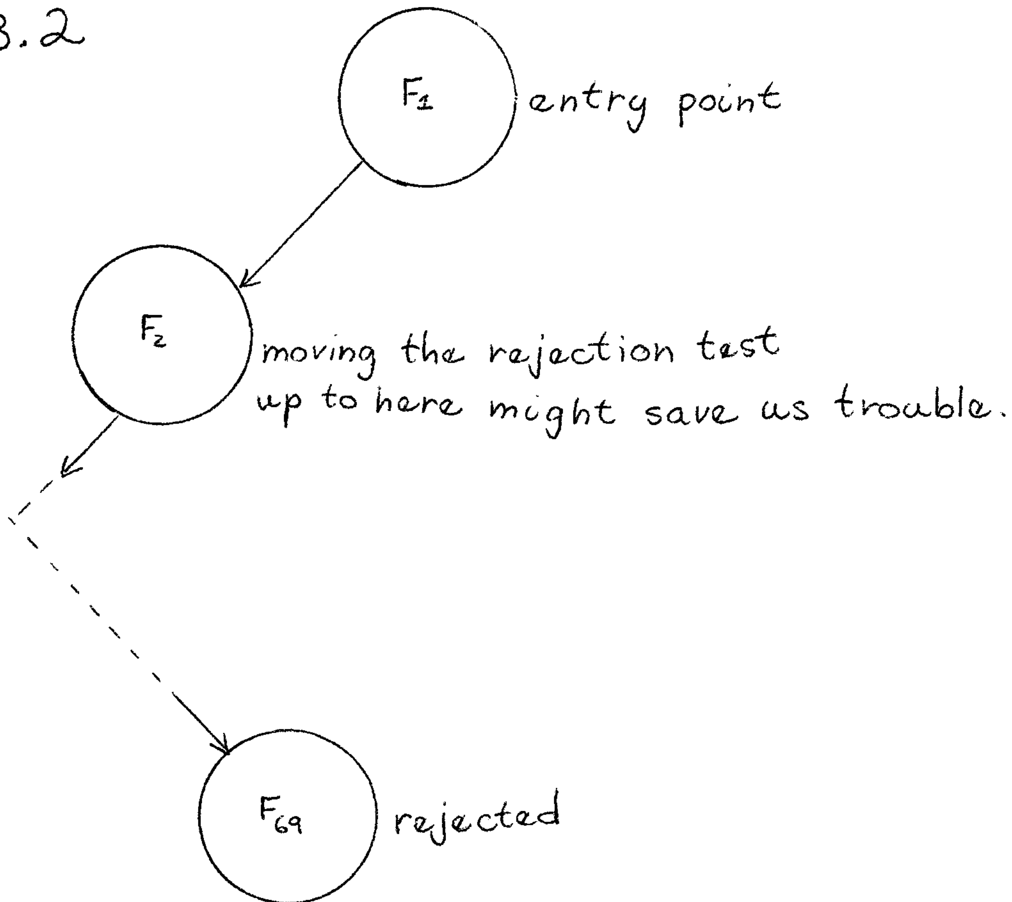
(must be fixed)

F_2 Test i

F_3 rejected

F_4 would be accepted here

Fig. 3.2



Then we must follow the path down to our desired class, if necessary amending further rejection criteria and changing frame pointers as in (2).

These routines will enable us to patch most errors discovered in our system. If we accept an SD in a certain category which should be rejected, we either (figure 3.1) fix the frame where a wrong choice was made or (figure 3.2) place a rejection pointer close to the point at which we entered the hierarchy. If the SD belongs somewhere higher up, or on a separate branch, we must create a new terminal and entry point for it and then test to make sure processing of the SD will be accepted by the terminal.

If we reject an SD before we get to its proper class, we invoke procedure (4). If, on the other hand, we have taken a wrong turn and hit a dead end, we must process the pointers in the frame where the wrong turn was taken. Perhaps the worst problem will occur when we were not even pointed into the proper hierarchy by any entry points for a given SD. Here we must begin at the top of the hierarchy and work our way down, changing each frame necessary to insure that the SD will reach the required classification. Certainly we will want to add an entry point for the new terminal created. Along with these methods for fixing hierarchies, we will want to be able to create a new hierarchy when we desire. This will be relatively simple. We simply create a top level entry point with skeleton being the function description and whose FM is the SD of our first example. It will initially be a terminal frame. The program should be informed when we desire a new hierarchy started, since

it will usually be incapable of handling descriptions so completely alien to its current structure. The hierarchy will then be built up by successive terminals added as new examples appear and are named.

We will also want the capacity to split up a frame when the number of subframes it points to gets too large. The number of subframes pointed to by a particular test-frame should be more or less between two and six. Since the subframe selection will generally be a serial pocess, more than six would be wasteful of time. Perhaps even more important, as the number of subframes grows, so does the amount of work needed each time the frame is patched. Of course the fewest number of decisions would be made if the frames had a balanced binary tree structure, but this would necessitate keeping a lot of fairly useless frames around -- taking up space and requiring maintenance.

Finally, we will want the capacity to specify that certain groups of classes should be given a collective name. We could either tell the program this or have it ask us. These are equivalent, since it would be simple to invoke a questioning routine which would take groups collected under one frame and asked if they had a name. We could also ask, every time we established a terminal corresponding to an SD, if this example was typical of a specific subclass of the general class under which it is placed.

3.3.2 General Debugging Techniques

It may prove desirable to be able to restructure the entry point system from time to time if we are winding up with too many entry points or too many of a specific nature. In the latter case, we may collect any group whose members are too similar and examine the FM's associated with test-frames at and below each corresponding entry point. If we can discern specific properties which apply to small enough subgroups, and are easy to find in a complicated structure, we may include them in the skeleton. We may test for this condition by trying some previously learned examples. In the former case, we may want to remove some entry points from particularly dense hierarchies. We can do this, provided we do not compromise the efficiency of finding an entry point close to the desired classification. If elimination of an entry point causes better matches to be generated at entry points elsewhere in the system we may put explicit rejection statements at each of these, thus providing that the time spent examining such irrelevant entry points be short. Again, we will want to test our alterations on known examples to be sure we haven't fouled things up somewhere else.

Fig. 3.3

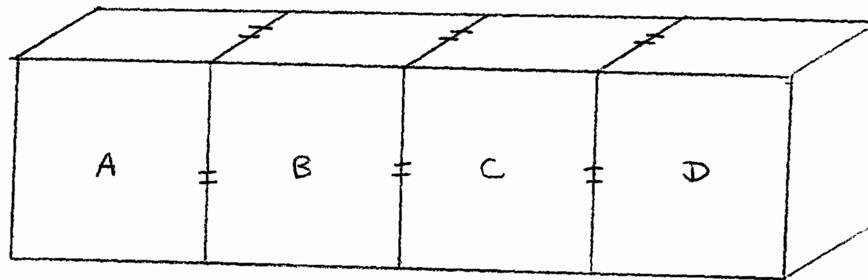


Fig. 3.4

WALL1

I I I I

A B C D

II, III, IV II, III, IV II, III, IV

V V V V

CUBE

Relations

I HAS-PART

II ATTACHED-TO

III NEXT-TO

IV MARRIES

V A-KIND-OF

3.4 An Example

This section will present an example which, it is hoped, will make some of the points described in this chapter clear.

Suppose we start with no frame system and an example of a wall (figure 3.3). Since walls do not generate functional descriptions we might choose as a skeleton some pattern which matches any set of objects all of which are connected by a chain of attachment and "NEXT-TO" pointers. (We will not be considering point-of-view problems, so there will be no distinction between left-of and right-of or in-front-of.) Our first hierarchy will have frame model of figure 3.4, no pointers, and skeleton as above, as well as the name "wall." Suppose now we introduce a tower (figure 3.5). The chain of attachment and support pointers which would form its skeleton only partially matches that of the wall hierarchy, and (since this is the only hierarchy we presently have) the tower description is matched against the wall description. The match is not perfect, but then there is nothing which expressly forbids the tower being a wall. So our program proposes that the new structure is a wall. We inform the program that this structure is actually a "tower" but that it should be included in the same hierarchy. The program now amends its test-frame structure so that the top frame matches a group of blocks which are connected by attachment pointers. The top frame will now test whether the blocks are connected by chains of support or "NEXT-TO" pointers, and point to the appropriate terminal frame. A problem now

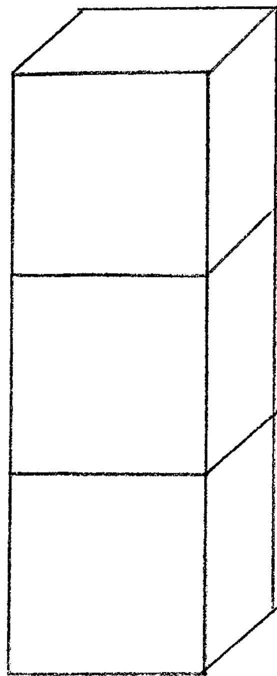


Fig. 3.5

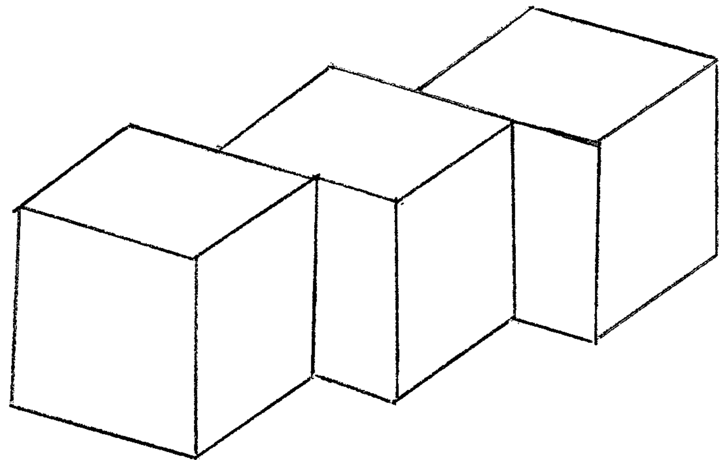


Fig. 3.6

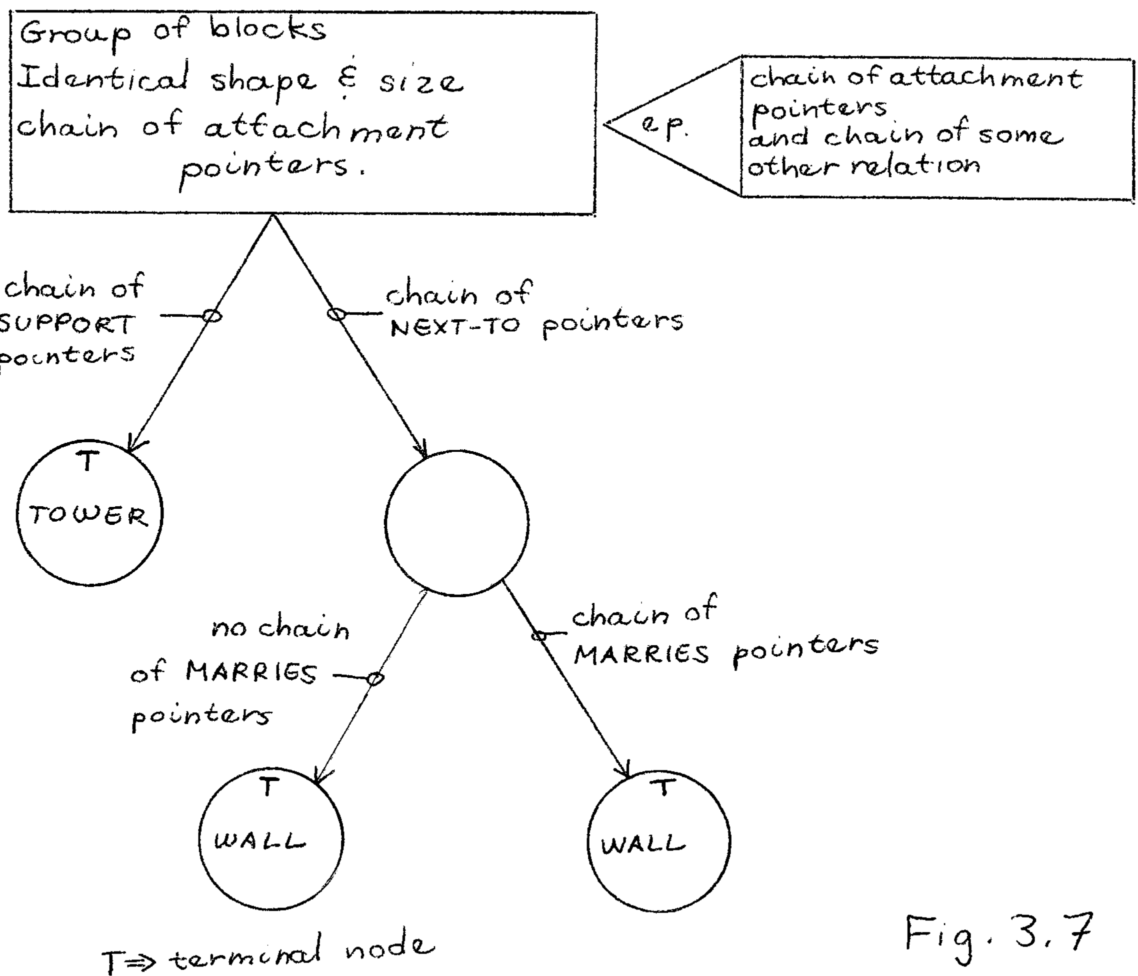


Fig. 3.7

arises with regard to the entry point skeleton, since it no longer matches the properties of all subframes. The skeleton (in this instance) will be amended to match any group of objects with chains of attachment and MARRIES pointers. This is not especially what we would like generally, so we introduce a new "wall" (figure 3.6) in which the blocks do not marry. This will be added to the hierarchy under the wall category. Now one might expect that the skeleton be reduced to the chain of attachment pointers. This is not particularly desirable since such a skeleton would be so general as to match nearly anything. We would expect that a store of general information about the domain (perhaps analogous to Gerry Sussman's "blocks world knowledge library" <5; p.41>) could communicate this to the skeleton builder, which might then add any restriction common to all subframes. For example:

(a) that all the objects must be of identical shape & size,

or

(b) the chain of attachment pointers be accompanied by a chain of pointers representing some other spatial relationship.

Assuming we pick b, our new hierarchy looks like fig 3.7. Notice that we perhaps have more test-frames than is really desirable, i.e. the two "wall" terminals could be collapsed. This, however, is an open question. Perhaps at some point (when the number of example walls is quite large) we will want to separate walls on the basis of whether they "look nice" or not. The decision to collapse test-frames should only be made when the structure becomes extremely cumbersome, and then on the basis of special purpose knowledge about the relative importance of

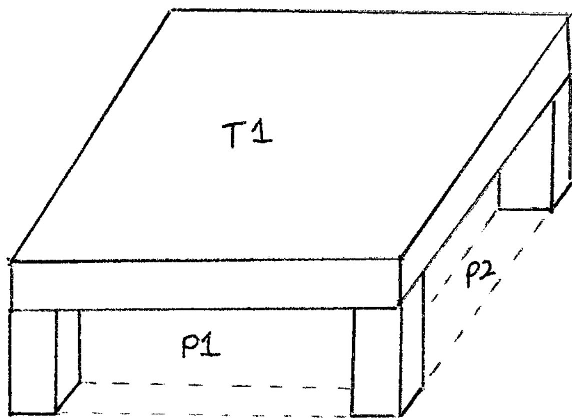


Fig. 3.8

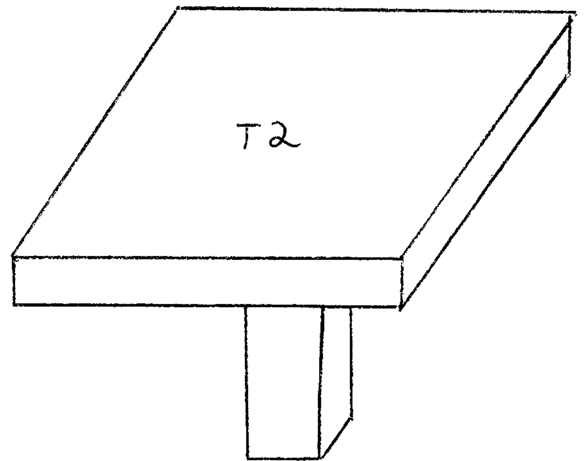


Fig. 3.9

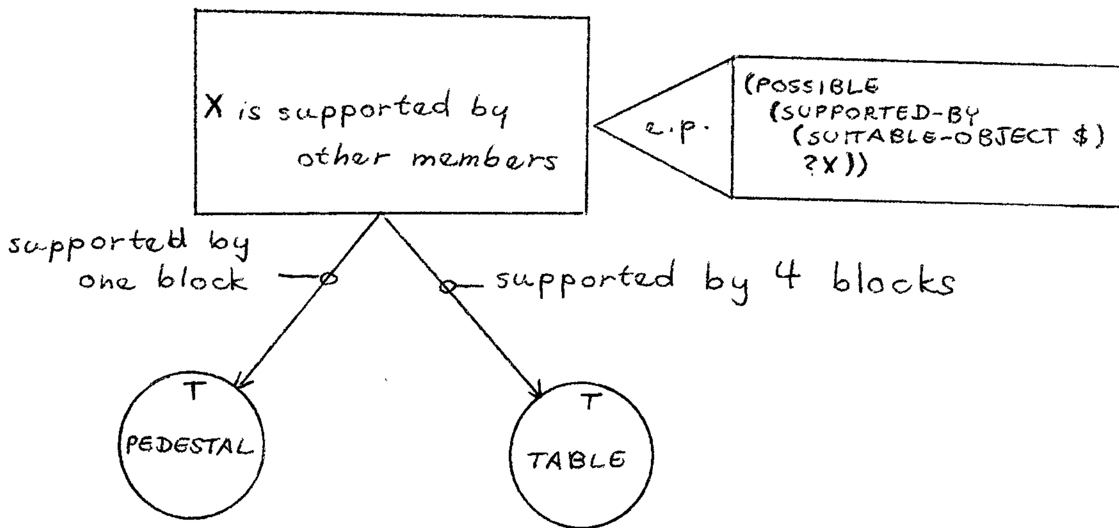


Fig. 3.10

relationships. At this point we only have a total of 5 frames, so we can well afford the luxury of two terminals for "wall".

Now, suppose we introduce a typical table (figure 3.8). The skeleton created will be:

```
(POSSIBLE (SUPPORTED-BY (SUITABLE-OBJECT $) T1)
(POSSIBLE (COVERED-BY (SUITABLE-OBJECT $) T1))
(POSSIBLE (PASS (SUITABLE-OBJECT $) P1 THRU))
(POSSIBLE (PASS (SUITABLE-OBJECT $) P2 THRU))
(POSSIBLE (PASS (SUITABLE-OBJECT $) P3 THRU))
(POSSIBLE (PASS (SUITABLE-OBJECT $) P4 THRU))
```

where T1 is the top block and P1 - P4 are the ports created by the legs. This skeleton matches no current entry point in the frame system. So we have good reason to believe that we should create a new hierarchy. We do so, giving the entry point skeleton shown above. Now, suppose we introduce a pedestal (figure 3.9) into the system. the skeleton created will be:

```
(POSSIBLE (SUPPORTED-BY (SUITABLE-OBJECT $) T2)).
```

This partially matches the entry point for table and we attempt a match there. The match will not be particularly successful, but we don't have anything else to do. So our machine will half-heartedly propose that what we have here is another table. We will, with paternal patience, instruct the machine that this is in fact a pedestal. The machine will reform the hierarchy, giving us the one shown in fig 3.10. The entry point skeleton will tentatively be reduced to:

(POSSIBLE (SUPPORTED-BY (SUITABLE-OBJECT $) ?X))
(HAS-PART $ X)

This may be somewhat weak, (though clearly not as weak as a chain of attachment pointers) but we will have the option of strengthenting it or adding new entry points later. For now such a weak skeleton is fine.

Suppose now we introduce the pedestal of fig 3.11. The chain of SUPPORTED-BY pointers will match the entry point in the wall-tower hierarchy, while the support offered by the top block will match the entry point in the table-pedestal hierarchy. This will be the first time we have had a choice of two entry points for an SD. We will perhaps need to call a routine to classify them, as described in section 3.2. Using the knowledge that groups of similar attached objects are much more likely to be substructures than complete structures we would place the wall-tower entry point in class (a) and the table-pedestal in class (c). Using the entry point in (a) first we would match the tower substructure. This would cause us to modify our structural description of T3's supporting group by the designation

(TOWER (some pointer to the example frame))

and hang the example under the appropriate terminal. Then we would compare our modified SD with the entry point in class (c). Once we got into the frame, we would not be able to progress further because the pointers to terminals refer to the blocks, and in the current strucutral description we have a tower instead of a block. Since we know that the tower is a grouped substructure, we may undo the grouping and proceed. Now we make it down the the pedestal terminal (though perhaps for the wrong reason, i.e. that the large block is supported by the top block of

the tower. This fulfills the support requirement but fails to account for the rest of the tower.) But here we have matching problems because there are several blocks unaccounted for. Our program will most likely propose a substructure "pedestal" consisting of the top two blocks in the structure. Now we are stuck because we have two overlapping substructures, but no classification for the whole, and we ask for help. The program is informed that the whole structure is actually a pedestal. This type of problem may be expected quite often in a recognition system which is always attempting to generalize its descriptions, and there seem to be many ways to surmount it. The error made by our program was actually in backing up. If it had stuck with the tower subgroup and admitted defeat at that point, it would have learned that a pedestal may be supported by a tower, and have been able to fix the frame pointer so that a single tower or block would point toward selection of a pedestal. But since we have discarded the original tower substructure, we are faced with the prospect of embedding an additional tower description (and worse, not even a general one) under the pedestal description. Furthermore, since the test-frame pointer specifying support by a single block eventually worked, it has little chance of being modified. It seems we will be screwed by our own conservatism. One suggestion might be to treat structures in the tower-wall hierarchy as single units, since they are such functionally, and also have some claim to being the most basic of all substructures. (Any block may be replaced by a group formed by slicing the block up and gluing the pieces back together.) But this patch will only help us when the group concerned is a tower or wall.

Perhaps a better strategy would be to make the program wary of backing up too often. When a substructure in the SD occurs in one-to-one correspondence with a specific block formation noted in the frame system, the program could call a routine which checks if the substructure may serve the same purpose as the formation. Or, at each point where it is forced to discard a previously accomplished substructure, it could suspect lack of generality as the problem (especially if the substructure is a fairly basic one), and temporarily alter the frame system to see if it got to the right place. Still another alternative would be to take a more general view of the frame pointer tests. In this specific example, the machine might notice that the essential difference in pointers was between one and four, the number of supports, and the best match would clearly be the "one". This technique appears most promising in terms of generality, because if the assumption proves valid, we do not have to invoke the hairy debugging procedure, because we know exactly which parts of which test-frames should be modified (i.e. in the pointer under consideration, the description "block" would be matched against "tower" and if we succeeded we would then return to include towers where previously only blocks were permitted.) When the program reached a correct acceptance it would know to amend such pointers, and if it reached an incorrect acceptance it would be capable of placing rejection conditions in that frame to prevent the same process recurring. Of course, this latter strategy requires the program's ability to compare and contrast sections of its own code. If the program has an adequate mechanism for modeling its own commands, the comparisons may be made by a

Fig. 3.11

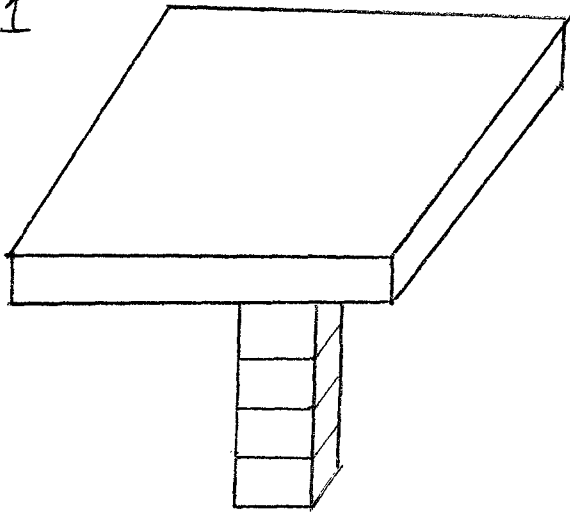


Fig. 3.12

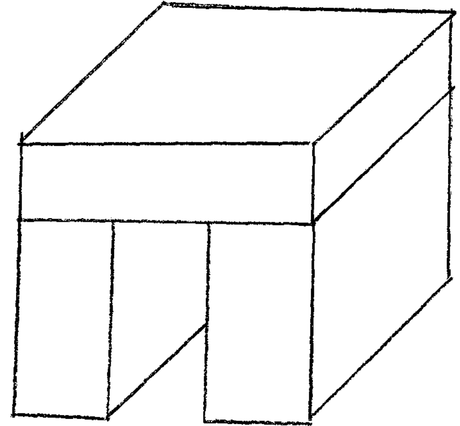


Fig. 3.13

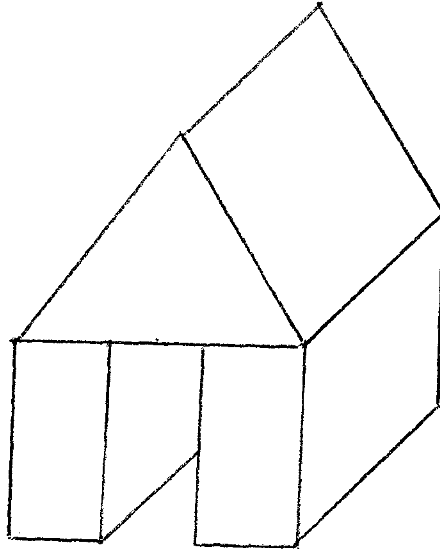
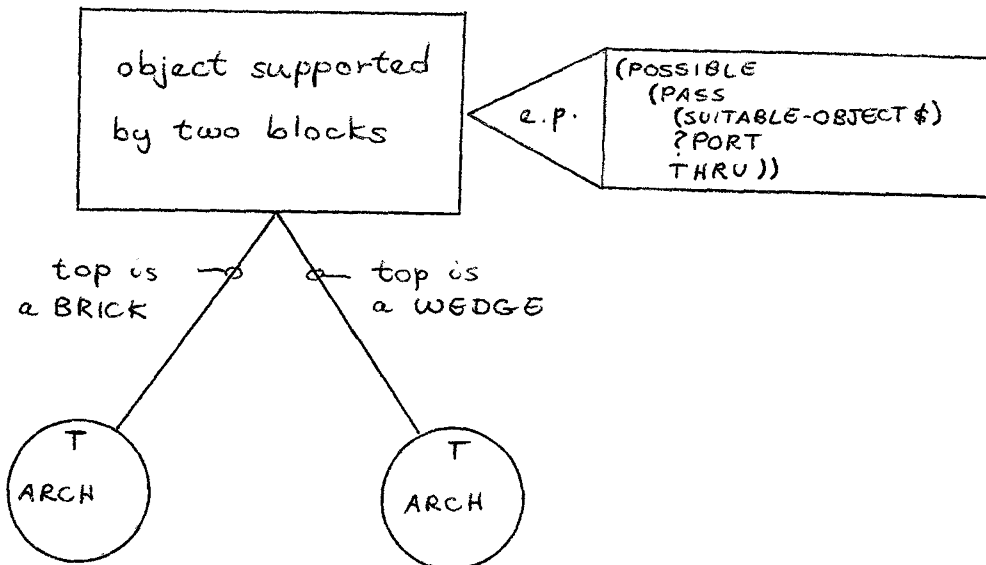


Fig. 3.14



program similar to Winston's analogy problem solution <6; p.105>. A related problem is that of a structure which gets classified, but which has an important subgroup not recognized at all before the classification is made. This problem will be discussed in the section on grouping.

Anyhow, let us say the above problem is resolved as desired, i.e. that the pedestal now accepts towers as legs. The program is now fed a standard arch (figure 3.12). We may wish it to create a separate hierarchy for the arch. But suppose not. The program will concievably determine that the arch is capable of supporting something as well as having the required port. Its skeleton would thus be:

```
(HAS-PART $ ?X)
(POSSIBLE (SUPPORTED-BY (SUITABLE-OBJECT $) X))
(POSSIBLE (PASS (SUITABLE-OBJECT $) $ THRU))
```

It will match in the table-pedestal hierarchy, and cause a new frame "arch" under the hierarchy, pointed to in case there are two supports instead of one or four. Alternatively, (depending on our whim) we could inform the program that it is a special type of table, in which case the pointer to table frames would be generalized from "4" to "more than one." Supposing the former happens, we now feed in a pointed arch (figure 3.13). Its skeleton will be

```
(POSSIBLE (PASS (SUITABLE-OBJECT $) P1 THRU))
```

which matches no entry point. Our program will create a new hierarchy for this item. Imagine the machine's suprise on being told that this too is an arch! It now has two terminals labeled "arch" in completely different hierarchies. Such a situation is intolerable and must be fixed. The new hierarchy cannot be incorporated under the old arch frame

because it will not match the skeleton. The only thing to do is to move our old arch description under the new hierarchy, since its skeleton will match the others to some extent. This will entail deleting a frame from the table-pedestal hierarchy. Our final arch hierarchy will look like figure 3.14.

Thus, the frame system gets constructed by gradual debugging as new examples are added. Much of the structure will be dependent on what we choose to tell the program. But regardless of the actual structure, we can be reasonably sure of having a system which will classify items much faster than an attempted match with all known models.

4. Grouping

Many of the unresolved problems mentioned in this paper stem from the lack of a good theory of adequate grouping mechanisms. Since the purpose of grouping is to provide for simple generalization of description along many lines, many different types of activities will come under the heading of "grouping". A grouper will really be a collection of mechanisms looking for a chance to apply themselves rather than one simple coherent procedure.

Winston's program uses two general grouping heuristics:

(a) sequences of objects chained by the same pointer <6; p.84>

and

(b) sets of objects or structures with sufficiently similar properties rated on a percentage basis. <6; p.87>

No doubt (a) is a valuable heuristic, but I have the feeling that (b), as Winston constructed it, may be of doubtful use in a function oriented system. Rather than debate the merits of Winston's grouper, however, I would like to offer some suggestions for a more general set of procedures which I feel will be useful to my proposed system.

Since the system is expected to have knowledge of attachment between blocks, one strong technique will be to divide an SD into its attached components. There are basically two reasons for this. First, objects attached to each other must move as a group and thus may to some extent be "melted down" into a simpler representation which differs only in the

number of elements, and not in any more fundamental relationships. It has already been observed, for example, that any block would serve exactly the same purposes if it were sliced up and then glued back together. Secondly, in the real world, objects are usually attached for some reason, and attachment is strongly indicative that the group has some basic functional property. Furthermore, if the attached subgroups of an SD are considered separately, and one at a time, much irrelevant detail may be discarded, and we will be likely to obtain a clearer picture of the more general structure of the total description. Consequently I think it advisable that when an SD of any complexity is being considered, it be "pre-processed" by classifying as well as possible all its attached subgroups as if they were independent and then modifying the SD so as to take this information into account. This raises many questions, of course, about how to represent intricate relationships between subgroups and how to provide for the complicated backup procedures which may be necessary. It may prove necessary to keep several descriptions of a given structure on hand, each at a different level of generality.

Another suggestion is to greatly increase the ability of the machine to ask questions. One fault of Winston's program is that it is in a sense expected to be smarter than humans. It is forced to provide alternative structures in many cases where a human student would stop and ask a question of the form "Must it be this way?" or "May it be modified like this?" Such questioning ability, however, if used too liberally, can subtly shift the "learning ability" right back to the programmer. It

seems quite feasible, for instance, to replace Winston's program with a much less clever one which would simply ask all possible important questions and produce the same results. But there are certain instances where questions would enable the program to learn much faster, though not give it any appearance of power it did not actually have. Such questions might be:

1) finalizing the description of substructures before attempting to classify the entire structure.

2) avoiding messy backups such as indicated in the example of section 3.4 by asking if a particular group could serve as well as a block in a particular instance.

3) requesting the names of subclassifications when they seem desirable.

4) asking if new hierarchies should be constructed when the program cannot find a suitable spot for a given structure.

Another form of grouping which seems quite desirable is a search for fundamental substructures in a large structure. The class (a) of entry points described in section 3.2 is a partial attempt to bring this about. But it may be useful to define simple structures as "basic" in the sense that they are exhaustively searched for in the pre-processing stage, regardless of whether their presence is indicated by a structure's skeleton. Walls and towers are good candidates because their essential properties are quite unlikely to appear in any condensed skeleton of a large structure. Others may be defined as "basic" if the general management programs discussed below discover that they are missed quite often.

Another heuristic which I feel might be valuable is what I would call the "immediate induction" heuristic <6;p87>, which, when it discovers a group of substructures sharing the same function whose number is three or greater, immediately generalizes to assume that the structure may contain any number greater than two of such elements. Using this heuristic, the number of elements permitted in towers, walls, and supports for a table in the example of section 3.4 would be immediately generalized. Groups of this class seem far more abundant than groups (like the sides of a triangle or the faces of a cube) which depend on a specific number of such elements.

One of the most obscure problems I have encountered in thinking about this proposed system is the question of what should be done when a structure manages to be classified, and yet a significant subgroup is not discovered, (or worse yet, not even known) before such classification. Too many occurrences of this situation could defeat the generality of the frame system and be very costly in terms of time or examples. It would be nice if our program were detecting clues of this sort of behavior (such as discovery of a terminal frame which is always arrived at easily only to require some small modification in the terminal frame model), but such clues are hard to come by. Perhaps a better suggestion would be a general management system which spends idle time searching for large ungrouped segments in the terminal frame models and attempting to replace them with an appropriate substructure pointer, or taking structures which have been recently defined and searching the frame system for instances of their occurrence. Of course such unsupervised "play" may have

dangerous consequences unless the machine is obsequious enough to present its findings for human approval. Far more desirable, though, would be a general algorithm for discovering such instances at classification time. However in cases where the desired substructure has not even been defined, the process will be difficult if not impossible.

One might criticise the heuristics above (as well as some aspects of the test-frame system operation) as being biased in favor of immediate and sometimes unwarranted generalization. I feel such bias is justified for two reasons. First, in the real world it is rare that some but not all of a given class of substructures is permitted in the general class of some larger structure. (One rarely builds a mansion with a plywood door, but would this be criterion for asserting that such a building could not be a mansion?) Secondly, a mistaken generalization can be undone (if as proposed we save examples known to be correct) by a single counterexample. A more conservative generalization process however, would require several examples of occurrences of a substructure in each structure in which it occurs, before the evidence is considered justified for generalization to a class of substructures. Furthermore, such a procedure would require constant observation of many points in the test-frame system which are current candidates for generalization. This would present a significant drain on the machine's energy. Ability to generalize, even if somewhat hastily, enhances the effectiveness of the learning process.

5. Conclusions

Since the ideas presented in this paper are still of such a tentative nature, it is hardly appropriate to make any far-reaching claims as to the possibilities of such a system. I feel, however, that the generality of the system proposed should enable us to extend some of these ideas to other areas. Discussion of the frame system already exists <3>. The representation of function is still far too limited to provide much information about wider applicability. However, I feel it is instructive to notice the possibilities of using relationships between different types of properties of objects. Such relationships are potentially of great use in environments in which it is necessary to process several different descriptions (i.e. from totally different criteria) of each object encountered.

REFERENCES

(1) Goguen L. "The Logic of Inexact Concepts"
Synthese 19 (1968-1969) pp. 325-375

(2) Lakoff G. "Hedges: A Study in Meaning Criteria and the Logic of Fuzzy Concepts"
University of Michigan and Center for the Advanced Study of the Behavioral Sciences

(3) Minsky M. "Frames"
A. I. draft; April 4, 1973

(4) Newell, A. and Simon, H. "Human Problem Solving"
Prentice-Hall; New Jersey; 1972

(5) Sussman, Gerry "A Computational Model of Skill Acquisition"
M.I.T. Artificial Intelligence Technical Report 297;
May 1973

(6) Winston Pat "Learning Structural Descriptions From Examples"
M.I.T. Artificial Intelligence Technical Report 231;
September, 1970