A FRAME SYSTEM MODEL OF THE REPRESENTATION OF KNOWLEDGE FOR UNDERSTANDING NATURAL LANGUAGE

by

DAVID BLAIR McDONALD

B.Sc., University of British Columbia, 1974.

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computer Science

We accept this thesis as conforming to the

required standard

The University of British Columbia

August, 1975

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date AUG. 26, 1975

Abstract

A model for the representation of knowledge is described. It was developed in the domain of teaching a student how to draw simple figures on a graphic terminal's screen. The student should not need to know anything about computers or graphics and should interact with the computer using a subset of English. The model was designed to represent the knowledge needed to understand sentences encountered in this domain.

The model is based on the ideas in Minsky's frame systems, Hewitt's actor formalism and semantic network theory. The basic components of the model and how they are used to represent knowledge is presented. A detailed example is described, giving some details of how the system performs inferences and how it responds to various inputs. Finally some conclusions are drawn about the design of this representation for knowledge.

Table of Contents

Table of Figures

Acknowledgement

The financial assistance of the Department of Communications and the National Research Council of Canada is gratefully acknowledged.

Chapter 1.

Introduction

This thesis describes a model of the representation of knowledge for understanding natural language. The first chapter gives a brief introduction to the problem domain. The second chapter contains a brief survey of the more important papers that influenced this thesis. The third chapter describes the basic components of the model and some of the built in functions that the system uses. The fourth chapter goes into the details of the frames that are necessary to process natural language and also goes into a detailed example of how the frames interact to understand English sentences. Finally some conclusions about frames and this system are presented.

In Artificial Intelligence there has been a great deal of effort to develop a representation for knowledge. Considerable effort has been devoted to the problem of developing a method to handle large amounts of both procedural and declarative knowledge. The lack of a definite model for representing knowledge has been a major stumbling block for Artificial Intelligence over the past few years. There have been some successful projects carried out which satisfactorily handle a very small subset of knowledge possessed and used every day by a human being. These systems have not been extensible to any great degree beyond the domain for which they were designed. So far there has been no model which can adequately represent and

process information at the level of competence of a person.

Computers have been used to greatly speed up the amount of computation that people can do in a given time period. However, this great speed can only be used to solve specific well defined problems with a well defined solution in a given domain. Programs can be written which can solve many different problems in a domain. However, once any input from outside this domain is given, the program is completely lost and will either give an aknowledgement of its loss for a solution (if the programmer has foreseen this problem), give incorrect results or may even produce a program interrupt.

Computers have allowed physical and social scientists to perform tasks which would otherwise be impossible. However, there is no room for error in the input. If incorrect input is given to a program, then the program will produce incorrect answers (i.e. garbage in, garbage out).

Humans on the other hand are "intelligent" in the sense that if presented with incorrect input, they will normally notice this and either complain or make some assumption about why the input is incorrect and carry on from there. In the domain of computational linguistics when a parser is presented with a syntactically incorrect sentence and is unable to parse the sentence, then normally the program will just fail and say something like "I couldn't understand that, please try again". People, however, in most cases seem to have the ability to digest syntactically incorrect sentences and still understand their underlying meaning. This is more of a criticism of the

standard approach to the solution of problems in Computational Linguistics than a statement that computers will never be able to understand natural language. What must be done is to take a approach which uses semantics at the same time as the operation of parsing is performed and possibly suspend the parsing of the whole sentence and just parse constituents such as noun phrases, prepositional phrases, verbs, etc.

In this thesis a model for representing knowledge is proposed. This model is not fully developed yet and still needs much work. The model was developed from problems associated with teaching a person who knew nothing about computers how to use a graphics terminal. A brief description of this problem domain and some of the problems associated with it will now be described.

1.1. Description of the Problem Domain

The motivation for this thesis arose from a project carried out at the University of British Columbia to develop a computer program which would be able to teach a person who knew nothing about computer graphics how to use a graphics terminal. The computer was to interact with the student using a subset of English, so that the student would be more at ease.

The idea was to display a scene or figure such as a simple house and then have the student reconstruct this figure, by asking the computer (in English) to draw subcomponents, to move subcomponents about the graphics screen, etc. This was meant to

give the student a feel for the capabilities of a graphics terminal. When the student had finished this phase, he would be shown how to use the hardware features of the graphics terminal to perform the same tasks much more efficiently and easily.

As far as natural language understanding was concerned the first phase of the teaching session would be by far the most difficult. In the second phase, the student would most likely be asking only for new components to be drawn and then manipulate them using the graphics hardware. The student at this point might want to ask questions about how to do something using the graphics hardware features. This sort of dialogue has been handled very well by such systems as SCHOLAR [3,5] and SOPHIE[2] using semantic networks. The first phase must be able to understand and respond to far more difficult utterances.

The following sample sentences could arise when the student is asked to draw a simple house.

Sentence	Action
Draw a big square.	This sentence is easy to handle and a big square will be displayed on the graphics screen.
Make it bigger.	The anaphoric reference "it" must be disambiguated. In this case there is only one possible candidate and so the square will be made bigger.
Draw a triangle.	This sentence is easy to handle. The knowledge that the triangle will eventually form part of the house may be used to perform inferences on the following sentences.

Make it big.	The knowledge that the triangle is part of the house must now be used to decide how big to make the triangle. The actual size should be such that the base of the triangle is the same width as the top of the square.
Move it up a bit.	Again the word "it" must be disambiguated. Also, the system must decide what amount "a bit" is.
Higher.	This is a sentence fragment which must be handled by the system. The system must realize that this fragment is asking for the square to be moved even higher.
Move it up an inch.	This sentence is again easy to handle. The system should use its knowledge to make the bottom of the triangle level with the top of the square and move it up about an inch but not necessarily exactly one inch. At this point the system must realize that the main structure of the house has been drawn. This will allow the student to refer to the structure as "the house".
Draw an upright rectangle.	This is again a simple sentence to handle. The system will assume that the rectangle is to be the main body of the door of the house.
Make it smaller.	The rectangle will be made smaller. The knowledge that the door must fit inside the square may be used to determine how small to make the rectangle.
Thinner.	Another sentence fragment which the system must be able to handle. In this case the door will be made thinner with the amount obtained from the fact that a door is supposed to be about half as wide as it is high.
Move it down.	The system should realize that the bottom of the rectangle must be equal to the bottom of the square. Thus the rectangle may be moved down to satisfy this relation.

Draw a very small circle.	This sentence will cause a very small circle to be drawn. The system will assume that it is meant to represent the door knob for the door. The student will have to move the circle around until it is in the correct position relative to the other components.
Draw a small square inside the front.	This sentence is easy to handle, except there is no figure on the screen with the name "front". The system must be able to recognize when a front of a house has come into existence. This will allow the student to reference components of a figure by name rather than a phrase such as "the big square under the triangle".

As can be seen from the examples, people tend not to utter complete syntactically well formed sentences, especially when they must type everything at a computer terminal. These examples begin to show that the system would have to be able to understand sentence fragments as well as syntactically well formed sentences. Some of the sentences are easy to handle using standard techniques, but, some of the fragments are extremely difficult. Also anaphoric references are used very heavily.

An associated problem is to decide on an internal structure which will allow a flexible method of representing the current state of the conversation, together with the current state of the figure being drawn. This information could then be used to help understand the utterances of the student. Also, sufficient information should be kept so that the system will have some expectation of what the student is going to do next. This will help the program to disambiguate utterances which would not

normally be understood.

Chapter 2.

A Brief Survey of Relevant Work

Several different and diverse areas of Artificial Intelligence have influenced the direction this thesis has taken. Minsky's frame systems [17] and Hewitt's actor formalisms [11,12,23] are the two most important ideas used. Semantic networks [2,3,18] have also played a significant role.

2.1. Minsky's Frame Systems

The most important aspects of this thesis are involved with the use of Minsky's frame systems to represent knowledge for understanding natural language. Minsky's frame systems paper [17] suggests an approach for the representation of knowledge. The idea behind frames is that when one encounters a new situation (or makes a substantial change in one's view of the current situation) a frame is selected from memory. This is a remembered structure which should very closely match the current situation. This frame may need to be adapted to fit the current situation.

A frame is a structure for representing a stereotype situation, such as what a graphical house should look like or what a graphical tree should look like, etc. Attached to each frame are several pieces of different information. Some of this information will consist of how the frame is to be used, some

expectation of what should happen next, what to do next if these expectations are not met, etc.

A frame can be considered as a network of nodes and relations between the nodes. In each frame some information is absolutely fixed and must be true for the supposed situation, no matter what. The frame also contains terminals which are slots which may be filled with specific instances or data. The slots may have associated with them conditions which must be met for the assignment to this slot to be allowed. The assignments to the slots are usually also frames (or subframes). Conditions may consist of a property which a subframe must have, such as it must be a square or it must be blue, etc. More complex conditions may be specified by relationships between the slot concerned and other slots of the frame. Collections of related frames are gathered together into frame systems.

Minsky is mainly concerned with problems of representation of knowledge necessary for visual image processing, even though he mentions other applications for his theory. In vision processing the different frames of the system will specify the scene from different points of view. This will mean that different frames of the system may share the same terminal slot assignment.

Much of the power of the theory hinges on the inclusion of expectation and other information in the frame. A frame's terminal slots are normally filled with default values. In this way a frame may contain many assumptions not necessarily provided by the actual situation. In the same way that a person

may have a favourite graphical object (e.g. a circle), frames may have this property also. For example, the frame for a graphical object could have a default value of a circle. When it is asked to draw an object on a graphics terminal screen rather than drawing a randomly chosen object, it could draw its favourite object, i.e. a circle. The person who originally designed and wrote the frame would most likely pick his own favourite object.

Frame systems are linked by an information retrieval net so that when the current frame does not successfully fit the current situation, a new frame may be provided by the retrieval net. Once a frame is proposed to represent the current situation, a matching process must be performed so that the terminal slots of a frame are correctly assigned according to the current situation. The frame itself will direct the pattern matching process according to the current goals of the system and will include information about how to handle surprises or anomalies.

The frames in my system are similar to but not the same as Minsky's. For one thing mine are designed specifically for processing natural language, rather than visual images. They must be able to perform some pattern recognition to be able to understand the sentences described in the Introduction. The frames in this system are meant to be specialists which know everything about one single concept. Anything which has to do with a concept must pass through this one frame. The concept cannot be used in any other way except through its frame. If a

word has more than one meaning then there must be frames defined for each of these meanings and these would be subsets of the frame for the word itself. The frames themselves are defined in the Conniver programming language using some of the ideas of Hewitt's actor formalism. Thus in this case a frame is not a data structure, but a procedure or an actor (using Hewitt's terminology).

2.2. Hewitt's Actor Formalism

Hewitt has developed from his original Planner language [10] a new formalism for Artificial Intelligence programming languages [11,12,23]. This formalism consists of what he calls actors. An actor is sent a message (by another actor) and carries out some action according to its "script" or body. The message is pattern matched against a pattern which is specified in the definition of the actor. If this pattern match fails then the actor is not applicable to the message and thus will not be activated. The script for an actor may cause messages to be sent to other actors or even to itself (recursion is immediately possible). Sending and receiving messages by actors is the only possible form of communication.

When a message is sent to an actor the message may contain both a continuation point (CP) and a complaint department (CD). These two parts of the message are optional and if they are not specified then the actor receiving the message must continue the computation on its own. The CP specifies where the result of

the computation is to be sent when the actor has completed its processing of the message. The CD is where an error message is to be sent when the actor detects an error condition. Both the CP and the CD are actors in their own right and will behave accordingly. An actor can easily simulate a procedure by having the actor which represents the calling procedure send a message to the actor defining the procedure, specifying the parameters to it. The actor defining the procedure would have a pattern in its definition which would assign the values of the parameters to variables. The CP of the parameter message to the actor defining the procedure could be made to return control to the calling actor in much the same way that a procedure returns control to its caller.

Hewitt claims that the actor formalism accounts for all possible data and control structures, including: semaphores, monitors, ports, descriptions, semantic nets, logical formulae, numbers, identifiers, demons, processes, contexts and data bases. That is all the above are special cases of the actor formalism which is supposed to be a general formalism for current programming language structures. As such it is a powerful concept useful for giving the programmer full control over the computation as it is carried out. A very important feature about the actor formalism is that the way an actor actually performs its computation is hidden from the user who has no access to it. The only information available about an actor is what Hewitt calls a contract. A contract is a specification of what an actor does and what it returns

according to the message sent to it. This does not specify how the actor performs this action. Thus an actor may be replaced by another actor which behaves exactly the same as the original (externally at least) even though it may perform the computation in a different manner. For example, an actor which uses recursion to perform its computation may be replaced by another actor which uses iteration to achieve the same result. The system should then perform as it previously did without changing any specifications of the other actors in the system.

Hewitt's actor formalism has been used in this thesis to implement a frame system in the spirit of Minsky's frame systems paper [17]. Changes have been made to the way that actors work even though they are similar to Hewitt's actors. The actor formalism is embedded in Conniver functions and thus are heavily dependent on the Conniver control and data base facilities. The basic idea is that actors can only interact with other actors through sending and receiving messages. Routines are provided to send and receive messages much in the same way that actors send and receive messages, except that the user usually has to explicitly state when to send and when to receive a message.

2.3. Semantic Networks

Semantic networks have been designed as a method for representing knowledge. They have been successfully used to represent knowledge for understanding natural language in both SOPHIE [2] and in SCHOLAR [3,5]. In both systems the problem

domain has been a very limited one. SOPHIE is limited to electronic circuit diagrams and SCHOLAR was originally designed for computer aided teaching of South American geography. SCHOLAR has been adapted to handle "how to" questions for teaching information about the ARPA network [5]. In all cases these systems have been very successful.

Semantic networks consists of nodes and links. The nodes usually denote words that the system knows about and the links specify relationships between the nodes. Some of the relationships which are often used in semantic networks are: superset, subset, is-a, synonym, has-part, etc. The relationships actually used in any one system depend heavily on what the domain of the system is. Most systems based on semantic networks use either the superset or is-a relationship. These relationships are used to specify hierarchies such as: a dog is a mammal is an animal, or a dog has superset mammal has superset animal.

The problem with semantic networks is that they can become very ad hoc. That is the types of links between nodes are not well defined and tend to be created as they are needed by the designer of the system. However, if the relationships are well defined such as superset, subset and others then semantic networks can greatly improve the efficiency and inferencing power of a system. This is so because there is no need to repeat redundant information, since by using superset or is-a links, the more general nodes can be used to represent general information about a concept. For example, if there is a

hierarchy of the form a parrot has superset bird, then the information that a parrot has feathers does not need to be associated with the node for parrot, but may be inherited from the node for bird (since all birds have feathers).

Semantic networks are used in this thesis in a very limited way. Only the more well defined relationships are used, such as superset, subset (the inverse of superset) and synonym links. The superset links are used in the system so that redundant information is not associated with many different nodes. More general information will be kept in a node and all the nodes which need access to this information will be subsets of this general node. Synonym links will point from one node to another node which has the same meaning as the first node so that synonyms can be used without actually having to keep multiple copies of the same information around. For example, the adjectives large and big are to all intents and purposes synonyms. Thus rather than having the same information for both, there will be one piece of information, say associated with the node for big.

2.4. Fahlman's Work on Frames

Since Minsky's frame systems paper, Fahlman has produced a thesis proposal [7] and a thesis progress report [8] on the implementation of frames. The thesis proposal suggested an implementation method for recognition tasks by specifying packets of demons which would be looking for expected features

of the object under consideration. These packets of demons could bring in more packets of demons when they considered it necessary.

In his progress report, Fahlman repudiates the approach described in his proposal and completely revises his recognition mechanisms. He is now proposing a system based on a network very similar to semantic networks. His system has nodes and links much in the same way that semantic networks do. He proposes a set of fairly rigid organizational principles (such as a hierarchy of is-a relations). The system will use parallel marking of nodes for recognition. Trees can be marked very efficiently by sending commands in parallel to the nodes via the links.

The is-a hierarchy is efficiently implemented. Members of a class inherit all the properties of a class and all its super-classes. Access to these properties is very fast. The system is capable of handling any number of orthogonal is-a hierarchies. These is-a hierarchies may be tangled to any desired extent. This system makes it possible to very easily find the intersection of two or more classes. These classes can be explicit groupings represented in the is-a hierarchy or they can be implicit groupings based on some common property of the members.

Each piece of information in the system will exist in some context and is only visible when that context is active. This allows the system to maintain many distinct world models. Information not in an active context is completely out of the

way and does not slow down the accessing machinery. Contexts form tangled hierarchies with different levels of generality. One can operate in a very general context with a lot of information available or in a very specialized context with much less potential for ambiguity and confusion. The system may focus its attention by moving up or down this hierarchy.

A major point of his work is that his system will need virtually unlimited parallel processing capabilities if the system is going to perform up to his expectations in a very large domain of knowledge.

Chapter 3.

The Basic Components of the Model

This chapter describes the basic components of a prototype model for the representation of knowledge used to solve some of the problems described in the Introduction. The structure of frames, how frames communicate with one another and how inferences are performed will be given. These components and how they work are not complete yet, at least for a general representation for knowledge, but seem to work well for the present domain.

3.1. The Structure of Frames

The first important step is to define what is meant by a frame. A frame is a node in a network. Each such node contains a packet of knowledge that is specific to that node. These nodes may be linked together in three different ways, by implicit, explicit and dynamic links.

Explicit links are physical links between two frames. That is there are physical pointers between the two frames. This is useful for defining superset - subset, synonym, and other such relationships. Thus the frame will have some method of accessing frames pointed to by such links, but will not need to know the name of the frames at the time of the definition of the original frame.

Implicit links do not consist of physical links between two frames. These links are dependent on what is contained in the two frames. An implicit link exists between two frames if one of the frames knows of the existence of the other frame. Then the first frame may communicate with the second. However, there need not be a reverse link from the second frame to the first.

A dynamic link is formed when one frame tells another frame about itself. Then the second may communicate with the first. This is in effect a method of forming a reverse link to an implicit or explicit link.

The frames themselves behave very much like Hewitt's actors [11,12,23], i.e. to activate a frame, a message must be sent to it. Optional CP and CD frames may also be sent with the message. When a frame receives a message, it will perform a pattern match on the message to determine what it is meant to do. If the match succeeds the frame may do one of two things:

1.) return control to the CP frame by sending a message to it (via the CP passed to the current frame).

or 2.) it may activate a new frame by sending a message to another frame. This new frame would then control what happens next.

Thus the external structure of a frame consists of a declaration of explicit links, such as superset, subset, synonym and other relationships in which it takes part with other frames. The external structure will also be specified by the messages that the frame knows about and how it responds to these messages.

The internal structure of a frame, on the other hand, consists of implicit links to other frames and the decoding of messages that are sent to the frame. The frame will then do whatever is necessary according to the message received. Thus the internal structure of a frame will contain information about how to perform actions which may be needed.

The internal structure also consists of what Minsky [17] calls slots. These slots will almost always be filled by a subframe of the current frame. For example, the frame for the front of a house which is to be drawn on a graphics terminal screen will have slots for a front body (the big square that is the major part of a graphical house), a door and a window. Also there are certain relationships that must be fulfilled by these subframes. The geometric relationships between the objects representing the front body, the door and the window may also be considered as slots inside the front frame which must also be filled before a front can be recognized by the system. Some or all of these slots may be optional.

3.2. Definition of Frames

In this section, some functions which are useful for defining frames are described.

1.)
```
(FRDEFUN NAME
     (BFRAME (LOCAL VARS) BODY)
     (EFRAME (LOCAL VARS) BODY))
```

FRDEFUN defines a frame. NAME is the name of the frame. BFRAME indicates that this frame is the basic frame (BF) and

contains the more general information about the concept to be represented by the frame. This form will always be present in a frame definition.

EFRAME, which is optional, indicates that this frame is an extension frame (EF) and contains information about a specific instance of the concept to be represented by the frame. A copy of the EF will be made to represent a specific object of this type when a new instance is to be created. This type of frame will have the same name as the BFRAME above with an integer concatenated on to the end of the name. This allows access to the EF by the BF.

LOCAL VARS are Conniver local variables which may be used inside the body of the frame without having to worry about their being destroyed by other procedures or frames which may be defined using the same local variables.

BODY consists of the body of the frame, i.e. Conniver code for decoding any messages received and performing any actions desired. This may consist of any Conniver code which is needed, as long as communication between frames is done only by the function SEND.

FRDEFUN may be used to define two frames at the same time. This is so that information about a concept can be split into two distinct frames. For example, the concept square is divided into two distinct frames. One frame (BF) consists of more general information about squares. For example, it would specify that the lengths of all the sides are equal. The other, more specific, frame (EF) will describe a specific square. For

example, it will contain information about the lengths of the sides of the square, the location of the square on the screen if it is being displayed, where the top of the square is, etc.

FRDEFUN generates code in front of BODY in order to receive a message when the frame is sent one. Thus the user need not explicitly write code to receive a message. The variables FRMESSAGE, FRCONTINUE and FRCOMPLAINT are assigned the message, the CP and the CD respectively. These variables can then be accessed by the user if he so wishes.

Other variables which are available when defining frames are: HISTORY-LIST, CURRENT-FRAME and PREVIOUS-FRAME. The variable HISTORY-LIST is a stack of all the messages sent to frames. Each element in this list contains the name of the frame sent the message, the message itself, the CP and the CD. This history list is used by the input frame for disambiguating sentences. This variable is also used by the superset processing routine to decide when the unwinding process of the superset processing should be terminated. CURRENT-FRAME is always bound to the name of the currently active frame. PREVIOUS-FRAME is always bound to the name of the frame which activated the currently active frame. This variable is often used by the input frame to help in disambiguating sentences.

2.) (CREATE FRAME)

Where FRAME is the name of a frame which has an EF. This function will create a copy of this EF and return the name of this new instance.

```
3.)  (CASES VARIABLE
            (PATTERN1 BODY1)
            (PATTERN2 BODY2)
            (PATTERN3 BODY3)
               .        .
               .        .
               .        .
            (PATTERNn BODYn)
```

where VARIABLE contains a message or some value to be used in pattern matching against the patterns PATTERN1, PATTERN2, ... , PATTERNn. If VARIABLE is not specified, then the variable FRMESSAGE will be used. PATTERN1, PATTERN2, ... , PATTERNn are patterns to be matched against the value of VARIABLE (see Appendix 1 for a brief description of the Conniver pattern matcher). If a pattern is the single atom ELSE:, then no matter what the value of VARIABLE is the pattern match will succeed.

BODYi are pieces of Conniver code which are executed when PATTERNi is successfully matched against the value of VARIABLE. Only the first successful match will count, i.e. CASES acts very much like a SELECT clause in LISP.

```
4.)  (EXPECT
            (PATTERN1 BODY1)
            (PATTERN2 BODY2)
            (PATTERN3 BODY3)
               .        .
               .        .
               .        .
            (PATTERNn BODYn)
```

where PATTERN1, PATTERN2, ... , PATTERNn and BODY1, BODY2, ... , BODYn have the same meaning as for CASES above. EXPECT causes a message to be received by the currently active frame. The variables EXPECT-MESSAGE, EXPECT-CONTINUE and EXPECT-COMPLAINT are assigned the values of the message, the CP and the CD sent to the current frame respectively. These variables are defined

only within an EXPECT clause. EXPECT is normally used immediately after a message is sent to another frame. EXPECT is used to check that when a return is made via the CP with some message, this message is one of the expected ones.

5.) (SEND FRAME MESSAGE CONTINUATION COMPLAINT)

SEND sends the message MESSAGE to the frame specified by FRAME. MESSAGE is a Conniver structure which will be evaluated. CONTINUATION is optional, and if specified should evaluate to the frame to which control should return after FRAME has finished its processing. If CONTINUATION is not specified, then the frame sent the message must not attempt to return control via the CP. COMPLAINT is also optional, and if specified should evaluate to a Conniver label in the current frame. This label will be made to look like a frame and will be sent to FRAME. If FRAME or possibly some of the frames it activates detect an error then control may be returned to this CD and the CD must decide what to do depending on what error message is returned.

All this information is put in a special Conniver context used solely for messages. This global context is available to all frames. At this point the current frame is suspended and the frame being sent the message is activated or reactivated depending on whether or not it has been used in the current computation. All the information concerning the activation of a frame is contained on the property list of the name of the frame.

6.) (RECEIVE MESSAGE CONTINUATION COMPLAINT)

RECEIVE is used to receive a message sent to the currently

active frame. MESSAGE is a variable which will be assigned the actual message sent to the current frame. CONTINUATION is optional, and if present will be assigned the CP if one was specified when the message was sent. If no CP was specified when the message was sent, then CONTINUATION will be set to NIL. If CONTINUATION is not specified, the CP sent with the message will be ignored. COMPLAINT is also optional, and if present will be assigned the CD if one was specified when the message was sent. If no CD was specified when the message was sent, then COMPLAINT will be set to NIL. If COMPLAINT is not specified, the CD information sent with the message will be ignored.

```
7.)  (DEF-SUBSET (FRAME1 SUB11, SUB12, ...  , SUB1n)
                 (FRAME2 SUB21, SUB22, ...  , SUB2n)
                    .                .
                    .                .
                    .                .
                 (FRAMEm SUBm1, SUBm2, ...  , SUBmn)
```

This will define the frame FRAMEi to have as subset frames the frames SUBi1, SUBi2, ... , SUBin and the inverse, the frames SUBi1, SUBi2, ... , SUBin have as superset frame FRAMEi. A frame may have an arbitrary number of supersets.

When a frame receives a message which does not match any of the expected patterns, the frame may send this message to its supersets, to see if they know what the message means. If one of the superset frames does accept the message, then it will return control to the CP with some message, specifying what happened. If a superset frame is unable to understand a message, then it will send a message to the CD sent with the message it received, after making all efforts to understand the

message. When an invalid message is encountered superset frames may be invoked to as high a level as possible. Eventually control will return (via the CD's) to the original frame and it wil send an error message to its CD.

If all possible superset paths have been traced with the message, without success, then an error message will be sent to the CD corresponding to the message causing all the problems.

The following Conniver function performs this task.

8.) (SUPERSET-PROCESSOR MESSAGE) where MESSAGE is the message which so far has not been understood by the current frame. This routine also generates the appropriate error message and sends it to the CD if the superset processing fails.

The main purpose of having superset processing is to reduce the amount of information that must be contained in a frame. There are often many facts associated with a frame which are not specific to that one frame. Using the notion of supersets from semantic network theory [2,3,18], it is possible to associate more general information about a frame with the superset of that frame. Thus instead of putting as much as possible into a frame, the more general information will be put into its superset frame. The superset of a frame has that frame as a subset, so that processing may go in either direction, if necessary.

```
9.)  (DEF-SYNONYM (FRAME1 SYN11, SYN12, ...  , SYN1n)
                  (FRAME2 SYN21, SYN22, ...  , SYN2n)
                  (FRAME3 SYN31, SYN32, ...  , SYN3n)
                     .              .
                     .              .
                     .              .
                  (FRAMEm SYNm1, SYNm2, ...  , SYNmn))
```

This function defines FRAMEi to be the canonical frame for a word and SYNi1, SYNi2, ... , SYNin to be synonyms for the word.

When a message is sent to one of the synonyms for a frame, the procedure SEND checks to see if the frame being sent the message is defined. If it is not then SEND checks to see if it is defined as a synonym. If it is then rather than generating an error message, SEND sends the message to the canonical frame which defines the meaning of all the synonyms for that concept.

3.3. Error Processing

Error processing capabilities are very important for a system to be able to react correctly to an erroneous message passed to a frame or an erroneous assumption made by a frame. Erroneous assumptions by a frame are the most difficult errors to handle. Such errors must be handled specially and involve a great deal of effort to return the system to a state which consists of the correct assignments to the slots of the frames.

The detection of erroneous assumptions is also a major concern. It means that there must be demons present to check for certain conditions which cannot normally occur after a decision has been made. For example, if the student is in the process of drawing a house and the system suddenly discovers that the square it thought to be the front body is inside the square it thought to be the window, the system must take some corrective action. This involves reassigning slots in the front

body frame and the window frame to the correct squares.

Invalid messages sent to frames, on the other hand, are relatively easy to handle. When such a message is first detected, supersets of the frame detecting the error are first checked to see if any of them can comprehend what the message means. If the superset processing succeeds in understanding the message, then the frame that was originally sent the message will remain in control of the computation. If superset processing fails, then an error message is sent to the CD for the frame.

When an error occurs a message of the form:

(*ERROR TYPE ERROR-MESSAGE)

is sent to the CD, where TYPE specifies the type of the error. Some of the types that are presently in use are:

1.) *MESSAGE - This signifies that the error was caused by a message which did not match any of the patterns associated with the frame and that none of the superset frames of the frame could decode the message. ERROR-MESSAGE is just the erroneous message that caused the error.

2.) *FATAL - This is a message, which may only be sent to the top level frame. This message will be sent when a frame has detected an internal system error. At this point the system has no idea of what has gone wrong or what to do next. Such errors arise when a frame is activated, but no message was sent for it to receive, or a message is sent to an invalid frame, or possibly when a frame sends a message to a frame and receives back an unexpected message. At present these are the only times

that a fatal error may occur. Under a fatal error, the current computation is aborted.

3.) *FULL - This type of error is caused when a frame has already filled the slot corresponding to a message just received. *FULL errors are not fatal and are handled in the same way as the *MESSAGE error, except no superset processing is done.

Other types of errors may be defined as the need arises. There are always two frames necessary in any error type interaction. One frame must be able to detect an error and send a message to the second. The second frame must then be able to process the error when it occurs. This other frame must be the CD sent with any message that may cause this type of error.

To facilitate sending error messages to a frame, a function is provided.

10.) (COMPLAIN FRAME TYPE MESSAGE) where FRAME should evaluate to the name of the frame to which the error message is to be sent, TYPE is the error type, which is not evaluated. TYPE may be one of the errors mentioned above or it may be one the user wants to use. MESSAGE is optional, and is meant to provide more information about the error to the CD.

3.4. Inferencing and Demon Processing

Inferencing plays an important role in any system which attempts to understand natural language. In this system the messages passed from one frame to another perform this function.

To perform any inferencing in the system there is usually some interaction between the frames of the system. In some cases inferences will be performed within one frame, but this will not be the usual case. Inferences are needed when some action is to be performed by the system, or when some question has been asked of the system. Also, inferencing will be necessary in the process of generating some expectation for what the next input to the system might be.

Any time a message is sent to a frame inferencing is being performed. The message must be accepted by the receiving frame and should cause it to perform some actions and possibly respond with some answer. In the process of performing these actions the frame receiving the message must make sure that the message does not contradict any knowledge of the system (or at least not contradict any certain facts, even though some uncertain facts may be contradicted). The frame after performing any action must either return control to the CP associated with the message or continue the computation on its own.

Sending messages to a frame may be thought of as the invocation of a consequent theorem, such as in Micro-Planner[24]. However, there is no indiscriminant invocation of frames when a message is sent, as there may be when Micro-Planner theorems are invoked. Only the frame which is meant to receive the message can receive it, so it is the only one which will get control of the computation. This provides far greater control over the computation than is possible with Micro-Planner in general. Every theorem which

matches the pattern of the goal may be invoked in Micro-Planner, whereas in this frame system only the one frame may be invoked. This means that control is maintained within the system and there should be no chance of a runaway computation to occur as can happen in a Micro-Planner system.

Demon processing for the frame system corresponds to the antecedent theorem of Micro-Planner or the If-added method of Conniver. This type of process is the basis of the expectations generated by the system. In the same way that antecedent theorems in Micro-Planner are used to expect some fact to become evident, demons are used to expect something to happen within the system. When some frame is waiting for something to happen, it may send a demon message to the frame corresponding to this action. If a message which corresponds to the demon is encountered by the frame, then after finishing processing the message, it will send a message to the frame specified in the demon message. A function has been provided to perform this task.

11.) (DEMON-PROCESSOR DEMON-PATTERN MESSAGE CONTEXT) where DEMON-PATTERN is the pattern which will be used to retrieve the demon information from the context CONTEXT. MESSAGE will be used to construct the message that will be sent to the frame which is to receive the demon message. The actual message sent will be of the form:

(*OCCUR MESSAGE).

where MESSAGE is the same one that was passed to the function DEMON-PROCESSOR. The frame activated by the *OCCUR message will

then continue with its processing, but it must return control to the frame which sent the *OCCUR message. The computation will then be carried on from that point normally. This is very much like the control structure of the Micro-Planner antecedent theorem or the Conniver If-added method.

To facilitate sending demon messages a special function, DEMON is provided.

12.) (DEMON FRAME RFRAME MESSAGE) where FRAME specifies the frame to be sent the demon message, RFRAME is the frame to return control to (usually the currently active frame) when the event expected happens. Message specifies the information about what event is wanted.

13.) (CANCEL FRAME RFRAME MESSAGE)

Where FRAME, RFRAME and MESSAGE have the same meaning as for DEMON. CANCEL will cancel a demon which was created by the function DEMON.

The messages and demons of the frame system provide all the inferencing power that is necessary to understand the subset of natural language in the domain described in the Introduction. Micro-Planner uses backup to perform inferences, i.e. when one branch of the solution path fails, Micro-Planner backs up to the last decision point and tries another branch. It is not easy in Micro-Planner to use information about what went wrong to decide what branch should be taken next. Conniver, on the other hand, does allow the programmer the opportunity to make such decisions by allowing access to the possibilities list. This frame system allows at least as much control as Conniver, using the frame

system's error processing capabilities. When a frame makes a decision to send a message to another frame, it will normally send the name of its CD along with the message. In this way, when the frame being sent the message detects an error it may send an error message to the CD specifying what type of error has occured and an error message. When the computation is blocked because of a wrong decision, an error message will be sent to the CD. Using the information gathered from going down the wrong path and the error message, the CD can then make a wiser decision about which choice to try next. Thus the simple backup technique of Micro-Planner could easily be implemented by just blindly choosing the next branch of the path using the technique of this type of error processing. In most cases the system should pick the right path the first time, rather than trying many different paths before finding the correct one.

3.5. Context and Focus

Context plays an important role in the understanding of natural language. Context must often be used to comprehend English sentences that are ambiguous. The context for the system will be specified by the currently active frames and the currently active Conniver contexts. The frames will have slots filled and will contain other information which will specify the state of the system and will help in disambiguating many sentences.

This system depends heavily upon Conniver contexts to help

in the processing for the frame system. There is one global context which contains global information known to the system. It will also contain contexts for conversations and situations which are currently not applicable. This global context will contain information which is always true and is available to all the frames through the Conniver variable GLOBAL. This global context will contain information which will not usually be used during a computation once a subject has been chosen and its context has been instantiated.

When the system is first initialized a blank context will be instantiated. This context will be used by the frames to store information about the computation that takes place, as well as for storing assertions the frames need to use and the demon structures that the frames may send to one another.

These Conniver contexts must be used together with the Conniver data base functions. The use of general data base functions such as those provided by Conniver tend to slow down retrieval of information when large amounts of data are in a context. Each time information is retrieved from a context, all the items in the context must be pattern matched with the goal pattern. Thus if each frame were to add information to this current context, so much information would quickly accumulate, that the retrieval will become very inefficient. Also, with all the frames storing information in one context, there may be some information which the wrong frame thinks is information for it, but is actually not.

The problem is solved by more fully utilizing the power of

the Conniver contexts. This is done by giving each frame the possibility of having one or more of its own contexts. Conniver provides the ability to create new contexts at will, so that frames may create contexts whenever they want. When these contexts are created by the frame, they will be added to the current topic context so that the frame may access its context whenever it needs it. The frame will do this so that when a new topic is encountered, the frame will not contain a variable which points to the context for the old topic. Every time a frame needs its context it must do a data base search to retrieve the current value for its context. The current topic context will contain all the contexts of frames which need them for the current conversation.

When a change in topic is encountered, the current topic context can be saved and a new topic context can be instantiated. This new context will then be used by the frames for storing any information in the data base that they need. The old context together with all the information stored in it by the frames may be saved and later restored. This stored information will contain all the subcontexts created by the frames invoked for that topic. These subcontexts will be hidden from the frames that created them. Thus when a new topic is introduced, the old topic can be completely removed from the computation so that it will not interfere with the new topic. Frames which represent instances of objects will still be in existence, but they will be hidden frame the current topic context. Frames which do not represent instances of objects

will cause no problem, since all the information that they ever contain will only be in their contexts, which will be saved along with the old topic context. They will have to create a new context the next time they need one. When the old topic context becomes the current topic context again, all this information will be returned and will now be available for use by the frames. If necessary all information is available through the global context but the current topic context will have no direct access to it. by the system. This change of topic capability is important, since it must be possible to change topics with ease. All that is necessary is to reinstate a context to return to a topic previously discussed.

In this way the previous computation will be protected from destruction by the new computation and the new computation will be protected from interference from information stored by the old computation. Or at least there will be no direct interaction between the two topics.

The use of contexts focuses on the current topic of the conversation rather than having no current topic. Using the information from the current topic context and information about the current frames, the syntactic and semantic analysis frames should be able to better understand the sentences or sentence fragments input into the system. In many cases this information will be absolutely necessary for the system to understand the input utterance. This will be especially true with sentence fragments, which often occur in human conversation. Also, this information will be useful in determining referents when

pronouns and other ambiguous forms of reference are used in sentences and in sentence fragments.

3.6. Representation of Nouns

Enough information has now been presented to allow a discussion of how various syntactic elements of English are specified using frames.

Nouns will usually have two frames associated with them, one (the BF) will contain the more general information about the concept represented by the noun and one (the EF) which contains information about a specific instance of the concept. Sometimes this latter frame will not be needed. This usually occurs with nouns which are general in scope, and could denote any one of a number of concepts. For example, the noun "object" is a noun which embodies many different concepts (e.g. squares, triangles, circles, doors, etc.) and will not have an EF. The structure of nouns was split into two frames so that it would be possible to have one frame in control over all the specific frames. This means that the more general frame has to decide when to create a new instance of the specific frame, rather than the programmer having to check to see if a single frame for an object would have to be instantiated or not. This also means that the general frame can keep track of all the instances of the specific frame and return these names to any frame which wants to have a current list of all the frames of one concept.

It would be wasteful to have an object frame for each

specific object that the system knows about, especially since there are already frames which contain specific knowledge about all the objects the system knows about. In this case, the object frame will know more general information about objects as a class. For example, the object frame will know how to move an object around on a graphics screen. This information is common to all objects known by the system and should not be contained in a more specific frame.

What actually goes inside these frames depends very much on what the noun represents. A specific example of what information is contained in a frame for a noun or a physical object will now be given. The example used will be taken from the problem domain described in the Introduction. The following definition is the outline for what the house frame would look like. Words in lower case which are preceded by a semi-colon(;) are comments.

```
(FRDEFUN HOUSE
  (BFRAME (ADJECTIVE-LIST HOUSE OBJECT-TYPE OBJECT HOUSE-CONTEXT)
:HOUSE-LOOP
    (CASES
      ((!>OBJECT-TYPE !>OBJECT)
          ; an object which could be part of a house has
          ; been drawn. Create a house EF and send the message
          ; to it.
        (CSETQ HOUSE (CREATE 'HOUSE))
        (DEMON 'FRONT HOUSE) ; send demons so the new house
        (DEMON 'ROOF HOUSE) ; EF is told whena front & roof occur.
        (SEND HOUSE FRMESSAGE FRCONTINUE 'HOUSE-ERROR))
      ((DRAW . (!>ADJECTIVE-LIST))
          ; the system has been asked to draw a house.
          ; ADJECTIVE-LIST contains the properties that
          ; the house should have.
        (CSETQ HOUSE (CREATE 'HOUSE)) ; create  house EF to
                                      ; to represent new house.
        (DEMON 'FRONT HOUSE)
        (DEMON 'ROOF HOUSE)
```

```
        ; draw the front of the house.
       (SEND 'FRONT '(DRAW) CURRENT-FRAME 'HOUSE-ERROR)
       (EXPECT ((FRONT !>FRONT))
               (ELSE: error))
        ; draw the roof of the house.
       (SEND 'ROOF '(DRAW) CURRENT-FRAME 'HOUSE-ERROR)
       (EXPECT ((ROOF !>ROOF))
               (ELSE: ERROR))
         ; make the front support the roof.
       (SEND 'SUPPORTS (LIST 'MAKE FRONT ROOF)
              CURRENT-FRAME 'HOUSE-ERROR)
       (EXPECT (T)
               (ELSE: ERROR))
         ; make the front "married" to the roof.
       (SEND 'MARRY (LIST 'IS FRONT ROOF)
              CURRENT-FRAME 'HOUSE-ERROR)
       (EXPECT (T)
               (ELSE: ERROR))
       ; process properties specified by ADJECTIVE-LIST.
       ; a house has been recognized, perform demon
       ; processing to tell those frames waiting for
       ; a house.
       (DEMON-PROCESSOR '(*DEMON !>RFRAME) (LIST 'HOUSE HOUSE)
                        'HOUSE-CONTEXT)
       (SEND FRCONTINUE (LIST 'HOUSE HOUSE)))
       ; must accept messages which create and
       ; destroy demons.
     ((*DEMON !>RFRAME))
     ((*CANCEL !>RFRAME))
       .
       .
       .
    ) ; END OF CASES
      ; send message to receive next input.
      ; and go to top of loop to receive
      ; message and perform required action.
   (SEND 'INPUT-FRAME '*NEXT CURRENT-FRAME)
   (GO 'HOUSE-LOOP)
:HOUSE-ERROR
   ; error processing for the house BF.
 ) ; end of BF for house.

 ; definition of EF for the concept house.
  (EFRAME (       )
    (CASES
      .
      .
      .
      )))
```

As can be seen the structure for a graphical house will consist of two actual frames, since a house represents a specific object. One frame will consist of information about a specific instance of a graphical house. The more general or BF for a graphical house will consist of information about graphical houses in general. In particular, it will contain information about how to draw a house, it will be able to return a list of all the houses the system knows about, in case the semantic analysis routines of the system have need of such information. The information for drawing a house will consist of code for sending messages to the subframes of the house. This frame must also be able to accept demon messages from frames wanting to know when a house comes into existence. This means that it may have to send a demon to the more specific house frame, which it has created, so that it will know when a house has actually been completed. The house BF must be able to create a house EF house frame, when it receives a message specifying some object which could be part of a house has been created. This will happen when the first object for representing the house comes into existence.

When an EF is created and a message is sent to it, the CD specified in this first message will be saved. This CD should specify the EF that has as part the current EF. This is done so that when a message is sent to the current EF and the message is not understood, the system may back up through the control links rather than sending an error message to the CD associated with the message causing the problem. This type of error processing

will only be used with errors of type *MESSAGE and *FULL. This creates a network of pointers between the currently active EFs which will be used for performing back up error processing.

The information in the EF consists of the actual slots for a front and a roof frame. It must also contain information about the relationships between the two subframes. For example, in the specific house frame the following two relationships must hold:

1.) Support - The front of the house must support the roof of the house.

2.) Marry - This relationship is taken from Winston's work [27]. This means that the top of the front must be in direct contact with the bottom of the roof of the house. The two surfaces must be exactly aligned, without any overhang by either the roof or the front of the house.

Also, the frame must be able to return information about this instance of a house to any frame which may request it. For example, the house EF must be able to return the location of itself on the graphics screen, its top, its bottom and other physical characteristics which a graphical house may possess. This information will be needed by the frames for Marry and Support, since they must know the coordinates of the graphical house to either check to see if the relationship is satisfied or to make the relationship hold.

3.7. Representation of Properties

Properties modify nouns or objects and further specify the objects described. Properties correspond most closely with adjectives and in fact most of the property frames correspond exactly with adjectives. Properties also correspond to participles such as moving, i.e. a physical object may have the property that it is moving. These verbs are handled in exactly the same way as adjectives.

The frame for a property is very much like that for a noun, however, there is always only one frame associated with a property. There are no EF's for instances of properties, since the only thing that these would be useful for would be to tell which object had that property. The property BF itself will contain a list of all such objects already. Thus this single frame is in complete control of the one property. Any requests to give or remove a property from an object or to check if an object does or does not have a property must pass through the frame for that property. An example of a property frame is given below.

```
(FRDEFUN BIG
  (BFRAME (OBJECT BIG-CONTEXT)
    (CASES ((MAKE !>(OBJECT (ATOM !,OBJECT)))
              ; pretest code to make sure object
              ; can be made big.
              ; code to assign variable BIG-CONTEXT
              ; to the current value of BIG's context.
             (ADD (LIST 'SIZE OBJECT 'BIG) BIG-CONTEXT)
              ; code to actually make object BIG.
             (DEMON-PROCESSOR (LIST '*DEMON
                                    !>RFRAME
```

```
                                    OBJECT)
                                (LIST 'BIG OBJECT)
                                BIG-CONTEXT)
            ; perform demon processing in case
            ; any frames are waiting for this
            ; object to be made big.
           (FSEND FRCONTINUE (LIST 'BIG OBJECT))
            ; return to caller with the message that
            ; the object is now big.
         ) ; end of first case.
        ((IS !>(OBJECT (ATOM !,OBJECT)))
            (COND ( ; is information in data base.
                    ; if so then return T.
                    (FSEND FRCONTINUE T))
                  ( ; check to see if object is
                    ; big using coordinates etc.
                    ; if so then return T.
                    (FSEND FRCONTINUE T))
                  (T ; otherwise return NIL.
                    (FSEND FRCONTINUE NIL)))
         ) ; end of second case.
           .
           .
           .
       ) ; end of cases.
  )) ; end of definition.
```

The actual frame should be able to handle a message such as:

(MAKE OBJ).

Which means to make OBJ have the property corresponding to the frame being sent the message. This frame must be able to detect whether OBJ may or may not have the property. This is a pretest that must be made by the frame before it goes ahead and makes OBJ have the property.

Another message the frame should be able to handle is:

(IS OBJ).

This message is asking if OBJ has the property associated with the frame being sent the message. If the object OBJ does have

the property, then the message T will be sent to the CP, otherwise NIL will be sent. The actual test will usually involve a data base search to see if OBJ has the property associated with the frame. If the search fails, then more work has to be done to actually check to see if OBJ has the property. This will usually be done by checking the actual coordinates of the object. In a more general domain, this might involve invoking vision procedures on the object to see if it actually has the property. This seems very much the way humans work. If we know something already has a given property from information stored in memory, we assume this information is correct. Otherwise we will have to actually check whether the object has the property, usually by looking at it.

Another message the frame should accept is one of the form:

(ASSERT OBJ).

Which is stating that OBJ has the property associated with frame. This message corresponds to an utterance such as:

The square is big

i.e. it corresponds to a statement by the student that OBJ has the property associated with the frame being sent the message. This assertion is first checked to see if the property is true for the object or could possibly be true for an object. The check starts in the Conniver data base and then must continue to see if the property is true for an object if the data base search fails. The conditions for this search should be more relaxed since this may be a personal preference of the student and ordinarily may not be accepted if the system itself were to

assign the property to the object.

For example, if an object is coloured red and the student makes a statement of the form:

The object is blue.

Then the system should balk and not execute it. Rather it should complain to the student about the statement and in effect ignore it. What the student must do is utter a sentence such as:

Make the object blue.

Such a statement would have the desired effect and could be carried out by painting the object or whatever. In this case there would be no conflict, the old colour of the object can be deleted from the current context and the new one added.

Other messages which a property frame must be able to handle are the negations of the messages above. These messages will take the form:

(MAKE (NOT OBJ))

(IS (NOT OBJ))

(ASSERT (NOT OBJ))

respectively. So this means that it is possible to make an object not have some property, to check to see if an object does not have some property and to assert that an object does not have a property.

Unfortunately, it is not possible to use the negation of a property to represent the opposite of that property. For example, big is the opposite of small, so it would be nice to be able to use the negation feature for say big to represent small

within the same frame. However, this is impossible since small is not necessarily the same as not big. That is something may be of medium size, in which case it is neither big nor small

3.8. Representation of Relations

Relations are predicates of two or more arguments. Relations correspond to prepositions and composite phrases such as son of, front of, greater than, less than, etc. Those relationships which are derived from prepositions and composite phrases not using "of" are frames, much like the frames corresponding to properties.

Associated with each relationship there is only one frame, i.e. there will be no EF's for a relation. There is no need for frames corresponding to instances of relationships, since the only thing needed to be contained in such frames would be the two objects which were fulfilling that particular instance of the relationship. This can be more easily done using the Conniver data base facilities for storing the information. Thus all requests to make a relation hold between two objects or to destroy the relationship between two objects must pass through the single frame for the relationship. An example of a relation frame follows.

```
(FRDEFUN BIGGER
  (BFRAME (OBJECT1 OBJECT2 BIGGER-CONTEXT)
    (CASES ((MAKE !>(OBJECT1 (ATOM !,OBJECT1))
                  !>(OBJECT2 (ATOM !,OBJECT2)))
```

```
            ; pretest code to make sure OBJECT1
            ; can be made bigger than OBJECT2.
            ; code to assign variable BIGGER-CONTEXT
            ; to the current value of BIGGER's context.
           (ADD (LIST 'BIGGER OBJECT1 OBJECT2)
                BIGGER-CONTEXT)
            ; code to actually make OBJECT1
            ; bigger than OBJECT2.
           (DEMON-PROCESSOR (LIST '*DEMON !>RFRAME
                                  (LIST OBJECT1 OBJECT2))
                            (LIST 'BIGGER
                                  OBJECT1
                                  OBJECT2)
                            BIG-CONTEXT)
            ; perform demon processing in case
            ; any frames are waiting for
            ; OBJECT1 to be made bigger than
            ; OBJECT2.
           (FSEND FRCONTINUE (LIST 'BIGGER
                                   OBJECT1
                                   OBJECT2))
            ; return to caller with the message that
            ; OBJECT1 is bigger than OBJECT2
         ) ; end of first case.
        ((IS !>(OBJECT1 (ATOM !,OBJECT1))
             !>(OBJECT2 (ATOM !,OBJECT2)))
            (COND ( ; is information in data base.
                    ; if so then return T to CP.
                    (FSEND FRCONTINUE T))
                  ( ; check to see if OBJECT1 is
                    ; bigger than OBJECT2.
                    ; if so then return T to CP.
                    (FSEND FRCONTINUE T))
                  (T ; otherwise return NIL to CP.
                    (FSEND FRCONTINUE NIL)))
         ) ; end of second case.
           .
           .
           .
       ) ; end of cases.
  )) ; end of definition.
```

The frame itself, must be able to handle a message of the form:

(MAKE OBJ1 OBJ2)

which means to make OBJ1 and OBJ2 have the relationship between

them that corresponds to the frame which was sent the message. A pretest must be made by the frame to make sure that OBJ1 and OBJ2 may have that relationship between themselves. If not then an error message will be sent to the CD for the message sent to the relation frame.

Another message the frame must be able to respond to is of the form:

(IS OBJ1 OBJ2).

That is, do the two objects, OBJ1 and OBJ2, satisfy the relationship corresponding to the frame being sent the message? If they do then the message T will be sent to the CP, otherwise NIL will be sent. The actual processing of such a request will involve a simple data base search to see if OBJ1 and OBJ2 are in the correct relationship with each another. This search would not just check to see that OBJ1 is in the correct relationship with OBJ2, but depending on the relationship, the frame may perform more complex processing. For example, with the relationship inside, the two objects may not be stored with the inside relationship between them but there may be a series of objects such that OBJi+1 is inside OBJi for all i over the series. If the first element of the series is inside OBJ1 and OBJ2 is inside the last element of the series, then OBJ2 is inside OBJ1. This sort of inference is often made by people and will be easy to perform within the one frame. Note that only the context for the one frame corresponding to the relationship need be used in the inferencing. This should reduce the necessary amount of search to an acceptable level. If the data

base search fails, then more processing must be done to actually check to see if the relationship holds between the two objects. In this case the system would have to use the coordinates of the two objects to check that the relationship does indeed hold. In a more general system this might involve a visual system which looks at the two objects involved and uses this information to check if the relationship holds.

A further message that the frame must be able to accept is of the form:

(ASSERT OBJ1 OBJ2)

This simply states that OBJ1 and OBJ2 are in the correct relationship with one another. However, such a relationship must be checked to see if it is valid or at least possible in which case the relationship will be instantiated in the data base. If there is no possibility of such a relationship holding, then an error will occur and an error message sent to the CD of the assertion message. For example, someone might say:

John is beside the building.

If no explicit check can be made, i.e. you can't see the building where John is, then the relationship between John and the building will be assumed true, at least until contradictory information comes to light.

Other messages which a relationship frame must be able to handle are the negations of the messages above. These messages will have the form:

(MAKE (NOT OBJ1 OBJ2))

(IS (NOT OBJ1 OBJ2))

(ASSERT (NOT OBJ1 OBJ2))

respectively. This means that it is possible to make two objects not have some relationship between them, to see if two objects do not have some relationship between them and to assert that two objects do not have some relationship between them.

The other type of relationship is of the form: son of, front of, etc. Such relations can be considered as selectors for the slots of a frame. For example, consider

the front of the house.

This is selecting the front slot of the frame representing "the house". Every house must have a slot to contain the frame representing its front. Actually this will be a pointer to an instance of a frame representing the front.

With a relationship such as the one above, a message is sent to the frame corresponding to the object on the right side of the "of". Since there may be a construct such as:

the window of the front of the house

such phrases must be processed recursively. The first thing to be done is to determine the referent for "the house". Once this has been determined a message may be sent to this frame to select the slot corresponding to the front of "the house". This "house" frame must be able to respond with the name of the frame filling a slot when an appropriate message is sent to it. Once the frame for representing "the front of the house" has been determined, a message may be sent to this frame to obtain the name of the frame corresponding to the slot for "window". This

frame would then be the referent for the phrase "the window of the front of the house".

The "of" relationship is totally dependent on the objects in the system and all frames associated with them must be able to handle messages such as those described above. When a message is sent to a frame wanting a value of a slot to be returned, a list of all the candidates that fill slots of that type in the frame will be returned to the CP. The frame requesting the list of values for a slot of a frame must then decide which if any of the frames is the one desired. If no frame is filling the slot when the message is sent, NIL will be returned to the CP.

3.9. Representation of Actions or Verbs

Actions correspond to verbs. Verbs do not have distinct frames associated with them. They form part of messages to frames and usually correspond to the first atom in the message. Thus actions may only be performed by a frame representing the subject of the action. All frames which may perform an action specified by some verb must be able to handle a message with that verb as the first word in the message. For example in the current problem domain described, all the specific object frames must be able to accept a message to draw themselves. Thus to draw a square on the graphics screen a call to SEND is made of the following form:

```
(SEND 'SQUARE '(DRAW) CONTINUATION COMPLAINT).
```

Where SQUARE specifies that the message is to be sent to the square BF. (DRAW) is a message which specifies that a square is to be drawn. CONTINUATION is where the computation should continue when a square has been drawn on the graphics screen. COMPLAINT is where to go if an error is encountered while drawing the square of if the message is not decoded properly.

Verbs which specify actions for a specific frame only must be handled by that specific frame. For example, the verb draw specifies a specific action to the object being drawn. The object itself is the only frame which knows how to draw itself. In general this is not always the case, for example, for moving any object about the graphics screen, it is not necessary for each object type to know how to move itself around the screen. What can be done is to create a superset frame of all these objects which can be moved around on the graphics screen. This superset frame would know all about moving objects.

Some actions must be associated with a specific frame and others may be associated with a more general frame. Thus superset processing is used heavily in representing actions which may be performed on a frame or by a frame. A graphical object frame will contain information on how to move graphical objects (squares, rectangles, triangles, circles, etc.) around a graphics screen. This frame must also accept messages corresponding to sentences like:

Draw an object.

where object refers to a graphical object. In this case the

graphical object frame would have to pick one of its subsets at random (or possibly its favourite type of graphical object) and send a message to this object to draw itself. Thus it might pick a square to be drawn and send a message to the square frame specifying that a square is to be drawn. This is not like superset processing, where a message is sent to all the supersets of a frame if a message cannot not be understood. The graphical object frame must be able to decode any of the actions that any of its subframes can perform. This means that it must be able to decode a message caused by a sentence such as the above example and then decide what to do. There can be no automatic subset processing in the same way that there is automatic superset processing.

With the many different actions which may be encountered in a sentence, there are many different actions that a frame must be able to handle. This number of actions may be reduced through the use of primitive actions similar to those that Schank proposes [20]. In this way the amount of information contained by a frame may be reduced and the amount of inferencing needed to handle sentences should also be reduced.

In the graphics domain there are very few primitive actions necessary. The three actions ASSERT, IS and MAKE as described above must be present in the system. Besides these actions the only other ones needed are: DRAW and MOVE. DRAW will be needed for drawing new objects on the graphics screen. This action would also correspond to different verbs. For example, the sentence:

Place a square inside the front

would call for a new object to be drawn on the screen and the primitive action DRAW would be used to handle this sentence. Similarly the primitive action MOVE will be used to handle other verbs besides the verb "move". For example, the sentence:

Place the small square inside the front

will cause the primitive action MOVE to be used to move the square (which would have to be on the graphics screen already) to a location inside the front of the house.

Chapter 4.

The Model in Use

In this chapter an example of how frames interact and perform inferences for understanding natural language will be described. This example will consist of the necessary inferences needed when a person goes through the process of drawing a graphical house using English commands. This problem may at first seem very simple, but it actually involves many difficult problems of natural language understanding.

4.1. Specification of Top Level Frames

In a system for natural language processing, there must be some frames which control the understanding process. What is proposed is that there be a frame which can be thought of as a "consciousness" frame, which will be in control of the complete computation. This frame should be ready to respond to any inputs received from any of its sensory modes. In this case the only sensory mode is the input from the terminal keyboard typed in by the user. The "consciousness" frame must be able to accept all inputs, some of which may be ignored; others may be suppressed. This "consciousness" frame will send a message to a conversation frame when it detects an utterance from some input source (e.g. a person), or when it wants to begin a conversation with a person.

The "consciousness" frame will contain enough information to decide which subframes are to be activated on receiving some input. That is when some form of stimulus is applied to the "consciousness" frame, it will decide which subframe to invoke. In the case that a stimulus which is the beginning of a conversation is applied, a conversation subframe will be invoked.

In a system which is dealing only with natural language processing and does not have any other form of stimulus, there is no need for a "consciousness" frame, since it will immediately see that a conversation is to take place and will just invoke the conversation frame.

The conversation frame itself will consist of two major subframes and must also have the ability to send messages to other frames. This frame will usually be the frame that begins the process of making an assumption about what is being talked about. The first thing the conversation frame will do is send a message to an input frame which will input utterances and convert them into an internal format. This internal format will consist of messages to be sent to other frames involved in the understanding of the utterance. The input frame will consist of a syntactic analyzer which will be passed the input utterance. This frame will then begin to parse the utterance as far as possible. The syntactic analysis frame will have full access to all the frames known by the system.

The parsing frame, having full access to all the information stored in the system will also be able to perform

semantic analysis of the utterance at the same time, to help in disambiguating sentences. In this way the parse of an utterance can be guided by the semantic information available in the system's knowledge base. What this parsing frame will produce is one or more messages and the frames to which these messages should be sent. These messages will cause the correct action to be performed, or will update the data base with new information, or return information in response to a question.

The subject of the sentence will be the noun that determines which frame the message or messages are to be sent to. By subject of a sentence, I mean the main noun of a sentence. In an imperative sentence this will usually be the actual object of the sentence. In a declarative sentence this will normally be the actual subject of the the sentence. The decision of which noun is the main one is a difficult problem to solve in general. In some cases, there will only be one noun in a sentence, in which case, the decision will be simplified. In other cases syntactic and even semantic knowledge will have to be used.

Often noun phrases will have to be disambiguated by the input frame. This should be done by the parsing function in most cases. The method will be to send messages to the main noun of the phrase and to all the frames that represent the properties that this noun must have. These messages will specify that a list of all the specific instances of the noun is to be returned and that a list of all the objects that have the properties associated with the property frames sent the message

be returned. In this way, the input frame will have lists of all the specific instances of the main noun and a list of all the objects which have the properties specified in the noun phrase. These lists may then be intersected with one another to produce a list of all the possible referents for the noun phrase. This information can then be used to determine an actual referent for the noun phrase. Through the use of Conniver contexts, the whole data base need not be searched but only the currently active data base.

Messages will then be sent to the frames corresponding to the main noun of the sentence. What the input frame does depends on the type of the sentence. When a command is encountered, the input frame will send a message to the correct frame specifying as CP, the one specified in the message to the input frame. When a declarative sentence is encountered, a message will be sent to the correct frame specifying the input frame as the CP. The input frame will, when it regains control, read in the next input sentence. When a question sentence is found a message will be sent to the correct frame specifying the input frame as the CP. The response to the question will be returned to the input frame. The input frame would then send this information to the output frame so that an English sentence may be constructed to be output to the student.

Once the semantics of an utterance have been determined and the associated semantic actions have been performed the system will have to output some message to the user. This would involve the second important subframe of the conversation frame.

This frame would have to generate surface level English responses to the input utterances. This would be passed the original utterance together with the messages derived from this utterance and the responses that were returned by the frames.

The output frame would have to be responsible for generating surface level English sentences from information passed to it via messages. Any frame in the system would have to be able to send a message to this frame so that it may output a message to the user. The actual processes used to generate surface level English sentences would be completely contained in this one frame together with its subframes.

The rest of this chapter will assume that the input and output frames can be developed to a level necessary to actually make the system useable. This thesis is more motivated by the internal workings of the system and not so much with the input and output problems associated with natural language. Examples of how inferences are made and how expectations are used to drive the computation will now be given.

4.2. A Detailed Example

In this section I shall give a complete example of the system in use. The actual messages sent between the frames will be given. These will be presented in a tabular form, the first entry will be the message, the second will be the frame sent the message, the third will specify the CP and the fourth will give the CD. The frame sending the message will be the one just

above the entry in the table. A sentence will be presented, and then the table with the messages sent to process the sentence and finally a comment on why the system sent the messages in this manner. Frames which have numbers after their names are EF's; the others are BF's.

When the system is run the top level frame (TLF) will be invoked. It would begin by giving the student a brief introduction to what the system could do. It would explain to him how to input sentences to the system using the terminal keyboard. It would also display the built-in figures of the system, so that he will have some idea of what figures he may use. The system would then begin the process of teaching the student to use the graphics terminal by asking him to reconstruct a graphical house, for example. At this point the system would prompt the student for input and begin the session.

*NEXT	INPUT FRAME	TLF	TLF

The TLF sends the message *NEXT to the input frame which causes it to input and process a sentence from the student.

1.) Draw a big square.

(DRAW BIG)	SQUARE	TLF	INPUT FRAME
(MAKE SQUARE1)	BIG	SQUARE	SQUARE
(BIG SQUARE1)	SQUARE		
(SQUARE SQUARE1)	TLF		

At this point in the computation, a big square will have

been drawn (see Fig. 1). This square will eventually be used to represent the front body of the house. When the BIG frame is asked to make the square big, it will have no information available to decide on how big to make the square. What must be done is to attach context information to the square BF so that the BIG frame may use this knowledge to make the decision. One possible method is to put a list of three numbers on the property list of the square BF. These numbers would be multiplication factors for making the default square small, medium and large (e.g. 0.5 1.0 2.0). Another possible way would be to attach a Conniver context to the square BF which would contain examples of different sized squares. The BIG frame could then use this information to make the decision. The TLF will now have to decide which subframe to make active. Since the student was asked to draw a house, this decision can be made easily and the house frame will be invoked.

(SQUARE SQUARE1)	HOUSE	TLF	TLF
(*DEMON HOUSE)	HOUSE2	HOUSE	TLF
T	HOUSE		
(SQUARE SQUARE1)	HOUSE2	TLF	TLF
(*DEMON HOUSE2)	FRONT	HOUSE2	HOUSE2
T	HOUSE2		
(*DEMON HOUSE2)	ROOF	HOUSE2	HOUSE2
T	HOUSE2		
(SQUARE SQUARE1)	FRONT	HOUSE2	HOUSE2
(*DEMON FRONT)	FRONT3	FRONT	FRONT

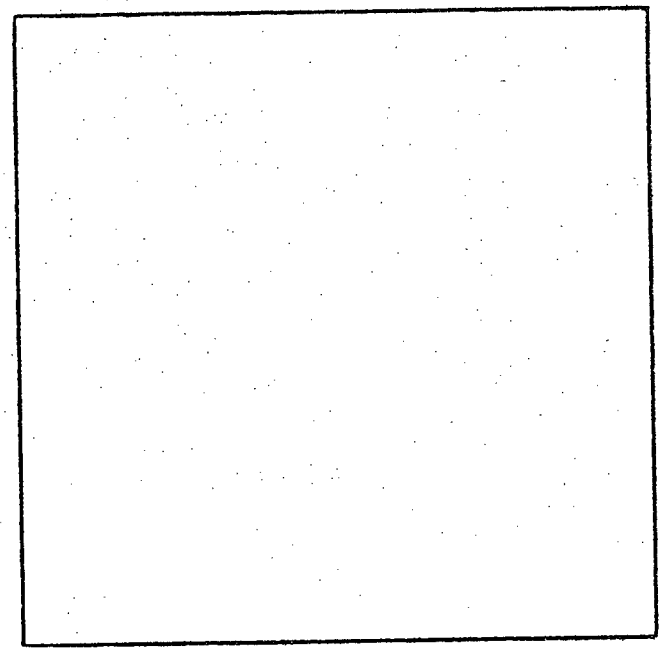

Figure 1.

T	FRONT		
(SQUARE SQUARE1)	FRONT3	FRONT	FRONT
(*DEMON FRONT3)	FRONT-BODY	FRONT3	FRONT3
T	FRONT3		
(*DEMON FRONT3)	DOOR	FRONT3	FRONT3
T	FRONT3		
(*DEMON FRONT3)	WINDOW	FRONT3	FRONT3
T	FRONT3		
(SQUARE SQUARE1)	FRONT-BODY	FRONT3	FRONT3
(*DEMON FRONT-BODY)	FRONT-BODY4	FRONT-BODY	FRONT3
T	FRONT-BODY		
(SQUARE SQUARE1)	FRONT-BODY4	FRONT3	FRONT3

The message for the square has now been filtered down to the front body EF which will assert that it is represented by the square. The above sequence of the messages is relatively straight forward. The only time any major decision must be made is when the front EF must decide whether the square is to represent the front body or the window. In this case the decision that the square should represent the front body is made easier by the extra knowledge that the square is big. In other cases the same assumption will normally be made but for a different reason. When a person is drawing a figure, he tends to draw the major components first before the minor ones. This assumption may be used to make the decision that the square will represent the front body. If, the square had been drawn small, the decision may be made that the square should represent the

window. The many demon messages are so that when a component of the house is recognized by the system, the frame of which the component is part will be notified. Since the front body has been recognized, this information will begin to filter back up the chain through the demons specified by the higher frames.

(*OCCUR (FRONT-BODY FRONT-BODY4))	FRONT-BODY	FRONT-BODY4	FRONT-BODY4
(*OCCUR (FRONT-BODY FRONT-BODY4))	FRONT3	FRONT-BODY	FRONT-BODY
(*OCCUR (FRONT FRONT3))	FRONT	FRONT3	FRONT3
(*OCCUR (FRONT FRONT3))	HOUSE2	FRONT	FRONT
T	FRONT		
T	FRONT3		
T	FRONT-BODY		
T	FRONT-BODY4		

The demon processing will back up all the way up to the house EF, since a front which just consists of a front body is just as good as one which contains a front body, a door and a window. Control will be returned to the front body EF after the demon processing has finished. At this point, the front body EF will pass control to the square so that when the word "it" is encountered in a sentence by the input frame, the input frame will be able to determine the referent for "it" easily.

(SQUARE SQUARE1)	SQUARE1	FRONT-BODY4	FRONT-BODY4
*NEXT	INPUT FRAME	SQUARE1	

Note that this is the first time that the square EF frame has been entered.

2.) Make it bigger.

(MAKE BIGGER)	SQUARE	SQUARE1	INPUT FRAME
(SQUARE1 (MAKE BIGGER))	SQUARE	SQUARE1	SQUARE1
(SQUARE1 (MAKE BIGGER))	OBJECT	SQUARE	SQUARE
(MAKE SQUARE1 SQUARE1)	BIGGER	OBJECT	OBJECT
(BIGGER SQUARE1 SQUARE1)	OBJECT		
(BIGGER SQUARE1 SQUARE1)	SQUARE		
(BIGGER SQUARE1 SQUARE1)	SQUARE1		
*NEXT	INPUT FRAME	SQUARE1	

The above sequence of messages is an example of superset processing. When the square EF is sent the original message, it is not able to recognize it. Rather than immediately causing an error message to be sent, the square EF initiates superset processing to see if any of the superset frames know what the message means. In this case the graphical object frame does know what the message means. It realizes that an object is to have some action performed on it and uses the fact that BIGGER

is a relation to construct the message sent to the actual BIGGER frame (see Fig. 2).

3.) A triangle, please.

(DRAW)	TRIANGLE	SQUARE1	INPUT FRAME
(TRIANGLE TRIANGLE5)	SQUARE1		
(*ERROR *MESSAGE (TRIANGLE TRIANGLE5))	FRONT-BODY4		
(*ERROR *MESSAGE (TRIANGLE TRIANGLE5))	FRONT3		
(*ERROR *MESSAGE (TRIANGLE TRIANGLE5))	HOUSE2		

When the square receives the message from the triangle, it has no idea what the message means. It then tries superset processing (which wasn't shown) which fails because none of its supersets will know what the message means either. It then generates a *MESSAGE type error which travels up to the house EF which does know what the message means. The house EF will assume that the message is meant for the roof frame and thus will send it to the roof BF (see Fig. 3).

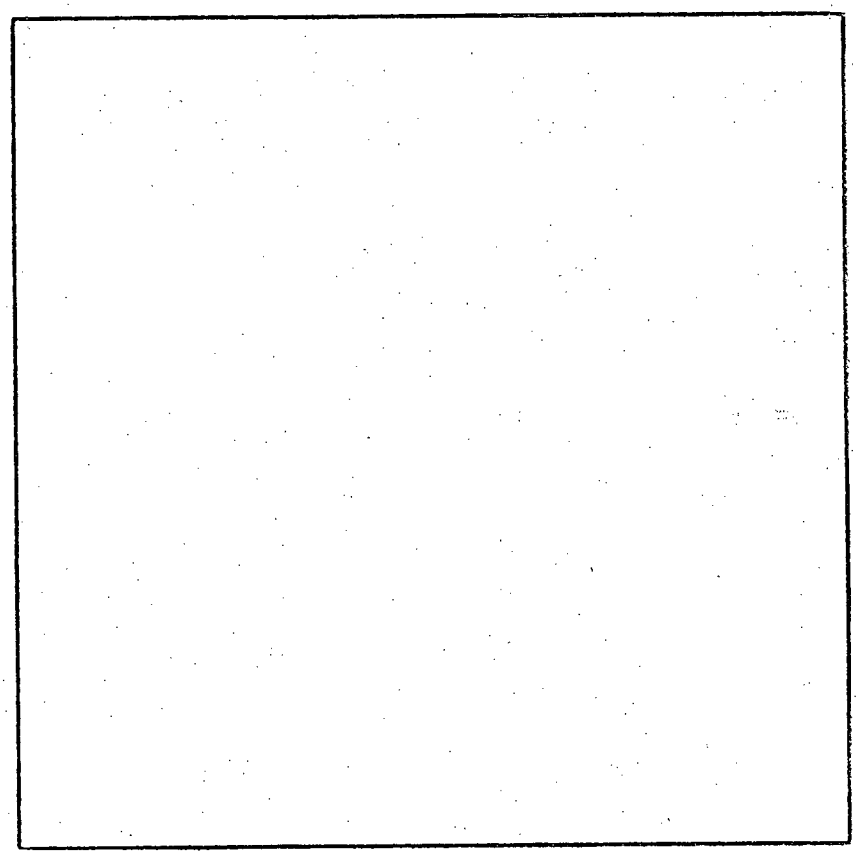

Figure 2.

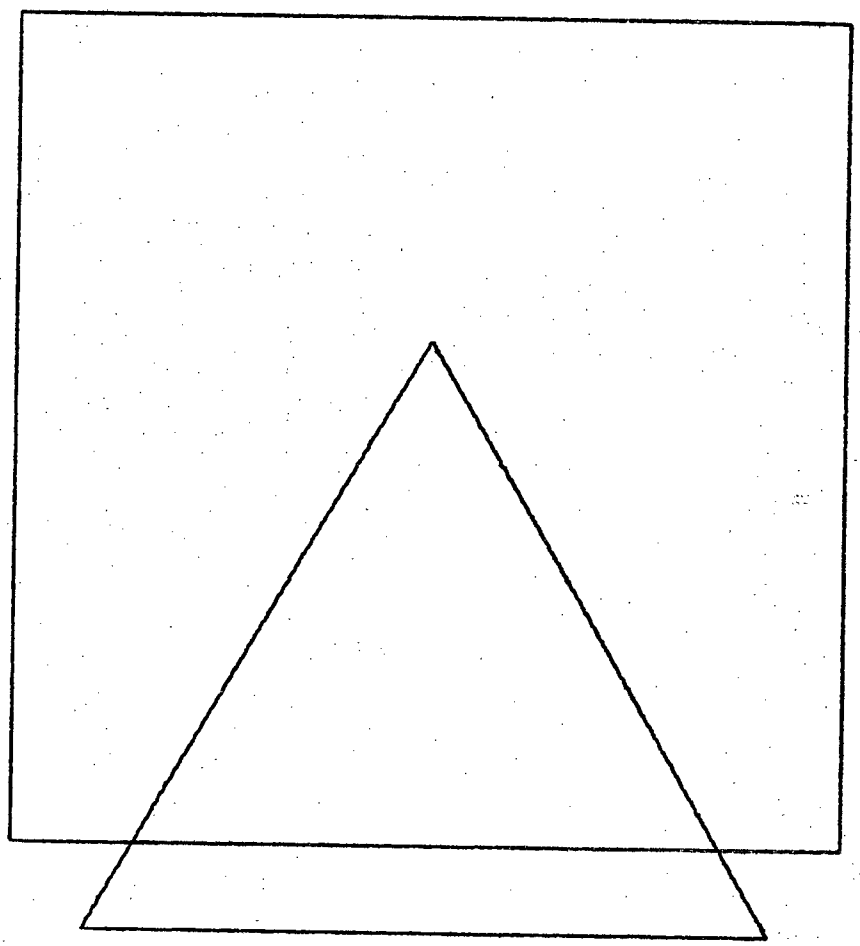

Figure 3.

(TRIANGLE TRIANGLE5)	ROOF	HOUSE2	HOUSE2
(*DEMON ROOF)	ROOF6	ROOF	HOUSE2
T	ROOF		
(TRIANGLE TRIANGLE5)	ROOF6	HOUSE2	HOUSE2
(*OCCUR (ROOF ROOF6))	ROOF	ROOF6	ROOF6
(*OCCUR (ROOF ROOF6))	HOUSE2	ROOF	ROOF
(IS FRONT3 ROOF6)	SUPPORTS	HOUSE2	HOUSE2
NIL	HOUSE2		
(*DEMON HOUSE2 (FRONT3 ROOF6))	SUPPORTS	HOUSE2	HOUSE2
T	HOUSE2		
(IS FRONT3 ROOF6)	MARRY	HOUSE2	HOUSE2
NIL	HOUSE2		
(*DEMON HOUSE2 (FRONT3 ROOF6))	MARRY	HOUSE2	HOUSE2
T	HOUSE2		
T	ROOF		
T	ROOF6		
(TRIANGLE TRIANGLE5)	TRIANGLE5	ROOF6	ROOF6
*NEXT	INPUT FRAME	TRIANGLE5	TRAINGLE5

The house EF decided that the triangle was meant to be part of the roof. When the triangle has been assigned the slot for representing a roof, demon processing will be performed telling the house EF of the existence of the roof. At this point, both

the major components of the house will be in existence. This means that the slots for the relationships between the roof and the front will be checked. In this case neither relation will hold and demon messages are sent so that the house EF will be told when they do hold.

4.) Make it big.

(MAKE BIG)	TRIANGLE5	TRIANGLE5	INPUT FRAME
(TRIANGLE5 (MAKE BIG))	TRIANGLE	TRIANGLE5	TRIANGLE5
(TRIANGLE5 (MAKE BIG))	OBJECT	TRIANGLE	TRIANGLE
(MAKE TRIANGLE5)	BIG	OBJECT	OBJECT
(BIG TRIANGLE5)	OBJECT		
(BIG TRIANGLE5)	TRIANGLE		
(BIG TRIANGLE5)	TRIANGLE5		
*NEXT	INPUT FRAME	TRIANGLE5	

This is another example of superset processing in action. This time, however, the message sent to the graphical object frame specifies that the triangle is to be assigned the property big. Now there is information available through demon processing and the existence of the square for the BIG frame to make a decision as to how big the triangle should be made. That is the system now has sufficient facts to begin to make inferences about actions asked for by the student. In this case the triangle will be made big enough so that its base is the same width as the top of the square (see Fig. 4).

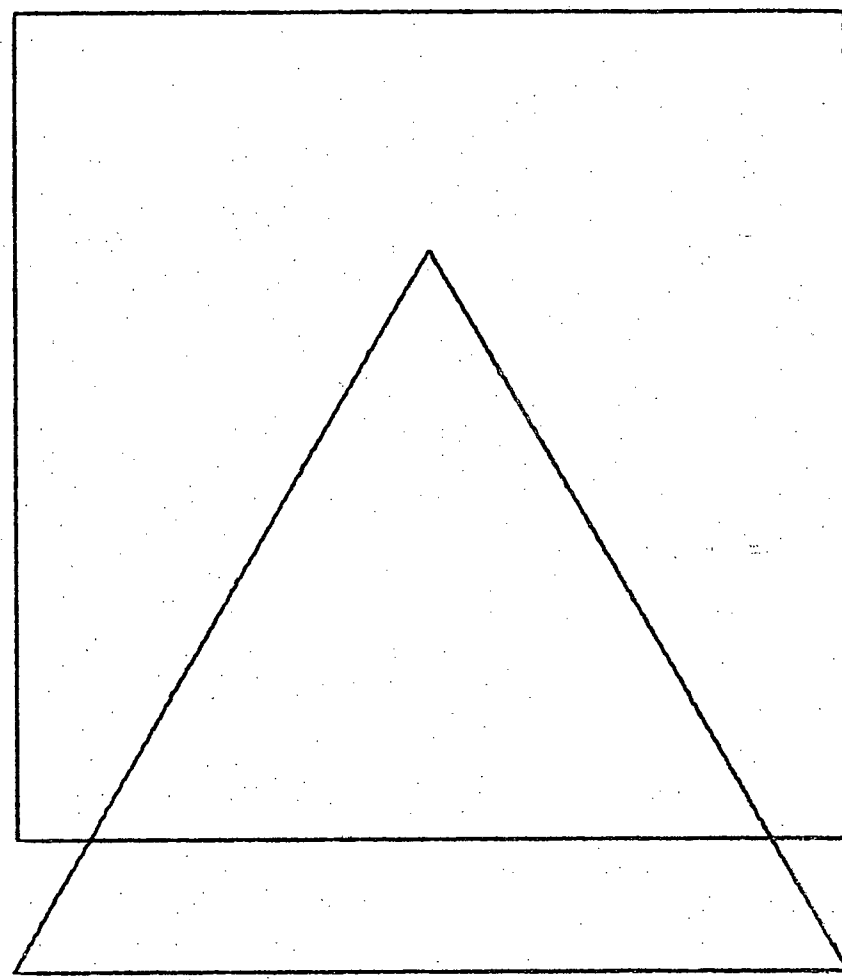

Figure 4.

5.) Move it up a bit.

(MOVE UP BIT)	TRIANGLE5	TRIANGLE5	INPUT FRAME
(TRIANGLE5 (MOVE UP BIT))	TRIANGLE	TRIANGLE5	TRIANGLE
(TRIANGLE5 (MOVE UP BIT))	OBJECT	TRIANGLE	TRIANGLE
T	TRIANGLE		
T	TRIANGLE5		
*NEXT	INPUT FRAME	TRIANGLE5	

Again superset processing is necessary since the triangle frame does not know how to move itself up. The graphical object frame, however, did know and thus performed the action. Using the demon information specified by the house EF for the front and the roof, the graphical object frame knows exactly how far up the triangle should actually be moved. In this case it will be too much to just move the triangle up a bit, so that the triangle will onlybe moved up some default amount (say about an inch) (see Fig. 5). Thus no demons will be satisfied yet.

6.) Higher.

(MAKE HIGHER)	TRIANGLE5	TRIANGLE5	INPUT FRAME
(TRIANGLE5 (MAKE HIGHER))	TRIANGLE	TRIANGLE5	TRIANGLE5
(TRIANGLE5 (MAKE HIGHER))	OBJECT	TRIANGLE	TRIANGLE

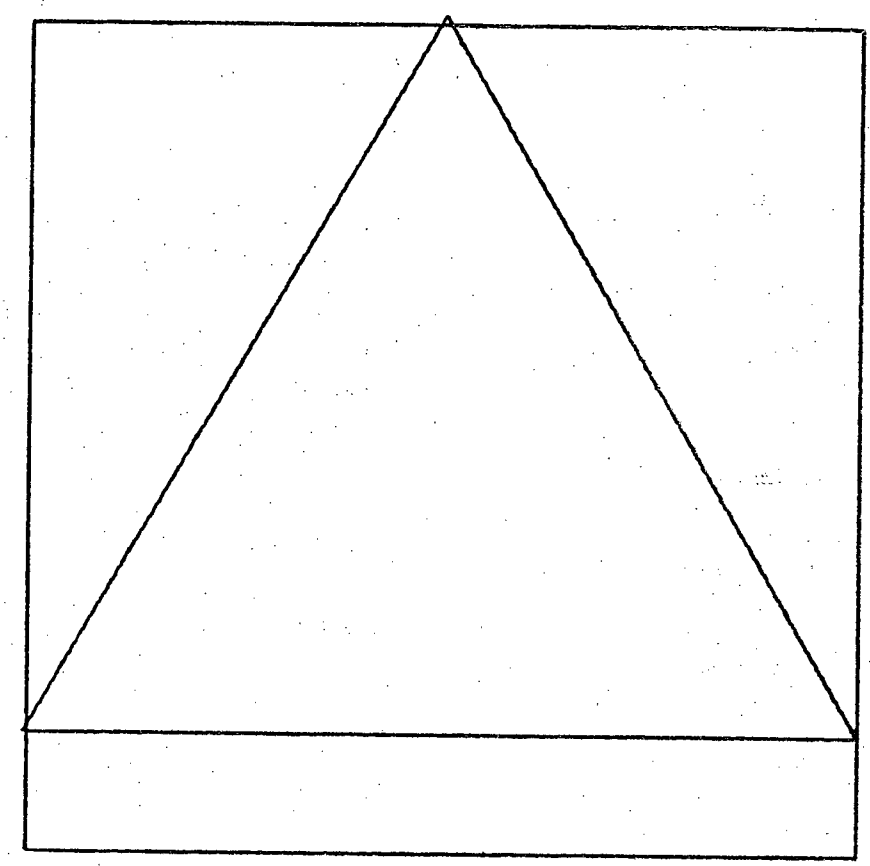

Figure 5.

(MAKE TRIANGLE5 TRIANGLE5)	HIGHER	OBJECT	OBJECT
(HIGHER TRIANGLE5 TRIANGLE5)	OBJECT		
(HIGHER TRIANGLE5 TRIANGLE5)	TRIANGLE		
(HIGHER TRIANGLE5 TRIANGLE5)	TRIANGLE5		
*NEXT	INPUT FRAME	TRIANGLE5	

This is a sentence fragment which must be processed by the input frame. From context and the fact that "higher" is a relationship it should be easy to realize that the sentence means to make the triangle higher up the graphics screen. Thus the original message will be constructed and sent to the triangle EF. The rest of the processing is straight forward using superset processing. Again the graphical object frame will not move the triangle all the way up to the top of the square, but will leave it about an inch short (see Fig. 6). This will normally be done to frustrate the student somewhat, which will later show him how useful the graphics hardware is.

7.) Move it up an inch.

(MOVE UP (INCH 1.0))	TRIANGLE5	TRIANGLE5	INPUT FRAME
(TRIANGLE5 (MOVE UP (INCH 1.0)))	TRIANGLE	TRIANGLE5	TRIANGLE5

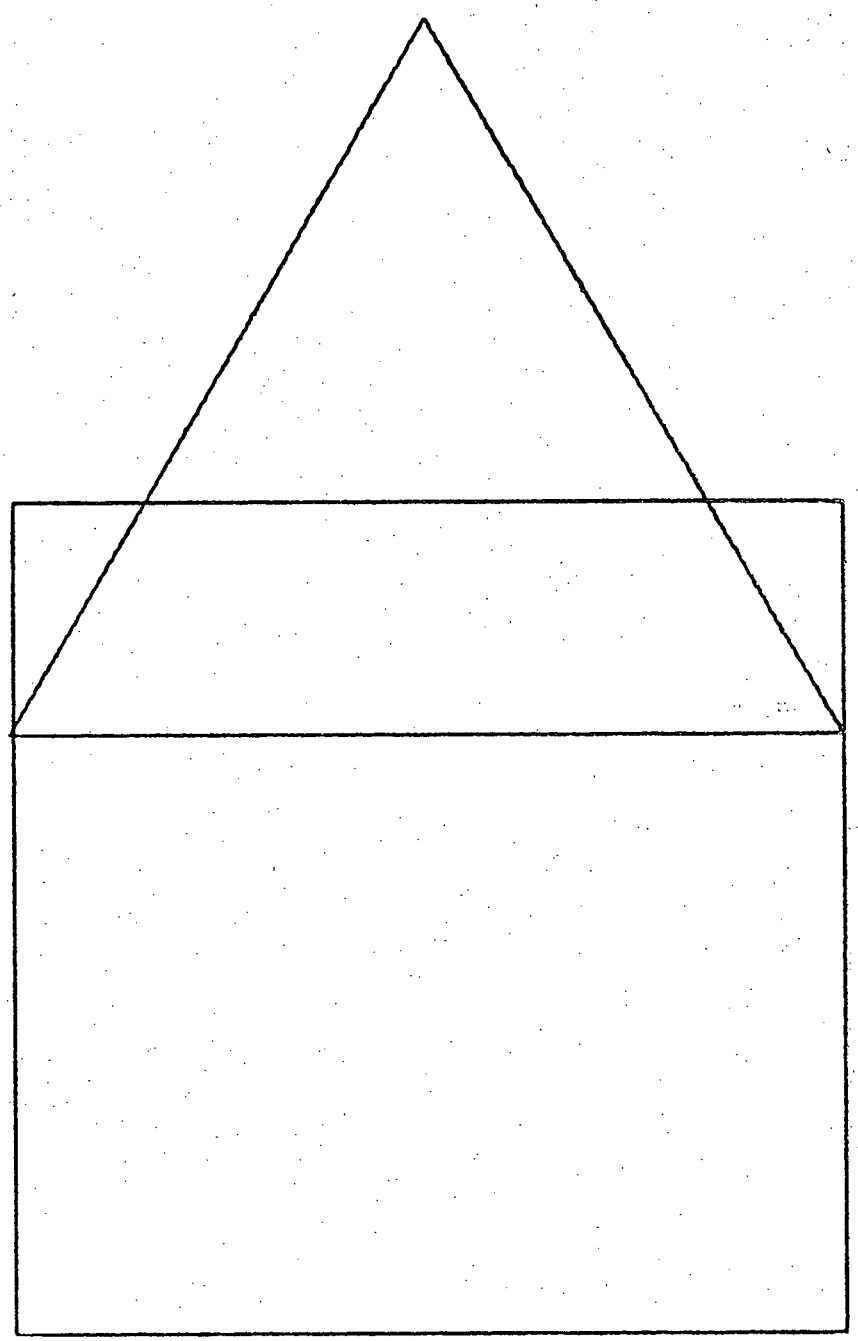

Figure 6.

(TRIANGLE5 (MOVE UP (INCH 1.0)))	OBJECT OBJECT	TRIANGLE TRIANGLE	TRIANGLE TRIANGLE
T	TRIANGLE		
T	TRIANGLE5		
*NEXT	INPUT FRAME	TRIANGLE5	

This sentence will be easily handled by the input frame. The graphical object frame will move the triangle up about an inch (see Fig. 7). This time I will assume that the bottom of the triangle is an inch below the top of the square (with the possibility of a small margin of error). Thus the triangle will be moved up sufficiently so that the bottom of the triangle is equal to the top of the square even though it may not be exactly one inch. At this point the two relations between the front and the roof will hold. Thus demon processing will be performed.

(*OCCUR (SUPPORTS FRONT3 ROOF6))	HOUSE2	SUPPORTS	SUPPORTS
T	SUPPORTS		
(*OCCUR (MARRY FRONT3 ROOF6))	HOUSE2	MARRY	MARRY
(*OCCUR (HOUSE HOUSE2))	HOUSE	HOUSE2	HOUSE2
(*OCCUR (HOUSE HOUSE2))	TLF	TLF	TLF
T	HOUSE		
T	HOUSE2		
T	MARRY		
T	OBJECT		

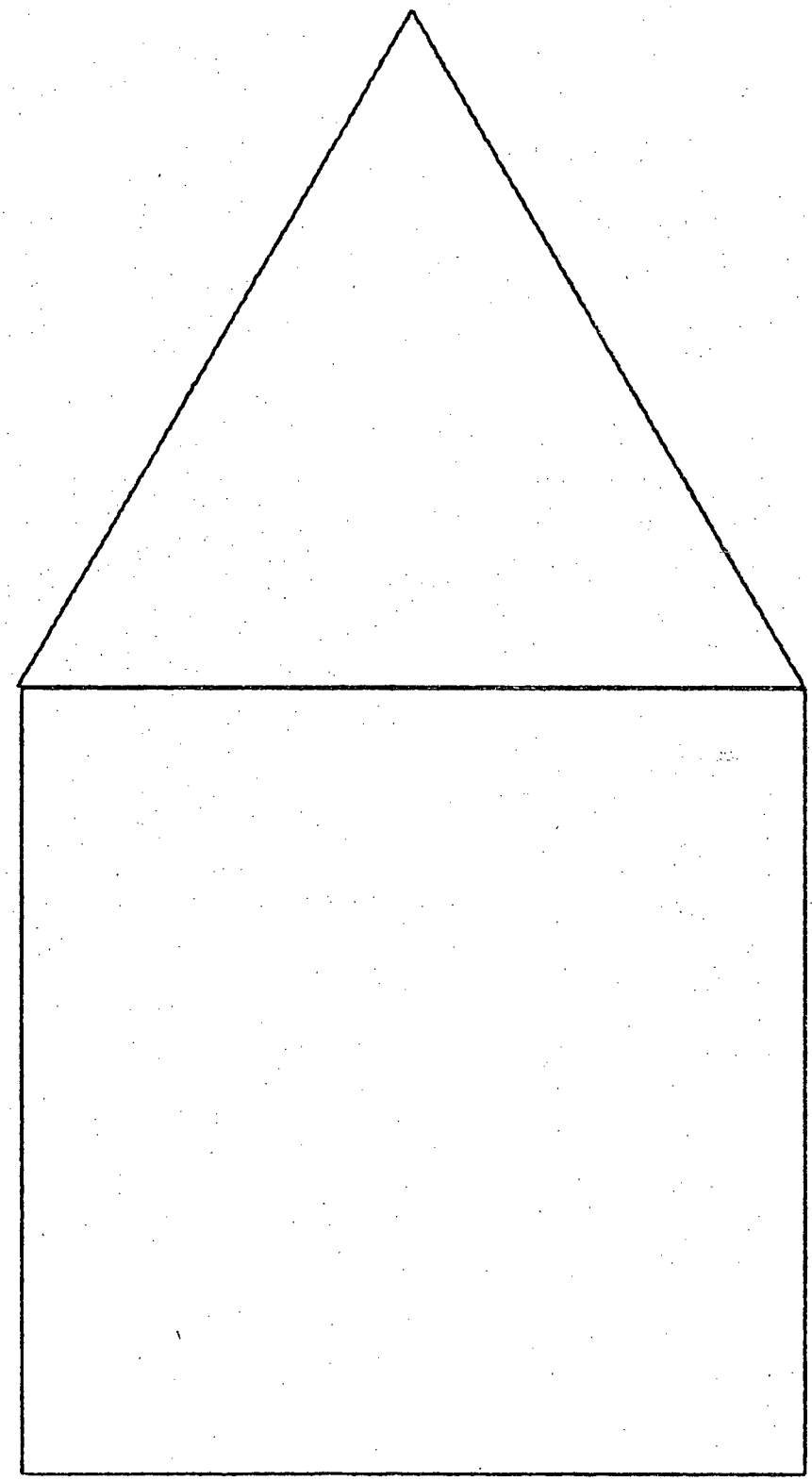

Figure 7.

T	TRIANGLE		
T	TRIANGLE5		
*NEXT	INPUT FRAME	TRIANGLE5	

When the object frame moves the triangle up it will notify the supports and marry frames that the triangle has been moved. The relation frames in turn will check if the correct relations hold. In this case they will be satisfied and demon processing will notify the house EF that the relations between the front and the roof have been satisfied. The house EF will in turn notify the house BF and the TLF. What this means is that the student may reference these two objects as one by the word "house". When the door and the window are added to the picture, they will not be actually part of the house until they are in the correct relation with the front body. This means that if the student refers to "the house", then only the square and the triangle will be considered as the house until the door and the window are in the correct relationship for them to be included in the house.

8.) Draw an upright rectangle.

(DRAW UPRIGHT)	RECTANGLE	TRIAGNGLE5	INPUT FRAME
(MAKE RECTANGLE7)	UPRIGHT	RECTANGLE	RECTANGLE
(UPRIGHT RECTANGLE7)	RECTANGLE		
(RECTNAGLE RECTANGLE7)	TRIANGLE5		

(*ERROR *MESSAGE (RECTANGLE RECTANGLE7))	ROOF6		
(*ERROR *MESSAGE (RECTANGLE RECTANGLE7))	HOUSE2		
(RECTANGLE RECTANGLE7)	FRONT3	HOUSE2	HOUSE2
(RECTANGLE RECTANGLE7)	DOOR	FRONT3	FRONT3
(*DEMON DOOR)	DOOR8	DOOR	DOOR
T	DOOR		
(RECTANGLE RECTANGLE7)	DOOR8	DOOR	FRONT3
(*DEMON DOOR8)	DOOR-BODY	DOOR8	DOOR8
T	DOOR8		
(RECTANGLE RECTANGLE7)	DOOR-BODY	DOOR8	DOOR8
(*DEMON DOOR-BODY)	DOOR-BODY9	DOOR-BODY	DOOR-BODY
T	DOOR-BODY		
(RECTANGLE RECTANGLE7)	DOOR-BODY9	DOOR-BODY	DOOR8
(*DEMON DOOR-BODY9 ((WIDTH RECTANGLE7) (TIMES 0.5 (HEIGHT RECTANGLE7)))	EQUAL	DOOR-BODY9	DOOR-BODY9
T	DOOR-BODY9		

(RECTANGLE RECTANGLE7)	RECTANGLE7 RECTANGLE7	DOOR8 DOOR8	DOOR8 DOOR8
*NEXT	INPUT FRAME	RECTANGLE7	

Again an error situation is encountered. The system backs up to the house EF and then follows the path down to the door frame. The system assumes that the rectangle will eventually form the body of the door (see Fig. 8). A demon is sent to the equal frame, so that when the width of the door is approximately half the height, the door body frame will be notified.

9.) Make it smaller.

(MAKE SMALLER)	RECTANGLE7	RECTANGLE7	INPUT FRAME
(RECTANGLE7 (MAKE SMALLER))	RECTANGLE	RECTANGLE7	RECTANGLE7
(RECTANGLE7 (MAKE SMALLER))	OBJECT	RECTANGLE	RECTANGLE
(MAKE RECTANGLE7)	SMALLER	OBJECT	OBJECT
(SMALLER RECTANGLE7 RECTANGLE7)	OBJECT		
(SMALLER RECTANGLE7 RECTANGLE7)	RECTANGLE		
(SMALLER RECTANGLE7 RECTANGLE7)	RECTANGLE7		

The SMALLER frame will be able to use information, that the door body must be inside the front of the house to decide how small to make the rectangle (see Fig. 9).

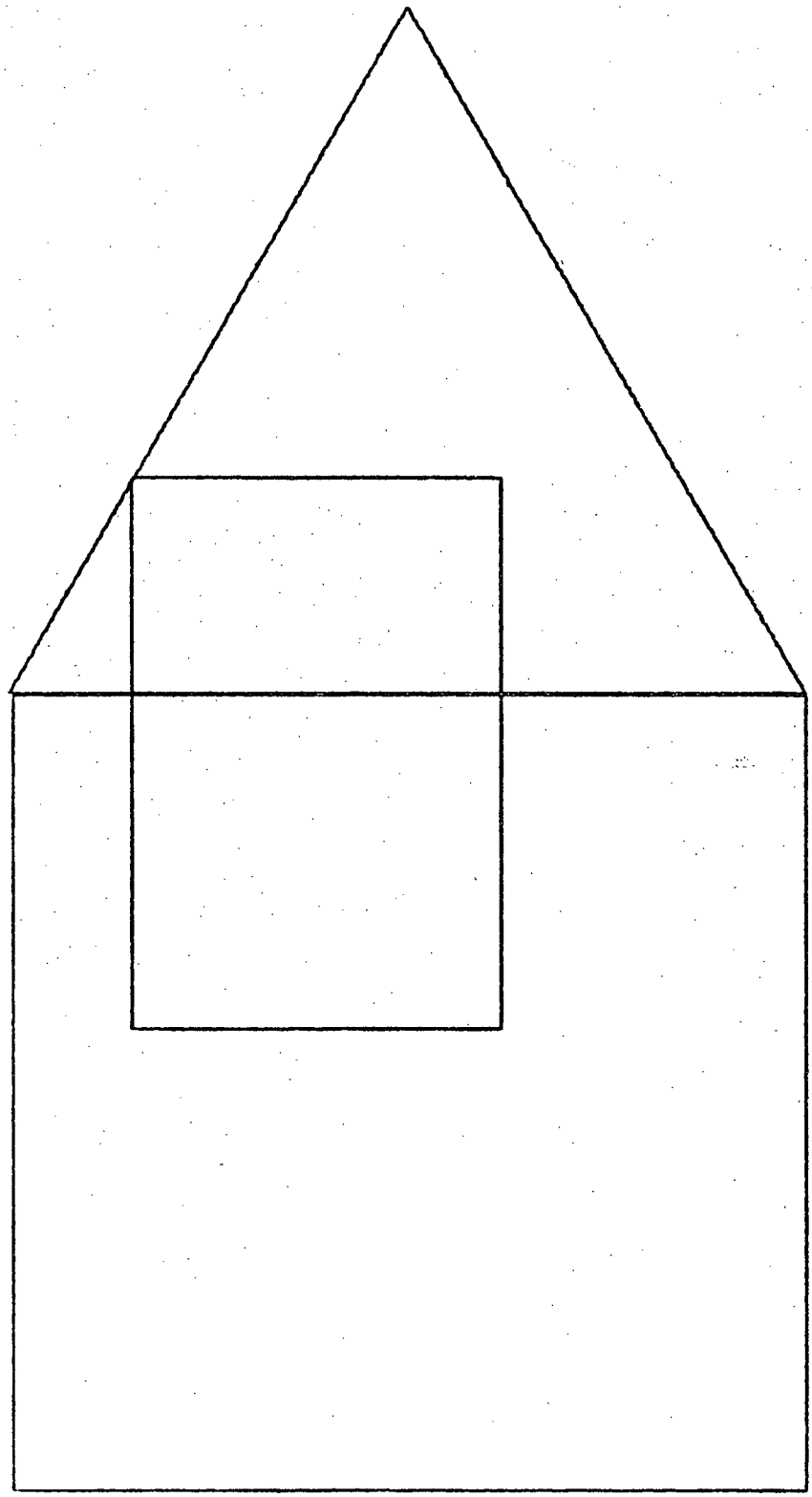

Figure 8.

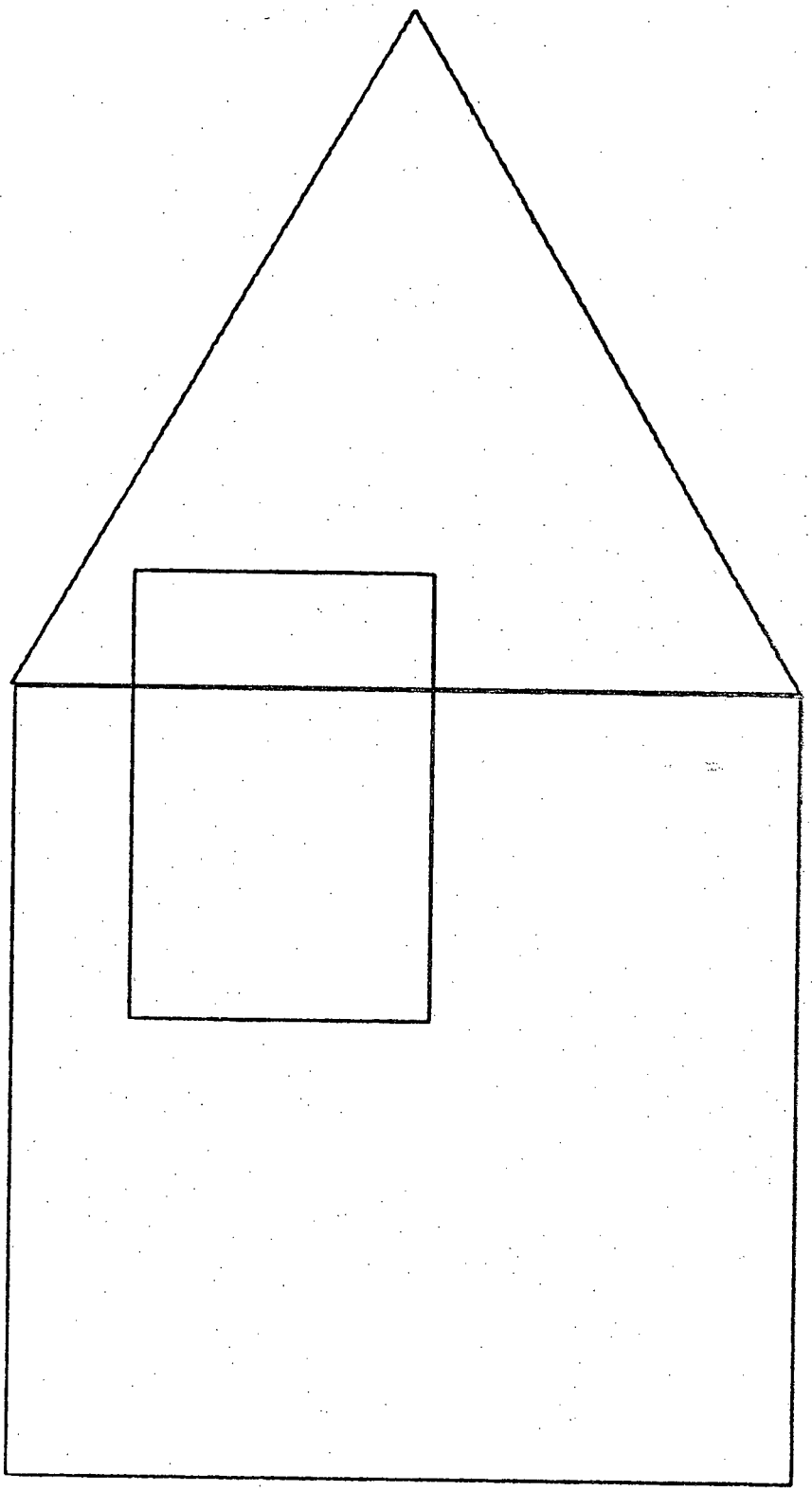

Figure 9.

10.) Thinner.

(MAKE THINNER)	RECTANGLE7	RECTANGLE7	RECTANGLE7
(RECTANGLE7 (MAKE THINNER))	RECTANGLE	RECTANGLE7	RECTANGLE7
(RECTANGLE7 (MAKE THINNER))	OBJECT	RECTANGLE	RECTANGLE
(MAKE RECTANGLE7)	THINNER	OBJECT	OBJECT

At this point, the door body will be of the correct dimension and thus the door body frame will be notified of the fact (see Fig. 10). This in turn will cause demon processing to occur.

(*OCCUR (EQUAL (WIDTH RECTANGLE7) (TIMES 0.5 (HEIGHT RECTANGLE7))))	DOOR-BODY9	EQUAL	EQUAL
(*OCCUR (DOOR-BODY DOOR-BODY9))	DOOR-BODY	DOOR-BODY9	DOOR-BODY9
(*OCCUR (DOOR-BODY DOOR-BODY9))	DOOR8	DOOR-BODY	DOOR-BODY
(*OCCUR (DOOR DOOR8))	DOOR	DOOR8	DOOR8
(*OCCUR (DOOR DOOR8))	FRONT3	DOOR	DOOR
(IS (FRONT-BODY4 DOOR8))	INSIDE	FRONT3	FRONT3
NIL	INSIDE		

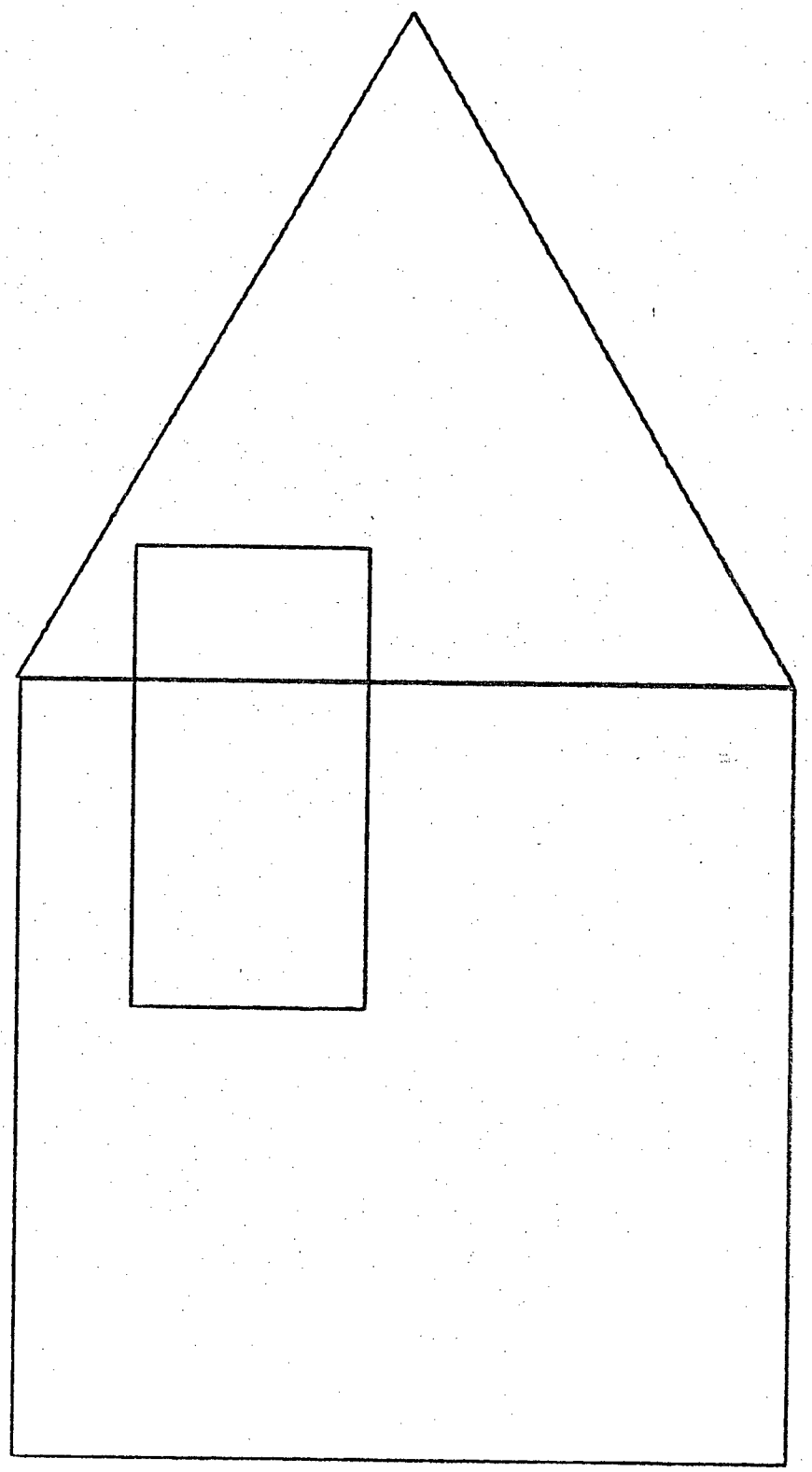

Figure 10.

(*DEMON FRONT3 (FRONT-BODY4 DOOR8))	INSIDE	FRONT3	FRONT3
T	FRONT3		
(IS (BOTTOM FRONT-BODY4) (BOTTOM DOOR8))	EQUAL	FRONT3	FRONT3
NIL	FRONT3		
(*DEMON FRONT3 ((BOTTOM FRONT-BODY4) (BOTTOM DOOR8)))	EQUAL	FRONT3	FRONT3
T	FRONT3		
T	DOOR		
T	DOOR8		
T	DOOR-BODY		
T	DOOR-BODY9		
T	THINNER		
(THINNER RECTANGLE7 RECTANGLE7)	OBJECT		
(THINNER RECTANGLE7	RECTANGLE RECTANGLE7)		
(THINNER RECTANGLE7 RECTANGLE7)	RECTANGLE7		
*NEXT	INPUT FRAME	RECTANGLE7	

The demon processing backs all the way back up to the front because a door is a door when it is just represented by the rectangle without the circle for the door knob. The front frame then checks to see if the relationships between the door and the front body are correct (i.e. the door is inside the front body

and the bottom of the door is equal to the bottom of the front body). In this case the relationships will not hold and so demon messages will be sent to the relation frames so that when they do hold, the front frame will be notified.

11.) Move it down.

(MOVE DOWN)	RECTANGLE7	RECTANGLE7	INPUT FRAME
(RECTANGLE7 (MOVE DOWN))	RECTANGLE	RECTANGLE7	RECTANGLE7
(RECTANGLE7 (MOVE DOWN))	OBJECT	RECTANGLE	RECTANGLE
(*OCCUR (EQUAL (BOTTOM FRONT-BODY4) (BOTTOM DOOR8)))	FRONT3	EQUAL	EQUAL
T	EQUAL		
(*OCCUR (INSIDE FRONT-BODY4 DOOR8))	FRONT3	INSIDE	INSIDE
T	INSIDE		
T	RECTANGLE		
T	RECTANGLE7		
*NEXT	INPUT FRAME	RECTANGLE7	

In the above sequence, the rectangle is moved down so that the two relationships now hold (see Fig. 11). The front EF is then notified via demon processing and assigns the slots for the two relationships. The system will know that the rectangle is to be moved down until its bottom is equal to the bottom of the door because of demon messages sent to the relations.

12.) Draw a very small circle.

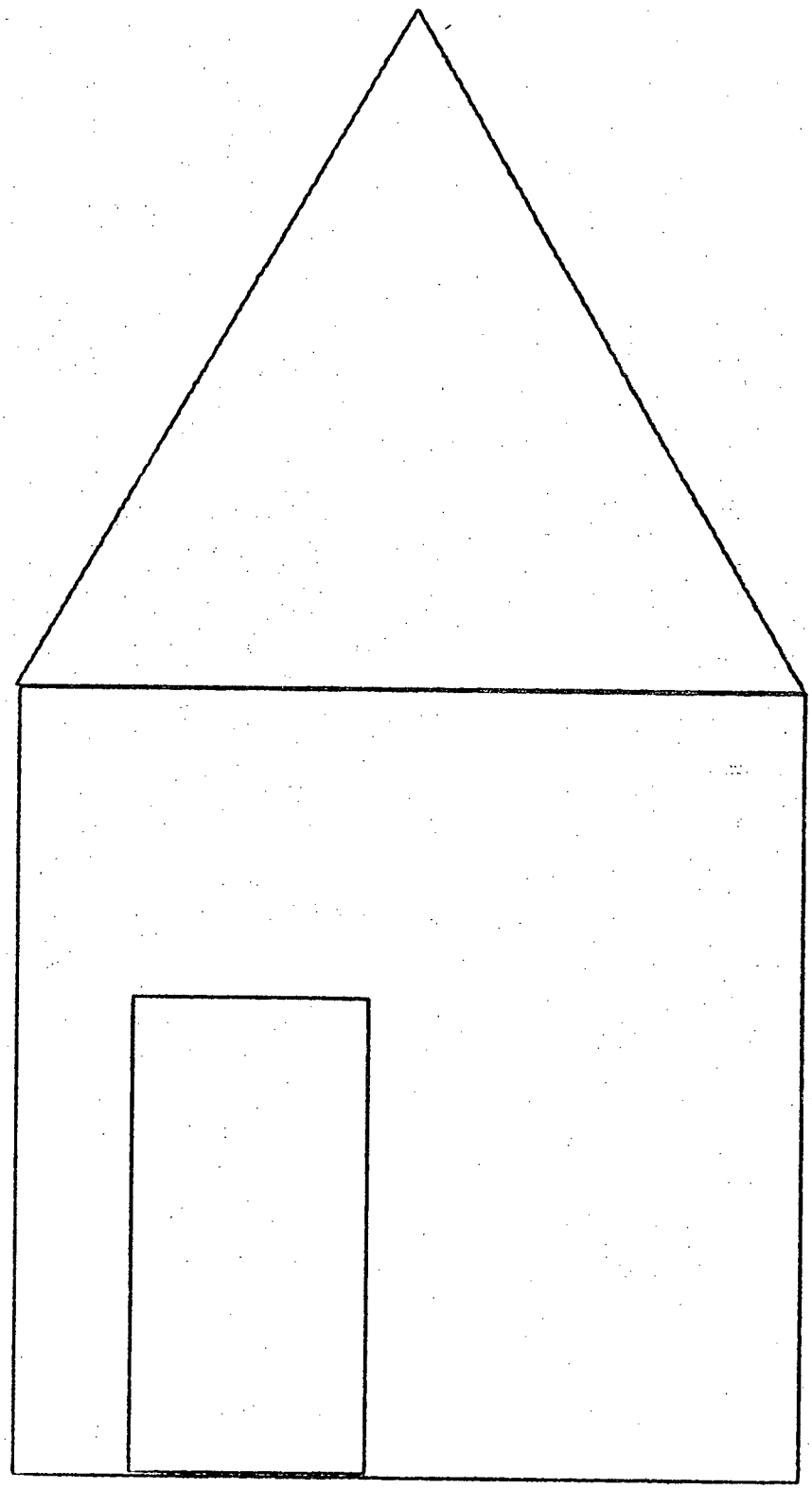

Figure 11.

(DRAW (SMALL VERY))	CIRCLE	RECTANGLE7	INPUT FRAME
(MAKE CIRCLE10 VERY)	SMALL	CIRCLE	CIRCLE
(SMALL CIRCLE10)	CIRCLE		
(CIRCLE CIRCLE10)	RECTANGLE7		
(*ERROR *MESSAGE (CIRCLE CIRCLE10))	DOOR-BODY9		
(*ERROR *MESSAGE (CIRCLE CIRCLE10))	DOOR8		
(CIRCLE CIRCLE10)	DOOR-KNOB	DOOR8	DOOR8
(*DEMON DOOR-KNOB)	DOOR-KNOB11	DOOR-KNOB	DOOR-KNOB
T	DOOR-KNOB		
(CIRCLE CIRCLE10)	DOOR-KNOB11	DOOR-KNOB	DOOR8
NEXT	INPUT FRAME	CIRCLE10	

The system assumes that the very small circle will represent the door knob (see Fig. 12). The door EF sent demon messages so that when the circle is in the correct position, it will be notified. I am going to assume that the student will move the circle around until it is in the correct position. The sentences encountered will be similar to those already described.

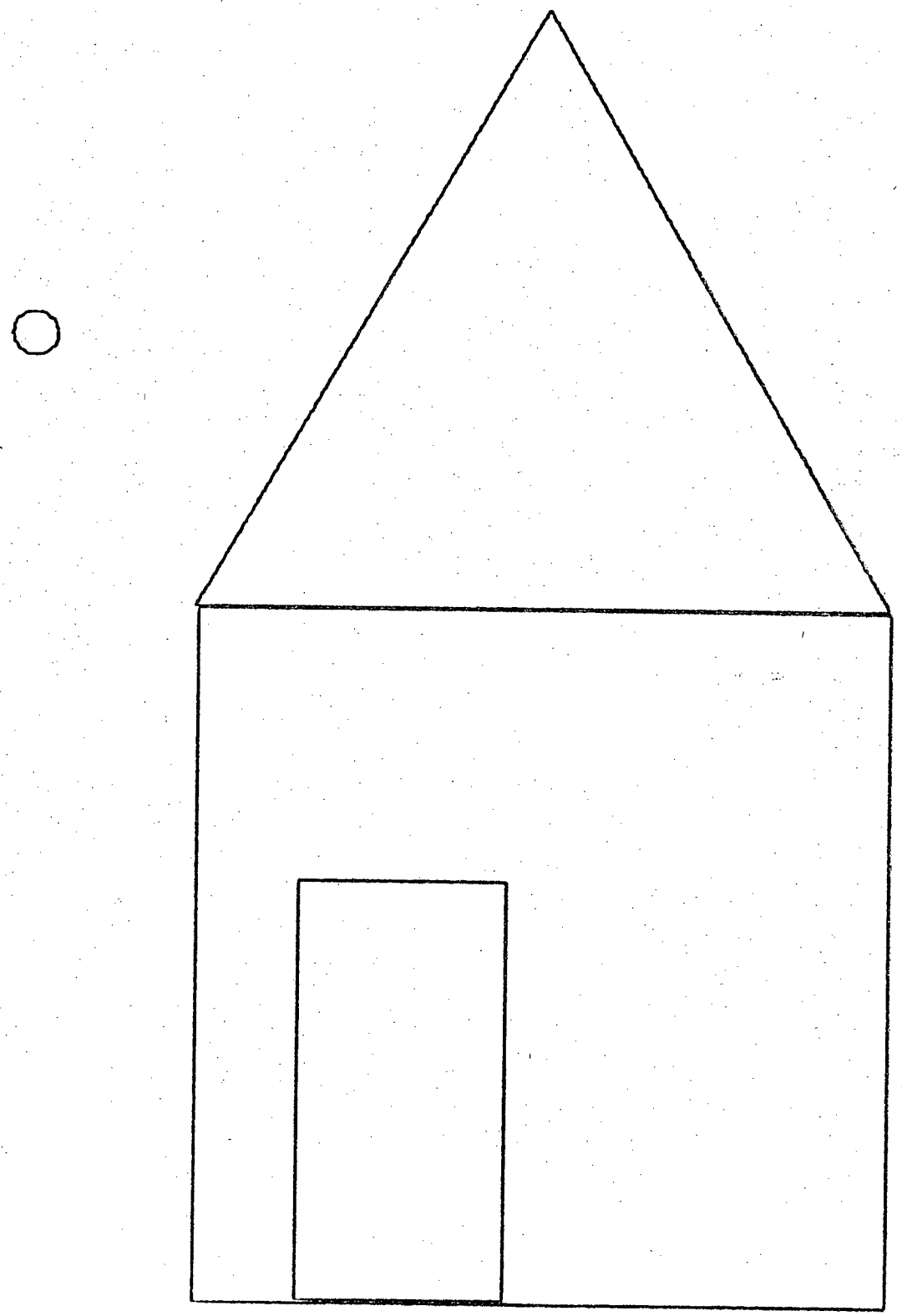

Figure 12.

(DOOR-KNOB DOOR-KNOB11)	DOOR-KNOB	DOOR-KNOB11	DOOR-KNOB11
(DOOR-KNOB DOOR-KNOB11)	DOOR8	DOOR-KNOB	DOOR-KNOB
T	DOOR-KNOB		
T	DOOR-KNOB11		
(CIRCLE CIRCLE10)	CIRCLE10	DOOR-BODY11	DOOR-BODY11
*NEXT	INPUT FRAME	CIRCLE10	

The "door knob" will now be in the correct position in relation to the other components (see Fig. 13). The system will now have the expectation that a window for the house is to be drawn.

13.) Draw a small square inside the front.

(DRAW SMALL (INSIDE FRONT3 ?))	SQUARE	CIRCLE10	INPUT FRAME
(MAKE SQUARE12)	SMALL	SQUARE	SQUARE
(SMALL SQUARE12)	SQUARE		
(MAKE FRONT3 SQUARE12)	INSIDE	SQUARE	SQUARE
(INSIDE FRONT3 SQUARE12)	SQUARE		
(SQUARE SQUARE12)	CIRCLE10		

The part: "(INSIDE FRONT3 ?)" in the message of the first entry of the table specifies that the square to be created is to be placed inside the front of the house. The "?" is necessary,

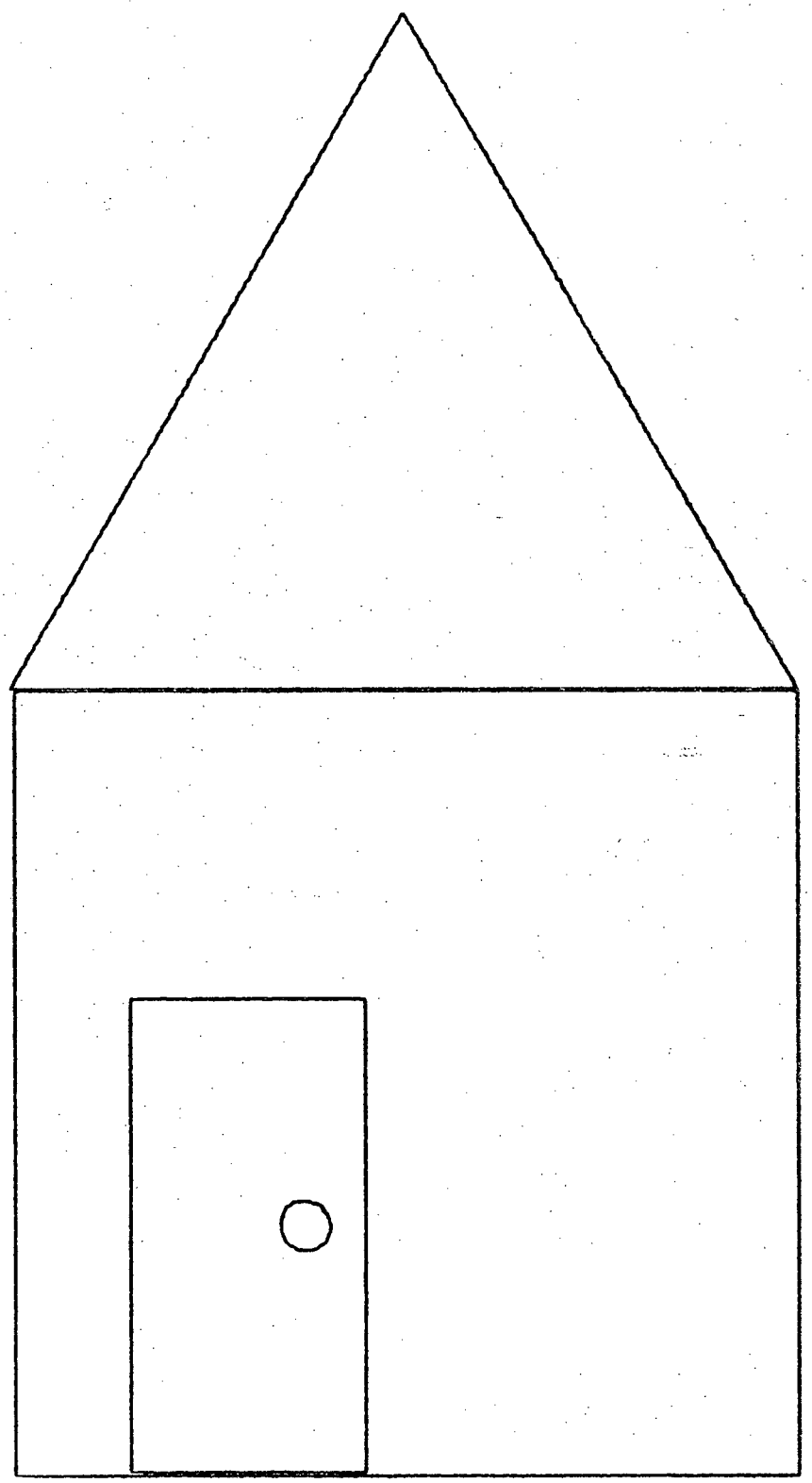

Figure 13.

since when the sentence is processed by the input frame, it will not know the name of the new square. This allows the square frame to construct the correct message to be sent to the INSIDE frame (see Fig. 14). At this point error processing will occur since the circle EF knows nothing about squares.

(*ERROR *MESSAGE (SQUARE SQUARE12))	DOOR-KNOB11		
(*ERROR *MESSAGE (SQUARE SQUARE12))	DOOR8		
(*ERROR *MESSAGE (SQUARE SQUARE12))	FRONT3		
(SQUARE SQUARE12)	WINDOW	FRONT3	FRONT3
(*DEMON WINDOW)	WINDOW13	WINDOW	FRONT3
T	WINDOW		
(SQUARE SQUARE12)	WINDOW13	FRONT3	FRONT3

The system assumes that the square will represent the window. I am again going to assume that the student will move the square around until it is in the correct position. When this happens, demon processes will notify the front EF (see Fig. 15).

(*OCCUR (WINDOW WINDOW13))	WINDOW	WINDOW3	WINDOW3
(*OCCUR (WINDOW WINDOW13))	FRONT3	WINDOW	WINDOW
T	WINDOW		

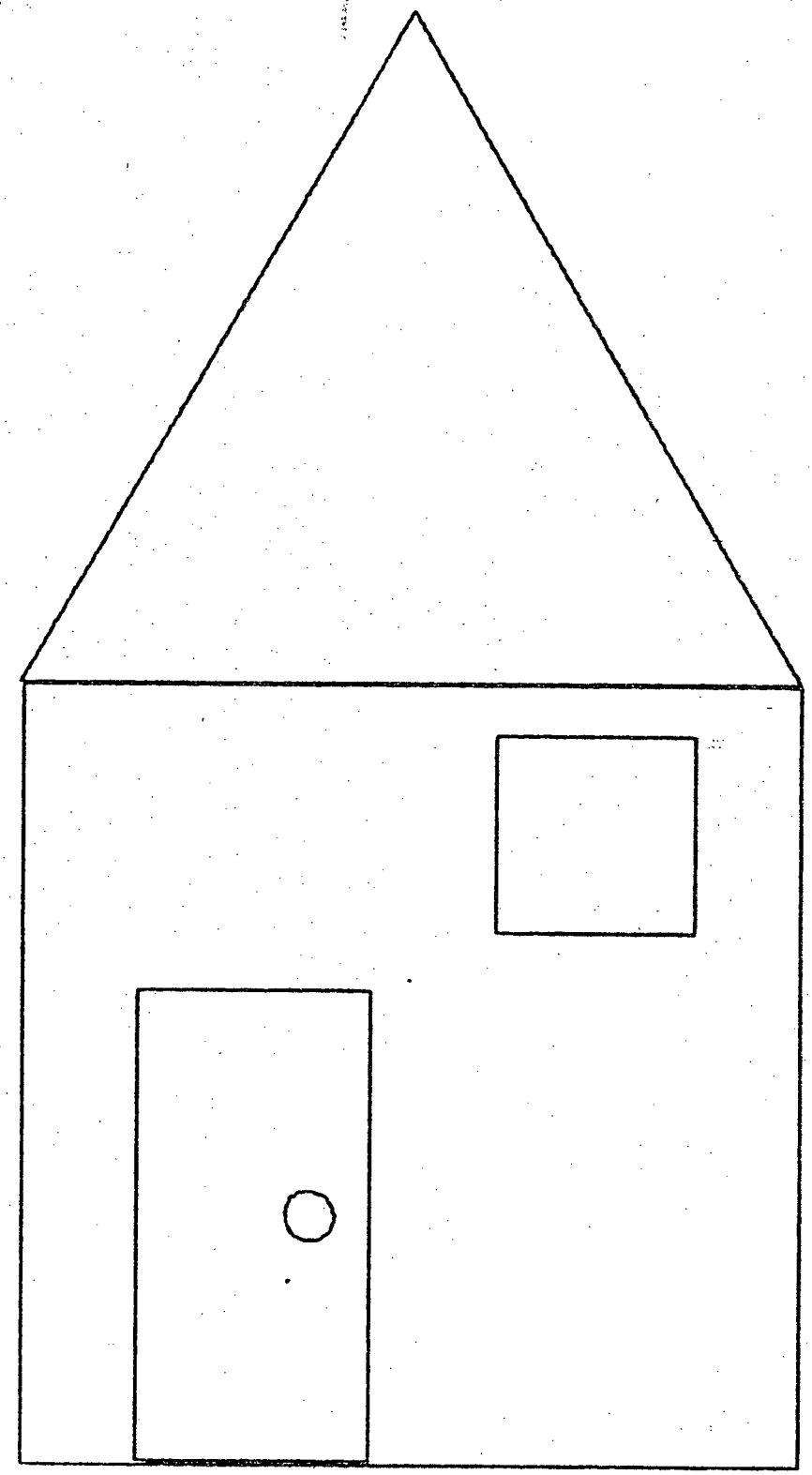

Figure 14.

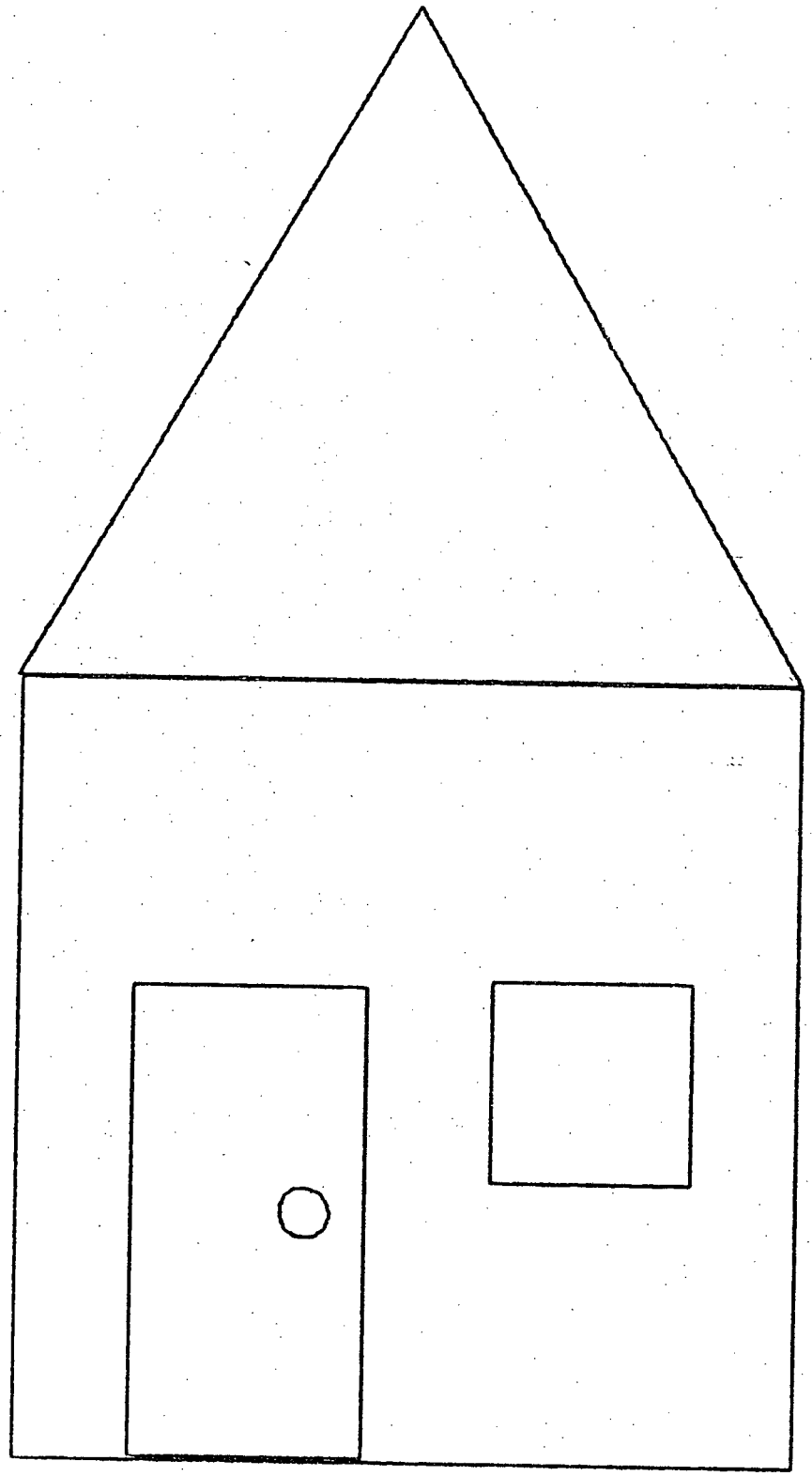

Figure 15.

T	WINDOW13		
(SQUARE SQUARE12)	SQUARE12	WINDOW13	WINDOW13
*NEXT	INPUT FRAME	SQUARE12	

At this point the house will have been completed and the student would normally be asked to redraw the same figure. This time, however, he would be told how to use the hardware of the graphics terminal to perform the tasks that he had previously done using English. This will begin to show the student that it is relatively easy to manipulate the objects on the graphics screen using the hardware available rather than having to specify every action in English or an even less natural form.

4.3. Comments on the Example

Most of the example described above has been implemented. However, some of the frames must have additions made, so that they will be able to handle some of the details of the system. The system is incremental, in that more detail may be added by writing new frames or adding information to current frames. No implementation of the input or output frame has been attempted.

The example above has gone into many of the features of the system; however, a greater emphasis is needed on how error processing is performed when an assumption has been made in error. This is an important aspect since any system is bound to make some assumptions which later turn out to be incorrect. People sometimes assume something in a conversation which later

turns out to be untrue and are able to correct themselves without any great difficulty. Sometimes the information that contradicts the assumption can be used to redirect the computation to the correct conclusions. Thus there must be some facility in the system to destroy frames that have been created due to incorrect assumptions and to reassign slots of frames which have already been instantiated. There is a problem with this in that when an object is originally drawn, it may violate a condition but will be moved about the screen before it is in its final position. Determining when an object is in its final position is a problem in itself and cannot be easily ascertained. In fact it is impossible since the student may at any time change it by moving it around the graphics screen or by increasing or decreasing its size or by performing some other action.

There must be some method of determining when an assumption has been contradicted. For example, when constructing the house as in the above example, the first square drawn will normally assumed to be the front body of the house. In some cases a person may start to draw the minor details of the house before drawing the major details. That is the first square drawn may not represent the front body but may be meant to represent the window of the house. This incorrect inference at the beginning will not cause any major difficulties, since all the actions will be directed to the EF for the square just drawn. The only problem originally will be that the front body EF will have the wrong square representing it.

The system must continually check to see if new information introduced into the system by the sentences input by the student contradict any of the current information. The system must not be too quick to decide that new information contradicts current information. This must be done so that if the contradiction is only a temporary one, then there won't be too much work changing information back and forth between frames. In the case of drawing a house, the first square to be drawn will be assumed to be the front body for the house. However, if a new square is drawn which completely surrounds it then the system must decide whether to change the assignments of the slots in the front body frame and the window frames or not. This decision should be made at one of two times: when the system is about to change the focus or when the system must actually use the information about which square represents what to disambiguate a sentence. A change of focus occurs when a sentence causes a message to be sent to a frame other than the referent for "it". When the system is about to change the focus of the conversation from one object to another, then it must decide how the slot assignments of the affected frames are to be assigned. This rule is used since the current topic focus will usually be one of the frames that is causing the problem. When the student is satisfied with its position, the system may make assumptions from the current positions of the objects on the graphics screen and reassign slots of the frames affected. This processing will only happen when there is some doubt about what the various objects represent in the figure. In the example given above, none of

this processing would be necessary since the conversation follows the "expected" path. When the assumptions have been made, the computation will continue as it normally would have under the new state of the world. When the decision has to be made to disambiguate a sentence the same form of processing as in the above case will be performed. The system will use the information about the current state of the objects on the graphics terminal screen.

An important question at this point is how does the system recognize that the new information contradicts information that has been assumed from previous information. The system must have some form of expectation for such errors to occur. This immediately leads to the idea that demons be sent to the frames which will be the ones to recognize the contradiction in order to tell the frame that must make the decision when such conditions arise. For example, in the case of drawing a house and the possible confusion between the front body and the window, the front EF is the one which must eventually make the decision about which square represents the front body and the window. When the front EF has assigned the front body and the window slots, it will check to see if the relationship between the front body and the window are such that the two should reverse their assignments. If this is the case the front frame must send a demon message to the input frame to tell it to notify the front EF when either the front body or the window are referenced or when it changes the focus from the current object. When this happens the front EF will be notified about the need

to make the decision. It may then use the current information available to make a decision as to whether the slot assignments of the two frames should be changed or not. The slots in fact may change places many times in a computation (even though this hopefully will not be the case) depending on what the student or conversant says to the system.

In the example, all the error processing associated with invalid messages is done by backing up to the frame in which the current frame is part. This does not have to be the way the processing needs to be done. What could happen is that when an invalid message is encountered, control may be immediately sent to the top level frame so that it may decide what to do with the message. In most cases the message will be sent down the same path the original message was sent. In this way the highest level frame which can accept the message will gain control of the computation. The difference is a matter of emphasis. In the back up method the first component on the way back up which recognizes the message will carry on the computation. In the top down method, the top most component which receives the message will gain control of the computation. The distinction is between filling in minor or major detail first. Once some of the major details have been filled it is likely that the student will fill in the minor detail starting with the last major component drawn. Which way this type of error is processed may be made to depend on what type of figure is being drawn, or at what point in the computation the system is.

Something which was not fully specified in the example was

how the graphical object frame would be able to use the demon information to decide on how much to move an object. When a demon message is sent to a relation frame, the frame will send a message to the graphical object frame specifying that the relation frame is interested in the objects passed in the demon message. When the graphical object frame is about to move an object it will check to see if there are any demons which contain that object. If there are, then the graphical object frame will send a message to the relationship frame asking how far that object must be moved to satisfy the relation. The relation frame will return a list of two elements. The two elements will be a range that the object may be moved in the X direction and the Y direction respectively. For a relation such as inside, the ranges may be large, but for other frames there may only be one position which will satisfy the relationship. The same idea can be used for other types of relationships. Using the combination of all such information, the graphical object frame can actually decide how much to move the object.

Chapter 5.

Conclusions

A model which will handle the semantic knowledge necessary to understand a subset of natural language has been described. The model is incomplete; however, it shows that it is possible to develop a model which handles a small subset of the English language. It was not designed as a general representation of knowledge. However, it could be generalized to represent information for more general domains of knowledge to be used for understanding natural language.

The structure of frames is such that when a new scene or figure is to be used in teaching the student the use of a graphics terminal, the whole system will not have to be rewritten. Some new frames will be written which correspond to the components of the new figure. Thus the new figure will cause only a few new frames to be written rather than major changes having to be made to the system. The top level frame will also have to be rewritten, so that it will be able to begin the process of recognizing this new figure. All the relation frames, property frames, built in object frames, the input and output frames and the superset frames will remain the same. Some new relation and/or property frames may have to be written to handle the new figure, but once they have been written they can be used by all succeeding figures. This makes it relatively easy to make additions to the system.

Most frames will know everything about the concept they represent. This is not quite true about frames representing nouns, since there will be superset hierarchy among most of the noun frames in the system. Thus frames which represent specific objects will contain only information which is specific to that object alone. This frame will have associated with it superset pointers to other frames which will contain more general information about the concept. This means that information that is general in nature will not be duplicated in many different frames but may be consolidated into a general one.

Assertions in the system for teaching a student how to use a graphics terminal are all assumed to be absolutely true. This means that the various frames will have to keep checking to make sure that the assertions remain true. In a more general domain facts should have some indication of the validity of the assertion. How to implement assertions which have a variable truth factor is a very difficult problem. One simple method using a range of numbers (say 0 to 10) according to how certain the assertion is (e.g. 10 could mean absolutely true, 0 could mean absolutely false, 1 could mean that the assertion may be true but almost certainly not, etc.). This sort of method does not appear very appealing, since people do not seem to use some arbitrary scale to decide on the validity of an assertion.

A major difference between this system and other systems which have been written to understand natural language is that verbs are not separate procedures which are invoked when the action corresponding to the verb is to be performed. In this

system, the verb is part of a message to the frame that represents the object that is the subject of the action (i.e. the object on which the action is to be performed). Thus to move a square higher up on the graphics screen a message will be sent to the square frame specifying the action move. In some cases the square frame will contain the required information to perform the action, in other cases a superset of the square frame will contain the desired information. In most other systems the verb is a procedure (or theorem) which is passed the name of the object for which the action is to be performed. This is really only a matter of emphasis, since both methods should cause the same actions to be performed correctly. The reason that I did it this way is that the information about how to perform actions on an object is information which should be associated with the object and not be a separate frame. The frame for the object knows more about how it should react to an action to be performed on it than some general procedure which would have to perform some processing to see how to perform the action.

In a specific domain such as described in the Introduction, there is no difference because each verb will usually only have one method of performing an action for all objects in the domain. However, in a more general domain it is possible that the same verb will have a different action associated with it depending on the subject of the action. If a general procedure for an action is used then it will have to perform extra processing to decide how exactly the action on the parameter is

to be carried out. Using the method described above this extra processing would not be necessary (even though superset processing may be necessary) since the frame (or one of its supersets) originally sent the message will know exactly how to perform the action. If an error is encountered in the default processing of the action, then the frame with which that action is performed will also be the best frame to decide what is to be done next, whether to try some other method or give up. The superset links used in the system will give the ability to associate actions with as general a concept as possible and thus all subsets to this concept will have access to this method of performing the action and all superset frames will not normally have direct access to the information.

A problem which is associated more with frames and the representation of knowledge rather than with natural language is how is learning performed. For example, when drawing a graphical house if a second small square is drawn to represent a second window for the front of the house, what is the system to do. In the present system, the system would complain to the student that there should not be a third square. What is actually needed is to create a new frame using the old front frame as the prototype and to make the necessary changes to add a second window to the new one. This is another whole problem which will need much work before it is solved. People seem to be able to perform such learning tasks quickly and easily in most cases and so should any sytem for representing knowledge.

Context is a very important aspect of a system since it

must know what the current topic is. This is necessary in a system since this information will often be the only way that some sentences will be disambiguated. Context is also useful in reducing the amount of information that needs to be looked at for understanding a sentence. This should greatly reduce the amount of time needed to understand a sentence.

Demon processing is handled much differently than in most other systems where there are general demon processes in existence which must be checked every time information is added or deleted from the data base. The patterns associated with the demon processes may match items on which the demon is not meant to work. In this system the actual parameters of the demon process are known when the demon is made active. Thus the demon will only be invoked when the parameters sent in the original demon message are used. Most demons will be associated with specific relationships between two objects or with the creation of new instances of objects. When a demon is successfully invoked a message will be sent to frame frame waiting for the demon, saying that the action or object that was wanted has occurred.

A problem which is associated with frames is how to select a frame to represent the current situation. In this system the major decision has already been made since the student is told which figure to draw. This is a major problem which must be solved before frame systems can become fully useful as a general representation of knowledge.

Several conclusions have been made about the design of the

model and how it represents knowledge for understanding natural language. The frame approach to the representation of knowledge seems to work quite well. This is mainly due to the fact that the internal structure of a frame is not important, as long as the frames which may send message to a frame know what messages the frame will accept and what responses it may make. This means that the contents of a frame may be changed without affecting the performance of the system as a whole as long as the external part of the frame remains the same. The frame may be made to accept more messages and may be changed as to how it actually handles the processing of already existing messages without affecting other frames in the system.

Bibliography

[1] Bobrow, D. and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments", CACM, Vol. 16, No. 10, pp 591-603, 1973.

[2] Brown, J., Burton, R., Bell, A., "SOPHIE A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting (An Example of AI in CAI)", BBN Report No. 2790, 1974.

[3] Carbonell, J. R., "Mixed-Initiative Man-Computer Instructional Dialogues", Ph.D. Thesis, BBN Report No. 1971, 1970.

[4] Charniak, E., "Toward a Model of Children's Story Comprehension", M.I.T. AI Lab TR-266, 1972.

[5] Crignetti, M. and Warnock, E., "Mixed-Initiative Information System for Computer Aided Training and Decision Making", BBN Report No. ESD-TR-73-200, 1973.

[6] Fahlman, S. E., "A Planning System for Robot Construction Tasks", in Artificilal Intelligence Vol. 5, No. 1, pp 1-49, 1973.

[7] Fahlman, S. E., "A Hypothesis-Frame System for Recognition Problems", M.I.T. AI Lab Working Paper 57, 1974.

[8] Fahlman, S. E., "Thesis Progress Report: A System for Representing and Using Real World Knowledge", M.I.T. AI Lab Memo No. 331, 1975.

[9] Havens, W., "A Model for Artificial Intelligence Programming Languages", Thesis Proposal, U.B.C., 1975.

[10] Hewitt, C., "Description and Theoretical Analysis (Using Schemata) of Planner", Ph.D. Thesis, M.I.T. AI Lab TR-258, 1972.

[11] Hewitt, C., Bishop, P. and Steiger, R., "A Universal Modular Fromalism for Artificial Intelligence", IJCAI-73, pp 235-245, Stanford, California, 1973.

[12] Hewitt, C., and Grief, I., "Actor Semantics for Planner-73", Sigplan - Sigart Conference, 1975.

[13] de Kleer, J., "Qualitative and Quantitative Knowledge in Classical Mechanics", M.I.T. AI Lab Working Paper 88, 1975.

[14] Kuipers, B. J., "Frames: Representing Knowledge for Recognition", M.I.T. Lab, Xerox Palo Alto Research

Centre, 1975.

[15] McCalla, G., "Private Communications", Summer, 1975.

[16] McDermott, D. V. and Sussman, G. J., "The Conniver Reference Manual", M.I.T. AI Lab Memo No. 259, 1972.

[17] Minsky, M., "A Framework for Representing Knowledge", M.I.T. AI Lab Memo No. 306, 1974.

[18] Quillian, M., "Semantic Memory", Ph.D. Thesis, Carnegie-Mellon University, reprinted in Minsky, M. (ed.) Semantic Information Processing, MIT Press, Cambridge, Mass., 1968.

[19] Reiger, C., "Understanding by Conceptual Inference", Computer Science Department, University of Maryland, 1974.

[20] Schank, R., "Identification of Conceptualization Underlysing Natural Language", in Schank and Colby, Computer Models of Thought and Language, pp 187-248, W. H. Freeman, 1973.

[21] Schank, R. and Colby, K., "Computer Models of Thought and Language", W. H. Freeman, 1973.

[22] Simmons, R., "Semantic Networks: Their Computation and Use for Understanding English Sentences", in Schank and Colby, Computer Models of Thought and Language, pp 63-113, W. H. Freeman, 1973.

[23] Smith, B. and Hewitt, C., "Towards a Programming Apprentice", AISB Summer Conference, University of Sussex, 1974.

[24] Sussman, G., Winograd, T. and Charniak, E., "Micro-Planner Reference Manual", M.I.T. AI Lab Memo No. 203a, 1973.

[25] Winograd T., "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language", Ph.D. Thesis, M.I.T. AI Lab TR-84, 1971.

[26] Winograd T., "Frame Representation and the Declarative/Procedureal Controversy", to appear in Essays in Memory of Jaime Carbonell, 1974.

[27] Winston, P., "Learning Structural Descriptions from Examples", Ph.D. Thesis, M.I.T. AI Lab TR-231, 1970.

Appendix 1.

A Brief Description of Conniver

Since some of the terminology used in the thesis comes from the Conniver programming language, this appendix gives a brief view of some of the features of Conniver which will be useful in understanding some of the constructs in the thesis. In particular, the Conniver pattern matcher and control structure, which are used heavily in the thesis will be described.

1.1. The Conniver Pattern Matcher

All the pattern matching done by the system is done using the Conniver pattern matcher, so that a brief description of the more important aspects of the pattern matcher used in the body of the thesis is in order. Most of this description is from the Conniver Reference Manual[16].

The Conniver pattern matcher is called in the following way:

(MATCH VARPAT DATAPAT).

Where the variable pattern VARPAT is a pattern which may contain variables which are to be assigned values from the data pattern DATAPAT or whose values are to be used in the pattern match. The pattern matcher works to any level, so that variables may occur at any level in the variable pattern, unlike the Micro-Planner pattern matcher which only matches variables at

the top level. The data pattern DATAPAT is the pattern against which the variable pattern will be matched. This data pattern may contain variables, but this feature is mainly useful for Conniver's if-added, if-removed and if-needed methods, and are not needed by the frame system, so they will not be described.

MATCH returns NIL if no match is possible and returns a list of two association lists if the pattern match is successful. The first association list corresponds to the variable pattern and associates with each variable in the variable pattern the sequence in the data pattern which was matched against that variable. The second association list corresponds to the data pattern and associates with each variable in the data pattern the sequence in the variable pattern which it was matched against that variable. In most cases the second association list will be NIL. Note that the variables in the data pattern must match against variables in the variable pattern.

Another version of the Conniver pattern matcher, which is not actually part of the conniver system but has been found useful, is called in the following way:

(FRMATCH VARPAT DATAPAT).

Where VARPAT and DATAPAT have the same meaning as for the description of MATCH above. However, FRMATCH return T if the pattern match is successful and NIL otherwise. As a side effect the variables in the variable pattern are set to the values corresponding to the section of the data pattern that they matched. The variables are only set if the pattern match

succeeds. This is useful when the data patterns are messages sent to a frame.

As with Micro-Planner, Conniver variables in patterns must be specified by the use of prefixes. There are several possible prefixes which are available for use. Each of the prefixes cause the pattern matcher to perform different actions. Some of the more important prefixes are:

1.) !>VAR - This is the basic Conniver pattern matching variable and will match any expression which does not contain any variables. The special form !>(VAR RESTRICTION1, RESTRICTION2, ... , RESTRICTIONn) matches any form which does not contain any variables. Also, all the restrictions, RESTRICTION1, RESTRICTION2, ... , RESTRICTIONn, must evaluate to a non-NIL value for the match to succeed.

2.) !,VAR - This does not bind a variable, but refers to the current value associated with the Conniver variable VAR. This value may have been produced by a previous !> binding or by the Conniver binding of the variable VAR before the pattern matching began.

3.) !,(VAR VALUE) - This binds VALUE to VAR and matches anything that VALUE would match.

There are other Conniver pattern matching prrefixes which are available, but they are not used in the current system. Following are some examples of how the Conniver pattern matcher works. The following call to MATCH:

(MATCH '(FOO !>X) '(FOO BAR))

will return the list of the two association lists (((X BAR))

NIL). Where the NIL is the association list for the data pattern (FOO BAR). The pattern:

(GRANDFATHER !>X !,X)

will match all items corresponding to people who are their own grandfathers. This could be used in an If-added pattern to make sure no contradictory information of this sort is added to the data base. The pattern:

!>(X (ATOM !,X))

will match only atoms. The pattern:

(FUNCT-OF !>FORM !,(F (CAR !,FORM)))

will match (FUNCT-OF (FACTORIAL 3) FACTORIAL), but will not match (FUNCT-OF (PLUS 2 2) MINUS).

The pattern:

((FREDS !>X) . !>REST)

matches ((FREDS FATHER) WHISTLES) assigning FATHER to X and (WHISTLES) to REST. It also matches ((FREDS FATHER) WHISTLES DIXIE) assigning FATHER to X and (WHISTLES DIXIE) to REST.

1.2. The Conniver Control Structure

The Conniver control structure differs from that of most other languages. LISP's control structure is recursive. In Micro-Planner the basic control structure is its backup facility which can be easily simulated using the recursive control of Lisp. Conniver, however, does not have recursive control as such, but a structure similar to that proposed by Bobrow and Weigbreit[1]. Conniver's control structure is a generalization

of Lisp's. This generalized control allows a routine to be invoked and then suspended leaving its environment in-tact.

The Conniver control structure uses the idea of a stack frame developed by Bobrow and Weigbreit[1]. A stack frame consists of an association link which is a pointer to another stack frame. This association link is used as a variable association list and is used for associating values with variable names. There is also a control link which is a pointer to the stack frame which caused the current stack frame to come into existence. There is also a location in a stack frame for storing the program counter of the next instruction to be executed when control is returned to the current stack frame. Whenever a routine is called a new stack frame is created for it to carry out its computation. When a routine is suspended a pointer must be created which points to the stack frame for the routine. This means that the control and association links will remain in existence until this pointer is destroyed, since the stack frame will not be garbage collected until the pointer no longer exists.

The Conniver control structure is a very general one which allows parallel processing or simulated parallel processing in this case. At any one time there may be many different frames which may be activated and in existence. Each such frame will have associated with it a pointer to the stack frame corresponding to an invocation of the frame.

1.3. The Conniver Data Base

The Conniver data base is a flexible facility allowing for adding and deleting information, and performing other actions on the data in the data base. The data base consists of a hierarchy of contexts. Each context may or may not contain some piece of information. The Conniver data base allows for the creation of new contexts, pushing and popping of contexts. Also, there are methods available for performing inferences similar to those available in Micro-Planner. When Conniver is first run the two Conniver variables CONTEXT and GLOBAL are assigned the value of the same empty context. The variable CONTEXT is used as the default value used in all functions which require a context. In the following, variables enclosed in brackets ([ ]) are optional and will default to the value of the Conniver variable CONTEXT.

The following functions manipulate contexts only:

(PUSH-CONTEXT [CONTEXT])

This function pushes a new layer onto the context specified by CONTEXT and returns this new context as value. The information in the old context is still available in the new context, but any changes to this information will only be reflected in the new context and not in the old one. This allows a function to create a hypothetical context in which it may change information without worrying about destroying the old context in case the function fails.

(POP-CONTEXT [CONTEXT])

This function pops a layer from the context specified by CONTEXT and returns this new context as its value. Thus any information

in this layer will be lost. This restores the context to the state just before a PUSH-CONTEXT was performed.

(FINALIZE [CONTEXT])

This function has the same value as POP-CONTEXT, with the side effect of making its argument an equivalent context to its superior. That is all data will have the same properties in the super context that they had in the original context.

The following functions all manipulate data which are in a context.

(ADD ITEM [CONTEXT])

This function adds item to the context specified by the context CONTEXT. For example,

(ADD '(IS-A SQUARE SQUARE1))

will add the data item "(IS-A SQUARE SQUARE1)" to the current context.

(REMOVE ITEM [CONTEXT])

This function deletes item from the context specified but the context CONTEXT.

(FETCH PATTERN [CONTEXT])

This function will return a possibilities list of all the data items in the context specified by the context CONTEXT which match the pattern PATTERN. For more information on the possibilities list, see the Conniver Reference Manual [16].

(IF-ADDED ATOM PATTERN BODY)

This function defines a method with name ATOM whose body BODY will be invoked if any data item which matches pattern is added to a context in which the method is defined.

(IF-NEEDED ATOM PATTERN BODY)

This function defines a method with name ATOM whose body BODY will be invoked if any fetch is done which matches pattern in a context in which the method is defined.

(IF-REMOVED ATOM PATTERN BODY)

This function defines a method with name ATOM whose body BODY will be invoked if any data item which matches pattern is removed from a context in which the method is defined.

The above three methods may be added to a context by use of the ADD function described above, by either adding the name of the method or by adding the method itself.