ＴＲ－Ｃ－０１２４

Knowledge Representation and Acquisition for 3-D shape Ontologies

ホァキン　デルガド　　　ジュリ　ティヘリノ
Joaquin DELGADO　　　Yuri TIJERINO

１９９５　９．２２

ＡＴＲ通信システム研究所

Knowledge Representation and Acquisition for 3-D shape Ontologies

*Joaquin DELGADO and **Yuri TIJERINO

*Nagoya Institute of Technology
Gokiso-cho, Showa-ku, Nagoya 466 Japan
E-mail:jdelgado@egg.ics.nitech.ac.jp

**ATR Communication Systems Research Laboratories
2-2 Hikaridai, Seikacho, Sorakugun, Kyoto 619-02 Japan

Abstract: In order to build more *intuitive* virtual environments, there is a need for the computer to understand the meaning of virtual objects in a way that reflects common sense knowledge of such objects. This report introduces several mechanisms for building a 3-D shape ontology that eventually would add such common sense knowledge to the What You Say is What You See (WYSWYS) framework [1]. Principally, this can be achieved through ontologies of primitive 3-D shapes and of combination of such shapes into more complex shapes. The representation used for expressing 3-D shapes is accomplished through a two-level ontology approach, that is a knowledge level and a symbol level, where the former is a representation closest that of humans while the later is a high level computer representation in the form of implicit parametric functions. Under the same approach, this report proposes a more detailed ontology hierarchy, mainly concentrating on the evaluation of formal methods for the edition and construction of the symbol level ontology using *Ontolingua* [2], and in knowledge acquisition through knowledge reuse and machine learning techniques, for building the knowledge level ontology. It presents a modification of the case based reasoning (CBR) algorithm PROTOS [3] for this last purpose. To the end, this report also gives guidelines for future implementation and further research topics.

1. Introduction

Communication of mental images in an intuitive way is a very important goal to achieve in human-to-human and human-to-machine interaction. In computer supported cooperative workspace (CSCW) [4] and more recently in the Virtual Space Teleconferencing System --VSTS [5], being developed at ATR, the need for integration of language and gestures for the manipulation and modification of 3-D shapes led to the WYSWYS framework, where a new communication paradigm was proposed for such virtual environments. One of the most important features of this proposal was to point out the existence of a two-level ontology for bridging the knowledge gap between human and computers. To clarify what the word ontology is, in philosophy it means the study of existence, but in artificial intelligence circles it usually means the set of most primitive terms or concepts that describes something. Applying it to the task of adding visual knowledge to the computer, we can use ontologies to get the machine to understand the meaning of virtual objects in such way that reflects common sense knowledge of such objects. This approach for 3-D shape representation consists of a symbol level and a knowledge level, where the former is a representation closest to that of humans while the later is a high level computer representation in the form of implicit parametric functions that capture the meaning of a shape based on the interpretation of a set of parameters and their values. Common sense about complex 3-D shapes can be expressed through these ontologies of primitive 3-D shapes and of combination of such shapes into more complex shapes at the knowledge level. Indeed, it has been suggested that easy decomposition into parts is the characteristic of basic level categories that determines their high degree of distinctiveness and informativeness [6]. Going further on that line of research this report proposes a more detailed ontology hierarchy and actually describes how each part of this hierarchy can be made.

One of the main problems in building any ontology is to choose the most appropriate tool for creating and editing it, looking towards portability and standards in knowledge representation and current technology. For that purpose, an ontology editor tool from the

Knowledge Sharing Lab. at Stanford University, called *Ontolingua*, has been evaluated by translating the existing rule base system of *shapes* and *adjectives* as part of ATR's most recent VSTS prototype [7], into a formal symbol level ontology in the editor. That implied the creation and design of such ontology that did not exist before.

On the other hand, knowledge acquisition about 3-D shapes, can be achieved either by the reuse of previously built high level ontologies or by machine learning techniques. In that sense the case-based knowledge acquisition algorithm PROTOS was first evaluated and later modified in terms of more adequate learning heuristics for the 3-D shape domain, resulting in a modified PROTOS system. All this in order to build systems that can acquire knowledge and be able to learn, under user guidance, about the complex 3-D shapes that the user builds himself for teaching the computer.

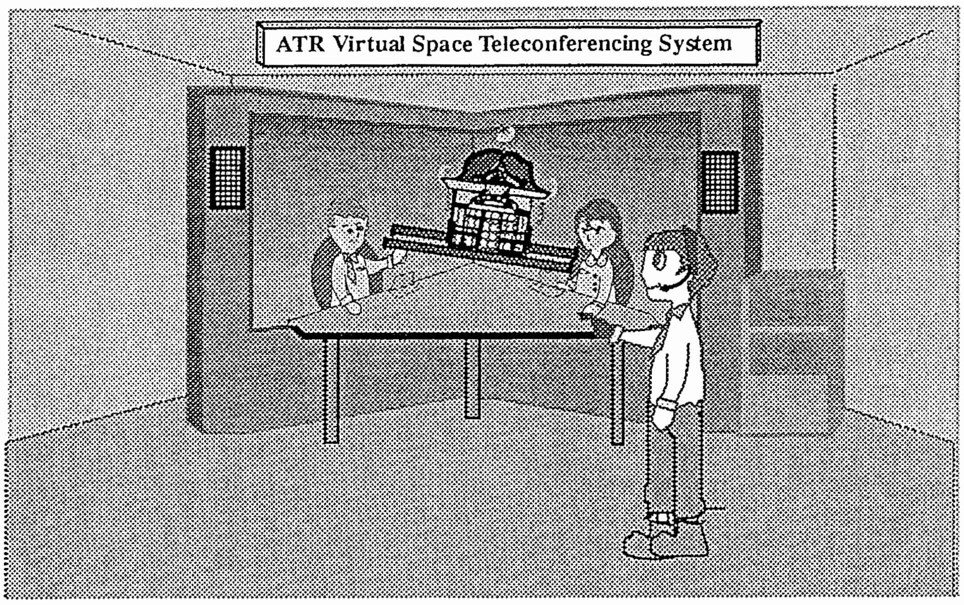

Fig.1 A virtual teleconferencing room. We want to be able to interact with the computer in a more intuitive and intelligent manner.

This report is organized in the following manner. Section 2 gives an overview of the ontology hierarchy, showing the components or sub-ontologies needed to define well the symbol level and knowledge level ontologies. Section 3 actually shows how the symbol level ontology was designed and partially built using Ontolingua. Section 4 introduces the knowledge level ontology for complex shapes and also gives a detailed example of how this ontology can be incrementally built through a knowledge acquisition tool specifically modified for this purpose. Section 5, discusses further possibilities of enriching this ontology with component-relation and shape modification ontologies and how the two level ontology approach allows knowledge reuse. Several appendixes were added, including implementation guidelines for a translator between Ontolingua and Nextpert (the current knowledge base system), and the interface between the knowledge level and system level.

2. Ontology Hierarchy

As shown in Figure 2, The knowledge representation hierarchy consists of three levels: the machine level representation, the symbol level ontology and the knowledge level ontology. While the machine level is not an ontology, and therefore left out of the scope of this report, emphasis will be done in describing both the symbol level and the knowledge level ontologies more in detail, with the recent advances and difficulties in constructing such ontologies.

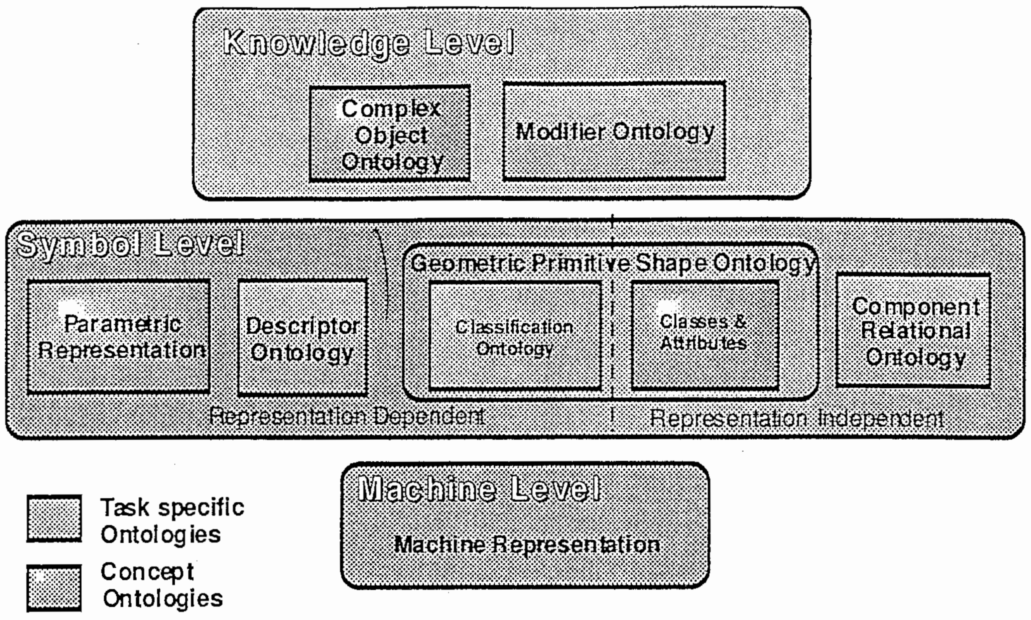

Figure 2. The conceptualized ontology hierarchy

In the figure, *Concept Ontologies* refers to Gruber's [2] definition of an ontology, in other words it refers to a set of definitions of content-specific knowledge representation primitives: classes, relations, functions, and object constants. On the other hand *Task Ontologies*, refers, according to [8], to a vocabulary of concepts for describing the problem solving structure of all the existing tasks in a domain-independent manner.

3. Symbol Level Ontology Design using Ontolingua

In the 3-D shape ontology context a symbol level ontology is the one most proximate to the low-level computer representation of 3-D shapes. Definitions for the parametric representation, chosen for the 3-D shapes, descriptors related with the parameters and basic 3-D primitive shapes such as sphere, cone, cube, etc..., and their representation independent features, are stated in this ontology. Therefore, it is more less a static ontology in the sense that once it is defined, no learning process is needed for further knowledge acquisition. Furthermore the general conception (idea) of an object is not only related to its basic form or shape but also to the way this complex object is organized from other primitives, this is the component-relation ontology, that also belongs to the symbol level, and which formalizes relations between 3-D primitive shapes in order to complete the definition of complex 3-D shapes. It is left for the discussion section.

The existing prototype system for the symbol level ontology had integrated a visualization module with a rule based system for achieving the classification of 3-D primitive shapes. This system was based only on rules and not on a formal specification of an ontology. One of the goals was to create a formal specification of the ontologies listed in the ontology hierarchy. The first problem was how to build such ontologies?

For this purpose we decided to evaluate Ontolingua.

We chose to evaluate Ontolingua, an on-line network service of the Knowledge Sharing Laboratory from Stanford University, for editing and constructing portable ontologies (**), basically because of:

1) Its a tool that gives a methodology and facilitates building complex ontologies in a cooperative way. It also checks consistencies while supporting ontology creation or definition.

Ontolingua: Ontology Editor and Data Base

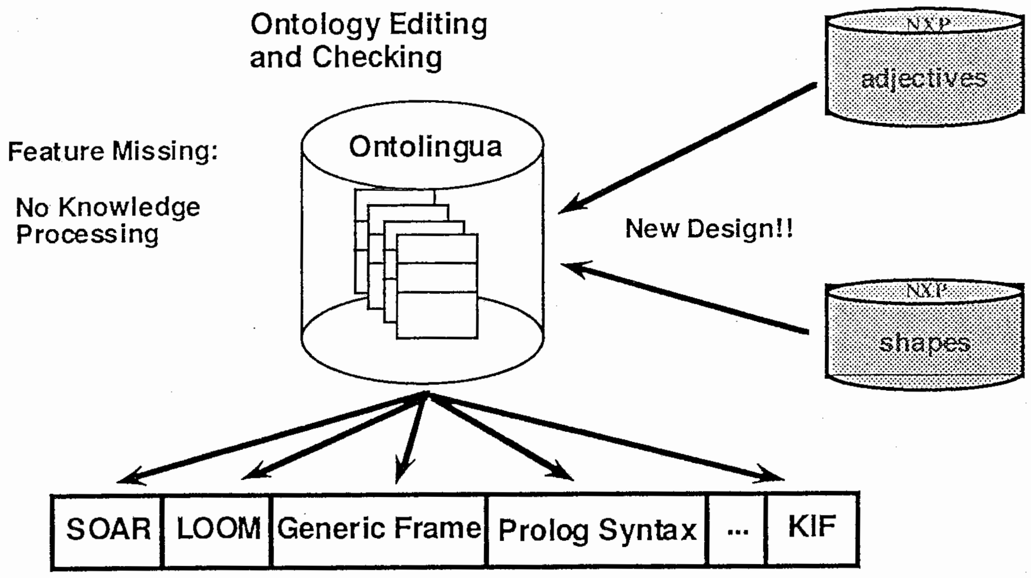

Figure 3. The *Ontolingua* system: an On-line network service of the Knowledge Sharing Laboratory from Stanford University

2) Allows portability, knowledge reuse and sharing to some extent. It offers a wide range of existing ontologies (ontology library) which can be integrated into our ontology

3) It is compatible with current agent processing technology. The future need for a distributed knowledge processing platform entails us to assume a multi-agent perspective for further research, in which knowledge sharing between agents will be essential

As shown in figure 3, the translation of the existing two level rule base **adjectives** and **shapes**, from its original source written as rules in Nextpert Object, an expert system shell, into Ontolingua was done as the first step (see apendix A). Although Ontolingua does not support Knowledge Processing --it wasn't intend to do so, it does support automated translation to several Knowledge Based System's shells and knowledge representation languages such as SOAR, LOOM, Prolog Syntax, Knowledge Inteerchang Format (KIF) etc... The next step is to build a translator between and Nextpert Object for which guidelines are given in section 6.2. Anyway, the actual design of the ontologies were also reflected in the existing system and new definitions and changes were also done in Nexpert Object because of actual system requirements.

In the next two subsections, there is a partial description of how the symbol level ontology was designed and defined using Ontolingua. For more details, the reader can refer to the Appendix A that contains a full description of the ontologies built in Ontolingua.

(**) http://www-ksl-svc.stanford.edu:5915/

3.1. The 3D-Parametric-Models

This ontology is intended to be for the conceptualizing of parametric 3-D solid models for computer graphics. Therefore it can contain *any class of mathematical model for representing 3-D shapes*. Particularly we have defined one class **Sq-Parametric-Model** that stands for the candidate representation called *superquadrics* recognized as a powerful representation for intuitively building 3-D models in [9]-[11] and [16]. Supercuadrics are defined by the flowing vector:

$$x = f(z)\cos^{\varepsilon 1}\alpha(\text{cose}2\omega + a4)$$
$$y = g(z)\sin^{\varepsilon 1}\alpha(\text{cose}2w + a4)$$
$$z = a3\sin^{\varepsilon 2}\omega$$

where, $f(z) = a1 * (1 - k1 * x / a3)$; and
$g(z) = a2 * (1 - k2 * z / a3)$

In Ontolingua we have classes, relations, functions, and object constants. Here is an example of the definition of the **Sq-Parametric-Model.**

```
;;; Sq-Parametric-Model

(Define-Frame Sq-Parametric-Model
              :Own-Slots
              ((Arity 1)
               (Documentation
                 "This is a 3d solid math model based on a parameter space
                 named superquadrics")
               (Instance-Of Class) (Subclass-Of Thing))
              :Template-Slots
              ((A1 1 (Slot-Value-Type Real-Number))
               (A2 1 (Slot-Value-Type Real-Number))
               (A3 1 (Slot-Value-Type Real-Number))
               (A4 0 (Slot-Value-Type Real-Number))
               (E1 1 (Slot-Value-Type Positive))
               (E2 1 (Slot-Value-Type Positive))
               (K1 0 (Slot-Value-Type Real-Number))
               (K2 0 (Slot-Value-Type Real-Number)))
              :Issues
              ((:Source
                "The definition of the superquadrics model is
                 given by Hirikoshi, as follows:

                        x = f(z) cos(alpha)^E1 *(cos(omega)^E2 + A4);
                        y = g(z) sin(alpha)^E1 *(cos(omega)^E2 + A4);
                        z = A3 * sin(omega)^E2;

                 where,        f(z) = A1 * (1 - K1 * x / A3); and
                               g(z) = A2 * (1 - K2 * z / A3)")))
```

No relations were defined but 25 functions defined as follows:

- A1, A2, A3, A4, K1, K2 as function that returns the *values* of the parameters of the superquadrics.

-Descriptor functions, that represents the Descriptor Ontology:

Descriptor	Formal representation
Flat	$a_i < min(a_j, a_k)/3; i \neq j \neq k$
Holed (top-bottom)	$a_4 > 1.0$
Long	$\lvert a_i - a_j \rvert < \Delta a; max(a_i, a_j) < a_k < min(2a_i, 2a_j); i \neq j \neq k$
Round latitude	$\lvert e_2 - 1.0 \rvert < \Delta_e$
Round longitude	$\lvert e_1 - 1.0 \rvert < \Delta_e$
Short	$\lvert a_i - aj \rvert < \Delta a; a_k < min(a_i, a_j); i \neq j \neq k$
Similar scale parameters	$\lvert e_2 - 1.0 \rvert < \Delta_e; i \neq j$
Squared latitude	$\lvert e_2 \rvert < \Delta_e$
Squared longitude	$\lvert e_2 \rvert < \Delta_e$
Tall	$\lvert a_i - a_j \rvert < \Delta a; max(a_1, a_2) < a_3 < min(2a_1, 2a_2)$
Tapered top	$0 < k_2 \leq 1$ or $0 < k_1 \leq 1$
Very long	$\lvert a_i - a_j \rvert < \Delta a; a_i \geq 2a_k, i \neq j \neq k; i, j = 1, 2$
Very tall	$\lvert a_i - a_j \rvert < \Delta a; a3 \geq min(2a_1, 2a_2)$

Table 1. Descriptor ontology, where i, j, k = 1, 2, 3; $\Delta e = 0.3$; $\Delta a = 1/30 \sum_{m=1}^{3} a_m$

Here is a list of all the **boolean functions** that are axioms of the class Sq-Parametric-Model as shown it Table 1:

Has-Hole	Has-Round-Latitude
Has-Round-Longitude	Has-Sharp-Top
Has-Sharpened-Side-1	Has-Sharpened-Side-2
Has-Similar-Scale-Parameters	Has-Square-Latitude
Has-Square-Longitude	Is-Flat
Is-Long	Is-Short
Is-Tall	Is-Tapered-Top-E-W
Is-Tapered-Top-N-S	Is-Very-Long
Is-Very-Tall	

Even if these functions are part of the 3-D-Parametric-Models ontology they actually are the Descriptor Ontology (a task ontology) shown in the ontology hierarchy. It also important to point out, due to impact on higher level ontology, that these functions are not strictly defined in terms of the parameters but they also include ranges of tolerance that later allows approximation in formal shape descriptions.

3.2. 3-D-Shapes

This ontology is for defining 3-D Shapes either simple primitives of complex shapes, independent of their graphical representation. It gives semantics to graphical objects than can be part of a system.

3d-Shapes includes the following ontologies

3d-Parametric-Models	/* Representation Dependent */
Component-Assemblies	/* Included as part of an integration experiment

Class hierarchy (15 classes defined):

* 3d-Geometric-Primitive

Cone	Cube	Pyramid	Shpere
Cylinder	Cylindrical-Rod	Rectangular-Rod	Unknown
Cylindrical-Tube	Disk	Rectangular-Sheet	
Hoop	Ingot	Rectangular-Tube	

* 3d-Complex-Shape /* To be used further at the knowledge level */

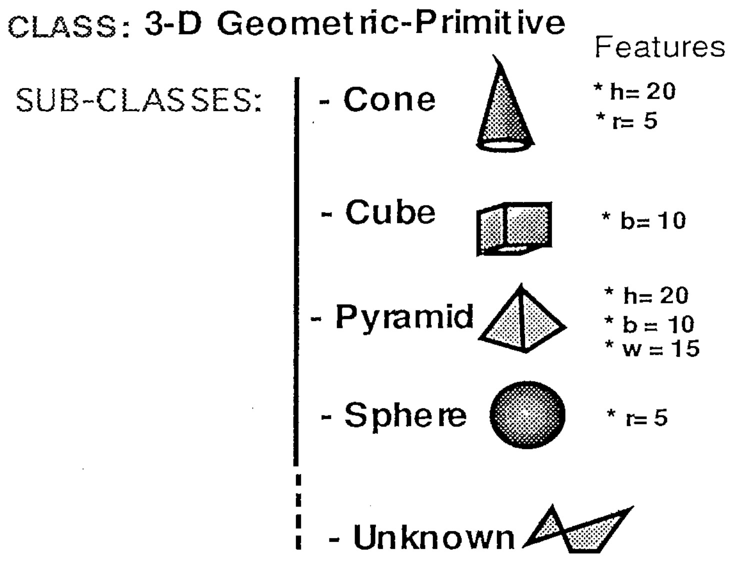

Figure 4. 3-D PrimitiveShape ontology. Based on the so called "Platonic Shapes" and some other basic topological shapes.

It is important to point out that intrinsic attributes of simple shapes, such as the **radio** of a Sphere, is declared in this ontology as independent from the geometric representation but its calculated and extracted from the parameters of the graphical representation.

Axioms of this ontology follows the rules stated in the following table:

● Condition holds true
— Implicit false or non-applicable
▨ Condition holds false

Some primitive descriptors \ Some primitive shapes	Cone	Cube	Cylinder	Cylindrical rod	Cylindrical tube	Disk	Hoop	Ingot	Pyramid	Rectangular rod	Rectangular sheet	Rectangular tube	Sphere
Flat	—	—	—	—	—	—	—	—	—	—	●	—	—
Holed (top-bottom)	▨	▨	▨	▨	●	▨	●	▨	▨	▨	▨	●	▨
Long	—	—	—	—	—	—	—	●	—	—	—	—	—
Round latitude	●	—	●	●	●	●	●	—	—	—	—	—	●
Round longitude	—	—	—	—	—	—	—	—	—	—	—	—	●
Short	—	—	—	—	—	●	●	—	—	—	—	—	—
Similar scale parameters	—	●	—	—	—	—	—	—	—	—	—	—	●
Squared latitude	—	●	—	—	—	—	—	●	●	●	●	●	—
Squared longitude	●	●	●	●	●	●	●	●	●	●	●	●	—
Tall	●	—	●	—	—	—	—	—	●	—	—	—	—
Tapered top	●	▨	▨	▨	▨	▨	▨	▨	●	▨	▨	▨	▨
Very long	—	—	—	—	—	—	—	—	—	●	—	—	—
Very tall	—	—	—	●	●	—	—	—	—	—	—	●	—

Table 2. Basic 3-D primitive shape ontology based on descriptors

For the *classification* of a primitive 3-D shape (an object of the class Sq-Parametric-Model) the boolean functions for the descriptors are used. Note that through out these functions independence of the underlying geometric representation is achieved. This ontology of shapes is not intended to be exhaustive by any means. On the contrary they are given with the intention of allowing further definition of other shapes.

We have then created, in Ontolingua, two extendable ontologies: the *Sq-Parametric-Model* ontology and the *3d-Shape* ontology. The former one used for the shape's parametric representation and their description through dimensional adjectives, and the later corresponds to the geometric primitive shape ontology, also described in the ontology hierarchy (figure 2). These are the results of formalizing and re-designing the preexistent two-level rule based system (**adjectives** and **shapes**) into well defined ontologies. We have also achieved through them graphical representation independence, being able to build the next level of the knowledge representation with primitive 3-D shapes and some of their known attributes, as addressed in visual knowledge common sense, without having to deal with their underlying graphical representation.

4. Knowledge Level Ontology

The knowledge level is the closest to that of human representation of 3-D shapes. Based on primitive shapes as components and relations between these components (symbol level ontology), we can think of an intuitive way of building more complex 3-D shapes as meaningful objects in terms of the common sense that this shape reflects. The final goal is to apply this constructive paradigm to as may domains as possible. This immediately recalls the need for the computer to learn new concepts. One of the limitations of the formalization described in previous sections is that it requires much effort to formalize all sort of common sense shapes. For this reason it becomes necessary to automate or semi-automate the formalization process. Basically this can be achieved either by machine learning techniques or by knowledge sharing and reuse. While a machine learning algorithm, named *Protos*, was evaluated and modified for suiting our problem, knowledge sharing and reuse is left as further discussion. We also give an example of how a complex object ontology can be created using the modified Protos algorithm. The shape modification ontology that will enable shape deformation in terms of conceptualized transformations, using language and gestures, is also in the discussion section.

4.1. Knowledge Acquisition

To employ case based reasoning (CBR) for interactive on-line learning new complex shapes seemed adequate and was demonstrated by evaluating and modifying of Protos algorithm. The off-line learning by data presented to the computer, although not discussed in this report, can be achieved by using ID3 [12], Explanation-Based Learning (EBL) [13] or any other generalization-based method, but is has been shown [14] that these methods are not likely to perform as well as exemplar-based methods in domains with weak theories. We have chosen exemplar-based methods which are a specialization of case based reasoning, because we are dealing with weak theories.

4.1.1 Protos Algorithm and Modification

Protos is a machine learning program which acquires knowledge for performing heuristic classifications. The program learns by the by-product of performing classification under guidance of a human teacher. When presented with a case to be classified, Protos classifies the case by recalling an appropriate previous case and explaining its similarity to the new case. If classification fails, the teacher is asked to supply an explanation. By learning from explanations in addition to examples, Protos is able to classify from little training and to justify its conclusions. It is considered a *case-based* or *exemplar-based* classification system -- it takes input in the form of a new *case* and then tries to find a similar past case that was retained as an *exemplar* of a particular category/concept. Its output is the category (i.e., the classification of the case), accompanied by the specific exemplar and a set of explanations of how the features of the case are similar to the features of the exemplar. Here is the general Protos algorithm :

```
---------------------------------------------------------
Given:
   a set of exemplar-based categories C = {c_1,c_2,...,c_n}
   and a case (NewCase) to classify.

REPEAT
   Classify:
     FIND an exemplar of c_j belonging to C that STRONGLY MATCHES
     NewCase and classify NewCase as c_j.
        EXPLAIN the classification.

   Learn:
     IF the expert disagrees with the classification of explanation
          THEN
               acquire classification and explanation knowledge
               and ADJUST C so that NewCase is correctly classified
               and explained.
UNTIL the expert approves the classification and explanation.
-------------------------------------------------------------
```

For explaining exactly where the modification of the algorithm occurred a more detailed explanation about Protos and how it works follows .

Protos learns about four things -- features, exemplars, categories, and relations -- from the teacher (Protos also autonomously learns indexing and featural importance). Let's examine each of these.

* *Features* are terms used to describe a case. For example, I could describe a the case "1994 western house" with the following features: roof, walls, chimney, TV_antenna, (doors 2), garage, garden and (windows 4). Obviously, a feature can be in the form of a proposition or a predicate with arguments.

* An *exemplar* is a case which Protos has processed and retained. An exemplar is an instance of a concept. For example, the exemplar "1994 Western House" is an instance of the concept "house".

* A *category* is a collection of instances of a concept. The category "house" might consist of the exemplars "1994 Western House", "1889 Japanese House", and "1966 Muscovite House".

**Relations* are learned from *explanations*. For example, Protos will ask what the relevance of "door" is to "house". The teacher might respond with the explanation "door has function entrance is part of house". From this, Protos installs two relations in its knowledge base: "door has function entrance" and "entrance is part of house".

The efficient use of exemplars during classification requires that they be indexed so that they can be efficiently retrieved when they are likely to be similar to a new case. Now we should describe how Protos learns four type of indices -- remindings, censors, prototypicality, and difference links.

1) *Remindings* are initially learned from feature-to-category explanations given by the teacher when a new exemplar is added to the category network. For example, when the teacher explains that *entrance is part of house* Protos learns a reminding from *entrance* to *house* The strength of a reminding estimates *p(C/f)*, the probability of the classification given the feature. This strength comes from the strength of the explanation.

2) *Censors* are negative remindings, i.e., an index to the category network saying that the presence of a certain feature tends to rule out a given classification.

3) The *prototypicality* of an exemplar is a measure of how typical the exemplar is of the category. Every time that an exemplar is successfully used in matching a new case, its prototypicality is automatically increased.

4) Protos learns *difference links* as a result of a "near miss" matches during classification. It connects two exemplars (in the same or different categories) and indicates important features which discriminate between the two.

Having mentioned the important concepts, now a more detailed Protos algorithm for using and learning indexing knowledge will be presented, in order to focalize the modified part.

```
GIVEN: a case (NewCase) to classify.

To find an exemplar matching NewCase:

1     Collect remindings from New Case's features to categories.
2     Combine remindings to related categories (**)
3     Retain the N categories with the strongest combined remindings.
4     Select, in order of prototypicality, several exemplars of
          each category.
5     Collect remindings from NewCase's features to exemplars,
          and add these to list of       exemplars. Order the
          list by reminding strength.
      REPEAT (consider the exemplars in decreasing order)
6         Let Exemplar1 be the exemplar with the next highest
              reminding strength.
7         Determine the similarity of NewCase and Exemplar1.
      UNTIL a sufficiently strong match is found.
8     Use exemplar differences from Exemplar1 to locate a better
          match (Exemplar2).
```

Exemplar2's category is used to classify *NewCase* and the match between *Exemplar2* and *NewCase* is used to explain this classification. Indexing knowledge is acquired by discussing this classification and explanation with the expert as follows.

```
      IF the expert rejects the classification or explanation
9     THEN Reassess the remindings  from NewCase's features.
      ELSE (the expert accepts the classification and explanation)
10        Increase protopypicality ofExemplar2.
          IF NewCase is retained as an exemplar
11        THEN Learn remindings for NewCase. (*)
          IF NewCase was initially classified or explained
          incorrectly
12        THEN record exemplar differences.
```

Steps 1 through 8 describes how Protos uses indexing knowledge to find an exemplar matching a given case. Steps 9 through 12 describes how Protos acquires indexing knowledge. The point that was directly modified was **step 11** (*), i.e.., the core process of learning remindings. This affects the overall learning process and it is reflected mostly in step 2 (**) in the matching part of the algorithm.

The major modification was due to the fact that in the process of learning a reminding by compiling an expert-supplied explanation of a case feature, Protos heuristically analyzes each explanation to *determine the category or exemplar to which the reminding should refer* and the strength of the reminding. That means that it chooses only one category of the network to which refer the reminding. Although this simplifies the algorithm it leaves out the possibility of *propagating the reminding though the category network* that is an important aspect to include in order to accelerate learning and minimize misses. This is clearly evident in part-whole and generalization-specialization relations that are much of our concern. Say for example, that the following expanation is presented to Protos:

prism suggests *roof* witch is part of *house*

Protos would then prompt asking to which one and only one of the categories involved (*roof* or *house*) should be referred the reminding from the feature prism. We have found that assigning the reminding to only one category produces inconsistencies and many errors at the hypothesis formation time for classification since the hypothetical categories are only drawn based on the reminders. If we choose only to refer the reminding from *prism* to *roof* then if later, we present a more complex case, say a *prism* and a *cube*, it would be considered as a *roof* without even listing among the categories candidates for classification, the correct one, i.e., a *house*.

Therefore the modification consisted in the propagation of remindings through the category network creating remindings not only to the main category target but also to its relatives (inferred from part-whole and generalization -specialization relations). The strength of the reminders are weakened towards more higher hierarchies. In case of the explanation given as example, a *strong* reminder will be created between *prism* and *roof*, but also a less stronger reminder will be established between *prism* and the category *house* enabling a more accurate list of hypothesis for faster and better learning.

4..2. Complex Object Ontology through an example

House ontology obtained by using Protos.

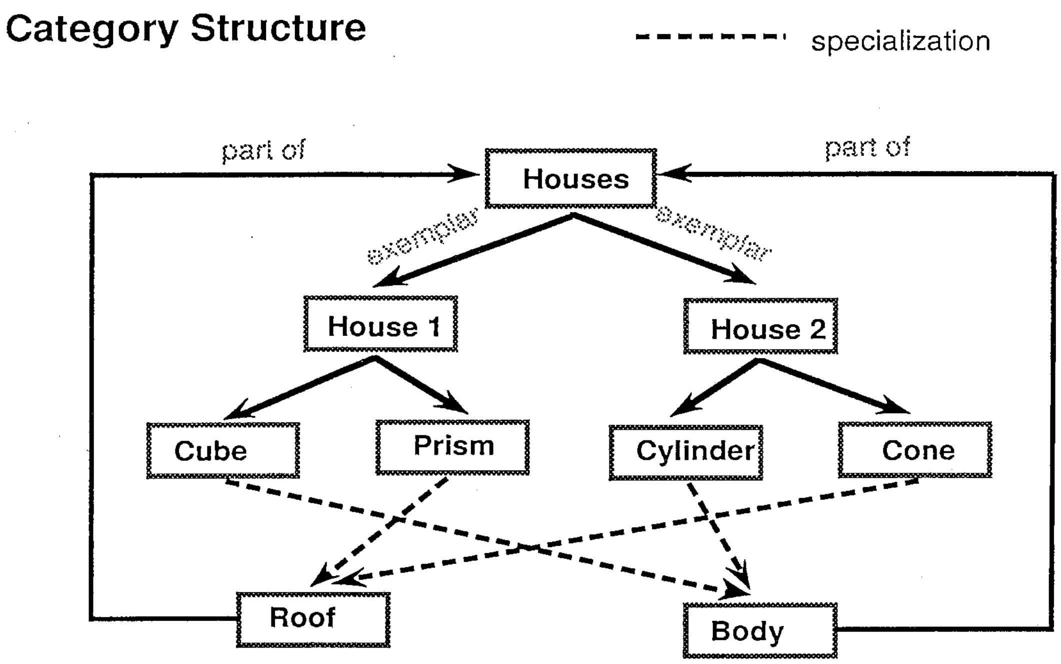

Figure 4. Partial description of the category network obtained when building a simple house ontology with modified Protos.

We have built an experimental simple house ontology (only considering the part-whole relations of the complex objects), with the help of the modified Protos program. We now present part of the outputted knowledge base that can serve as basis for constructing the formal ontology. It was generated in Common Lisp and, once the features and learningmechanism of Protos is understood, its comprehension is straight forward.

```
(term :name         house
      :importances ((story_2 0.70) (rectangular_sheet 0.34)
                        (semi-cylinder 0.34)
                        (square_base_pyramid 0.34) (cylinder 0.34) (cone 0.34)
                        (cube 0.95) (prism 0.78) (roof 0.70)
                        (walls 0.70))
       :exemplars      (proto_house humble_house dome_house indian_house
                        small_house two_story_house)
       :relations      (((house has part roof) "Joaquin" "A roof is part
                          of a house")
                        ((house has function (shelter person)) "Joaquin"
                     "A house has the function of shelter for persons")
                        ((house has part walls) "Joaquin" "walls are part of
                          a house ")))

(term  :name           proto_house
       :comment        "A general abstract house "
       :category       house
       :features       (walls roof)
       :typicality     6.50)

(term  :name           walls
       :remindings     ((house 0.67))
       :importances    ((cylinder 0.48) (cube 0.48))
       :exemplars      (walls1 walls2)
       :relations      (((walls equivalent story_2) "joaquin"
                     "Story 2 are walls of the second floor")
                        ((walls equivalent story_1) "joaquin"
                     "Story_1 are the walls of the first floor")
                        ((walls is part of house) "Joaquin" "Walls are
                          part of a house ")
                        ((walls sometimes is inferred from cube) "Joaquin"
                     "A cube suggest a house's base because of it's form.")
                        ((walls sometimes is inferred from cylinder) "Joaquin"
                     "A cylinder suggest a base of a house because of
                      it's form.")))

(term  :name           roof
       :remindings     ((house 0.70))
       :importances    ((triangle_base_pyramid 0.48)
                        (rectangular_sheet 0.48)
                        (semi-cylinder 0.48)(square_base_pyramid 0.48)
                        (cone 0.48) (prism 0.48))
       :exemplars      (proto_roof roof5 roof2 roof1 roof3 roof4)
       :relations      (((roof is part of house) "Joaquin" "A roof is part
                      of a house")
                        ((roof sometimes is inferred from prism) "Joaquin"
                     "A prism always suggest a roof because of its form")
                        ((roof sometimes is inferred from cone) "Joaquin"
                     "Cone sometimes suggest roof because of it's form")
                        ((roof sometimes is inferred from square_base_pyramid)
                      "Joaquin"
                     "A square base pyramid suggest roof because of it's form")
                        ((roof sometimes is inferred from semi-cylinder)
                      "Joaquin"
                     "A semi cylinder suggest roof because of it's form")
                        ((roof sometimes is inferred from rectangular_sheet)
                      "Joaquin"
                     "A rectangular sheet suggest roof because it's form")
                        ((roof sometimes is inferred from
                      triangle_base_pyramid) "Joaquin"
                     "A triangle base pyramid suggest roof because of
                      it's form"))) ...
```

Even if a generic house (*proto_house*) is considered to be composed of **roof** and **walls**, we have two more shape dependent exemplars, the *small_house* and the *indian_house*. While the *small_house* is composed if a **prism** and a **cube** (as prism is an exemplar of roof and cube an exemplar of walls), the *indian_house* or *hut* (*) is composed of a **cylinder** and a **cone** (in the same way exemplars of roof and walls) .

(*) Protos also allows the definitions of *synonyms* of terms, that is very convenient as we may need to refer to the same term with different names, for example, a *cube* could also be called a *box*..

In this section we have discussed why is it necessary to automate the knowledge acquisition process of formalizing 3-D common sense shapes, and we have also shown the modifications made to a machine learning program called *Protos* for such purpose. In addition, a simple house ontology is described as a experimental result of using this modified program for learning about combinations of primitive shapes than can somehow, describe the shape of a house.

5. Discussion

5.1. Component-Relation Ontology

According to Lang's proposal [15]:

a) Knowledge of shape is object-centered and based on axes of symmetry.

b) Knowledge of orientation is based on the relation of the object-centered axial system to the axes of local space.

c) The axes of local space consists of the vertical axis (determined by gravity) and the observer's line of sight.

Now merged with these concepts, first we had adopted the "Position-Orientation" theory for representing the relations between two primitives, at the geometric representation level, as two pairs of vectors, where each pair represents the general position and the orientation of a primitive component in space, in respect to the local space coordinates. The local space of the complex object is defined as for the bounding box that covers all the components. This although quite straightforward was not sufficient for building a case of a complex 3-D shapes for learning purpose.

Based on this and other analyses, we have reached the following conclusions:

- Definitions of complex objects has to be user's view point dependent.

- Definitions of complex objects, within a range, should be scale independent. For example, no matter if it's big or small, a shape of house should be identified as a house.

- A hierarchy of objects (components) has to be drawn and relations between them should be stated in a logic predicate form. This enables intelligent processing and learning at the knowledge level. Numerical constraints that truly holds the relations expressed through the position and orientation of each component, are verified only at the lowest level and only when it's needed.

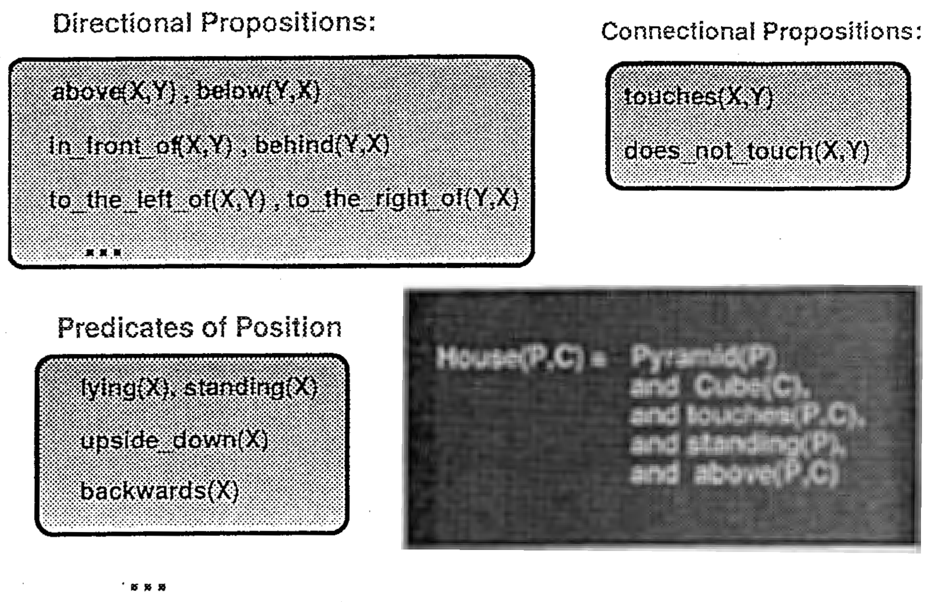

Figure 5. Partial description of a proposed Component-Relation Ontology

One of the problems that arises with the model of component-relation ontology, shown in figure 5, is to define well the "touches" or "connected_with" relations between primitive shapes. Bearing the predicates of position for each component and the directional propositions that establishes spatial relations between the components, where and how do two shapes touch can be important information when learning about complex shapes.

At this first glance we could use elements of the primitive shapes , such as faces, edges, vertices and other properties such as radius, height, etc..., that can be useful for establishing such relations and can be defined in the 3d-Shape ontology. Each geometric primitive could have "slots" corresponding to elements that belongs to them. In that sense we give the following examples:

- Pyramid:-	(Rectangular_face(base),Triangle_face(t1,t2,t3,t4), Edge(e1, e2, e3, e4, e5, e6, e7, e8), Vertex(top, v1, v2, v3, v4), Scalar(hight))
- Cube:-	(Rectangular_face(top, bottom,l eft, right, front, rear), Edge(e1,...,e12), Vertex(v1,v2,...,v8), Scalar (hight, length, width))
- Sphere:-	(Scalar(radius),Curve_surface(body))
-Cylinder:-	(Circular_face(top, bottom),Curve_surface(body), Scalar(inner_radius, outer_radius))
- Hoop:-	(Circular_face(top,bottom),Curve_surface(body), Scalar(inner_radius, outer_radius, Circular_hole(hole))

Taking this into account the exemplar of "house" in figure 5, would be redefined as:

```
House(P, C) =  Pyramid(P)
               and  Cube(C)
               and P.base "connected_to" C.top
               and standing(P)
               and  above(P,C)
```

Cases then will be represented as a group of features predicates and propositions that describe relations between these features giving a more qualitative description of the world. If quantitative descriptions become necessary they could be included in the description in form of predicates and can eventually be transformed into qualitative descriptions or left as numeric ones that can be suitable for a precise description of an exemplar.

Further research has to be done in order to build a well defined component-relation ontology but we do believe that this approach could give some insights in achieving it.

5.2. Knowledge Share & Reuse

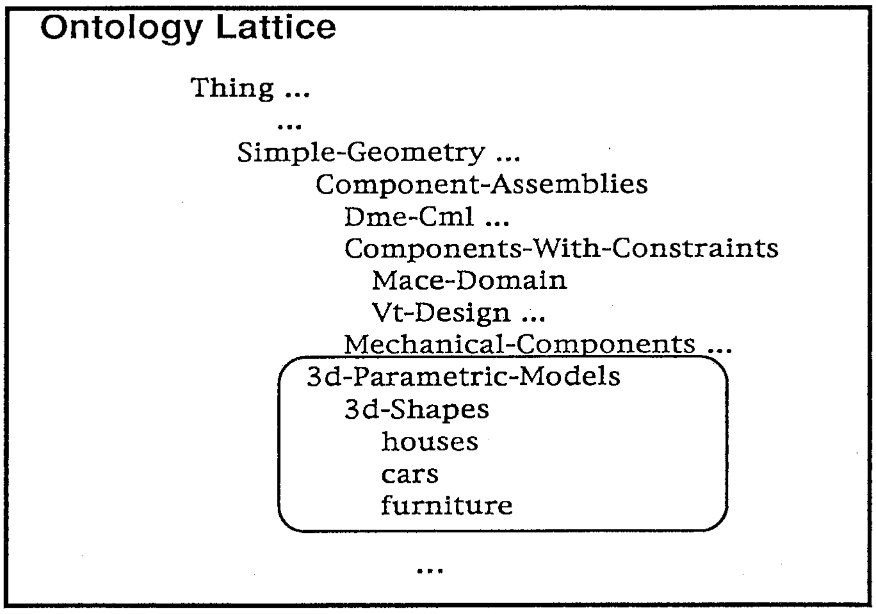

Figure 6. Ontology Lattice

As analized by Simmons in [16], this interesting question arise: what are the prospects for developing re-usable ("generic") ontologies of objects that encode knowledge of characteristic shapes for natural language processing and virtual environments? Biederman [17], estimates that there are about 3,000 identifiable and easily distinguishable basic level object categories in the cognitive and linguistic repertoire of typical adults, and that there may be as many as 10 geometric types for each category, yielding about 30,000 shapes that each of us knows about (he suspects that this estimate may be too large in about one order of magnitude). Many of these 3,000 categories overlap between individuals, but some of them correspond to idiosyncratic knowledge stemming from personal experience and expertise: an architect, for example, can distinguish a wide variety of roof shapes that all look the same to the rest of us. Leaving besides the fact that idiosyncratic categories such as house roofs may be very important for many practical AI applications; thus the size of a truly generic ontology of object shapes that would be of interest to many projects may be one order of magnitude larger than Biederman's estimate: on the order of a hundred thousand object categories.

Assembling an object taxonomy that large by hand would be a horrendous project, possibly taking as much as decades to complete. But there are numbers of ways the problem can be reduced some of them already addressed in this report. One would be to structure a generic ontology as hierarchy of useful sub-ontologies, such as a taxonomy of roof shapes. This is the top-down approach, adopted by the Cyc project [18] for building a huge "common sense" ontology. The other is to allow users to build ontologies in a more independent way, that later could be integrated, in a bottom-up fashion. For that purpose an ontology integration tool is needed, striving for commitment, called ontological commitment by Gruber [2], on the definitions of terms that want to be shared by different systems. This is the **Ontolingua**'s approach, that can also be used as a powerful tool for building ontologies in a cooperative way, even for agent systems purpose. As shown in figure 6, our ontologies *Sq-Parametric-Models* and *3-d-Shapes* could be (at least in theory) integrated in a more wide ontology lattice that Ontolingua offers as an ontology library. But even then, it would not be efficient to build a shape ontology by hand; using a learning algorithm that matches the performance of human beings, is surely a better aproach. Although not consider to match the performance of a human being, *Protos* has shown, by experiments in the medical audiology domain, to be quite accurate (98 %). The modification proposed in this report gives more power to its learning process and allows Protos to handle more efficiently "part-of" and "generalization-specialization" category hierarchies suitable to the shape ontology problem. We believe that it will not be long until we could teach the computer the shape of a house, just building one out of primitive shapes, or maybe inputting it as a picture, and say, "This is a house".

5.3. Shape Modification Ontology

In order to add knowledge to the computer that not only contains the common sense knowledge that these shapes reflect, but also shape knowledge about possible transformations of such objects, a *modification ontology* has to be created at the knowledge level. Based on adjectives (dimensional, functional, of form, ..., etc.; some of them already part of the description ontology at the symbol level) we can think of verbal derivative, such as lengthen, shorten, rounded, straighten, etc., that can serve as basis for more complex transformations of 3-D shapes. Say, for example, that we want to make a house more *modern*. This desired transformation could have several underling adjectives related to the concept modern, in the house domain, that can indicate what type of shape modification has to be performed. Say an architect defines a modern house to be *simple,* with more *horizontal* lines, *open* to the exteriors, etc. This is indeed difficult to conceptualize, but based on the **exemplar model,** we have thought of two approaches, at lower level, that might lead us to achieve the desired goal. Let's give an example for better illustrating these two approaches: Having a prototypical exemplar of an *old style Japanese house* A and other prototypical exemplar of a *modern Japanese house* B and the transformation A ~ B, meaning "make A more modern", then:

1) **The Parameter Interpolation approach**: The parameters of the geometric model (say supercuadrics) underling both shapes A and B, are interpolated resulting in another shape C, that will have both characteristics of A and B. C can be considered either a "modernized old Japanese house" or a "old fashioned modern Japanese house".

2) **The Genetic Algorithm approach**: This is similar to the previous approach but rather than interpolating the parameters, a GA algorithm can be used to produce an offspring C of the parents A and B rather more accurate in terms of inherited qualities. Different weights could be given to the genetic combination to reflect qualifications such as "more", "less", "very", 'barely", etc. in transformations such as "make A more modern".

5.4. Proposed System Architecture

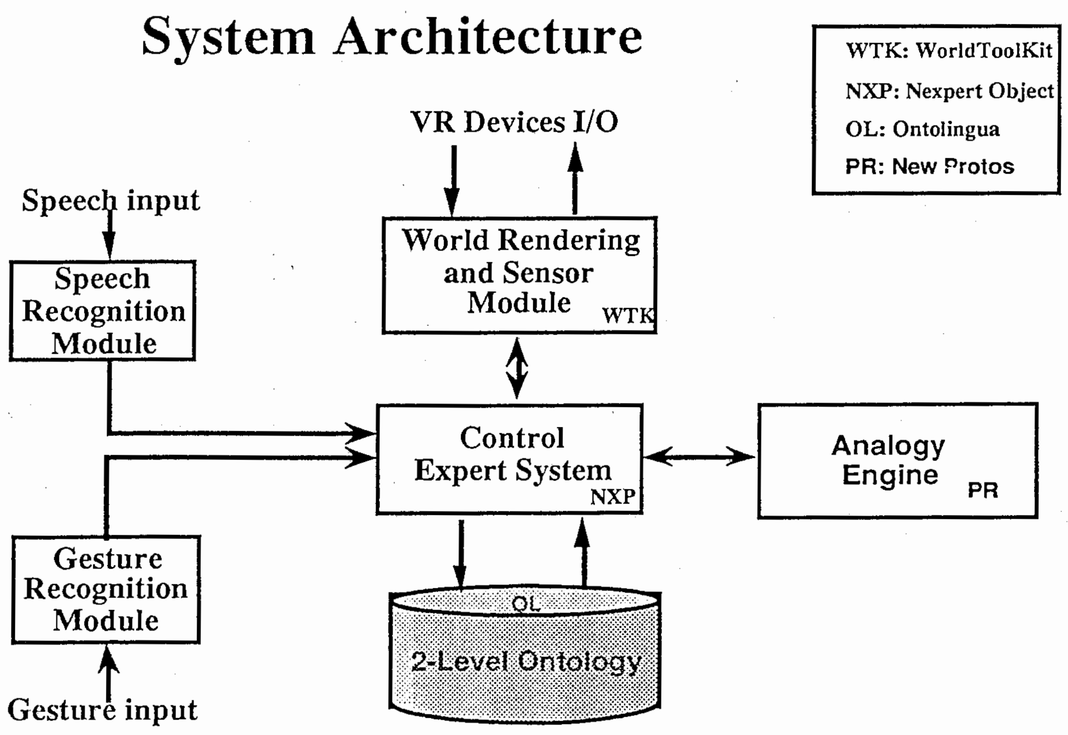

Figure 7. Proposed System Architecture

Towards an actual implementation of a prototype for the *intuitive to use* virtual environment, this new system architecture was proposed as the result of this research.

6. Concluding Remarks

* We have constructed a simple ontology of 3-D shapes and subjectively came to the conclusion that adding common sense knowledge to virtual objects is possible due to the 2-Level Ontology approach.

* We evaluated whether Case-Based Reasoning can be used to acquire such knowledge. Demonstrated with simple cases of houses that this can be done. Though further integration has to be made between the case-based reasoning system and Ontolingua.

7. Acknowledgments

We would like to thanks ATR Communication Systems Research Laboratories and all its personnel, for its support during this research. We specially thank Fumio Kishino and Professor Naohiro Ishii for allowing the realization of this project and Mr. Roberto Lopez for his contributions and critics.

References

[1] Tijerino, Y.A., Abe, S., Miyasato, T. and Kishino F. (1994) What You Say Is What You See, --Interactive Generation, Manipulation and Modification of 3-D Shapes Based on Verbal Descriptions--, *Artificial Intelligence Review Journal* Vol. 8, No. 2, 123-142

[2] Gruber, T. (1992). A translation approach to portable ontology specifications, Standford University Knowledge Systems Laboratory, Technical Report KSL 92-71

[3] Bareiss, R. (1989). *Exemplar-based knowledge acquisition: a unified approach to concept representation, classification, and learning*. Academic Press, Inc.

[4] Takemura, H. and Kishino, F 1992). Cooperative work evironment using virtual workspace, in *Proc. of CSCW'92*, pp. 226-232.

[5] Kishino, F. (1990). Communication with realistic sensations. *3-D image*, 4, 2. (In Japanese)

[6] Tverskey, B. and Hemenway, K. (1984). Objects, parts and categories. *Journal of Experimental Psycology:* General 113, pp. 169-193

[7] Tijerino, Y. A., Yoshida, M., Abe, S., and Kishino F. (1995) A shape knowledge representation scheme and its application on a multi-modal interface for a virtual space teleconferencing system. In *Proc. 4th IEEE International Workshop on Robot and Human Communication*, Tokyo, Japan, pp. 259-264.

[8] Mizoguichi, R., Tijerino, Y. A. and Ikeda, M. (1992). Two-level mediating representation for a task analysis interview system, In *Proc. of AAAI-92 Workshop for Knowledge Representation Aspects of Knowledge Acquisition.* San Jose, Ca., pp. 107-114

[9] Pentland, A. P. (1986). Perceptual organization and the representation of form, *Artifical Intelligence*, 28, pp. 293-331.

[10] Hirokoshi, T and Kasahara, H. (1990). 3-D Shape indexing language, in *Proc. of the 1990 International Conference on Computers and Communications*, pp. 493-499.

[11] Terzopoulos, D. (1991). Dynamic 3D models with local and global deformations: deformable superquadrics. *IEEE Trans. Pattern Anal. Machine Intelligence*, 13 (7), pp. 703-714.

[12] Quinlan R. (1986). Induction of decision trees, *Machine Learning*, Vol. 1, No. 1, pp. 81-106.

[13]Dejong, G. F. (1986). Explanation-based learning. In R. S. Michalsky, J. C. Carbonell, T. M. Mitchell (eds.), *Machine Learning: An artificial intelligence approach,* Vol. II. Los Altos, CA: Morgan Kaufmann.

[14] Porter, B. W., Bareiss, R. and Holte, R. C. (1990) Concept learning and heuristic classification in weak-teory domains. *Artificial Intelligence Journal*, vol. 45 (nos. 1-2), pp. 229-264.

[15] Lang, E. (1989). The Semantics of dimensional designation of spatial objects. In M. Berwisch, E. Lang (eds.), pp. 263-417.

[16] Simmons G. (1993). Towards an integrated theory of geometric knowledge in object concepts: where language and vision meet. In N. Guarino and R. Poli, Ladseb-CNR Internal Report 01/93: *Internal Workshop on Formal Ontology in Conceptual Analysis and kKowledge Representation.*

[17] Biederman, I. (1988). Aspects and Extensions of a Theory of Human Image Understanding. In Z. W. Pylyshyn (ed) *Computational processes in human vision: An interdisciplinary perspective.* Norwood, NJ: Ablex. pp. 370-428.

[18] Lenant, D. B. and R. V. Guha (1990). Cyc: Towards programs with common sense. *Comm. ACM,* 33 (8), pp. 30-49.

Appendix A

Ontologies in Ontolingua

This appendix summarizes the ontologies **3D-PARAMETRIC-MODELS** and **3D-SHAPES**, built with the help of an ontology editing tool called *Ontolingua.* The Knowledge Sharing Laboratory (KSL) at Stanford University, offers this tool as an interactive online service on the World Wide Web for cooperative creation and edition of "portable" ontologies, also being able to serve as an ontology library. Among the things it offers we can mention the automatic translations of ontologies into several knowledge representation languages and systems, and the consistency checking between its primitives for creating and editing such ontologies, - -classes, relations, functions, and object constants-- proved to be very usefull. It can be found in the WWW at http://www-ksl-svc.stanford.edu:5915/.

Ontology 3D-PARAMETRIC-MODELS

```
(In-Package "ONTOLINGUA-USER")

;;; Written by user Jdelgado from session "test" owned by group JUST-ME
;;; Date: Aug 31, 1995  19:15


(Define-Ontology
     3d-Parametric-Models
     (Frame-Ontology)
   "An ontology of 3d solid models for Computer Graphics"
   :Io-Package
   "ONTOLINGUA-USER"
   :Intern-In
   ((Kif-Meta Logconst)
    (Kif-Numbers Positive Real-Number Min < > Number Max * >= + / - Abs =<)
    (Kif-Relations Function Binary-Relation) (Kif-Sets True)))


(In-Ontology (Quote 3d-Parametric-Models))



;;; ------------------ Classes --------------

;;; Sq-Parametric-Model

(Define-Frame Sq-Parametric-Model
              :Own-Slots
              ((Arity 1)
               (Documentation
                "This is a 3d solid math model based on a parameter space")
               (Instance-Of Class) (Subclass-Of Thing))
              :Template-Slots
              ((A1 1 (Slot-Value-Type Real-Number))
               (A2 1 (Slot-Value-Type Real-Number))
               (A3 1 (Slot-Value-Type Real-Number))
               (A4 0 (Slot-Value-Type Real-Number))
```

```
                (E1 1 (Slot-Value-Type Positive))
                (E2 1 (Slot-Value-Type Positive))
                (K1 0 (Slot-Value-Type Real-Number))
                (K2 0 (Slot-Value-Type Real-Number)))
               :Issues
               ((:Source
                 "The definition of the superquadrics model is given by
Hirikoshi, as follows:

          x = f(z) cos(alpha)^E1 *(cos(omega)^E2 + A4);
          y = g(z) sin(alpha)^E1 *(cos(omega)^E2 + A4);
          z = A3 * sin(omega)^E2;

          where, f(z) = A1 * (1 - K1 * x / A3); and
                 g(z) = A2 * (1 - K2 * z / A3)")))



;;; ------------------ Relations --------------


;;; ------------------ Functions --------------

;;; A1

(Define-Function A1
                 (?Frame)
                 :->
                 ?Value
                 "Scale on the X axis"
                 :Def
                 (And (Sq-Parametric-Model ?Frame) (Number ?Value)))


;;; A2

(Define-Function A2
                 (?Frame)
                 :->
                 ?Value
                 "Scale on the X axis"
                 :Def
                 (And (Sq-Parametric-Model ?Frame) (Number ?Value))
                 :Documentation
                 "Scale on the Y axis")


;;; A3

(Define-Function A3
                 (?Frame)
                 :->
                 ?Value
                 "Scale on the X axis"
                 :Def
                 (And (Sq-Parametric-Model ?Frame) (Number ?Value))
                 :Documentation
                 "Scale on the Z axis")


;;; A4
```

```
(Define-Frame A4
               :Own-Slots
               ((Arity 2)
                (Documentation "'expansion' radius factor; >1.0 for
toroidal")
                (Domain Sq-Parametric-Model)
                (Instance-Of Binary-Relation Function) (Range Number)))


;;; E1

(Define-Frame E1
               :Own-Slots
               ((Arity 2)
                (Documentation "east/west deformation exponential
(positive)")
                (Domain Sq-Parametric-Model)
                (Instance-Of Binary-Relation Function) (Range Number))
               :Issues
               ((:Example "E1 < 1.0 makes square corners;
E1 = 1.0 makes smooth circles;
E1 > 1.0 makes hyperbolics slopes")))


;;; E2

(Define-Frame E2
               :Own-Slots
               ((Arity 2)
                (Documentation
                 "north/south deformation exponential (positive)")
                (Domain Sq-Parametric-Model)
                (Instance-Of Binary-Relation Function) (Range Number)))


;;; Has-Hole

(Define-Frame Has-Hole
               :Own-Slots
               ((Arity 2)
                (Documentation "Determines if a supercuadric is ¥"holed¥
".")
                (Domain Sq-Parametric-Model)
                (Instance-Of Binary-Relation Function) (Range Logconst))
               :Axioms
               ((<=> (Has-Hole ?X True) (> (Abs (A4 ?X)) 1.0))))


;;; Has-Round-Latitude

(Define-Frame Has-Round-Latitude
               :Own-Slots
               ((Arity 2)
                (Documentation
                 "Determines if a supercuadric has ¥"round latitude¥".")
                (Domain Sq-Parametric-Model)
                (Instance-Of Binary-Relation Function) (Range Logconst))
               :Axioms
               ((=> (Has-Round-Latitude ?X)
                 (<=> (Has-Round-Latitude ?X True)
                  (=< (Abs (- (E2 ?X) 1.0)) 0.3)))))
```

```
;;; Has-Round-Longitude

(Define-Frame Has-Round-Longitude
              :Own-Slots
              ((Arity 2)
               (Documentation
                "Determines if a supercuadric has ¥"round longitude¥".")
               (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Logconst))
              :Axioms
              ((=> (Has-Round-Longitude ?X)
                 (<=> (Has-Round-Longitude ?X True)
                   (=< (Abs (- (E1 ?X) 1.0)) 0.3)))))


;;; Has-Sharp-Top

(Define-Frame Has-Sharp-Top
              :Own-Slots
              ((Arity 2)
               (Documentation
                "Determines if a supercuadric is ¥"sharp¥" pointed.")
               (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Logconst))
              :Axioms
              ((<=> (Has-Sharp-Top ?X True)
                 (And (Has-Sharpened-Side-1 ?X True)
                   (Has-Sharpened-Side-2 ?X True)))))


;;; Has-Sharpened-Side-1

(Define-Frame Has-Sharpened-Side-1
              :Own-Slots
              ((Arity 2)
               (Documentation
                "Determines if a supercuadric has one side ¥"sharp¥".")
               (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Logconst))
              :Axioms
              ((<=> (Has-Sharpened-Side-1 ?X True)
                 (=< (Abs (- (K1 ?X) 1)) 0.1))))


;;; Has-Sharpened-Side-2

(Define-Frame Has-Sharpened-Side-2
              :Own-Slots
              ((Arity 2) (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Logconst))
              :Axioms
              ((<=> (Has-Sharpened-Side-2 ?X True)
                 (=< (Abs (- (K2 ?X) 1)) 0.1)))
              :Documentation
              "Determines if a supercuadric has one side ¥"sharp¥".")


;;; Has-Similar-Scale-Parameters
```

```
(Define-Frame Has-Similar-Scale-Parameters
                :Own-Slots
                ((Arity 2)
                 (Documentation
                  "It's  true  if  the  supercuadric  has  ¥"similar  scale
parameters.")
                 (Domain Sq-Parametric-Model)
                 (Instance-Of Binary-Relation Function) (Range Logconst))
                :Axioms
                ((<=> (Has-Similar-Scale-Parameters ?X True)
                   (And
                    (=< (Abs (- (A1 ?X) (A2 ?X)))
                     (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                    (=< (Abs (- (A1 ?X) (A3 ?X)))
                     (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                    (=< (Abs (- (A2 ?X) (A3 ?X)))
                     (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))))))


;;; Has-Square-Latitude

(Define-Frame Has-Square-Latitude
                :Own-Slots
                ((Arity 2)
                 (Documentation
                  "Determines if a supercuadric has ¥"square latitude¥".")
                 (Domain Sq-Parametric-Model)
                 (Instance-Of Binary-Relation Function) (Range Logconst))
                :Axioms
                ((<=> (Has-Square-Latitude ?X True) (=< (Abs (E2 ?X)) 0.3))))


;;; Has-Square-Longitude

(Define-Frame Has-Square-Longitude
                :Own-Slots
                ((Arity 2)
                 (Documentation
                  "Determines if a supercuadric has ¥"square longitude¥".")
                 (Domain Sq-Parametric-Model)
                 (Instance-Of Binary-Relation Function) (Range Logconst))
                :Axioms
                ((=> (Has-Square-Longitude ?X)
                  (<=>  (Has-Square-Longitude  ?X  True)  (=<  (Abs  (E1  ?X))
0.3)))))


;;; Is-Flat

(Define-Frame Is-Flat
                :Own-Slots
                ((Arity 2)
                 (Documentation
                  "Determines if a supercuadric is considered ¥"flat¥"")
                 (Domain Sq-Parametric-Model)
                 (Instance-Of Binary-Relation Function) (Range Logconst))
                :Axioms
                ((<=> (Is-Flat ?X True)
                  (Or
                   (And (< (A3 ?X) (Min (A1 ?X) (A2 ?X)))
                    (< (A3 ?X) (/ (Min (A1 ?X) (A2 ?X)) 3)))
                    (And (< (A2 ?X) (Min (A1 ?X) (A3 ?X)))
                     (< (A2 ?X) (/ (Min (A1 ?X) (A3 ?X)) 3)))
                    (And (< (A1 ?X) (Min (A2 ?X) (A3 ?X)))
                     (< (A1 ?X) (/ (Min (A2 ?X) (A3 ?X)) 3)))))))


;;; Is-Long

(Define-Frame Is-Long
              :Own-Slots
              ((Arity 2)
               (Documentation "Determines if a supercuadrics is ¥"long¥
".")
               (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Logconst))
              :Axioms
              ((<=> (Is-Long ?X True)
                 (Or
                  (And
                   (=< (Abs (- (A1 ?X) (A2 ?X)))
                    (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                   (> (A3 ?X) (Max (A1 ?X) (A2 ?X)))
                   (< (A3 ?X) (* 2 (Min (A1 ?X) (A2 ?X)))))
                  (And
                   (=< (Abs (- (A1 ?X) (A3 ?X)))
                    (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                   (> (A2 ?X) (Max (A1 ?X) (A3 ?X)))
                   (< (A2 ?X) (* 2 (Min (A1 ?X) (A3 ?X)))))
                  (And
                   (=< (Abs (- (A2 ?X) (A3 ?X)))
                    (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                   (> (A1 ?X) (Max (A2 ?X) (A3 ?X)))
                   (< (A1 ?X) (* 2 (Min (A2 ?X) (A3 ?X)))))))))


;;; Is-Short

(Define-Frame Is-Short
              :Own-Slots
              ((Arity 2)
               (Documentation "Determines if a supercuadrics is ¥"short¥
".")
               (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Logconst))
              :Axioms
              ((<=> (Is-Short ?X True)
                 (And
                  (=< (Abs (- (A1 ?X) (A2 ?X)))
                   (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                  (< (A3 ?X) (/ (Min (A1 ?X) (A2 ?X)) 2))))))


;;; Is-Tall

(Define-Frame Is-Tall
              :Own-Slots
              ((Arity 2)
               (Documentation "Determines if a superquadrics is ¥"tall¥"")
               (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Logconst))
              :Axioms
```

```
                  ((<=> (Is-Tall ?X True)
                    (And
                     (>= (Abs (- (A1 ?X) (A2 ?X)))
                      (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                     (>= (A3 ?X) (Max (A1 ?X) (A2 ?X)))
                     (< (A3 ?X) (* 2 (Min (A1 ?X) (A2 ?X))))))))


;;; Is-Tapered-Top-E-W

(Define-Frame Is-Tapered-Top-E-W
                 :Own-Slots
                 ((Arity 2)
                  (Documentation "Determines if a supercuadric is tapered at
the
top in East West direction.")
                  (Domain Sq-Parametric-Model)
                  (Instance-Of Binary-Relation Function) (Range Logconst))
                 :Axioms
                 ((<=> (Is-Tapered-Top-E-W ?X True)
                   (And (=< (K2 ?X) 1.0) (> (K2 ?X) 0.0)))))


;;; Is-Tapered-Top-N-S

(Define-Frame Is-Tapered-Top-N-S
                 :Own-Slots
                 ((Arity 2)
                  (Documentation "Determines if a supercuadric is tapered at
the
top in North-South direction.")
                  (Domain Sq-Parametric-Model)
                  (Instance-Of Binary-Relation Function) (Range Logconst))
                 :Axioms
                 ((<=> (Is-Tapered-Top-N-S ?X True)
                   (And (=< (K1 ?X) 1.0) (> (K1 ?X) 0.0)))))


;;; Is-Very-Long

(Define-Frame Is-Very-Long
                 :Own-Slots
                 ((Arity 2)
                  (Documentation
                   "Determines if a supercuadrics is ¥"very long¥".")
                  (Domain Sq-Parametric-Model)
                  (Instance-Of Binary-Relation Function) (Range Logconst))
                 :Axioms
                 ((<=> (Is-Very-Long ?X True)
                   (Or (Is-Very-Tall ?X True)
                    (And
                     (=< (Abs (- (A1 ?X) (A3 ?X)))
                      (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                     (>= (A2 ?X) (* (Max (A1 ?X) (A3 ?X)) 2)))
                    (And
                     (=< (Abs (- (A2 ?X) (A3 ?X)))
                      (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                     (>= (A1 ?X) (* (Max (A2 ?X) (A3 ?X)) 2)))))))


;;; Is-Very-Tall
```

```
(Define-Frame Is-Very-Tall
              :Own-Slots
              ((Arity 2)
               (Documentation
                "Determines if a supercuadrics is ¥"very tall¥".")
               (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Logconst))
              :Axioms
              ((<=> (Is-Very-Tall ?X True)
                 (And
                  (=< (Abs (- (A1 ?X) (A2 ?X)))
                   (/ (+ (A1 ?X) (A2 ?X) (A3 ?X)) 30))
                  (>= (A3 ?X) (* 2 (Max (A1 ?X) (A2 ?X))))))))


;;; K1

(Define-Frame K1
              :Own-Slots
              ((Arity 2) (Documentation "tapering factor on North/South")
               (Domain Sq-Parametric-Model)
               (Instance-Of Binary-Relation Function) (Range Number)))


;;; K2

(Define-Function K2
                 (?Frame)
                 :->
                 ?Value
                 "Scale on the X axis"
                 :Def
                 (And (Sq-Parametric-Model ?Frame) (Number ?Value))
                 :Documentation
                 "tapering factor on East/West")



;;; ------------------- Instance --------------


;;; ------------------- Axiom --------------


;;; ------------------- Other --------------

Date: Thu, 31 Aug 1995 19:16:36 -0700
To: jdelgado@atr-sw.atr.co.jp
From: KSL Network Services <Ontology-librarian@HPP.Stanford.EDU>
Subject: Ontology 3D-SHAPES

(In-Package "ONTOLINGUA-USER")

;;; Written by user Jdelgado from session "test" owned by group JUST-ME
;;; Date: Aug 31, 1995  19:16


(Define-Ontology
     3d-Shapes
     (3d-Parametric-Models Component-Assemblies)
```

```
   "This ontology is for defining 3-D Shapes. It gives
semantics to graphical objects than can be part of
a complex system."
   :Io-Package
   "ONTOLINGUA-USER"
   :Intern-In
   ((Frame-Ontology Slot-Value-Type Thing Alias Subclass-Of Class Instance-Of
     Documentation Arity)
    (Kif-Numbers Number Positive) (Kif-Sets True)))


(In-Ontology (Quote 3d-Shapes))



;;; ------------------ Classes --------------

;;; 3d-Geometric-Primitive

(Define-Class 3d-Geometric-Primitive
              (?X)
              "Primitive 3-D shapes."
              :Def
              (And (Thing ?X)))


;;; Cone

(Define-Frame Cone
              :Own-Slots
              ((Arity 1)
               (Documentation "The class of all shapes ¥"like¥" cones.")
               (Instance-Of Class)
               (Subclass-Of 3d-Geometric-Primitive Sq-Parametric-Model))
              :Template-Slots
              ((Has-Round-Latitude True) (Has-Square-Longitude True)
               (Is-Tall True))
              :Axioms
              ((<=> (Cone ?X)
                 (And (Sq-Parametric-Model ?X) (Is-Tall ?X True)
                  (Has-Square-Longitude  ?X  True)  (Has-Round-Latitude  ?X
True)
                  (Has-Sharp-Top ?X True) (Not (Has-Hole ?X True))))))


;;; Cube

(Define-Frame Cube
              :Own-Slots
              ((Alias Sq-Parametric-Model) (Arity 1)
               (Documentation "The class of all shapes ¥"like¥" cubes.")
               (Instance-Of Class)
               (Subclass-Of 3d-Geometric-Primitive Sq-Parametric-Model))
              :Template-Slots
              ((Has-Hole) (Has-Similar-Scale-Parameters True)
               (Has-Square-Latitude True) (Has-Square-Longitude True))
              :Axioms
              ((=> (Cube ?X) (Sq-Parametric-Model ?X))
               (=> (Cube ?X) (Not (Is-Tapered-Top-E-W ?X True)))
               (=> (Cube ?X) (Not (Is-Tapered-Top-N-S ?X True)))
```

```
                 (=> (Cube ?X) (Not (Has-Hole ?X True)))
                 (<=> (Cube ?X)
                  (And (Sq-Parametric-Model ?X)
                   (Has-Similar-Scale-Parameters ?X True)
                   (Has-Square-Longitude  ?X  True)  (Has-Square-Latitude  ?X
True)
                   (Not (Is-Tapered-Top-E-W ?X True))
                   (Not (Is-Tapered-Top-N-S ?X True))
                   (Not (Has-Hole ?X True))))))


;;; Cylinder

(Define-Frame Cylinder
                :Own-Slots
                ((Arity 1)
                 (Documentation  "The  class  of  all  shapes  ¥"like¥"
cylinders.")
                 (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
                :Axioms
                ((<=> (Cylinder ?X)
                  (And (Sq-Parametric-Model ?X) (Is-Tall ?X True)
                   (Has-Square-Longitude  ?X  True)  (Has-Round-Latitude  ?X
True)
                   (Not (Is-Tapered-Top-E-W ?X True))
                   (Not (Is-Tapered-Top-N-S ?X True))
                   (Not (Has-Hole ?X True))))))


;;; Cylindrical-Rod

(Define-Frame Cylindrical-Rod
                :Own-Slots
                ((Arity 1)
                 (Documentation
                  "The class of all shapes ¥"like¥" cylindrical rods.")
                 (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
                :Axioms
                ((<=> (Cylindrical-Rod ?X)
                  (And (Sq-Parametric-Model ?X) (Is-Very-Tall ?X True)
                   (Has-Square-Longitude  ?X  True)  (Has-Round-Latitude  ?X
True)
                   (Not (Is-Tapered-Top-E-W ?X True))
                   (Not (Is-Tapered-Top-N-S ?X True))
                   (Not (Has-Hole ?X True))))))


;;; Cylindrical-Tube

(Define-Frame Cylindrical-Tube
                :Own-Slots
                ((Arity 1)
                 (Documentation
                  "The class of all shapes ¥"like¥" cylindrical tubes.")
                 (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
                :Axioms
                ((<=> (Cylindrical-Tube ?X)
                  (And (Sq-Parametric-Model ?X)
                   (Or (Is-Very-Tall ?X True) (Is-Tall ?X True))
                   (Has-Square-Longitude  ?X  True)  (Has-Round-Latitude  ?X
True)
```

```
                    (Not (Is-Tapered-Top-E-W ?X True))
                    (Not (Is-Tapered-Top-N-S ?X True)) (Has-Hole ?X True)))))


;;; Disk

(Define-Frame Disk
               :Own-Slots
               ((Arity 1)
                (Documentation "The class of all shapes ¥"like¥" disks.")
                (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
               :Axioms
               ((<=> (Disk ?X)
                  (And (Sq-Parametric-Model ?X) (Is-Short ?X True)
                    (Has-Square-Latitude  ?X  True)  (Has-Round-Longitude  ?X
True)
                    (Not (Is-Tapered-Top-E-W ?X True))
                    (Not (Is-Tapered-Top-N-S ?X True)) (Has-Hole ?X True)))))


;;; Hoop

(Define-Frame Hoop
               :Own-Slots
               ((Arity 1)
                (Documentation "The class of all shapes ¥"like¥" hoops.")
                (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
               :Axioms
               ((<=> (Hoop ?X)
                  (And (Sq-Parametric-Model ?X) (Is-Short ?X True)
                    (Has-Square-Longitude  ?X  True)  (Has-Round-Latitude  ?X
True)
                    (Not (Is-Tapered-Top-E-W ?X True))
                    (Not (Is-Tapered-Top-N-S ?X True)) (Has-Hole ?X True)))))


;;; Ingot

(Define-Frame Ingot
               :Own-Slots
               ((Arity 1)
                (Documentation "The class of all shapes ¥"like¥" ingots.")
                (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
               :Axioms
               ((<=> (Ingot ?X)
                  (And (Sq-Parametric-Model ?X) (Is-Long ?X True)
                    (Has-Square-Longitude  ?X  True)  (Has-Square-Latitude  ?X
True)
                    (Not (Is-Tapered-Top-E-W ?X True))
                    (Not (Is-Tapered-Top-N-S ?X True))
                    (Not (Has-Hole ?X True))))))


;;; Pyramid

(Define-Frame Pyramid
               :Own-Slots
               ((Arity 1)
                (Documentation
                 "The  class  of  all  shapes  ¥"like¥"  rectangular  base
pyramids.")
```

```
                (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
               :Axioms
               ((<=> (Pyramid ?X)
                 (And (Sq-Parametric-Model ?X) (Is-Tall ?X True)
                  (Has-Square-Longitude  ?X  True)  (Has-Square-Latitude  ?X
True)
                  (Has-Sharp-Top ?X True) (Not (Has-Hole ?X True))))))


;;; Rectangular-Rod

(Define-Frame Rectangular-Rod
               :Own-Slots
               ((Arity 1)
                (Documentation
                 "The class of all shapes ¥"like¥" rectangular rods.")
                (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
               :Axioms
               ((<=> (Rectangular-Rod ?X)
                 (And (Sq-Parametric-Model ?X) (Is-Very-Long ?X True)
                  (Has-Square-Longitude  ?X  True)  (Has-Square-Latitude  ?X
True)
                  (Not (Is-Tapered-Top-E-W ?X True))
                  (Not (Is-Tapered-Top-N-S ?X True))
                  (Not (Has-Hole ?X True))))))


;;; Rectangular-Sheet

(Define-Frame Rectangular-Sheet
               :Own-Slots
               ((Arity 1)
                (Documentation
                 "The class of all shapes ¥"like¥" rectangular sheets.")
                (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
               :Axioms
               ((<=> (Rectangular-Sheet ?X)
                 (And (Sq-Parametric-Model ?X) (Is-Flat ?X True)
                  (Has-Square-Longitude  ?X  True)  (Has-Square-Latitude  ?X
True)
                  (Not (Is-Tapered-Top-E-W ?X True))
                  (Not (Is-Tapered-Top-N-S ?X True))
                  (Not (Has-Hole ?X True))))))


;;; Rectangular-Tube

(Define-Frame Rectangular-Tube
               :Own-Slots
               ((Arity 1)
                (Documentation
                 "The class of all shapes ¥"like¥" rectangular tubes.")
                (Instance-Of Class) (Subclass-Of 3d-Geometric-Primitive))
               :Axioms
               ((<=> (Rectangular-Tube ?X)
                 (And (Sq-Parametric-Model ?X)
                  (Or (Is-Very-Tall ?X True) (Is-Tall ?X True))
                  (Has-Square-Longitude  ?X  True)  (Has-Square-Latitude  ?X
True)
                  (Not (Is-Tapered-Top-E-W ?X True))
                  (Not (Is-Tapered-Top-N-S ?X True)) (Has-Hole ?X True)))))
```

```
;;; Sphere

(Define-Frame Sphere
              :Own-Slots
              ((Alias Sq-Parametric-Model) (Arity 1)
               (Documentation
                "This is the class of primitive 3-D shapes that are ¥"like
¥"
spheres.")
               (Instance-Of Class)
               (Subclass-Of  3d-Geometric-Primitive  Sq-Parametric-Model
Test))
              :Template-Slots
              ((Radio 1 (Slot-Value-Type Positive))
               (Radio 1 (Slot-Value-Type Positive)))
              :Axioms
              ((<=> (Sphere ?X)
                 (And (Sq-Parametric-Model ?X)
                  (Has-Similar-Scale-Parameters ?X True)
                  (Has-Round-Latitude ?X True) (Has-Round-Longitude ?X True)
                  (Not (Is-Tapered-Top-E-W ?X True))
                  (Not (Is-Tapered-Top-N-S ?X True))
                  (Not (Has-Hole ?X True))))))


;;; ------------------ Relations --------------


;;; ------------------ Functions --------------

;;; Radio

(Define-Function Radio
                 (?Frame)
                 :->
                 ?Value
                 "The Radio of a Sphere."
                 :Def
                 (And (Sphere ?Frame) (Number ?Value)))


;;; ------------------ Instance --------------


;;; ------------------ Axiom --------------


;;; ------------------ Other --------------
```

Appendix B

Modification of Protos program

The original code of the CL-Protos program (The common lisp version of Protos) was found at the software repository of the Artificial Intelligence laboratory, Department of Computer Sciences, University of Texas at Austin. All the copyrights are given to Daniel L. Dvorak, who had rewritten the original Protos system (implemented in Prolog) into Common Lisp, and to the authors of the Protos algorithm and following research, Ray Bareiss and Bruce W. Porter.

This program has been modified with the permission of the authors, on the behalf that it was used strictly for research purposes.

CL-Protos is a machine learning program which acquires knowledge for performing expert heuristic classification. It has been modified in it's heuristics to allow a more flexible learning proccess, sometimes in detriment of its computational cost. All the modifications have been stated in the program's text as comments and the original lines have been quoted, facilitating the direct comparision between the original version and the modified one.

```
-----------------------------------------------------------------
;;; -*- Mode: Lisp; Syntax: Common-lisp; Base: 10; Package: PROTOS -*-
;;;     Copyright (c) 1988, Daniel L. Dvorak.
;;;
;;; Last modified by Joaquin Delgado, Aug. 29 1995

(in-package 'protos)

(defvar *cur-exemplar*)    ;;global of current exemplar being used.
(defvar *feat*)            ;;global of current feature
(defvar *cat*)             ;;global of current category
(defvar *exp*)             ;;global of current exemplar


;;;=============================================================
;;;
;;;           D I S C U S S    M A T C H    W I T H    T E A C H E R
;;;     -----------------------------------------------------------
;;;
;;;  Overview:    This file contains the functions that Protos uses to
discuss
;;;          a proposed exemplar-to-new case match.  The top-level function
;;;          where it all starts is "discuss-match".
;;;
;;;  Functions:     add-new-exemplar
;;;                get-approximate-importances
;;;                discuss-differences
;;;                discuss-high-importances
;;;           discuss-match
;;;           discuss-success (*)
;;;           discuss-failure
;;;           discuss-exemplars
;;;           discuss-relevances
;;;           discuss-removals
;;;           discuss-unmatched
;;;               discuss-unmatched-importances
;;;              enter-explanation
;;;         read-category
;;------------------------------------------------------------
;; (*) This part was modifed, specifically the functions:
;; 1)  set-remindings2
;; 2)  select-riminding-target
;;;==========================================================



(defparameter *exemplar-menu* (make-menu
  :label  "~%What would you like to do?"
  :items   '((#¥T  .  ("Try  the  most  prototypical  remaining  exemplar."
return  next-exemplar))
           (#¥S . ("Select a specific exemplar of this category."    return
show-exemplars))
           (#¥C . ("Create new exemplar from this case."           return
add-exemplar))
           (#¥A . ("Abandon this case."                          return
abandon)))))


;;;-----------------------------------------------------------
;;;  Function:  (discuss-match  match)
;;;
;;;  Given: match, the details of an exemplar-to-newcase match
;;;
;;;  Returns:   2 values: action and match, where:
;;;             -- action is either:
;;;                'done  -- match contains an approved match structure;
;;;                'newex -- match contains a newly created exemplar;
;;;                'retry -- retry finding a match for this case;
;;;                'redo  -- redo the match to this exemplar;
;;;                'next  -- try the next hypothesis.
;;;
;;;  Called by: compare
;;;
;;;  Notes: Discuss-match presents the match to the teacher and asks
;;;         if the classification is correct.  Regardless of the answer,
;;;         the teacher is presented with other questions which may
;;;         modify the current match.  It is in the subordinate functions
;;;         of discuss-match where Protos learns from the teacher.
;;;-----------------------------------------------------------
(defun discuss-match (match)
  (print-match match)
  (examine-explanations match)
  (let ((prompt1 (format nil "~%Is ~A the correct classification for this
case? "
                 (getname (exemplar-category (match-exemplar match))))))
    (if (prompt prompt1 nil 'y-or-n nil nil)
      (discuss-success match)
      (discuss-failure match))))

;;;-----------------------------------------------------------
;;;  Function:  (discuss-success  match)
;;;
;;;  Given: match, the results of an exemplar-to-new case match where the
;;;                the teacher has approved the classification (but hasn't
;;;                    yet approved the exemplar that the case was matched
to).
;;;
```

```
;;;  Returns:     -- the same match if the teacher agrees that the chosen
;;;                  exemplar is appropriate;
;;;               -- a different match if the teacher chooses a different
;;;                  exemplar; or
;;;               -- nil if no more action is needed on this match (such as when
;;;                  the teacher decides to make a new exemplar from the case or
;;;                      the teacher decides to abandon the case).
;;;
;;;  Caller:      discuss-match
;;;-----------------------------------------------------------------
(defun discuss-success (match)

  (let* ((newcase (match-newcase match))
         action)

    ;; Set the classification of the new case.
    (setf  (case-category  newcase)  (exemplar-category  (match-exemplar
match)))

    (multiple-value-setq (action match) (discuss-exemplars match))

    ;; Return if teacher decided to make exemplar of the new case.
    (if (eql 'newex action)
      (return-from discuss-success (values 'newex match)))

    ;; Return if the case is being abandoned by the user.
    (if (eql 'done action)
      (return-from discuss-success (values 'done match)))

    ;; Ask about unmatched features.
    (discuss-unmatched match)

    ;; Ask for unfocused instruction.
    (discuss-unfocused)

    ;; Update prototypicality of the exemplar.
    (if (mergeable match)
      (progn
        (if (prompt (format nil
                        "~%~%This  case  (~A)  is  similar  enough  to  ~A  that
Protos~
                                 ~%is prepared to forget the case (and just
retain ~A).~
                               ~%~%Will it ever be important to distinguish
between these two? "
                   (case-name newcase)
                   (getname (match-exemplar match))
                   (getname (match-exemplar match)))
                       nil 'y-or-n nil nil)

              (progn
              (update-prototypicality (match-exemplar match) 0.75)
              (return-from discuss-success (values 'newex (add-new-exemplar
newcase))))
              (progn
              (update-prototypicality (match-exemplar match) 1.0)
              (merge-exemplar match)
              (return-from discuss-success (values 'done match)))))

      ;; match was not mergeable
      (progn
        (format t "~%This case is different enough from ~A~
                     ~%that it will be made into an exemplar."
                (getname (match-exemplar match)))
         (if (>= (match-similarity match) *strong-match*)
             (update-prototypicality (match-exemplar match) 0.50)
             (update-prototypicality (match-exemplar match) 0.25))
         (return-from  discuss-success  (values  'newex  (add-new-exemplar
newcase)))))))


;;;-----------------------------------------------------------------
;;;  Function:  (update-prototypicality  exemplar  amount)
;;;
;;;  Purpose:   This  function  increments  the  given  exemplar's  typicality
value
;;;                   and  then  (re-)sorts  the  affected  category's  list  of
exemplars
;;;             in decreasing order of typicality.
;;;-----------------------------------------------------------------
(defun update-prototypicality (exemplar amount)
  ;; Add amount to the exemplar's typicality value.
  (incf (exemplar-typicality exemplar) amount)
  ;; Re-sort the category's exemplars by typicality value.
  (let ((category (exemplar-category exemplar)))
    (setf (category-exemplars category)
          (sort  (category-exemplars  category)  #'>=  :key  #'exemplar-
typicality))))
```

```
;;;-------------------------------------------------------------
;;;  Function:  (discuss-exemplars match)
;;;
;;;  Given: match, a match to a Protos-selected exemplar of the correct
;;;         category;
;;;
;;;  Returns:     3 values, action and match, where:
;;;             -- action may be either 'mrgex, 'done or 'newex, and
;;;             -- match is either a teacher-approved exemplar of the same
;;;                category, or is a new exemplar (as when the teacher
;;;                decides to make a new exemplar of the case).
;;;
;;;  Caller:      discuss-success
;;;-------------------------------------------------------------

(defun discuss-exemplars (match)
  (declare (special closed-exemplars))

  (let* ((exemplar (match-exemplar match))
       (newcase  (match-newcase  match))
       (viewed-exemplars (list exemplar))
       (category (exemplar-category exemplar))
       (exemplars (category-exemplars category))
       (exemplist (copy-list exemplars)))

    (loop
      (tagbody
      (if (prompt "~%Is this a suitable exemplar for matching the new case?
"
                nil 'y-or-n nil nil)
          (return-from discuss-exemplars (values 'mrgex match)))

      ;; The teacher has rejected this match, saying that this is not a
      ;; suitable exemplar.  So, remember this fact so that the function
      ;; "test-hypotheses2" will skip over this exemplar if it appears in
      ;; the list of hypotheses.  This could happen if we reached this
      ;; unsuitable exemplar through a difference link.
      (push (match-exemplar match) closed-exemplars)

      ;; If this incorrect match is strong enough (i.e., is a "near miss"),
      ;; then remember it for possibly installing a difference link later
      ;; when the case finally is correctly classified.
      (if (>= (match-nth-root-of-similarity match) *near-miss-threshold*)
          (push match *near-misses*))

      ;; Teacher has said no, this isn't a suitable exemplar for the new
case.
      ;; So, Protos now offers the teacher one of 4 choices:
      ;;    1.  Take the most prototypical exemplar remaining and match to
it.
      ;;     2.  Display the names of this category's exemplars and allow
teacher
      ;;        to select one of them.
      ;;    3.  Install this case as a new exemplar of its category.
      ;;    4.  Abandon this case.

      (ecase (menu-select *exemplar-menu*)
        (next-exemplar
          ;; Find the next most prototypical exemplar that hasn't already
been viewed.
          (do ()
              ((endp exemplist))
              (setq exemplar (pop exemplist))
              (if (not (member exemplar viewed-exemplars))
                (go SHOW-EXEMPLAR)))

            ;; No more exemplars left on exemplist.
            (if (prompt "~%You have rejected all exemplars in this category.~
                           ~%The only choice now is to either make a new
exemplar~
                           ~%of this case or abandon the case.~
                         ~%Shall I make a new exemplar of this case? "
                   nil 'y-or-n nil nil)
              (return-from discuss-exemplars (values 'newex (add-new-exemplar
newcase)))
              (progn
                (format t "~%Then this case is being abandoned.")
                (return-from discuss-exemplars (values 'done nil)))))

          (show-exemplars
            ;; Print the names of all the exemplars of this category.
            (terpri)
            (print-node-names exemplars t ", ")

            (loop
              (let ((input (prompt "~%Please enter one of the above names:  "
                             nil 'termname category 'category)))
              (setq exemplar (find input exemplars :test #'equal :key #'node-
name))
              (if exemplar (go SHOW-EXEMPLAR)))))

          (add-exemplar
            ;; Make a new exemplar of the new case.
            (setf (case-category newcase) category)
            (return-from discuss-exemplars (values 'newex (add-new-exemplar
newcase))))

          (abandon
            (return-from discuss-exemplars (values 'done nil))))


      SHOW-EXEMPLAR
        ;; This is the bottom of the loop that begins by asking if the
exemplar
        ;; in the displayed match is a suitable match for newcase.  We arrive
        ;; here only if the teacher has selected a different exemplar to
examine.
        (push exemplar viewed-exemplars)
        (setq match (compare-new-case exemplar newcase))
        (print-match match)
        (examine-explanations match)))))
```

```
;;;-------------------------------------------------------------
;;;  Function:  (add-new-exemplar  newcase)
;;;
;;;  Given: newcase, a case from which an exemplar is to be built and
;;;                 installed;
;;;
;;;  Returns:     This function is executed for its side-effect of creating
and
;;;          installing a new exemplar in the category network.
;;;
;;;  Callers:     discuss-success, discuss-failure, discuss-exemplars.
;;;
;;;  Design:      -- If newcase's category is unknown, prompt for it.
;;;          -- Ask if any features of newcase should NOT become part of
;;;             the exemplar, and then remove them.
;;;          -- Create the exemplar from newcase and install it as a member
;;;             of the given category in the category network.
;;;          --  For  each  feature  that  is  newly  occurring  within  the
category,
;;;             ask about its relevance to the category.  For each explana-
;;;             tion given by the teacher, glean remindings from F-->C
;;;             explanations and estimate importances from C-->F explana-
;;;             tions.
;;;          -- Initialize prototypicality of new exemplar to 1.0.
;;;          -- Install pending difference links.
;;;-------------------------------------------------------------

(defun add-new-exemplar (newcase)

  (let ((category  (case-category newcase))
      exemplar)

    ;; If newcase's category is unknown, then prompt for it.
    (if (null category)
      (progn
        (setq category (read-category))
        (setf (case-category newcase) category)))

    ;; Ask if any features of newcase should NOT become features of the
    ;; exemplar.  If so, remove them from the newcase.
    (discuss-removals newcase)

    ;; Create the exemplar from newcase and install it as a member
    ;;   of the given category in the category network.
    (setq exemplar (make-term :name     (case-name     newcase)
                              :comment  (case-comment  newcase)
                              :features (case-features newcase)
                              :category (case-category newcase)
                              :typicality 1.0))
    (push exemplar *history*)

    (setq *cur-exemplar* exemplar)

    ;; Create the symbol that has the print-name of this exemplar.
    ;; (note that this is a 'set', not a 'setf').
    (set (case-name newcase) exemplar)

    ;; Update number of exemplar-containing categories.
    (if (null (category-exemplars category))
      (incf *number-of-ec-categories*))
    ;; Add this new exemplar to the category's list of exemplars (which are
    ;; sorted by typicality).  Since this is a brand new exemplar, it will
    ;; have the lowest possible typicality and can therefore simply be
    ;; added to the end of the list.  Also, install the "has-exemplar" and
    ;; "is-exemplar-of" relations between the category and exemplar.
    (setf (category-exemplars category)
        (nconc (category-exemplars category) (list exemplar)))
    (install-relation (list category) (list exemplar) nil *verb-hasExemplar*
nil)

    ;; For each feature of this exemplar, save a pointer to this exemplar
    ;; in the feature's structure.  This is used in combine-remindings
    ;; (specifically, strengthen-features) so that if featural exemplars
    ;; are present in a match, they will be matched first.
    (dolist (feature (exemplar-features exemplar))
      (push exemplar (feature-of-exemplars feature)))

    ;; Discuss any fault variables that should be added to the category.
    (discuss-faultvars exemplar)

    ;; Discuss relevance of features new to this exemplar's category.
    (discuss-relevances exemplar)

    ;; Discuss category's unmatchable high-importance features.
    (discuss-high-importances category exemplar)

    ;; Install pending difference links.
    (discuss-near-misses exemplar)

    ;; End of function.
    (if *trace-new-exemplar*
      (progn
        (format t "~%The following new exemplar has been created:~%")
        (print-exemplar exemplar t 1)))

    exemplar))
```

```
;;;-----------------------------------------------------------------
;;;  Function:     (read-category)
;;;
;;;  Returns:      A category node.
;;;
;;;  Design:       This function prompts the user for a category name.  If
the
;;;          input is recognized as an existing category, then a pointer
;;;          to that category node is returned.
;;;
;;;  Callers:      add-new-exemplar
;;;-----------------------------------------------------------------
(defun read-category ()

  (loop
    (let* ((input   (prompt "~&What is this case's category?   " nil
'termname
                     nil nil))
         (object (check-term-name input 'ask)))

      ;; If the name entered by the user exists or has been created ...
      (if object

        ;; then make sure it is a category (as opposed to, say, an
exemplar)
        (if (my-exemplar-p object)
            (progn
            (format *query-io*
                  "~%Error: ~A is an exemplar, not a category.~
                       ~%Please enter a different category name: "
                  input))
            (return-from read-category object))

        ;; user apparently said to forget this term.
        (format *query-io* "~%Sorry, but you HAVE to enter a category name:
")))))
```

```
;;;------------------------------------------------------------------
;;;  Function:    (discuss-removals  newcase)
;;;
;;;  Given: newcase, a case to be made into an exemplar
;;;
;;;  Purpose:     The teacher is asked if any of the features of the new
case
;;;          are irrelevant to its future role as an exemplar of the
;;;          category.  If so, the teacher is allowed to remove those
;;;          features.
;;;
;;;  Called by:  add-new-exemplar
;;;------------------------------------------------------------------
(defun discuss-removals (newcase)

  (let ((features   (case-features newcase))
       (category    (case-category newcase))
       input)

    (format *query-io* "~%Case ~A is composed of the following features:~%"
                (case-name newcase))
    (print-node-names features *query-io* ", ")

    (if (not (prompt (format nil "~%Are any of these features IRRELEVANT~
                              ~%to its role as an exemplar of ~A ? "
                         (getname category))
                 nil 'y-or-n nil nil))
      (return-from discuss-removals (values)))

    (setq input (prompt "~%Then please enter, one per line:" "~&---> "
                 'termname newcase 'case))

    ;; If no features to be deleted, then just return.
    (if (null input)
      (return-from discuss-removals (values)))

    ;; Verify that each feature to be deleted was spelled correctly.
    (dolist (fname input)
      (let ((feature (check-term-name fname 'ask)))
      (if feature
          (if (member feature features)
            (setq features (delete feature features))
            (format *query-io* "~%Skipping ~A -- it wasn't a feature of the
case anyway."
                    fname)))))

    ;; Done deleting features -- store results back in newcase.
    (setf (case-features newcase) features)

    (format *query-io* "~%Thank you.  Case ~A now contains only these
features:~%"
          (case-name newcase))
    (print-node-names features *query-io* ", ")
    (terpri)))
```

```
;;;-------------------------------------------------------------
;;;  Function:  (discuss-faultvars  exemplar)
;;;
;;;  Purpose:    This functions adds new fault variables to the category,
with
;;;             the teachers approval.
;;;
;;;  Design:    1.  Collect all the variables specified as features of this
;;;                 exemplar.
;;;             2.  Remove the variables that are already known to be fault
;;;                 variables of this exemplar's category.
;;;             3.  For each remaining variable, ask the teacher if the
;;;                 variable is a fault variable.
;;;-------------------------------------------------------------
(defun discuss-faultvars (exemplar)
  (let* ((category          (exemplar-category exemplar))
         (known-faultvars  (category-faultvars category))
         (maybe-faultvars  nil)
         pred)

    ;; Check every feature of exemplar for possible fault variables.
    (dolist (feature (exemplar-features exemplar))
      (if (and (setq pred (term-predicate feature))
               (predicate-relations pred)
               (not (member pred known-faultvars)))
        (push pred maybe-faultvars)))

    ;; Ask user about each candidate fault variable.
    (dolist (var maybe-faultvars)
      (if (prompt (format nil "~%Is ~A a fault variable of ~A ? "
                          (getname var) (getname category))
                 nil 'y-or-n nil nil)
        (push var (category-faultvars category))))))


;;;-------------------------------------------------------------
;;;  Function:  (discuss-near-misses  exemplar)
;;;
;;;  Given:      -- an exemplar which the new case has either been merged
with
;;;                or has been made into, and
;;;                 -- *near-misses*, a global list of strong matches to
exemplars
;;;                that were rejected by the teacher.
;;;
;;;  Does:        For each near miss, the teacher is shown the unmatched
features
;;;             of the new case and the near-miss-exemplar, and is asked to
;;;             specify which features are important discriminators.
;;;              If any of the discriminating features currently is of low
or
;;;               spurious importance to the target exemplar, then ask the
teacher
;;;             for a reassessment of the importance.
;;;-------------------------------------------------------------

(defun discuss-near-misses (chosen-exemplar)

  ;; Consider each near-miss in turn ...
  (dolist (match *near-misses*)
    ;; Guard against the unusual situation where a near-miss was recorded
to
    ;; what turned out to be the chosen exemplar.  This can happen in the
rare
    ;; case where the teacher changes his/her mind about a match.

    (if (not (eq chosen-exemplar (match-exemplar match)))
      (let* ((exemplar     (match-exemplar match))
             (c-unmatched  (match-unmatched match))
             (e-unmatched  nil))

        ;; Collect non-spurious features of exemplar not matched by current
case.
        (dolist (result (match-results match))
          (if (and (eql 'unmatched (result-type result))
                   (/=  0.0  (result-importance result)))
            (push (result-feature result) e-unmatched)))

        (format t "~%~%Protos previously mistook this case for the exemplar
~A."
              (getname exemplar))

        (discuss-differences e-unmatched chosen-exemplar exemplar t)
        (discuss-differences c-unmatched exemplar chosen-exemplar nil)))))

;;;------------------------------------------------------------
;;;  Function:  (discuss-differences  features  from-exemplar  to-exemplar
;;;                                      from-chosen)
;;;
;;;  Given:     a list of features of the to-exemplar that were not matched
by
;;;             the from-exemplar, and a 'from-chosen' flag that is non-nil
if
;;;             from-exemplar is the "chosen exemplar" for the new case.
;;;
;;;  Does:      installs a difference link in the from-exemplar pointing to
the
;;;             to-exemplar, annotated with a subset of 'features' approved
by
;;;             the teacher as important discriminating features.  If the
;;;                 teacher feels that none of 'features' are important
discrimina-
;;;             tors, then no difference link is installed.
;;;------------------------------------------------------------

(defun discuss-differences (features from-exemplar to-exemplar from-chosen)

  (if (null features)
      (return-from  discuss-differences  nil))

  (if from-chosen
      ;; Ask teacher about discriminating features of mistaken exemplar.
      (format t "~%The features of ~A that were not matched by the current
case are:~%"
            (getname to-exemplar))
      ;; Ask teacher about discriminating features of the chosen exemplar.
      (format t "~%The features of this case that were not matched by ~A
are:~%"
            (getname from-exemplar)))

  (let ((diff-features nil))
    (dolist (feature features)
      (format t "  ~A"  (getname feature)))

    (format  t  "~%~%Which  of  these  features  (if  any)  are  important
discriminators?")
    (dolist (feature features)
      (if (y-or-n-p  "~&--> ~A ?~26T"  (getname feature))
        (push feature diff-features)))

    ;; If no features were specified, then just return.
    (if (null diff-features)
      (progn
        (format  t  "~%Since  no  features  were  designated  as  important
discriminators,~
                          ~%then no difference link will be created/changed from
~A to ~A"
              (getname from-exemplar)  (getname to-exemplar))
        (return-from discuss-differences (values))))

    ;; Display the difference link to be installed.
    (let ((newdiff  (make-difference :node to-exemplar
                                 :features diff-features))
          (olddiffs (exemplar-differences from-exemplar)))

      (format t "~%Installing new difference link:~
                 ~%~12@A: "  (getname from-exemplar))
      (print-difference newdiff t 1)
      (terpri t)

      ;; Before installing the new difference link, check to see if there
      ;; is already a difference link between these two exemplars.
      ;; If so, ask the user if it should be retained or deleted.

      (dolist (olddiff olddiffs)
      (if (eq to-exemplar (difference-node olddiff))
          (progn
            (format t "~%A previous difference link exists:~
                         ~%~12@A: "  (getname from-exemplar))
            (print-difference olddiff t 1)
            (if (y-or-n-p "~&~%Do you want this to be deleted? ")
              (setf  (exemplar-differences  from-exemplar)  (delete  olddiff
olddiffs))))))

      (push newdiff (exemplar-differences from-exemplar))

      ;; All of the discriminating features should probably have at least
      ;; moderate importance to the exemplar or category possessing those
      ;; features.  If not, ask teacher to reassess the importance.
      (dolist (feature diff-features)
      (multiple-value-bind (imp node) (get-importance feature to-exemplar)
        (declare (ignore node))
        (if (or (null imp) (< (cdr imp) *reassess-importance-threshold*))
            (progn
            (format  t  "~%~%~A  is  a  discriminating  feature,  but  it  is
currently~
                               ~%of  ~A  importance.   Please  reassess  its
importance."
                    (getname  feature)  (qualitative-value  (cdr  imp)
'importance))
            (reassess-importance feature to-exemplar)))))))
```

```
;;;-----------------------------------------------------------------
;;;  Function:    (qualitative-value  strength  type)
;;;
;;;  Returns:     a word giving a qualitative interpretation of the strength
of
;;;          an importance or reminding.
;;;
;;;  Note:  There is a "clinical scale" of normal, mild, moderate,
;;;          severe, profound.
;;;-----------------------------------------------------------------
(defun qualitative-value (strength type)
  (if (null strength) (return-from qualitative-value 'unknown))
  (ecase type
    (importance  (cond ((null strength)                          'spurious)
                   ((>= strength *importance-necessary*)    'necessary)
                   ((>= strength *importance-high*)         'high)
                   ((>= strength *importance-moderate*)     'moderate)
                   ((>= strength *importance-low*)          'low)
                   (t                                 'spurious)))

    (reminding   (cond ((=  strength *reminding-absolute*)      'absolute)
                   ((>= strength *reminding-strong*)        'strong)
                   ((>= strength *reminding-moderate*)      'moderate)
                   ((>  strength (- *reminding-moderate*)) 'weak)
                   ((>  strength (- *reminding-strong*))   'moderate)
                   ((>  strength (- *reminding-absolute*)) 'strong)
                   (t                                       'absolute)))))

;; Function to compute the average of a set of numbers.
(defun average (&rest values)
  (/ (apply #'+ values) (length values)))


(defparameter *importance-alist*
          `((necessary  .  ,(average *importance-absolute*   *importance-
necessary*))
          (high      . ,(average *importance-necessary* *importance-high*))
          (moderate   .  ,(average  *importance-high*         *importance-
moderate*))
          (low       . ,(average *importance-moderate*   *importance-low*))
          (spurious   . ,0.0)))


(defparameter *decide-importance-menu* (make-menu
      :label  "~%Do you agree?"
      :items  '((#¥Y  .  ("Yes."                                      return
yes))
          (#¥E  .  ("no,  let  me  revise  the  Explanation"         return
revise-expl))
          (#¥I  .  ("no,  let  me  just  revise  the  Importance"   return
revise-imp)))))

(defparameter *importance-menu* (make-menu
      :label  "~%What do you believe its importance to be?"
      :twocol t
      :items  '((#¥N . ("Necessary"                 return  necessary))
          (#¥L . ("Low"                         return  low))
          (#¥H . ("High"                        return  high))
          (#¥S . ("Spurious"                    return  spurious))
          (#¥M . ("Moderate"                    return  moderate))
          (#¥Q . ("Quit (leave it unchanged)" return  nochange)))))


;;;-----------------------------------------------------------------
;;;  Function:  (reassess-importance  feature  node)
;;;
;;;  Purpose:    Given a feature and a node (an exemplar or category) for
which
;;;              we want to reassess the feature's importance, this function
;;;              shows the user the feature's current importance (if any) in
;;;                    qualitative form and asks the user for a revised
qualitative
;;;              value.
;;;-----------------------------------------------------------------
(defun reassess-importance (feature node1)
  (let (qual-imp1
      qual-imp2
      target)

    ;; Get importance of this feature, and node to which it is important.
    (multiple-value-bind (importance node2)
      (get-importance feature node1)

      ;; Set the target node which is to get the revised importance value.
      ;; If no importance already exists, then make the target be a
category
      ;; rather than an exemplar.
      (setq target (if importance
                   node2
                   (if (my-exemplar-p node1)
                     (exemplar-category node1)
                     node1)))

      ;; Convert importance value to a qualitative form (high, moderate,
etc.).
      (setq qual-imp1 (if importance
                    (qualitative-value (cdr importance) 'importance)
                    'unknown))
      (format t "~%~%~A currently is of ~A importance to ~A"
            (getname feature)
            qual-imp1
            (getname target))

      ;; Ask teacher for a revised qualitative value of importance.
      (setq qual-imp2 (menu-select *importance-menu*))
      ;; If it's not the same as the old value then modify the importance.
      (if (and (not (equal 'nochange qual-imp2))
              (not (equal qual-imp1 qual-imp2)))
        (let ((new-imp  (cdr (assoc qual-imp2 *importance-alist*))))
          ;; If there was a previous importance cons ...
          (if importance
            ;; Then modify its strength
            (rplacd importance new-imp)
            ;; Else create a new importance.
            (push (cons feature new-imp) (category-importances target)))
          ;; Sort the category's importances.
          (setf (category-importances target)
              (sort (category-importances target)  #'>=  :key #'cdr)))))))
;;;-----------------------------------------------------------------
;;;  Function:    (get-unknown-features  exemplar)
;;;
```

```
;;;  Given: exemplar, a newly-created exemplar
;;;
;;;  Returns:     a list of features of the new exemplar whose importance
is
;;;          unknown to its category or immediate general categories.
;;;
;;;  Design:      For each feature of the new exemplar, this function looks
to
;;;          see if the importance of this feature is known:
;;;          -- idiosyncratically to the exemplar,
;;;          -- to the exemplar's category, or
;;;          -- to the next-most-general category(s).
;;;
;;;          If not found in any of these places, then the feature is
;;;          included in the list of unknowns that is returned.
;;;          If the feature is found in the next-most-general category,
;;;          then its importance is copied into the exemplar's category.
;;;
;;;                !*!*!*!  I need to ask Ray if this last step is wise.
!*!*!*!
;;;
;;;  Called by:
;;;-----------------------------------------------------------------
(defun get-unknown-features (exemplar)

  (let ((category  (exemplar-category exemplar))
        (unknowns  (copy-list (exemplar-features exemplar))))

;     ;; Delete from unknowns all features having idiosyncratic importance.
;     (dolist (imp (exemplar-importances exemplar))
;       (setq unknowns (delete (car imp) unknowns :count 1)))

    ;; Delete from unknowns all features having importance to the category.
    (dolist (imp (category-importances category))
      (setq unknowns (delete (car imp) unknowns :count 1)))

    ;; Return now if all featural importances are known.
    (if (null unknowns) (return-from get-unknown-features nil))

    ;; Delete from unknowns all features having importance to the
    ;;   next-most-general category(s).
    (dolist (rel (node-relations category))
      (if (eq *verb-hasTypicalGen* (relation-verb rel))
        (dolist (cat (relation-to-nodes rel))
          (dolist (imp (category-importances cat))
            (if (member (car imp) unknowns)
              (push imp (category-importances category)))
            (setq unknowns (delete (car imp) unknowns :count 1))
            (if (null unknowns) (return-from get-unknown-features nil))))))
    unknowns))



;;;-----------------------------------------------------------------
;;;  Function:    (discuss-high-importances  category  exemplar)
;;;
;;;  Given: -- category, the category of the new exemplar, and
;;;             -- exemplar, the newly-created exemplar
;;;
;;;  Purpose:      If this category already has another exemplar, i.e., a
;;;              prototype, then check every high-importance feature of the
;;;             prototype to see if it can be matched to a feature of the
;;;             new exemplar.  If not, ask the teacher if he/she wants to
;;;             lower the importance of that feature.
;;;
;;;  Notes: This procedure was part of the original Prolog Protos but
;;;             was not mentioned in Ray Bareiss's dissertation since it
;;;             was considered a detail.
;;;
;;;  Caller:      add-new-exemplar
;;;-----------------------------------------------------------------
(defun discuss-high-importances (category exemplar)
  (let ((prototype  (first (category-exemplars category)))
      (e-features (exemplar-features exemplar))
      (imps       nil))

    ;; If not in learning mode, then just return.
    (if (null *learning-mode*)
      (return-from discuss-high-importances (values)))

    ;; If the new exemplar IS the prototype, then just return.
    (if (eq prototype exemplar)
      (return-from discuss-high-importances (values)))

    ;; For each feature of known importance to the category ...
    (dolist (imp (category-importances category))
      (let* ((feature  (car imp))
           (strength (cdr imp)))
      ;; If the feature is of high importance and is NOT a feature of
      ;; this new exemplar ...
      (if (and (>= strength *importance-high*)
             (member feature (exemplar-features prototype))
             (not (member feature e-features)))
        (push imp imps))))

    (if (null imps)
      (return-from discuss-high-importances (values)))

    (setq imps (nreverse imps))
    (format t "~2%The following are features of the prototype of ~A (~A)~
             ~%that are currently of high importance to the category.~
             ~%Protos is going to see if each of these features matches~
             ~%a feature of the new exemplar."
        (getname category) (getname prototype))

    (let ((column 0))
      (dolist (imp imps)
      (if (= column 0) (terpri))
      (setq column (mod (1+ column) 3))
      (format t " ~25A" (getname (car imp))))
      (terpri))

    (dolist (imp imps)
      (let ((feature  (car imp))
          (strength (cdr imp)))

      (format t "~%    ~A ... " (getname feature))

      ;; Then see if feature is related to an exemplar feature.
      (let ((result (kbpm 'FtoF feature strength prototype e-features)))

        (princ (result-type result))
        ;; If not, then ask teacher to reassess (lower) the importance
        ;; of this feature to the category.
        (if (eql 'unmatched (result-type result))
            (let ((qual-imp1 (qualitative-value strength 'importance))
                  qual-imp2)
            (format t "~2%~A is a feature of ~A importance to category ~A~
                            ~%that does not match any feature of this new
exemplar.~
                            ~%The importance of ~A may be set too high."
                    (getname feature)
                    qual-imp1
                    (getname category)
                    (getname feature))

            ;; Ask teacher for a revised qualitative value of importance.
            (setq qual-imp2 (menu-select *importance-menu*))

            ;; If it's not the same as the old value then modify the
importance.
            (if (and (not (equal 'nochange qual-imp2))
                     (not (equal qual-imp1 qual-imp2)))
                (let ((new-strength  (cdr (assoc qual-imp2 *importance-
alist*))))
                    ;; Then modify its strength ...
                    (rplacd imp new-strength)
                    ;; and re-sort the category's importances.
                    (setf (category-importances category)
                        (sort (category-importances category)  #'>=  :key
#'cdr))))))))))


;;;------------------------------------------------------------
;;;  Function:    (discuss-relevances  exemplar)
;;;
;;;  Given: exemplar, a newly-created exemplar
;;;
;;;  Purpose:     Given a newly-created exemplar, this function attempts to
;;;         determine the relevance of each     feature whose importance to
;;;         the category (or to the exemplar itself) is unknown.
;;;
;;;  Notes: This function contains the top-level control for asking about
;;;         the relevance of features to a category.  Details are handled
;;;         in the subordinate functions search-explanation,
;;;         ask-for-explanation, and discuss-explanation.
;;;
;;;         This function gives the teacher the option of submitting an
;;;         explanation before Protos attempts to find one itself.  This
can
;;;         save a lot of time in those situations where the teacher KNOWS
;;;         that Protos will not be able to find a feature-to-category
;;;         explanation (unsuccessful searches are time-consuming).
;;;
;;;  Caller:      add-new-exemplar
;;;------------------------------------------------------------
(defparameter *choose-explanation* (make-menu
      :label  nil
      :items  '((#¥E  . ("Enter an explanation."                return
enter))
            (#¥L  . ("let protos Look for an explanation."  return look))
            (#¥S  . ("declare the  feature Spurious."           return
spurious))
            (#¥U  .  ("leave  this  feature  Unexplained."         return
unexplain)))))



(defun discuss-relevances (exemplar)

  (let ((category  (exemplar-category exemplar))
        (unknowns  (get-unknown-features exemplar))
        (ask-first nil))

    ;; If all features accounted for, then there is nothing to do.
    (if (null unknowns) (return-from discuss-relevances nil))

    (format t  "~%~%Protos is currently unaware of the relevance~
              ~%of ~D feature~:P to the category ~A."
          (length unknowns)  (getname category))

    ;;  If  several  unknowns,  see  if  teacher  wants  to  be  asked  about  each
one.
    ;; If only 1 or 2 unknowns, then just go ahead and ask about each one.
    (if (< (length unknowns) 3)
      (setq ask-first t)
      (if (prompt "~%~%Should Protos ask you about each feature~
                   ~%before it tries to find an explanation itself? "
                    nil 'y-or-n nil nil)
          (setq ask-first t)))

    ;; Main loop -- determine the relevance of each unknown feature.
    (dolist (feature unknowns)

      ;; If teacher is to be prompted about each feature ...
      (if ask-first
        ;; then prompt the teacher ...
        (progn
          (format t "~2%What is the relevance of ~A to ~A ?~
                        ~%-----------------------------------------------------"
                (getname feature)  (getname category))

          ;; The teacher is given 3 choices:
          ;;        1 -- Protos will search for an explanation.
          ;;        2  --  Protos  will  ask  for  an  explanation  from  the
teacher.
          ;;        3 -- Teacher declares that the feature is spurious.

          (case (menu-select *choose-explanation*)
            (look   (search-explanation   feature  category t))
            (enter        (ask-for-explanation  feature  category))
            (spurious     (set-spurious            feature  category))
            (unexplain  nil)
            (otherwise  (format t "Error: discuss-relevances: menu returned
unexpected value."))))

        ;; else search the category network for an explanation.
        (search-explanation  feature  category t)))))
```

```
;;;-----------------------------------------------------------------
;;;  Function:    (search-explanation  feature  category  first-time)
;;;
;;;  Purpose:     This function tries to find an explanation relating the
given
;;;          feature to the given category.
;;;
;;;  Design:       -- Do search for explanation.
;;;          -- If found, call discuss-explanation,
;;;             else explain that no explanation was found and ask teacher
;;;                 if he/she wants to give an explanation.
;;;          -- If yes, call ask-for-explanation,
;;;             else declare the feature to be spurious.
;;;-----------------------------------------------------------------
(defun search-explanation  (feature  category  first-time)

  (let ((result (kbpm 'FtoT feature *importance-big* category (exemplar-
features *cur-exemplar*))))

    (case (result-type result)

      (explained  (discuss-explanation feature category (result-explanation
result)))

      (excluded   nil)  ; don't learn a reminding or importance here.

      (unmatched   (format t "~%Protos ~:[still~;~] could not find an
explanation.~
                                ~%What is the relevance of ~A to ~A ?
                          ~%------------------------------------------------------
"
                           first-time  (getname feature)  (getname category))

                     (case (menu-select *choose-explanation*)
                       (look      (search-explanation   feature  category nil))
                       (enter     (ask-for-explanation  feature  category))
                       (spurious       (set-spurious           feature  category))
                       (unexplain    nil)
                       (otherwise     (format t "Error: search-explanation: menu
returned unexpected value."))))

      (spurious     (set-spurious feature category))

      (otherwise   (format t "~%Error: search-explanation: result-type =
~A~%"
                          (result-type result))
                  (set-spurious feature category)))))


(defun set-spurious (feature category)
  (push (cons feature 0.0) (category-importances category)))


(defparameter *new-term-menu* (make-menu
  :label   "Is this a:"
  :items   '((#¥T  . ("New  term."                                     return
new-term))
           (#¥A . ("Abbreviation for an existing term."  return abbrev))
           (#¥S . ("Synonym for an existing term."       return synonym))
           (#¥Q . ("Quit (return to previous menu)."     return quit)))))
```

```
(defun enter-new-terms ()
  (let (action termname termname2 node)
    (loop
      (setq termname (prompt "~%~%Enter term name (terminate with blank
line) ---> "
                             nil 'termname nil nil))
      (if (null termname) (return (values)))
      (setq action (menu-select *new-term-menu*))
      (case action
      (new-term
        (check-term-name termname 'ask))
      ((abbrev synonym)
          (setq termname2 (prompt "~%   Enter name of existing term ---> "
nil
                                   'termname nil nil))
          (if (null termname2)  (return (values)))
          (if (not (symbolp termname2))
            (progn
              (format t "~%   Error: ~A is not a symbol." termname2)
              (return (values))))
          (if (not (boundp termname2))
            (progn
              (format t "~%   Error: ~A is unknown." termname2)
              (return (values))))
          (setq node (eval termname2))
          (if (not (node-p node))
            (progn
              (format t "~%   Error: ~A is a ~A, not a node."
                    termname2 (type-of node))
              (return (values))))
          (if (eql action 'abbrev)
            (setf (node-abbrev node) termname)
            (push termname (node-synonyms node)))
          (set termname node))
      (quit
        (return-from enter-new-terms (values)))))))
```

```
;;;-----------------------------------------------------------------
;;;  Function:     (ask-for-explanation  feature  category)
;;;
;;;  Purpose:      This function prompts the teacher for an explanation
relating
;;;           the given feature to the given category (or, the teacher can
;;;           give the missing fragment that will allow Protos to find an
;;;           explanation from feature to category.
;;;
;;;  Design:       -- Get explanation from teacher.
;;;           -- Verify that the given explanation provides a path through
;;;              the category network from the feature to the category.
;;;           -- If so, call discuss-explanation,
;;;              else harass the teacher for a correct explanation.
;;;-----------------------------------------------------------------
(defun ask-for-explanation (feature category)

  (let (explanation)

    (format t
          "~%~%Please explain how ~A is related to ~A (preferred)~
             ~%or how ~A is related to ~A (alternate form).~%"
          (getname feature)   (getname category)
          (getname category)  (getname feature))

    (setq explanation (enter-explanation))
    (if (null explanation) (return-from ask-for-explanation nil))

    (setq *feat* feature)
    (setq *cat* category)
    (setq *exp* explanation)
    ;; Try to verify the explanation by the simple syntactic strategy
    ;; of noticing if the explanation begins with 'feature' and ends
    ;; with 'category' (or vice versa).  If so, then we don't have to
    ;; do the more time-consuming search through the category network.

    (let ((to-nodes    (get-leaves explanation t nil))
          (start-term  (explanation-start-term explanation)))

      (cond  ;; Is this a feature-to-category explanation?
            ((and (eq feature start-term)  (member category to-nodes))
             (discuss-explanation  feature  category  explanation))

            ;; Is this a category-to-feature explanation?
            ((and (eq category start-term)  (member feature to-nodes))
             (discuss-explanation  feature  category  explanation))

            ;; Apparently this is a fragment, so must search the C.N.
            (t  (search-explanation feature category t))))))
```

```
;;;-----------------------------------------------------------
;;;  Function:    (discuss-explanation  feature  target  explanation)
;;;
;;;  Given: an explanation which may be either feature-to-target or
;;;         target-to-feature (where "target" is a category or exemplar).
;;;
;;;  Do:    Set both a reminding and an importance from the explanation.
;;;         The reminding comes from the feature-to-target explanation
;;;         and the importance from the target-to-feature explanation.
;;;         Simply invert the given explanation to get the other type.
;;;
;;;  Note:      If the target is related to a more general category, then
;;;               set-reminding will ask the user if the reminding should
really
;;;             go to the more general category.  If the user agrees, then
;;;             set-reminding returns the new (reminded) category so that
;;;               set-importance will install the importance in that new
category.
;;;
;;;  Callers:   ask-for-explanation, search-explanation
;;;-----------------------------------------------------------

(defun discuss-explanation  (feature  target  explanation)

  (let ((from-leaves  (get-leaves explanation nil nil))
      inverse-explanation)

    ;; If this is a feature-to-[category or exemplar] explanation ...
    (if (member feature from-leaves)
      ;; then set reminding, invert explanation, and set importance
      (progn
        (multiple-value-setq (target explanation)
          (set-reminding feature target explanation))
        (if target
            (progn
            (setq inverse-explanation (invert-explanation explanation))
            (set-importance feature target inverse-explanation))))

      ;; else this must be a [category or exemplar]-to-feature explanation,
      ;; so invert the explanation, set reminding, then set importance.
      ;; The reason why the reminding is always set before the importance
      ;; is that 'set-reminding' may truncate the explanation, changing
      ;; the target.
      (progn
        (setq inverse-explanation (invert-explanation explanation))
        (multiple-value-setq (target explanation)
          (set-reminding feature target inverse-explanation))
        (if target
            (set-importance feature target explanation))))))
```

```
;;;-------------------------------------------------------------
;;;  Function:    (set-reminding  feature  target  explanation)
;;;
;;;  Given: -- feature, the feature causing this reminding;
;;;         -- target, the category or exemplar that is the object of
;;;            the reminding to be installed;
;;;         -- explanation, a feature-to-target explanation.
;;;
;;;  Purpose:    Given  a  feature-to-category  or  feature-to-exemplar
explanation,
;;;         this function will install a reminding, subject to several
;;;         heuristics.
;;;
;;;  Returns:   Two values:
;;;                -- The  target  category  or  exemplar  that  the  reminding
really
;;;                points to.  It can be different than the supplied target
if
;;;                 the  target  is  a  category  and  the  teacher  agrees  to  let
the
;;;                reminding be to a more general category.
;;;             -- The  explanation  from  feature  to  actual  target.   This  may
be
;;;                a truncated version of the supplied explanation.
;;;
;;;   Heuristics:   --  If  the  explanation  contains  a  mutual  exclusion
relation,
;;;               then no reminding is installed.
;;;            -- If the target of the explanation is an exemplar, then
;;;               install the reminding without applying further heuristics.
;;;            -- If the explanation contains a "has-specialization",
;;;               "has-part", or a weak link, then truncate the explanation
;;;               at that point (thus changing the target of the reminding).
;;;-------------------------------------------------------------
(defun set-reminding  (feature  target  explanation)

  (let (quality)
    ;; If mutual exclusion found, don't install reminding.
    (if (check-mutex explanation)
      (return-from set-reminding (values nil nil)))

    ;; If target of explanation is an exemplar,
    ;; install the reminding and return.
    (if (my-exemplar-p target)
      (progn
        (setq quality (explanation-strength explanation))
        (set-reminding2 feature target quality)
        (return-from set-reminding (values target explanation))))

    ;; Apply heuristics for truncating the explanation.
    (setq explanation (truncate-explanation explanation))

    ;; If explanation truncated down to nothing, just return.
    (if (null explanation)
      (return-from set-reminding (values nil nil)))

    ;; Install reminding to target(s) of the possibly-truncated explanation.
    (setq quality (explanation-strength explanation))

    ;; For each target of this explanation ...
    (dolist (e-target (get-leaves explanation t nil))
      (set-reminding2 feature e-target quality))
    (values target explanation)))


;;;-----------------------------------------------------------------
;;;  Function:  (set-reminding2  feature  target  quality)
;;;  Modified by  Joaquin Delgado, 28 Aug. 1995
;;;
;;;   Purpose:    This function installs a reminding from 'feature' to
'target'
;;;                with the given 'strength' if there is not already a
reminding
;;;              between 'feature' and 'target'.  If there is, then it is
updated
;;;             only if this new reminding is stronger.
;;;-----------------------------------------------------------------

(defun set-reminding2 (feature target quality)
  (let ((rem (assoc target (feature-remindings feature))))

    ;; if there is an existing reminding to this target ...
    (if rem

      ;; then update the reminding strength only if this one is stronger
      (if (> quality (cdr rem))
          (progn
            (rplacd rem quality)
            (if *trace-new-remindings*
              (print-reminding feature target quality t))))

      ;; else create a reminding to it or a more general target.
      (progn
        (if (not (my-exemplar-p target))
            (setq target (select-reminding-target feature target quality)))
;;Original -----------------------------------------------------------
;;      (push (cons target quality) (feature-remindings feature))
;;      (if *trace-new-remindings*
;;          (print-reminding feature target quality nil))
;;------------------------------------------------------------------
;; Modified by Joaquin Delgado on Aug 28, 1995

;; IF target is a list
          (if (listp target)
            (dolist (e-target target)
            (push (cons e-target quality) (feature-remindings feature))
            (if *trace-new-remindings*
                (print-reminding feature e-target quality nil))
                (setq quality (/ quality 4)))
              ;;    The strenght of the reminder will be  0.25% less for
more
              ;;   general categories.
;; ELSE
              (progn
                (push (cons target quality) (feature-remindings feature))
            (if *trace-new-remindings*
              (print-reminding feature target quality nil))))

        (trim-remindings feature)))))

;;;-----------------------------------------------------------------
;;;  Function:  (select-reminding-target  feature  target  quality)
;;;  Modified by Joaquin Delgado on Aug 28, 1995
```

```
;;;
;;;  Given:     -- a feature that evokes a reminding,
;;;             -- a potential target for that reminding, and
;;;             -- the quality (strength) of the reminding.
;;;
;;;  Returns:   the actual target for the reminding, which may be either
the
;;;                    given target or a more general category in the
generalization,
;;;             functional, or partonomic hierarchies.  If the given target
has
;;;              one or more more-general categories, then the teacher is
asked
;;;             to select the desired target.
;;;-------------------------------------------------------------------

(defun select-reminding-target (feature target quality)
  (let ((relatives (collect-relatives target
                               (list *verb-hasTypicalGen*
                                     *verb-hasFunction*
                                     *verb-partOf*)
                               nil)))

    ;; If this target has no categories that are more general, then just
    ;; return the original target.
    (if (null relatives)
      (return-from select-reminding-target target))

    (format *query-io* "~%Protos is ready to install a ~4,2F reminding from
~A to ~A."
          quality (getname feature) (getname target))

    ;; IF only one more general category ...
    (if (null (cdr relatives))

      ;; THEN ask about it specifically

;;Original -----------------------------------------------------------
;;     (if (y-or-n-p "~%~%Should it instead install it to the more general
category ~A ? "
;;                  (getname (car relatives)))
;;          (setq target (car relatives)))
;;-------------------------------------------------------------------
;; Modified by Joaquin Delgado on Aug, 28 1995

      (if (y-or-n-p "~%~%Should it also install it to the more general
category ~A ? "
                  (getname (car relatives)))
          (setq target (list target (car relatives))))
           ;; target is a list of all the posible targets that the user
considers important
;-------------------------------------------------------------------
      ;; ELSE ask for a selection from the list of more general categories.
      (progn

;;Original-----------------------------------------------------------
;;      (format *query-io* "~%~%Should it instead install it to one of the
more general categories:~%")
;; Modified by Joaquin Delgado on Aug 29 1995
          (format *query-io* "~%~%Should it also install it to the more
general categories:~%")
;;-------------------------------------------------------------------
```

```
         (print-node-names relatives *query-io* " or ")
         (format *query-io* " ")
         (if (y-or-n-p)
;; added by Joaquin Delgado
              (progn
              (loop
;;Original -----------------------------------------------------------
;;                 (format *query-io* "~%Then please specify which one: ")
;;                 (setq target (check-term-name (read *query-io* nil nil)
'fail))
;;                 (if (member target relatives) (return (values)))
;;                 (format *query-io* "~%Error: you must enter one of the
above terms.")))))
;; Modified by Joaquin Delgado on Aug 29 1995

                  (setq temp (prompt "~%Please specify which ones, in
order of importance (from less to most)~
                            ~%one category per line (terminate with
blank line): ~%"
                             "~&---> " 'symbol nil nil))
                   (dolist (e-target temp)
                      (setq e-target (check-term-name e-target 'fail))
                      (if (not (member e-target  relatives))
                        (progn
                          (print-node-names relatives *query-io* " or
")
                          (format *query-io* "~%Error: you must enter
categories among of the above terms.")
                          (read-char)
                          (setq temp nil)
                          (return))
                        (progn
                          (if (not (listp target))
                            (setq   temp2   (list   target)))

                          (push e-target temp2))))
                   (if (not (null temp))
                          (return)))
               (setq target (reverse temp2))))))

;;-------------------------------------------------------------------
    target))


;;;------------------------------------------------------------------
;;;  Function:  (check-mutex  explanation)
;;;
;;;  Purpose:   Given an explanation, this function returns T if a mutual-
;;;                 exclusion relation is found anywhere within the
explanation.
;;;             Otherwise, it returns NIL.
;;;------------------------------------------------------------------
(defun check-mutex (explanation)
  (if (eq *verb-MEx* (relation-verb (explanation-relation explanation)))
      (return-from check-mutex t))
  (dolist (term (explanation-from-terms explanation))
    (if (explanation-p term)
      (if (check-mutex term)
          (return-from check-mutex t))))
  (dolist (term (explanation-to-terms explanation))
    (if (explanation-p term)
      (if (check-mutex term)
          (return-from check-mutex t))))
    nil)
```

```
;;;-------------------------------------------------------------------
;;;  Function:    (set-importance  feature  target  explanation)
;;;
;;;  Given: -- feature, the feature whose importance value is to be set;
;;;         -- target, the category or exemplar where the importance is
;;;            to be stored;
;;;         -- explanation, a category-to-feature or exemplar-to-feature
;;;            explanation.
;;;
;;;  Purpose:    Given  a  category-to-feature  or  exemplar-to-feature
explanation,
;;;         this  function  sets  the  importance  of  the  feature  in  the
category
;;;         or exemplar, respectively.
;;;
;;;  Notes: -- The importance value (a number in the range 0 - 1.0) is
;;;            taken from the strength of the explanation.
;;;
;;;  Callers:    discuss-explanation
;;;-------------------------------------------------------------------

(defun set-importance  (feature  target  explanation)

  (if (null explanation) (return-from set-importance (values)))

  (let* ((quality    (explanation-strength explanation))
       (qual-imp1  (qualitative-value quality 'importance))
       (category   (if (my-exemplar-p target)
                   (exemplar-category target)
                   target))
       qual-imp2)

    ;; Show explanation to teacher for possible modification.
    (format t "~%~%Protos believes that ~A is of ~A importance to ~A~
             ~%based on the explanation:~
                 ~%    "
          (getname feature)  qual-imp1 (getname category))
    (print-explanation explanation)

    ;; See if teacher agrees with this value of importance.
    (case (menu-select *decide-importance-menu*)
      (yes          ;; Teacher agrees with importance, so nothing to do.
      nil)

      (revise-expl  ;; Teacher wants to revise the explanation.
        (prog ()
         TOP
         (setq explanation
                (get-specific-explanation   feature   target   'TtoF
(exemplar-features *cur-exemplar*)))
           (if (null explanation)
             (progn
              (format t "~%~%Protos could not find an explanation from ~A
to ~A"
                 (getname feature) (getname category))
              (if (y-or-n-p "~%Do you wish to revise your explanation
again? ")
                  (go TOP)))
             (print-explanation explanation))
         (set-importance feature target explanation)
         (return-from set-importance (values))))
        (revise-imp   ;; Teacher wants to just revise the importance.
        (setq qual-imp2 (menu-select *importance-menu*))
        (if (and (not (equal 'nochange qual-imp2))
                 (not (equal qual-imp1 qual-imp2)))
            (setq quality  (cdr (assoc qual-imp2 *importance-alist*))))))

      ;; Install the feature's importance in the category.
      (push (cons feature quality) (category-importances category))))


;;;-----------------------------------------------------------------
;;;  Function:  (discuss-unmatched-importances  match)
;;;
;;;  Given:     a match of an exemplar of a wrong category to a newcase;
;;;
;;;  Purpose:   This function examines the exemplar's unmatched features to
;;;             determine which ones are believed to be of moderate or low
;;;                importance, since it is possible that the reason the
feature is
;;;             unmatched is because the importance was set too low.
;;;             Protos then asks the teacher to reassess the importance
;;;             of each of these features.  The teacher is told the current
;;;              qualitative strength of the feature's importance and then
asked
;;;             to revise the importance to be either "necessary", "high",
;;;             "moderate", "low", or "spurious".
;;;-----------------------------------------------------------------

(defun discuss-unmatched-importances (match)
  (let ((results    (match-results match))
        (first-time t))
    (dolist (result results)
      (if (and (eql 'unmatched (result-type result))
               (< (result-importance result) *importance-high*))
          (progn
            (if first-time
            (progn
            (setq first-time nil)
            (format t
     "~%~%The incorrectly matched exemplar contains unmatched features~
        ~%of moderate or low importance.  A possible reason that this case~
        ~%was classified incorrectly is because one or more of those
features~
        ~%is really more important than Protos currently believes.")))
            (reassess-importance (result-feature result) (match-exemplar
match))))))
```

```
;;;---------------------------------------------------------------
;;;  Function:    (get-approximate-importances  exemplar)
;;;
;;;  Given: exemplar, a newly-created exemplar
;;;
;;;  Returns:     an alist of (feature . importance) with one entry for
each
;;;           feature of the exemplar, sorted in decreasing order of
;;;           importance.
;;;
;;;  Caller:      discuss-relevances
;;;
;;;  Purpose:     The decision to create this function was based on the
belief
;;;           that when Protos discusses the relevance of each feature to its
;;;           category, it is best to discuss the most important features
;;;           first.  Without this function, Protos would present the
features
;;;           in the order in which they occurred in the new case (which may
;;;           bear no resemblance to importances).
;;;
;;;  Design:      For each feature, this function looks for its importance
value
;;;           in the following places (in this order):
;;;           -- in the exemplar's category;
;;;           -- in neighboring categories (immediate neighbors only);
;;;           -- otherwise, the value 0 is assigned.
;;;---------------------------------------------------------------
(defun get-approximate-importances (exemplar)

  (let* ((features1      (copy-list (exemplar-features exemplar)))
      (features2  (copy-list features1))
      (category   (exemplar-category exemplar))
      (c-importances    (category-importances category))
      (importances      nil))

    ;; ------------- Extract importances from the category. ----------------
    (dolist (feature features1)
      (let ((pair (assoc feature c-importances)))
      (if pair
          (progn
            (push pair importances)
            (setq features2 (delete feature features2)))))))

    (if (null features2)
      (return-from  get-approximate-importances
            (sort importances  #'>  :key #'cdr)))

    (setq features1 (copy-list features2))


    ;;  ---------  Extract  importances  from  neighboring  categories.
----------
    (dolist (relation (node-relations category))
      (dolist (to-node (relation-to-nodes relation))
      (if (my-category-p to-node)
          ;; -------- Extract importances from this neighbor. -----------
          (let ((c-importances (category-importances to-node)))
            (dolist (feature features1)
            (let ((pair (assoc feature c-importances)))
              (if pair
                (progn
                (push pair importances)
                (setq features2 (delete feature features2))))))

        (if (null features2)
          (return-from  get-approximate-importances
                (sort importances  #'>  :key #'cdr)))

        (setq features1 (copy-list features2))))))


;; ------- Assign zero importance to any remaining features. ----------
(dolist (feature features2)
  (push (cons feature 0.0) importances))

(return-from  get-approximate-importances
        (sort importances  #'>  :key #'cdr))))
```

```
;;;-----------------------------------------------------------------
;;;  Function:    (discuss-unmatched  match)
;;;
;;;  Given: match, an approved exemplar-to-newcase match;
;;;
;;;  Does:  For each non-spurious unmatched feature of the exemplar,
;;;         the teacher is asked to explain how a feature of the new case
;;;         is related to the unmatched exemplar feature.  The teacher
;;;         may either enter an explanation or enter "none", "n" or "no".
;;;
;;;  Caller:      discuss-success
;;;-----------------------------------------------------------------

(defun discuss-unmatched (match)

  (let ((introduction t)      ; Print introduction only on first feature.
        (results  (match-results match)))

    ;; Iterate through the match results looking for unmatched features.
    (do* ((results results (cdr results))
          (result  (car results) (car results)))
         ((endp results))
      (let* ((feature    (result-feature result))      ; pointer to feature
node
             (importance (result-importance result))    ; importance of this
feature
             (quality    (result-quality result))
               (unmatched    (match-unmatched match)))          ; unmatched
features

      (if (eql 'unmatched (result-type result))
          (progn
            (if introduction
              (progn
                (setq introduction nil)
                (format t
      "~%~%Protos would like to improve this classification by discussing~
       ~%some of the exemplar features it could not account for.~%")))

            (if (prompt (format nil
                        "~%Protos could not account for the exemplar
feature ~A.~
                                    ~%Is ~A related to any feature of the new
case? "
                          (getname feature)  (getname feature))
                     nil 'y-or-n nil nil)
                (let (newresult)
                  (format t "~%Please explain how ~A is related to some
feature of the new case."
                          (getname feature))
                  (enter-explanation)
                  (setq newresult (compare-feature feature
                                                   importance
                                                   (match-exemplar match)
                                                   (match-newcase match)))
                  (format t "~%~%The exemplar feature ~A now has the
following match:"
                        (getname feature))
                  (print-result 0 newresult)

                  ;; Replace the unmatched result with the explained result.
```

```
                    (rplaca results newresult)

                      ;; Depending on the type of result, adjust the list
                      ;; of unmatched case features.
                      (ecase (result-type newresult)
                        (identical
                        (setq unmatched (delete feature unmatched)))
                        (explained
                        (setq unmatched (nset-difference
                        unmatched (explanation-from-terms (result-explanation
newresult)))))
                        (excluded
                        (setq unmatched (nset-difference
                        unmatched (explanation-from-terms (result-explanation
newresult)))))
                          (unmatched nil)
                          (spurious  nil))

                        (setf (match-unmatched match) unmatched)

                    ;; Update the overall similarity by dividing by the old
                    ;; similarity and multiplying by the new similarity.
                    (setf (match-similarity match)
                        (* (/ (match-similarity match) quality) (result-quality
newresult)))
                    ))))))))

(defun discuss-unfocused ()
    ;; Ask for unfocused instruction.
    (format t "~%~%If there is any other instruction that you wish to
provide,~
                ~%you may now do so from the following menu:~%")
    (menu-select-2 '*unfocused-instruction-menu*))
```

```
;;;-----------------------------------------------------------------
;;;  Function:  (discuss-failure  match)
;;;
;;;  Given: match, the results of an exemplar-to-new case match where the
;;;               the teacher has rejected the match because it is to a
;;;                   wrong category.
;;;
;;;  Returns:     -- the same match with the action slot set to 'next if
the
;;;               teacher has said to try the next hypothesis;
;;;         -- the same match with the action slot set to 'done if the
;;;                 teacher decided either to create a new exemplar from
this
;;;               case or just abandon this case.
;;;
;;;  Caller:      discuss-match
;;;
;;;  Design:      -- Reassess remindings that led to this (incorrect)
category.
;;;         -- Ask for any revisions to the new case.
;;;         -- Ask for censors.
;;;         -- Ask for reassessment of importances of low-importance
;;;            unmatched features.
;;;         -- If strong match then record for later difference links.
;;;         -- Allow unfocused instruction.
;;;         -- Ask teacher whether to install this case as a new exemplar
;;;            or discard it and go on to the next hypothesis.
;;;-----------------------------------------------------------------

(defparameter *failure-menu1* (make-menu
  :label  "~%~%What do you want to do with this case now?~
           ~%-------------------------------------------"
  :items  '((#¥C . ("Create an exemplar from this case"   return  create))
           (#¥R . ("Retry finding a match for this case" return  retry))
           (#¥M . ("reMatch this case to same exemplar"  return  redo))
           (#¥T . ("Try next hypothesis"                 return  next))
           (#¥Q  .  ("Quit  (abandon  this  case)"               return
abandon)))))


(defun discuss-failure (match)
  (declare (special closed-categories))

  ;; The teacher has rejected this match because it is to the wrong
category.
  ;; So, remember that fact so that the function "classify2" will skip over
  ;; any other hypothesized exemplars of the same category.
  (push (exemplar-category (match-exemplar match)) closed-categories)

  (format t "~%~%Protos will now try to learn from this error.")

  ;; If this incorrect match is strong enough (i.e., is a "near miss"),
  ;; then remember it for possibly installing a difference link later
  ;; when the case finally is correctly classified.
  (if (>= (match-nth-root-of-similarity match) *near-miss-threshold*)
      (push match *near-misses*))

  (reassess-remindings match)
  (discuss-censors match)
  (discuss-unmatched-importances match)
  (discuss-unfocused)
  (case (menu-select *failure-menu1*)
    (create    (values 'newex (add-new-exemplar (match-newcase match))))
    (retry     (values 'retry match))
    (redo      (values 'redo  match))
    (next      (values 'next  match))
    (abandon   (values 'done  nil))))


(defparameter *censor-strength-menu* (make-menu
      :label  nil
      :items  `((#¥W . ("Weak"       return  ,-0.25))
            (#¥M . ("Moderate"   return  ,-0.50))
            (#¥S . ("Strong"     return  ,-0.75))
            (#¥A . ("Absolute"   return  ,*absolute-censor*)))))


(defun discuss-censors (match)
  (let* ((exemplar  (match-exemplar match))
         (category  (exemplar-category exemplar))
         (newcase   (match-newcase match))
         (features  (case-features newcase))
         input
         feature
         strength)

    (format *query-io* "~2%The features of the new case are:~%")
    (print-node-names features *query-io* ", ")
    (if (prompt (format nil  "~2%Are  any  of  these  features  mutually
exclusive~
                            ~%with the category ~A? "
                  (getname category))
            nil 'y-or-n nil nil)
      (progn
        (loop
          (format *query-io*
              "~%Please enter their name(s), enclosed in parentheses (or
type nil).~
                    ~%---> ")
          (setq input (read *query-io* nil nil))
          (if (or (null input) (listp input))
            (return (values)))
          (format *query-io* "~%Input error!"))

        (dolist (fname input)
          (setq feature (check-term-name fname 'ask))
          (if feature
            (progn
              (format  *query-io*  "~%Please  rate  the  strength  of  ~A's
negative evidence:"
                      (getname feature))
              (setq strength (menu-select *censor-strength-menu*))
              (push (cons category strength) (feature-remindings feature))
              (if *trace-remindings*
                  (print-reminding feature category strength nil)))))))))



(defun add-censor ()
  (let (name feature target strength)
    (format *query-io*
           "~%A censor lets a feature ¥"discount¥" a category or exemplar.~
              ~%Please enter a feature name ----------------> ")
    (setq name (read *query-io* nil nil))
    (if (null name)
      (return-from add-censor nil))
    (setq feature (check-term-name name 'ask))
    (if (null feature)
      (return-from add-censor nil))
    (format *query-io* "~%Please enter a category or exemplar name ---> ")
    (setq name (read *query-io* nil nil))
    (if (null name)
      (return-from add-censor nil))
    (setq target (check-term-name name 'ask))
    (if (null target)
      (return-from add-censor nil))
    (format  *query-io*  "~%Please  rate  the  strength  of  ~A's  negative
evidence:"
           (getname feature))
    (setq strength (menu-select *censor-strength-menu*))
    (push (cons target strength) (feature-remindings feature))
    (if *trace-remindings*
      (print-reminding feature target strength nil))))


(defun change-importance ()
  (let (name feature target)
    (format *query-io* "~%Enter name of feature --------> ")
    (setq name (read *query-io* nil nil))
    (if (null name)
      (return-from change-importance (values)))
    (setq feature (check-term-name name 'ask))
    (if (null feature)
      (return-from change-importance (values)))
    (format *query-io* "~%Enter category or exemplar ---> ")
    (setq name (read *query-io* nil nil))
    (if (null name)
      (return-from change-importance (values)))
    (setq target (check-term-name name 'ask))
    (if (null target)
      (return-from change-importance (values)))
    (reassess-importance feature target)))


(defun show-relations ()
  (format *query-io* "~%Please enter the name of any term ---> ")
  (let* ((name  (read *query-io* nil nil))
          (relations nil)
          node)
    (if (and (symbolp name) (boundp name))
      (setq node (eval name))
      (setq node (check-term-name name 'fail)))
    (if (and node (node-p node))
      (progn
        (setq relations (node-relations node))
        (if relations
            (progn
            (format t "~%Relations that begin with ~A:" name)
            (dolist (rel relations)
              (format t "~%     ")
              (print-relation rel t 1)))
        (format t "~%No relations involve this ~:[term~;predicate~]."
              (predicate-p node))))
  (format t "~%~A is not a term or predicate!" name))))
```

Appendix C

House knwoledge base produced by Protos

This knowledge base was generated by the modified Protos program, as the result of learning about general geometric forms of different examples of houses in the more general HOUSE domain.

```
;;; -*- Mode: Lisp; Syntax: Common-lisp; Base: 10; -*-
;;; Modified by Joaquin Delgado 08/23/95
;;; Comments: - Remindings to higher categories and classes were
;;;             first introduced by hand. PROTOS only sets remindings
;;;         one at a time, and does not propagate remindings in higher
;;;             herarchies
;;;

(term :name          walls
      :remindings    ((house 0.67))
      :importances ((cylinder 0.48) (cube 0.48))
      :exemplars     (walls1 walls2)
      :relations     (((walls equivalent story_2) "joaquin"
                     "Story 2 are walls of the second floor")
                        ((walls equivalent story_1) "joaquin"
                     "Story_1 are the walls of the first floor")
                        ((walls is part of house) "Joaquin" "Walls are part of
a house ")
                        ((walls sometimes is inferred from cube) "Joaquin"
                     "A cube suggest a house's base because of it's form.")
                        ((walls sometimes is inferred from cylinder) "Joaquin"
                     "A cylinder suggest a base of a house because of it's
form.")))

(term :name          roof
      :remindings    ((house 0.70))
      :importances ((triangle_base_pyramid 0.48) (rectangular_sheet 0.48)
                   (semi-cylinder 0.48)(square_base_pyramid 0.48) (prism
0.48))
      :exemplars     (proto_roof roof5 roof2 roof1
                      roof3 roof4)
      :relations     (((roof is part of house) "Joaquin" "A roof is part of a
house")
                        ((roof sometimes is inferred from prism) "Joaquin"
                     "A prism always suggest a roof because of its form")
                        ((roof sometimes is inferred from cone) "Joaquin"
                     "Cone sometimes suggest roof because of it's form")
                        ((roof sometimes is inferred from square_base_pyramid)
"Joaquin"
                     "A square base pyramid suggest roof because of it's form")
                        ((roof sometimes is inferred from semi-cylinder)
"Joaquin"
                     "A semi cylinder suggest roof because of it's form")
                        ((roof sometimes is inferred from rectangular_sheet)
"Joaquin"
                     "A rectangular sheet suggest roof because it's form")
                        ((roof sometimes is inferred from
triangle_base_pyramid) "Joaquin"
```

```
                    "A triagle base pyramid suggest roof because of it's
form")))

(term :name         house
      :importances ((story_2 0.70) (rectangular_sheet 0.34) (semi-cylinder
0.34)
                     (square_base_pyramid 0.34) (cylinder 0.34) (cone 0.34)
                     (cube 0.95) (prism 0.78) (roof 0.70)
                     (walls 0.70))
      :exemplars    (proto_house humble_house dome_house indian_house
                     small_house two_story_house)
      :relations    (((house has part roof) "Joaquin" "A roof is part of a
house")
                     ((house has function (shelter person)) "Joaquin"
                    "A house has the function of shelter for persons")
                     ((house has part walls) "Joaquin" "walls are part of a
house ")))

(term :name         proto_house
      :comment      "A general abstract house "
      :category     house
      :features     (walls roof)
      :typicality   6.50)

(predicate :name         shelter)

(term :name         person)

(term :name         (shelter person)
      :relations     ((((shelter person) sometimes is function of house)
"Joaquin"
                    "A house has the function of shelter for persons")))

(term :name         prism
      :remindings   ((walls -9999.00) (roof 0.78) (house 0.01))
      :relations    (((prism always suggests roof) "Joaquin"
                    "A prism always suggest a roof because of its form")))

(term :name         proto_roof
      :comment      "(This is a prototypical roof)"
      :category     roof
      :features     (prism)
      :typicality   2.25
      :differences  ((proto_house <-- (cube))))

(term :name         cone
      :remindings   ((walls -.75) (house 0.34))
      :relations    (((cone sometimes suggests roof) "Joaquin"
                    "Cone sometimes suggest roof because of it's form")))

(term :name         roof1
      :comment      "(Yet another example of roof)"
      :category     roof
      :features     (cone)
      :typicality   1.00)

(term :name         square_base_pyramid
      :remindings   ((walls -.75) (roof 0.80) (house 0.56))
      :relations    (((square_base_pyramid suggests roof) "Joaquin"
                    "A square base pyramid suggest roof because of it's
form")))
```

```
(term :name         roof2
      :category     roof
      :features     (square_base_pyramid)
      :typicality   1.00)

(term :name         semi-cylinder
      :remindings   ((roof 0.45) (house 0.34))
      :relations    (((semi-cylinder sometimes suggests roof) "Joaquin"
                  "A semi cylinder suggest roof because of it's form")))

(term :name         roof3
      :comment      "(Yet another example of roof)"
      :category     roof
      :features     (semi-cylinder)
      :typicality   1.00)

(term :name         rectangular_sheet
      :remindings   ((walls -.75) (roof 0.48) (house 0.34))
      :relations    (((rectangular_sheet sometimes suggests roof) "Joaquin"
                  "A rectangular sheet suggest roof because it's form")))

(term :name         roof5
      :comment      "(Yet another example of roof)"
      :category     roof
      :features     (rectangular_sheet)
      :typicality   1.00)

(term :name         triangle_base_pyramid
      :remindings   ((roof 0.48) (house 0.01))
      :relations       (((triangle_base_pyramid  sometimes  suggests  roof)
"Joaquin"
                  "A  triagle  base  pyramid  suggest  roof  because  of  it's
form")))

(term :name         roof4
      :comment      "(Yet another example of roof)"
      :category     roof
      :features     (triangle_base_pyramid)
      :typicality   1.00)

(term :name         cube
      :synonyms     (box)
      :remindings   ((roof -9999.00) (walls 0.78) (house 0.01))
      :relations    (((cube always suggests walls) "Joaquin"
                  "A cube suggest a house's walls because of it's form.")
                     ((cube  has  typical  generalization  parallelopypid)
"Joaquin"
                  "A cube is a type of parallelopypid")))

(term :name         walls1
      :comment      "(Yet another example of walls)"
      :category     walls
      :features     (parallelopypid)
      :typicality   2.25
      :differences ((humble_house <-- (rectangular_sheet))
                (proto_house <-- (prism))))

(term :name         cylinder
      :remindings   ((walls 0.45) (house 0.34))
      :relations    (((cylinder sometimes suggests walls) "Joaquin"
                  "A cylinder  suggest   walls  of  a  house  because  of  it's
form.")))
```

```
(term :name        walls2
      :comment     "(Yet another example of walls)"
      :category    walls
      :features    (cylinder)
      :typicality  1.00)

(term :name        parallelopypid
      :remindings  ((roof -.75))
      :relations     (((parallelopypid has typical specialization cube)
"Joaquin"
                 "A cube is a type of parallelopypid")))

(term :name        indian_house
      :comment      "(This is an example of an indian house, i.e. a cone
over a cylinder)"
      :category    house
      :features    (cone cylinder)
      :typicality  1.00
      :differences ((walls1 <-- (cube))))

(term :name        small_house
      :comment     "(This is a small house, e.i. a pyramid over a box)"
      :category    house
      :features    (cube square_base_pyramid)
      :typicality  1.00)

(term :name        dome_house
      :comment     "(This is a dome-like house,i.e. a semi-cylinder over a
parallelopypid)"
      :category    house
      :features    (semi-cylinder parallelopypid)
      :typicality  1.00
      :differences ((proto_roof <-- (prism))))

(term :name        humble_house
      :comment     "(This is an example of a humble_house)"
      :category    house
      :features    (parallelopypid rectangular_sheet)
      :typicality  1.00)

(case :name        two_story_house
      :comment     "(This is a two story house)"
      :creator     joaquin)

(term :name        story_1
      :remindings  ((house 0.70))
      :relations   (((story_1 equivalent walls) "joaquin"
                 "Story_1 are the walls of the first floor")))

(term :name        story_2
      :remindings  ((house 0.70))
      :relations   (((story_2 equivalent walls) "joaquin"
                 "Story 2 are walls of the seccond floor")))

(case :name        two_story_house
      :category    HOUSE
      :preclassify T
      :features    (story_1 story_2 prism)
      :disposition (BECAME TWO_STORY_HOUSE)
      :comment     "(This is a two story house)"
      :creator     joaquin)
```

```
(term :name        two_story_house
      :comment     "(This is a two story house)"
      :category    house
      :features    (story_1 story_2 prism)
      :typicality  1.00)
```

Appendix D

Implementation Guidelines

D-1 Ontolingua - Nextpert Translation

Both Ontolingua and Nextpert Object are Object Oriented (O.O.) and Rule-Based, and use an intermediate (programmer readable) language for representing both objects and rules in lisp like format. Both also have a pretty good interface for defining and editing ontologies in a "led by hand" mode, essential for a good quality knowledge design. The main difference is the expressiveness and the purpose of the underlying languages:

* In Ontolingua, KIF (Knowledge Interchange Format) is a very expressive language, that was not designed for KP but as an interchange format of knowledge. It is *self defined* by a KIF ontology and it is extensible in any extent due to the way it was designed. A good example is the Frame Ontology that is the basic data structure for KR in Ontolingua. Its application independent and as general as it can be for facilitating the translation between different sources of knowledge. It supports non-monotonicity, that is the capability of drawing conclusions based on the absence of knowledge from a database (incomplete information).

* In Nexpert Object, Nexpert is an application oriented language that is used for KR and KP as well. This makes it more complicated to understand and limits it's extensibility. On the other hand, its O.O. characteristics are more powerful as it supports methods for describing behavior and control of individual slots, objects or set of objects. This, in principle is not supported by KIF nor Ontolingua. It supports non-monotonicity as well.

Analogies were found not at the underlying language level but at the knowledge input level and the data structures used to represent knowledge. The constructions of ontologies in both systems have similar pattern, making it simpler, after understanding the analogies, to translate ontologies from one system into another. Some times, depending which is the source and the target system, some information might be incomplete.

Here is a list of all the mayor concepts needed to be understood in order to build any basic ontology in Ontolingua. Beside each question I will add, between parenthesis, what *possibly* are relative or equivalent concepts in the Nexpert Object system. For further anaysis please refer to the bibliography.

In Ontolingua.-

What is an Ontology? (A Knowledge Base)

An ontology is an explicit specification of some topic. For our purposes, it is a formal and declarative representation which includes the vocabulary (or names) for referring to the terms in that subject area and the logical statements that describe what the terms are, how they are related to each other, and how they can or cannot be related to each other. Ontologies therefore provide a vocabulary for representing and communicating knowledgeabout some topic and a set of relationships that hold among the terms in that vocabulary.

What is a Frame? (An Object)

A frame is a named data structure (or object) which is used to represent some concept in a domain. A frame can be a class, instance, slot, facet, function, relation, or axiom. A frame allows one to group some related statements about that concept. Associated with each frame is a group of slots, facets, and values on the slots or facets.

What is a Term? (An Object)

A term is any object that has a definition (e.g., slots, classes, instances, relations, functions). An axiom is not considered a term.

What is a Class? (A Class)

A class is a representation for a conceptual grouping of similar terms. For example, a computer could be represented as a class which would have many subclasses such as personal computers, mainframes, workstations, etc. Each class is described by a definition which specifies the slots and values that describe the class itself (not the members of the class), slots and values that describe the instances of that class, and logical statements (called axioms) that describe the class but can't be represented using slots and values.

What is an Instance? (An Object in its pure sense)

All of the terms in an ontology that have an associated definition (i.e.classes, slots, relations, functions, facets) are an instance of some class. Classes are instances of Class, functions are instances of Function, etc. An instance should not be confused with an Individual because an instance may be a class whereas an Individual cannot be a class.

What is a Facet? (A Meta-Slot)

Facets are used to represent information about a slot on an object. Usually facets represent some constraint on an instance slot. For example, Slot-Value-Type is a facet which can be used to represent that for instances of the class Person, the value of the slot Has-Mother must be an instance of Female-Person. The most commonly used facets are: Slot-Cardinality, Minimum-Slot-Cardinality, Maximum-Slot-Cardinality, and Slot-Value-Type.

What is an Instance Slot? (A Slot)

An instance slot is a slot on a class that is used to describe a property of the instances of a class rather than the class itself. For example, mass would be an instance slot because it is used to describe how much a particular instance weighs rather than what the class weighs (obviously a class doesn't have mass since it is an abstract concept). Whereas, subclass-of is an own slot because it is used to describe the class itself.

What is an Own Slot? (A Class Slot)

Own Slots are slots used to describe properties of the term itself. For most terms (i.e., any term except a class), the only slots they have are own slots. However, most classes also have instance slots that are used to describe properties of the instances of a class rather than the class itself. For example on the class Mother, subclass-of is an own slot because it is used to indicate a property of the class Mother. Whereas, has-children would be an instance slot because it describes a property of instances of the class Mother.

What is a Value? (A Special type of property - named "Value")

A value is a term which is related to the current definition through a slot or facet. For example, FemalePerson would be the value of the slot subclass-of for the class Mother. As another example, 1 would be the value of the facet Minimum-Slot-Cardinality for the slot Has-Children on the class Mother.

What is a Relation? (Is not defined explicitly in Nexpert Object)

A relation is used to describe a relationship among two or more terms. If a relation represents a relationship between only two terms, it is called a slot or a binary relation. If the relation describes a relationship among n terms such that there is a unique nth term corresponding to any set of the first n-1 terms, then the relation is called a function.

What is a Function? (Is not defined explicitly in Nexpert Object)

A function is a special type of relation which relates some number of terms to exactly one other term. That is, a function is a relation such that no two relationships of n terms in the relation have the same first n-1 terms. For example, mother is a function that relates an animal to exactly one female animal. A function may also be referred to as a slot if it relates only two terms.

What is Domain? (Is not defined explicitly in Nexpert Object)

The domain of a slot is a class that restricts the terms on which the slot can be added. A slot can only be added to terms which are an instance-of its domain.

Note that the domain is typically a superset of the exact-domain of a relation. For example, one can say that the domain of Mother-Of is Female-Animal, but since not all female animals are mothers, Female-Animal could not be the exact-domain of Mother-Of.

What is Range? (Is not defined explicitly in Nexpert Object)

The range of a relation is a class that restricts the values which the relation can have. A value can only be added to the relation if it is an instance-of its range.

Note that the range is typically a superset of the exact-range of a relation. For example, one can say that the range of Has-Mother is Female-Animal, but since not all female animals are mothers, Female-Animal could not be the exact-range of Has-Mother.

NOTE: *Relations, functions, domains and ranges* are implicitly defined when creating Classes and defining rules in Nexpert Object.

What is an Axiom? (A Rule)

An axiom is a sentence in first order logic that is assumed to be true without proof. In practice, we use axioms to refer to the sentences that cannot be represented using only slots and values on a frame.

Axioms must be entered in prefix notation. Use => to indicate logical implication, <=> to indicate logical equivalence, and to indicate conjunction, or to indicate disjunction, not to indicate negation, and exists to indicate existential quantification. Free variables are assumed to be universally quantified. Variable names must start with a question mark.

For example, you would represent the statement:

If any two animals are siblings,
then there exists someone who is the mother of both of them.

With the axiom:

```
 (=> (sibling ?sib1 ?sib2)
   (exists (?mom) (and (has-mother ?sib1 ?mom)
                  (has-mother ?sib2 ?mom)))
```

What is an Augmented Definition?

An augmented definition is rather like any other definition, only it is a specialization of a definition made in a different ontology. For example, if I have an ontology of vehicles, and another ontology of police vehicles that includes the ontology of vehicles, I might want to augment the definition of the class Vehicle so that all vehicles have radios and guns. This can be done by augmenting the definition of the class Vehicle in the police-vehicles ontology.

On The Nexpert Object side.-

We have several differences as follows:

METHODS: As said before methods describe the behavior of individual slots, objects, or sets of objects. Methods are composed primarily of a set of actions which when executed modify the behavior of the object upon which they act.

There is an effort for *"Extending Otolingua for Representing Control Knowledge"* by Eliana Cohelo and Guy Lapalme to be presented in IJCAI-95, but actually because KIF can be extended for an specific application, I don't see any problem in creating a "method ontology".

RULES: In Nexpert, rules have three basic parts:

- Left-hand side conditions
- The Hypothesis
- The right-hand side actions

Although this does not differs from any rule definition, in Nexpert the **Hypothesis**, is a boolean slot and it is treated as a special object as seen in the application interface.

Rules, in Nextpert, represent among other things: relations, heuristics, procedural knowledge, and temporal structure of knowledge. This might suggest that Ontolingua's *relations* and *functions* might be some kind of rules, but focused in an ontological way this is not true.

Partial Conclusions

Because the representation of Control Knowledge is not essential for the meanings of this project (building an 3-D Shape Ontology) and there is a way of interpreting Nextpert's KR Data Structures into Ontolingua's own data structures it is feasible to use Ontolingua for this project.

Bibliography

1) *Ontolingua,* on-line system "help", KSL-Interactive Network Services, URL:

http://ksl.stanford.edu

2) Neuron Data, Nexpert Object system manuals: *"Knowledge Design"*.

3) Michael R. Genesereth, Richard E. Fikes, *"Knowledge Interchange Format, Version 3.0, Reference Manual"* , Report Logic-92-1 , C.S. Department, Stanford University

D-2 Knowledge Level and Symbol Level Interface

Here is a rough description of what the interface between the symbol level and the knowoledge level might require. First I would like to remind you some important points:

- A case (or exemplar) description is a collection of features that represent attributes that the TEACHER (user) thinks are potentially RELEVANT to its classification.

- Each feature represents an independent attribute of the case.

- If RELATIONSHIPS BETWEEN ATTRIBUTES are relevant to the case's classifications in our system), they must be included in its featural description. The CONSISTENCY of these relations have to be enforced by the teacher (or by the system).

Here is an example:

Case: ARCH /* Formed by three blocks */

Features: block(a), block(b), block(c), above(c,a), above(c,b)

touches(c,a), touches(c,b), doNotTouch(a,b)

These relations have to be consistent with the VR world!!

What Is needed for the integration:

1) That Nextpert Object or whatever, decides and initiates the classification (or learning) process, by giving me the list of all the FEATURES of the new inputed case. Therefor it has to be decided when a newly formed case is ready for classifying. If there is an analogous case in the case base then PROTOS will create an hypothesis of what the case is.

2) That an interactive conversation could be held between the computer and the teacher in the learning process mainly for imputing domain knowledge as an explanation (using a fixed vocabulary). Note that the user has to be able to refer to any component of a complex object in an unambiguous way.

To get a grasp of what is intended to say in point 2 here is an example of a very idealized conversation with PROTOS:

SYSTEM: Protos, a new case has the features roof, walls, window, door, antenna, above(roof,walls),touches(roof,walls)

PROTOS: Based on this description, I am most strongly reminded of the category HOUSES. I will retrieve a past example, House-01, and will try to explain the similarity of these two cases by comparing their features. Many of the features of the new case match House-01 directly, but there are still some features I cannot match. First, House-01 does not have window.

TEACHER: That is true. However a window ENABLES a person inside a house to see to the outside, so it is a relevant feature to house in general.

PROTOS: I now believe that having a window is a moderated feature for HOUSES. In future cases if a window appears it will remind me of houses.

TEACHER: O.K.

PROTOS: Second, House-01 does not have antenna.

TEACHER: An antenna is not a important feature for a house (it's spurious).

PROTOS: Since this case doesn't differ from House-01 in many respects, I will not save it as a new exemplar of HOUSES. Is this a satisfactory disposition of this case?

TEACHER: Yes.